IMPLEMENTING LOW-LEVEL BROWSER-BASED SECURITY FUNCTIONALITY

Master Thesis

submitted to

Prof. Dr. Stefan Katzenbeisser
Fachgebiet Security Engineering
TECHNISCHE UNIVERSITÄT DARMSTADT

by

Ben Stock

Advisors: Dr. Martin Johns (SAP Research), Dr. Martin Mink

January 2013
Contents

List of Figures iv
List of Tables v
List of Listings vi

1 Introduction 2
   1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
   1.2 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
      1.2.1 Extensions of the Same-Origin Policy . . . . . . . . . . . . . . . . 3
      1.2.2 Detection of DOM-based Cross-Site Scripting . . . . . . . . . . . . 4
   1.3 Organization of this thesis . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2 Technical background 6
   2.1 HTML and its representation inside a browser . . . . . . . . . . . . . . . 6
   2.2 JavaScript . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
   2.3 Document Object Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
   2.4 The Same-Origin Policy . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
   2.5 XmlHttpRequests and Cross-Origin Resource Sharing . . . . . . . . . . . 13
   2.6 DNS Rebinding . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
   2.7 Cross-Site Scripting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
      2.7.1 Reflected Cross-Site Scripting . . . . . . . . . . . . . . . . . . . . . 17
      2.7.2 Stored Cross-Site Scripting . . . . . . . . . . . . . . . . . . . . . . 18
      2.7.3 DOM-based Cross-Site Scripting . . . . . . . . . . . . . . . . . . . . 19
   2.8 Taint tracking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

3 Layout of the Chromium web browser 29
   3.1 V8 JavaScript engine . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
   3.2 WebKit rendering engine . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
      3.2.1 Implementation of the HTML parser . . . . . . . . . . . . . . . . . . 33
      3.2.2 Implementation of the Document Object Model API . . . . . . . . . 33

4 Extending the Same-Origin Policy 36
   4.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.2 Concept of the extended Same-Origin Policy ... 37
4.3 Implementation ............................................. 38
  4.3.1 Script access to a viewport ....................... 39
  4.3.2 XmlHttpRequests ........................................... 40
  4.3.3 Modifications and performance ..................... 41
5 DOM-based Cross-Site Scripting detection using taint-tracking 43
  5.1 Basic concept ................................................ 43
  5.2 Implementation .............................................. 45
    5.2.1 First approach: code representation ............. 46
    5.2.2 Second approach: character-wise tagging ....... 50
    5.2.3 Evaluation ............................................... 66
6 Conclusion 69
  6.1 Summary ..................................................... 69
  6.2 Contributions ................................................ 69
    6.2.1 The extended Same-Origin Policy .................. 69
    6.2.2 Detecting DOM-based Cross-Site Scripting vulnerabilities .... 70
  6.3 Outlook ....................................................... 70
Bibliography 72
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Illustration of the methodology of the Same-Origin Policy</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>DNS Rebinding steps</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>Illustration of a Reflected Cross-Site Scripting attack</td>
<td>18</td>
</tr>
<tr>
<td>2.4</td>
<td>Illustration of a Stored Cross-Site Scripting attack</td>
<td>20</td>
</tr>
<tr>
<td>2.5</td>
<td>Parts of the URL explained</td>
<td>20</td>
</tr>
<tr>
<td>2.6</td>
<td>Illustration of a DOM-based Cross-Site Scripting attack</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>Overview of Chromium components</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>Implementation logic for XHR</td>
<td>41</td>
</tr>
<tr>
<td>5.1</td>
<td>Overview of exemplary data flow</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>Flow chart showing the lookup of document.location.hash</td>
<td>47</td>
</tr>
<tr>
<td>5.3</td>
<td>Illustration of where shadow data is stored</td>
<td>52</td>
</tr>
<tr>
<td>5.4</td>
<td>Illustration of the shadow information layout</td>
<td>59</td>
</tr>
<tr>
<td>5.5</td>
<td>Illustration of the HTML tree before and after executing the code in</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>Listing 5.11</td>
<td></td>
</tr>
<tr>
<td>5.6</td>
<td>Illustration of taint information assignment</td>
<td>63</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Same-Origin Policy for access by http://example.org/ex/index.html . . . 13
5.1 IPC Codes used in external lookup service . . . . . . . . . . . . . . . . . . 49
5.2 Initial list of source identifiers used for our taint-tracking approach . . . . 51
5.3 Bit sequence used to represent encoding functions . . . . . . . . . . . . . . 59
List of Listings

2.1 Example of functions and objects in JavaScript .......................... 7
2.2 Example of overwriting properties during runtime ...................... 8
2.3 Example of two classes in JavaScript ................................. 10
2.4 Example defining the function \texttt{area} in the prototype ............... 10
2.5 Example changing the prototype .................................. 11
2.6 Example of a simple vulnerable PHP application ...................... 17
2.7 Example showing DOM-based XSS .................................. 21
2.8 JavaScript Code Execution using events ............................ 21
2.9 Simple HTML example of navigation links implementing in JavaScript .. 24

3.1 Example JavaScript code to demonstrate the usage of hidden classes ... 31
3.2 Excerpt of the IDL file used to define the interface to arbitrary HTML elements ................................................. 34

5.1 Code example illustrating the code representation approach .......... 46
5.2 JavaScript code example leading to rivaling access paths ............ 50
5.3 Code excerpt showing the functions for setting and getting taint tags ... 53
5.4 Code excerpt showing the function for retrieving the taint information in consecutive strings ........................................... 54
5.5 Function implementing the tag copy process ........................ 55
5.6 Example code causing issues with central string storage .......... 56
5.7 Example code demonstrating pitfalls in use of unencoded data .... 57
5.8 C++ code implemented to allow adding of function tags .......... 58
5.9 C++ code implemented to allow clearing of function tags ....... 59
5.10 WriteTag function responsible for transfer of the taint information from V8 to WebKit ........................................... 61
5.11 Code example to explain tokenization process ...................... 62
5.12 Source code for \texttt{StringImpl::SetAllTag} ..................... 64
5.13 Complex test case to verify the functionality of our taint-tracking ... 67
5.14 Resulting log output when executing the test case code shown in Listing 5.13 67
Abstract

The principal policy in regards to isolation between web applications is the Same-Origin Policy (SOP). It provides said isolation employing the notion of origins to prevent unauthorized interaction between any two websites. However, there are two ways of circumventing the SOP, namely DNS Rebinding and Cross-Site Scripting. In this thesis, we try to tackle the issues caused DNS Rebinding and provide a prototyped implementation of a tool capable to detect a relatively new form of Cross-Site Scripting, namely DOM-based Cross-Site Scripting.

In pursuit of this, we propose an extension of the Same-Origin Policy which is able to prevent DNS Rebinding attacks. To verify the validity of our approach, we implemented this extended Same-Origin Policy into the popular Chromium web browser.

To allow for detection of DOM-based Cross-Site Scripting vulnerabilities in web applications, we employ the concept of dynamic taint-tracking during runtime of these applications. We therefore modified the key components of the Chromium web browser, namely the V8 JavaScript engine and the Document Object Model implementation residing in the WebKit rendering engine.

The thesis details all the necessary modifications and evaluates the efficiency of our followed approach. The evaluation of our implemented prototype shows that the concept of dynamic taint-tracking inside JavaScript and the DOM can be used to successfully detected DOM-based Cross-Site Scripting vulnerabilities.
1 Introduction

In this chapter, we shed light on the motivation behind this thesis and provide insight on related work which has been previously achieved. Finally, we give an outlook on the organization of this thesis.

1.1 Motivation

The Same-Origin Policy is the principal policy governing access to resources of mutually distrusting web applications. Its decision is based on the notion of the origins of the involved websites, such that only applications of the same origin can access each others resources. In this, it describes the boundaries applications based on their origin. If the SOP was not used in today’s web, attackers could easily gain access to privileged resources, by requesting access through scripted content embedded into a website under their control. For instance, considering applications like corporate wikis, an attacker could use this to retrieve highly sensitive corporate data. To prevent this scenario, the Same-Origin Policy has been part of the web ever since active content was introduced.

However, attackers trying to circumvent this policy to gain access to privileged information or to execute their malicious code in the context and origin of a given document have different ways of achieving this goal. One of these ways are attacks like DNS Rebinding which allow an attacker to interact with privileged content, whereas Cross-Site Scripting (XSS) enables the attacker to run his potentially harmful code in the context of the vulnerable web application. The goal of this thesis is to provide technology which effectively blocks attacks like DNS Rebinding and allows detection of a relatively new form of Cross-Site Scripting, namely DOM-based Cross-Site Scripting.

Although DNS Rebinding has been known since 1996 [38], its root cause has not been solved thus far. This is also shown by the fact that new attacks exploiting this vulnerability have been found in 2002 [39] and 2006 [40]. The proposed solution of DNS Pinning employed by modern browsers to prevent DNS Rebinding comes with several pitfalls [41]. Hence, the need for a different Same-Origin Policy arises, which provides better protection against the aforementioned attack. In this thesis, we therefore propose an extended Same-Origin Policy which is immune to DNS Rebinding attacks.

According to the OWASP Top 10 Vulnerability report from 2010 [37], Cross-Site Scripting is the second most prevalent type of vulnerability in web applications. The effects of Cross-Site Scripting range from leakage of privileged data to worm-like distribution of the attack code as shown by the XSS worm Samy, which exploited a vulnera-
bility in the well-known social networking site *myspace* [32]. For both classical types of Cross-Site Scripting, namely Reflected and Stored XSS, a number of tools for detecting vulnerabilities has been provided by researchers [47, 48]. However, due to the fact that the third kind of Cross-Site Scripting, DOM-based Cross-Site Scripting, was discovered quite recently and is not as prevalent as the two classical types, almost no tools exist which allow discovery of these vulnerabilities. Hence, another goal of this thesis is to implement a tool capable of dynamically analyzing web application to find possibly harmful vulnerabilities.

The goal of this thesis is to implement browser-based security functionality aimed at defeating and detecting the aforementioned attacks. Therefore, we will provide

- an enhancement of the Same-Origin Policy allowing it to become immune to DNS Rebinding attacks as well as the corresponding implementation and
- the design and implementation of a tool capable of detecting DOM-based Cross-Site Scripting vulnerabilities.

### 1.2 Related Work

In this section, we will present related work by other researchers. Since this thesis covers both problems with the Same-Origin Policy in regard to DNS Rebinding as well as an approach to detecting DOM-based Cross-Site Scripting, this section is divided accordingly.

#### 1.2.1 Extensions of the Same-Origin Policy

Different approaches to solving the problems caused by DNS Rebinding have been undertaken. The approach presented by Karlof et al. [50], called the "Strong Locked SOP", suggests extending the Same-Origin Policy using information extracted from TLS/SSL certificates. The concept they present makes use of so-called *locked web objects*. In their notion, a locked web object is an object which was retrieved over an encrypted connection and thus can be tied the corresponding certificate of said secure connection. For active content to gain access to such a locked object, not only the origins in the sense of the Same-Origin Policy need to match. Rather, an object accessing a locked object must have the same certificate information attached to it. Since an attacker is not able to deduce the private key of the web server involved, only the web server is able to produce valid certificates for the involved resources. Hence, the verification of the involved objects’ certificates is sufficient to ascertain whether they stem from the same server, and thus the same origin. However, the approach presented in this publication lacks support for standard HTTP connections since it is based purely on verification of the cryptographic certificates used in HTTPS.
1 Introduction

Jackon et al. [31] propose another means of tackling the problems caused by DNS Rebinding through the usage of reverse DNS lookups. In the Domain Name System, a so-called A record, represents a relation of a given domain name to one or multiple IP addresses [30]. Its counterpart, representing the relation between an IP address and a domain name, is called a PTR record and is also known as the reverse lookup of an IP address. In their paper, Jackson et al. suggest browsers use these reverse lookups to ensure that the IP address they are interacting with resolves back to the given domain name. However, this proposed solution relies on the DNS system itself to provide additional information and requires additional connection overhead to perform the reverse lookup.

1.2.2 Detection of DOM-based Cross-Site Scripting

The second part of this thesis covers our approach to detecting DOM-based Cross-Site Scripting vulnerabilities in web applications by employing taint-tracking.

One commonly used example of the use of taint-tracking is the Perl Programming Language. Its employs taint-tracking capabilities to provide security when dealing with untrusted data in potentially harmful functions [49]. This feature is automatically activated when a program is run with differing real and effective user and group IDs, but can also be activated manually. Specifically, when the taint-tracking is active, Perl traces all user-provided data throughout the program and executes additional checks before this untrusted data is passed to certain security-critical functions. In doing so, Perl effectively prevents user-provided data to cause changes outside the program itself.

The idea of using taint-tracking to provide security in applications was taken up in 2007 by Vogt et al. in their paper "Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis" [34]. They propose to use dynamic taint-tracking inside the web client to detect and stop Cross-Site Scripting attacks. In their notion, the main goal of Cross-Site Scripting is for an attacker to gather sensitive information from the client, such as a cookie. Since most web applications use cookies to store session information which allow authorization of a user, it is a popular target for attackers. To prevent this kind of information from leaking in an application prone to Cross-Site Scripting, they taint sensitive data. Once the borders of the web application are reached and any data is to be sent to a third party, their mechanism checks to see if said data has been tainted. If so, the user is prompted to allow or disallow the transfer of the sensitive data. In this way, they add a new layer of protection to the browser, capable of stopping the aforementioned effect of Cross-Site Scripting.

A recently developed tool aimed at detecting DOM-based Cross-Site Scripting is the DOMinator Project [45]. To allow detection of DOM-based XSS vulnerabilities, the authors extended the Mozilla Firefox browser to employ dynamic runtime tainting of strings and corresponding taint propagation. Their tool is made up of two components, namely additions to the C++ runtime code responsible for executing JavaScript and
allowing access to the DOM, and a JavaScript extension aimed at providing additional custom functionality such as logging and analysis.

In 2009, Chin and Wagner [46] presented an approach to use taint-tracking inside Java applications to defend against command injection attacks. The focus of their work lies on the detection of all kinds of injection attacks on the server-side. To achieve this, they propose a character-level tainting of all user-provided data, whereas their notion of user-provided data is any data sent in an HTTP request. Hence, they augment the Servlet API class used in conjunction with Java web applications to mark all strings stemming from HTTP requests as tainted. Beside this modification, they change the inner workings of the Java String class to allow taint propagation. To allow proper propagation, they explicitly taint every character in a given string rather than the string object itself. This way, if only parts of a tainted string are used in the execution of the program, the taint information is forwarded. Finally, they propose the augmentation of all security relevant functionality, such as database access or command invocation, to allow the detection of flows of tainted data into them.

1.3 Organization of this thesis

This thesis is structured in six chapters. Firstly, we give an introduction to the topics covered and provide an overview over related work on the subjects. In Chapter 2, we explain the technical background relating to the concepts provided and discussed in the thesis. The next chapter lays the groundwork towards understanding the implementation and the corresponding issues that were met in creation of our work, covering the layout of the Chromium web browser as such and providing details on the V8 JavaScript engine and the implementation of the Document Object Model API in the WebKit rendering engine. In Chapter 4, we describe the concept of the extended Same-Origin Policy and outline the details of the corresponding implementation we built into Chromium. After this, we depict the work done in conjunction with augmenting the Chromium web browser to allow dynamic taint-tracking in order to detect DOM-based Cross-Site Scripting, presenting the conceptual idea behind our approach as well as details on the implementation. The chapter is concluded with an evaluation of our approach. Ultimately, in Chapter 6 we present a summary of the thesis and its contributions as well as an outlook on future work regarding the presented topics.
2 Technical background

This chapter covers selected topics needed to effectively discuss this thesis. First, HTML and its representation inside a browser is covered and a short introduction to JavaScript is given. This is followed by the basics of the Document Object Model. Afterwards, the Same-Origin Policy, DNS Rebinding and Cross-Site Scripting are presented.

2.1 HTML and its representation inside a browser

The basic language used in the web is the Hypertext Markup Language (HTML). HTML as such is a plain text format to allow easy transfer from server to client. HTML consists of nested elements identified by tags. Each element can contain multiple attributes consisting of either only a name or a name and a value as well as multiple enclosed elements. This relation between the elements can also be shown in a tree-like structure as each element can have at most one parent element. The html element denotes the root of this tree, the content to be displayed to the user is to be written in the body element.

For HTML 4.01, there are three different Document Type Definitions (DTD), namely Strict, Transitional and Frameset. The entity governing the HTML standard, the World Wide Web Consortium (W3C) proposes that authors of web sites should use the Strict DTD but may fall back on the Transitional DTD. According to the W3C, the Strict DTD ”excludes the presentation attributes and elements that W3C expects to phase out as support for style sheets matures” [1], whereas the Transitional DTD allows for these. The Frameset DTD is identical to the Transitional DTD, except for the frameset element replacing the body element. Currently, the next version of the HTML standard – HTML 5 – is under development and has the status of a Working Draft.

Upon loading a website, the browser starts to parse the content and generates HTML tokens - this process is also called tokenization. From these tokens and their relations to each other, the HTML tree is formed and sent to the rendering engine of the browser. The rendering engine then displays the website by traversing the tree and showing the appropriate content for each node. The content to display is controlled by a multitude of factors like actual, textual content as well as information on style like the font size or color.

As the content created by the author of a site might not be well-formed according to the HTML standard, modern browsers implement their tokenizer in an error-correcting
manner. This ensures that small errors in the HTML source do not keep the browser from rendering the page at all.

2.2 JavaScript

JavaScript was first introduced under the name LiveScript in 1995, when Netscape implemented a simple scripting language into their Navigator web browser [20]. This new scripting language should enable otherwise static web sites to interact with a user and to change their own content dynamically. Nowadays, JavaScript is standardized under the name ECMAScript in Standard ECMA-262 [24]. At the time of writing this thesis, the version in use is 5.1.

JavaScript does not implement a object-oriented class concept but allows adding and removing properties to and from objects during runtime. In JavaScript, functions serve both as functions and constructors for objects. Objects in JavaScript can contain multiple properties. Such properties can either be values of basic types – like integers or strings –, other objects or functions. Listing 2.1 shows a basic example of a JavaScript object.

```
1 function DoB(day, month, year) {
2   this$retString = function() {
3     return day+"."+month+"."+year;
4   }
5   this.day = day;
6   this.month = month;
7   this.year = year;
8 }
9
10 function Person(name, dob) {
11   this$retString = function() {
12     return "My name is "+this.name+", my birthday is "+this.dob$retString();
13   }
14   this.name = name;
15   this.dob = dob;
16 }
17
18 var birthday = new DoB(24, 4, 1985);
19 var ben = new Person("Ben", birthday);
20 console.log(ben$retString());
```

Listing 2.1: Example of functions and objects in JavaScript

The above example shows all types of properties. The object DoB stores three values of the basic type integer. On the other hand, the Person object stores a string (name) as well as another object. The call in line 20 references the property retString, which
technical background

is called as a function as denoted by the parentheses behind the property’s name. The function itself in turn calls the `retString` function of its own property `dob`, which is a reference to an object created from the function `DoB`.

With classical object orientation, member functions of objects cannot be overwritten at runtime as the structure of any class and any of its objects must be defined before the program is started. In contrast to this, JavaScript allows for dynamic changes in the structure of objects during runtime. Considering the code shown above, we add the following new lines of code to our program.

```javascript
1 ben.retString = function () {
2   return "I have been overwritten during runtime.";
3 }
4 console.log(ben.retString())
```

Listing 2.2: Example of overwriting properties during runtime

The instruction in line 1 of Listing 2.2 overwrites the property `retString` of the object `ben` with the newly created function. Hence, when `retString` is called, it returns `I have been overwritten during runtime`. However, it does not change the original definition of the property defined in line 11 of Listing 2.1. Thus, any new objects that initiated by invoking the original `Person` function will still return the string `My name is ....`

As we have shown above, changing the property of an object does not change the property in the function this objects was created from. However, the definition of a property in the original function cannot be overwritten during runtime. Upon creation of a new object from a function, the properties of this new object are always generated from the original function definition. Thus, if a property should be removed or overwritten, this has to happen for each and every object created from the function.

Prototyping To allow more flexibility in its programs, JavaScript implements prototyping instead of classical inheritance as known from object-oriented languages like Java. Apart from the object types built into JavaScript (for example String or Number), objects can define a prototype. According to the ECMAScript standard a prototype is an "object that provides shared properties for other objects" [25]. Fundamentally, any function that is created in JavaScript is merely an object of type `Function`. Therefore, by default, the prototype of a constructor function is the Function object, which implements basic functionality like representing the function as a string [27]. However, the prototype can be changed to point to another object instead. For example, let us assume the following: We define a constructor function `func1` which implements no properties and a constructor function `func2` which implements a property `prop1`. From `func1`, we create an object called `obj1`. If we try to access the property `obj1.prop1`, the interpreter will throw an error as this property was not defined by the constructor function.
However, if we change the prototype of `func1` to `func2` and again instantiate an object `obj2` from `func1`, access to the property `prop1` is now possible as it was defined in the prototype of `func1, func2`. This simple example can be extended by adding another function and setting the prototype for `func2` to the newly created function. This way, we create a so-called prototype chain. On access to a property on an object `obj`, the interpreter will walk this chain upwards starting from the object `obj` until the property is found. If the prototype is set to NULL, the interpreter will throw an error stating that the requested property could not be resolved.

As we described above, the interpreter will try to resolve a property inside the prototype chain. This can be used by a programmer to implement common functionality used among different functions inside one object and modify the prototype chains for the other objects accordingly. This way, a programmer needs to implement this basic and shared functionality only once. In the following, we will discuss how prototyping works as shown on an example.

Listing 2.3 shows three different constructor functions, namely `Rectangle`, `Square` and `Circle` as well as objects instantiated from them. In lines 20 to 22, the property `area` of all three newly created objects is called as a function. Obviously, the call to `c.area()` works, as `c` was instantiated from function `Circle`, which defines the property `area` in line 13. However, the calls to `c.area()` and `s.area()` will fail as the corresponding property it not defined in either `Rectangle` or `Square`. In order to implement the functionality for the calculation of the area for these figures, we have different alternatives.

We can either define the property `area` inside the constructor functions or the objects themselves or we can change the prototype chains for the constructor function `Rectangle` and `Square`. As shown in Listing 2.2, we can simply overwrite or add a property to an existing object after it has been created. We can however not modify properties of the constructor after definition, so we would have to change the definition of the constructor function `Rectangle`.

The more elegant way to achieve this is to use prototyping. In Listing 2.4, we show a simple example which adds only the functionality for calculating the area. We first create a function called `commonarea` which implements the necessary calculation. Note that calling this function directly would return the value for `not a number` (NaN) [26] as the properties `width` and `height` were not defined inside the function. In lines 5 and 6, we define the property `area` for both prototypes of `Rectangle` and `Square`, whereas in line 7 we do the same for `Circle`. If these lines are added to the example in Listing 2.3 before instantiation of `r` and `s` respectively, the property access of `area` will now work as the property was defined inside the prototype chain. However, as discussed earlier, the interpreter will try to locate the requested property starting from the accessed object.

Note that for objects instantiated from the constructor function `Circle`, the property `area` is defined. Therefore, execution of line 20 from Listing 2.3 would result in the function from line 13 being called and not the newly added `commonarea` function.
2 Technical background

```javascript
function Rectangle(w, h)
{
  this.width = w;
  this.height = h;
}

function Square(w)
{
  this.width = w;
  this.height = w;
}

function Circle(d)
{
  this.diameter = d;
  this.area = function()
  {
    return (d/2) * (d/2) * 3.141592;
  }
}

r = new Rectangle(2, 3);
s = new Square(2);
c = new Circle(2);

console.log(c.area());
console.log(s.area());
console.log(r.area());
```

Listing 2.3: Example of two classes in JavaScript

```javascript
function commonarea()
{
  return this.width * this.height;
}

Rectangle.prototype.area = commonarea;
Square.prototype.area = commonarea;
Circle.prototype.area = commonarea;
```

Listing 2.4: Example defining the function area in the prototype

Another possibility to implement such functionality is to not add or modify one property in a given function’s prototype but to change the prototype completely. The example in Listing 2.5 shows this approach. Firstly, we define a variable figure, which contains an object implementing functions area and edgelength. In lines 6 and 7, we then change the prototype of both constructor functions Rectangle and Square. This way, all objects created from either of these functions will now contain the properties area and edgelength as defined in the example.

Next to prototyping for constructor functions as defined by ECMAScript, JavaScript also allows prototyping for already created objects. To change the objects prototype, a programmer can modify the __proto__ property and set it to any object [28]. Chang-
2 Technical background

```javascript
1 figure = function() {
2   this.area = function() { return this.width * this.height; }
3   this.edgelength = function() { return this.width * 2 + this.height * 2; }
4 }
5
6 Rectangle.prototype = figure;
7 Square.prototype = figure;

Listing 2.5: Example changing the prototype
```

changing the prototype for a specific objects however does not change the prototype for the constructor function.

**Dynamic code execution** Like other scripting languages like PHP [22] or Python [21], JavaScript also allows to run code generated from strings during runtime to be executed, which allows for a great deal of flexibility for a programmer. For example, the programmer can write his web application in such a way that new JavaScript code is downloaded from the server after the site has finished loading. The received code can then be executed directly calling `eval`. This way, the basic JavaScript functionality of a web application can be kept simple and the more complex parts can be loaded and evaluated as needed on the fly.

In this section, we discussed the basics of JavaScript, first and foremost the dynamic nature of this scripting language and the specifics on prototype chaining, JavaScript’s take on inheritance. In the following, we will discuss the Document Object model which is used to allow JavaScript access to the HTML tree.

2.3 Document Object Model

One of the basic blocks of each browser is its implementation of the Document Object Model or short DOM. As explained in the previous sections, a HTML document is represented in the form of tree inside a browser which can be changed by active content like JavaScript. Initially, the Document Object Model was implemented to ensure that access to the HTML tree was done in a structured manner. Therefore, it was designed and engineered as a language-independent interface accessible by any active content.

**DOM Level 1** The Document Object Model currently has three published specifications and one Working Draft. The first specification *DOM Level 1* was published in 1998 and originally provided two different application programming interfaces (APIs): DOM Core [2] and DOM HTML [3]. DOM Core provides the core functionality to access any DOM tree, whereas DOM HTML is the specialized API for handling HTML documents. The DOM HTML API allows for read and write access to nodes in the tree as well as
adding or deleting nodes. Also, it provides functionality to find certain nodes by, for ex-
ample, their name (getElementsByTagName). Apart from the access to the HTML tree,
the DOM API provides methods to read and write the value of the cookie belonging to
the document and get additional information like the referrer. The term referrer herein
adheres to the website that the user came from.

**DOM Level 2** With DOM Level 2, the CORE API was extended by new interfaces
and new methods for the existing interfaces [4]. Together with the changes to the CORE
API, multiple interfaces implemented by the HTML API were changed, making the new
API incompatible with the previous version of it [5]. Also, the Document Object Model
implemented four more specialized APIs. The DOM Style API [7] now allowed the
active content to access and change style sheet documents, whereas the DOM Events
API [8] provided methods to a generic event system like mouse input from the user.
Also, the Events API allows the programmer to trigger certain actions when the page
has finished loading, an element is hovered over or the focus is set on an element. The
DOM Traversal and Range API implemented functionality to traverse the HTML tree
in a structured manner as well as allowing the programmer to operate only on parts of
the tree using the Range API. Alongside these, the DOM Views API was introduced to
enable active content to gather information on how a page’s view was computed.

**DOM Level 3** In DOM Level 3, two more APIs were introduced, namely the DOM Load
and Save API and the DOM Validation API. The Load and Save API enables a script
to load content of an XML document into the DOM at runtime or retrieve a serialized
version of the DOM as XML. The Validation API provides methods for validating that
an update to the DOM by active content leaves the resulting DOM to be valid.

**Summary** To summarize, the Document Object Model or DOM provides methods for
active content to access and dynamically manipulate the HTML tree, the style informa-
tion associated with any node and additional information like the referrer or the cookie
in a structured and well-defined manner. Also, it allows enables a programmer to build
web sites which can react to user input using the Events API.

### 2.4 The Same-Origin Policy

In the early days of the web, web pages only contained static HTML and images. As the
web evolved, active content like JavaScript was introduced to allow pages to dynamically
interact with the user and change their own content after having been loaded. In recent
times, technologies like Flash or Silverlight also enhanced the user experience of web
sites.
2 Technical background

<table>
<thead>
<tr>
<th>Accessed URL</th>
<th>SOP fulfilled</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://intranet.corp/index.html">http://intranet.corp/index.html</a></td>
<td>No</td>
<td>Mismatch in host</td>
</tr>
<tr>
<td><a href="http://example.org:8080/ex/index.html">http://example.org:8080/ex/index.html</a></td>
<td>No</td>
<td>Mismatch in port</td>
</tr>
<tr>
<td><a href="https://example.org/ex/index.html">https://example.org/ex/index.html</a></td>
<td>No</td>
<td>Mismatch in protocol</td>
</tr>
<tr>
<td><a href="http://example.org/index.html">http://example.org/index.html</a></td>
<td>Yes</td>
<td>Path not part of the SOP</td>
</tr>
</tbody>
</table>

Table 2.1: Same-Origin Policy for access by http://example.org/ex/index.html

With the introduction of active content, the need for isolation of web pages arose to disallow access from one site to possibly privileged information on another. Considering an example of two web pages domain1 and domain2, we assume that domain2 contains sensitive information and therefore can only be accessed by authorized users. This authorization might be implemented in form of a password or IP address based authentication. If the pages domain1 and domain2 were not isolated from each other, the malicious page domain1 could retrieve content from the privileged page B and leak the sensitive information back to an attacker.

For this reason, the Same-Origin Policy was introduced. It is the principal security policy of the web and determines isolation borders based on the origin of the involved websites. Any active content – like JavaScript – running on site domain1 can only access content on site domain2 if their origin matches. In conjunction with the Same-Origin Policy the term origin refers to the triple of protocol, domain and port. Figure 2.1 shows the methodology of the Same-Origin Policy. In step 1, the client retrieves the HTML source containing scripted content from domain1 which is subsequently parsed in step 2. The web browser now invokes the JavaScript engine (as indicated in step 3), which tries to access content located on domain2. However, since the domain names for the involved resources do not match, access to the requested resource is blocked. Table 2.1 shows multiple examples of such decisions based on the Same-Origin Policy. In the first example, the external website example.org tries to access the corporate intranet site. Although the protocol (http) and port (default port 80) match, the host given does not match and thus access is disallowed. The second example denotes two sites that have the same host, but they differ in the port used to access each of them. Whereas the third example shows two different origins because of mismatching protocols, the fourth and last URL fulfills the Same-Origin Policy, as the path of the URL does not have to match to fulfill the SOP.

2.5 XmlHttpRequests and Cross-Origin Resource Sharing

JavaScript can initiate direct read/write HTTP connections to remote hosts using XMLHttpRequests (XHR). XHR was originally developed by Microsoft as part of Outlook Web Access 2000 and was standardized by the W3C [16]. XHR allows the creation of
synchronous and asynchronous HTTP GET and POST requests. The XML part of the name is misleading as the API supports requests for arbitrary, character-based data.

In the early days of the web, resources would typically be placed on the same origin, thus the need to interact with cross-origin resource was not given. However, with sophisticated web applications, such as Facebook, the requirement arose to use resources from foreign origins. For this reason, the W3C has specified Cross-Origin Resource Sharing (CORS) [13] to explicitly allow XHR access to certain cross-origin resources.

Before the introduction of CORS, the target URL of an XHR request was subject to the SOP, i.e., only URLs that satisfied the SOP in respect to the web page that contains the initiating script were permitted. Nowadays, modern browsers allow XHR to also request cross-domain URLs. In this case, the access check is performed after the HTTP response is received by the browser: If the response contained an **Access-Control-Allow-Origin** header which explicitly allows the requesting domain to access the resource, the response’s data is passed back to the requesting XHR object. If the header is missing or the requesting domain is not whitelisted, a security exception is thrown and access to the response is blocked.

In this context, CORS distinguishes between two types of requests - simple and complex. A simple request is a request that only contains a small number of allowed headers and does not use any credentials. This kind of request could also be generated by embedding an image or posting a form. In contrast, a complex request can use custom headers and can therefore only be generated by an XmlHttpRequest. Since a complex request allows the transmission of authentication credentials and cookies, operations performed by the web server in response to such a request may be state-changing. Because of this,
the browser must not allow a complex request to be sent to the server unless it has verified that the server explicitly allows this request. Therefore, when executing a complex request, the browser will first send an OPTIONS request to the server, stating the URL to be requested, the headers it wants to send and whether or not the request will use credentials. This is called a preflight request. The server then decides if it wants to allow access to the resource with the given criteria and replies accordingly.

For any XHR, the server implementing CORS will send at least the Access-Control-Allow-Origin HTTP header which contains a list of URLs that are allowed to access this resource. Thus, if a web site at http://www.domain.com:80 wants to access the resource on social-network.org, the server hosting social-network.org must set the Access-Control-Allow-Origin to either http://www.domain.com:80, http://*.domain.com:80 or *. For complex requests, this check is performed on the response to the preflight request and only if the check succeeds, the actual request is performed. For simple requests, the browser will send the actual request directly and block access to the response if the check fails. To sum up, CORS allows cross-origin resources to be accessed only if the server explicitly allows this access request.

2.6 DNS Rebinding

DNS Rebinding is a term for a technique used to circumvent the Same-Origin Policy and was introduced by Jackson et al. [31]. As we discussed in the previous sections, the Same-Origin Policy separates websites distrusting each other by denying interaction between them. Therefore, if a website A wants to interact with another website B, either their origins must match or B must implement CORS and explicitly allow access from A.

However, we have to assume that an attacker would not be interested in interacting with websites that allow this using CORS. In this case, we infer that the attacker has control over a website example.org and wants to access data on a corporate intranet. Figure 2.2 shows such a scenario. In the first stage, the victim (located on the lower right part of the figure) wants to access example.org. Therefore, he contacts the DNS server controlled by the attacker, which returns the IP address at which example.org is hosted (6.6.6.6) as shown in step 1 in the figure. The DNS reply has a very short time-to-live (TTL), meaning that the client requesting the lookup is told to re-iterate the lookup after a very short time. The client’s web browser now contacts the server at 6.6.6.6 and requests the content (step 2). The returned HTML source of example.org contains active content, which itself tries to load new resources from example.org. At this point, the browser needs to open a new connection to example.org. However, as previously stated, the attacker chose a low TTL. Therefore, the browser must now contact the DNS server again to resolve the address corresponding to example.org which is denoted as step 3. This time, the DNS server replies with the IP address of the intranet server located to
2 Technical background

Figure 2.2: DNS Rebinding steps

the lower left of the figure. The web browser now downloads the requested content from the intranet server, as shown in step 4. Before allowing interaction of the original script with the retrieved content, the browser now checks to see if the Same-Origin Policy is fulfilled. As neither port, protocol nor the host(name) was changed, the Same-Origin Policy is fulfilled and access to the resource is granted. Now, the attacker’s active content can leak the information gathered from the intranet host back to the attacker, which is denoted in step 5.

2.7 Cross-Site Scripting

In the web context, the term Cross-Site Scripting (XSS) is used for a class of attacks that allow an attacker to inject HTML or script code into a vulnerable web application. As discussed in the previous sections, the Same-Origin Policy governs interaction of web pages and disallows access to the content of website domain2 by website domain1 if their origins do not match. If however website domain2 is vulnerable to Cross-Site Scripting, this allows the attacker to execute his script code in the context of website domain2 and hence access the privileged content on that site. XSS can therefore also be seen as a means of circumventing the Same-Origin Policy. According to the Open Web Application Security Project (OWASP), Cross-Site Scripting nowadays is the second
most important class of vulnerabilities on the web [37].

To understand the impact of Cross-Site Scripting, we discuss the example of Google Mail. Google Mail presents the user with content that is meant only for the user himself and therefore access is only allowed after authentication. However, the user does not want to input his credentials before each new action – hence authorization is implemented in such a manner that the client has a special token that he presents to the server. Quite often, this authentication and authorization token is presented in the form of a cookie, which is only accessible from the same origin by which it was issued. As discussed, XSS allows an attacker to inject his own script code into the website and thus run in its context and origin. Now, the injected code can access the cookie and send it back to the attacker who in gains access to the emails of the victim.

Also, web applications vulnerable to Cross-Site Scripting might allow Phishing attacks. Considering the example of a vulnerable banking website, an attacker could use the injected code to eavesdrop on the entered credentials by not only sending the login information to the actual bank server, but also back to a server under his control. This way, he could retrieve login credentials even from users that validate the URL of the banking site.

Throughout this thesis, we will use the JavaScript code `alert(1)` as a possible payload of injected code. Please note that this merely represents arbitrary JavaScript code which can be executed inside the victim’s browser.

There are three different types of Cross-Site Scripting which we will discuss and explain in the following: Reflected, Stored and DOM-based Cross-Site Scripting.

### 2.7.1 Reflected Cross-Site Scripting

Reflected Cross-Site Scripting is the most commonly found variant of Cross-Site Scripting. Reflected XSS vulnerability usually come from user-provided data (most often parts of the URL) which is echoed back by the server [44]. Considering the simple example in Listing 2.6, we assume a website that takes the username as a GET parameter and welcomes the user. Normally, the URL a user would surf to would look like `http://example.org/?username=Ben`. This would show the string `Hello Ben, welcome back!` to the user accessing the site.

However, an attacker could abuse this by sending a specially crafted link to his victims. We see that all the input provided via the username parameter is presented back in the browser. Thus, the attacker could lure the victim to visit `http://example.org/?

```php
$username = $_GET['username'];
echo "<h1>Hello $username, welcome back!</h1>";
```

Listing 2.6: Example of a simple vulnerable PHP application
2 Technical background

Figure 2.3: Illustration of a Reflected Cross-Site Scripting attack

username=<script>alert(1)</script>, which would open a message box in the client’s browser. Real-world examples of reflected Cross-Site Scripting include search functionality which echoes back the user-provided search string (You search for: ...) and

Figure 2.3 depicts the process of a Reflected Cross-Site Scripting attack. In the first step, the attacker provides the victim with a crafted link. In the second step, the victim visits this link in the browser, unknowing of its malicious nature. The Cross-Site Scripting code embedded in the URL is send to the server as denoted by step 2 in the illustration. Due to its vulnerability, the server returns the injected code inside its HTML response. Upon loading of the page, the injected script code is executed by the client and leaks sensitive information back to the attacker as shown in step 4.

As mentioned before, an attacker exploiting a Reflected XSS vulnerability will lure his victims to a specially crafted link in the vulnerable web application. Typically, web servers generate log files in which all access to the web applications hosted on that server is stored. Therefore, this kind of Cross-Site Scripting usually leaves traces in the log files and can be detected by examining them.

2.7.2 Stored Cross-Site Scripting

In contrast to Reflected Cross-Site Scripting, which we explained in the previous section, Stored Cross-Site Scripting does not involve luring a victim to a crafted URL but refers to a vulnerability where the attacker can somehow store his malicious code in the web
application [44]. Thus, only web applications that allow users to store some information on the server are vulnerable to this kind of attack.

A commonly used example for Stored XSS are virtual guestbooks. Usually, the concept of a guestbook makes it necessary for users to provide information like name, email address and the message itself. This information is then stored inside a database on the server side and is retrieved when another user visits the guestbook to view the entries. If the user-provided data is inserted and retrieved from the database without filtering, an attacker can introduce his malicious code into either the name, email address or the message text.

In general, Stored Cross-Site Scripting describes a type of vulnerability where the attacker is able to persistently store the malicious code in the web application. Stored XSS can therefore also be used to implement self-replicating XSS worms. In 2005, the social networking site myspace.com was susceptible to Stored XSS as the web application did not implement proper filtering on the data provided in the user profile [32]. The worm, when executed in the victim’s browser, firstly added the author Samy as a friend and then copied the malicious code to the profile page of victim viewing the original, infected profile. In this manner, the worm created by Samy infected over 1 million profiles in less than 24 hours [32]. This real-world example shows the massive impact Stored XSS can have on large-scale applications.

Figure 2.4 depicts the steps in a Stored Cross-Site Scripting attack. In a first step, the attacker sends his malicious code to web application, which subsequently stores it. In step 2, the attacker lures the victim to the web application which now contains his injected code. The victim visits the vulnerable application and subsequently retrieves the previously stored code (steps 3 and 4). In the last step, the attacker-provided code now sends sensitive information back to the attacker.

As previously stated, Stored XSS indicates that the attacker is able to inject his malicious code once into the web application whereas every user visiting the site becomes a victim and executes the code. Thus, this type of attack is not recognizable using log files and very special care must be taken to ensure that user-provided data which is stored inside the application is either filtered or encoded.

### 2.7.3 DOM-based Cross-Site Scripting

In 2005, security researcher Amit Klein published an article describing a previously unknown kind of Cross-Site Scripting – DOM-based Cross-Site Scripting [33]. The common factor between both Stored and Reflected Cross-Site Scripting is that the injected code is contained in the HTML source which is provided by the server. With reflected Cross-Site Scripting, the code is directly echoed from the user-provided input, whereas with Stored XSS the payload is stored on the server and send to the victim at a later point. The major difference between Reflected, Stored and DOM-based Cross-Site Scripting is that for the latter, the aforementioned characteristic does not hold true. DOM-based
XSS purely relies on vulnerable client-side code whereas both Reflected and Stored XSS exploit vulnerabilities in the server-side code.

Listing 2.7 shows a simple example of an application susceptible to DOM-based XSS based on the example provided by Klein [33]. The code provided in the Listing is meant to be used to extract the username from the URL fragment of the currently viewed document. The URL fragment, as shown in Figure 2.5, is the last part of the complete URL, separated from the previous parts of the URL by the hash mark #. The complete URL refers to the protocol, hostname, port, path, search and URL fragment of the viewed document. The terms protocol, hostname and port are self-explanatory – the path denotes the location of a document on the web server. A very important difference between the URL fragment and the rest of the URL is the fact the fragment is not sent to the server and thus does not appear on a servers logfiles.

The code in line 4 searches for the position of the string username and adds an offset

Figure 2.5: Parts of the URL explained
of 9 (the length of the string `username=`) to the result. The next line then extracts the substring starting from the previously calculated position to the end of the string. In line 6, the extracted substring is assigned to the `p` node (in line 2) accessed by its identifier. The use-case this was designed for obviously is a website with an URL like `http://example.org/#username=Ben`. In that case, the website would normally greet the user similar to the example given in Section 2.7.1.

```html
<html>
  <p id="hello"></p>
  <script>
    var pos = location.hash.indexOf("username=")+9;
    var name = location.hash.substring(pos, location.hash.length);
    hello.innerHTML = "Hello +"+name+ " and welcome to our system";
  </script>
</html>
```

Listing 2.7: Example showing DOM-based XSS

However, this short example has a vulnerability. The user-provided input coming from the URL is not properly filtered or checked. Therefore, an attacker can lure his victim to this site via the URL `http://example.org/#username=Ben<\p>Attack code</p>`. The victims browser will execute the JavaScript snippet and will write both the intended username as well as the HTML source behind the username to the page. Note that in this example, the usage of `<script>` does not work as modern browsers will not evaluate the code in the newly added script block. Therefore, an attacker needs to use small work-around. As discussed in Section 2.3, the Document Object Model implements an Events API which can be used to trigger given actions on certain events. Listing 2.8 details such an event. When this code is injected, the browser will try to load the non-existent resource from the server. Once this download fails, the onError event will be triggered which in turn executes the JavaScript snippet. This way, the attacker can abuse the aforementioned example.

```html
<img src="non-existent" onError="alert(1)" />
```

Listing 2.8: JavaScript Code Execution using events

The complete process of exploitation through DOM-based Cross-Site Scripting is shown in Figure 2.6. The attacker, not unlike in the case of Reflected Cross-Site Scripting, provides a crafted link containing XSS code to the victim. In step 2 the victims requests the corresponding resource from the server which is provided in step 3. Note that, in this case, the vulnerability does not stem from insecure programming on the server, but rather from insecure client-side code as shown in Listing 2.7. Hence, the
response from the server does not contain the attacker-provided code. However, once the scripted content retrieved from the server is executed, it includes the XSS code as provided by the attacker. The client’s JavaScript engine therefore executes the injected code and provides the attacker with the desired privileged information.

Next to all parts of the URL, DOM-based Cross-Site Scripting can be caused by insecure handling of unfiltered data coming from a multitude of other sources. As the term *DOM-based Cross-Site Scripting* already implies, all sources of data used in DOM-based XSS are accessible via the Document Object Model. Among these are cookies as well as the referrer of a document. In the following, we provide a detailed list of several DOM sources usable in DOM-based Cross-Site Scripting and will present the different sinks which are relevant for DOM-based XSS.

**DOM-based Cross-Site Scripting sources**

As discussed above, DOM-based Cross-Site Scripting utilizes unfiltered input from so-called data sources. In this section, we will discuss the different sources provided by the Document Object Model. In our notion, a *source* depicts the origin of a certain piece of data.

**Location, documentURI and URL sources** The first category contains all sources corresponding to the URL of the document which is displayed in the browser. Inside a
browser, the objects `document` and `location` are both registered globally and can be accessed directly from JavaScript. The `document` object contains three relevant properties: `URL`, `documentURI` and `baseURI`. Whereas both `URL` and `documentURI` return a string containing the actual URL of the loaded document, the `baseURI` is the relative base used for references in conjunction with hyper links and resources like images. Depending on the browser, the `documentURI` and `baseURI` may not contain the URL fragment (previously depicted in in Figure 2.5). Also, the `documentURI` is also set if the accessed document is not HTML, whereas `URL` only exists for HTML documents [17].

Secondly, the `location` object contains multiple relevant properties. Among these is `href`, which contains the complete URL including the fragment. Also, the URL is split into different parts, which can be accessed via the properties `pathname`, `search` and `hash`. The path relative to the document root on the web server, which is denoted as the blue part in Figure 2.5, is accessible via `pathname` – including the leading slash. The `search` property refers to the part of the URL which is highlighted in green in Figure 2.5, including the question mark. The location object’s `hash` property contains the complete URL fragment, including the hash mark, depicted in red to the right of Figure 2.5.

**The cookie source** Considering an attacker-controlled sub-domain `sub.example.org`, this sub-domain can set a cookie for `.example.org`, which can then be accessed by any sub-domain of `example.org`. This way, the attacker can inject possibly malicious data and thus, the `cookie` must also be considered another source of possible attacker-controlled data.

**The referrer source** A third type of DOM-based Cross-Site Scripting source is the referrer. As discussed earlier in Section 2.3, the referrer is a string whose value is the URL of the page from which the user came to the current page. The referrer is automatically set by the browser accordingly when a link is clicked and is empty if a URL is typed directly into the location bar.

Often, a website will contain navigation links which allow the user to return to the previously viewed page. Listing 2.9 shows such an example. The designated usage writes the URL of the referring page to the `href` attribute of the anchor element. However, an attacker could lure the victim to a page in form of `http://example.org/attack.html" onclick="alert(1)`. The JavaScript code provided in the Listing would then write the following to the document:

```html
<a href="http://example.org/attack.html" onclick="alert(1)">Go back</a>
```

This way, the user would execute the attacker provided code when clicking on the newly created `Go back` link. However, in modern browsers, the referrer is encoded and therefore cannot be used. Nevertheless, researchers as Minded Security found a vulnerability in
Internet Explorer, which does not encode the hostname of the referrer [35]. Therefore, the attacker can use a sub-domain containing his malicious code which refers the user to a vulnerable web application.

1 <script>
2 document.write('<a href="'+document.referrer+'">Go back</a>');
3 </script>

Listing 2.9: Simple HTML example of navigation links implementing in JavaScript

**The window.name source** The `window` object has a optional property named `name` which is primarily used to give distinct names to opened popup windows. However, an attacker can open an arbitrary popup window using `window.open` and can assign a name to it. Thus, if the `window.name` property is used in a web application, that application might be susceptible to DOM-based XSS. Therefore, `window.name` must also be considered a source of attacker-controllable data.

**Other sources** Apart from the aforementioned sources, there are also other properties that can be thought of as sources for DOM-based XSS. Next to the previously discussed sources, the DOM-based XSS Test Cases Wiki [36] also lists the Local and the Session Storage as other sources for DOM-based Cross-Site Scripting. The Storage API is part of the Draft for the new HTML5 standard and is currently specified as a Candidate Recommendation [15]. Both the Local and the Session Storage are used to store data in a key-value pair in a web client. This technique is designed such that web applications can persistently store data in the client in order to reduce its bandwidth consumption. The difference between the Local and the Session Storage is that data stored in the Local Storage is to be kept until explicitly deleted, whereas the Session Storage only persists data for the duration of a session. However, the W3C Recommendation states that a session does not necessarily have to be related to the lifetime a user agent process. This means, that a session in the sense of the Session Storage might be kept even if the browser has been closed and re-opened. In a sense, the storages are similar to cookies as they store information in a key-value pair on the client side. The difference however is that each storage has its own origin (cf. Section 2.4) and thus cannot be manipulated by other sub-domains like cookies can.

Attacks using the Local or Session Storage require that an attacker was able to compromise the web application earlier to store his malicious payload. The attacker can however not lure the victim to a different sub-domain (like with cookies) or persist the data directly in the storages. Due to these limitations, although we mention the Local and Session Storage as a potential source for DOM-based Cross-Site Scripting, we do not include it in our implementation.
2 Technical background

DOM-based Cross-Site Scripting sinks

In the previous section, we listed the sources which can be used for DOM-based Cross-Site Scripting. In this section, we will give an overview over sinks which are used in conjunction with DOM-based XSS. In our notion, a sink is some sort of security-critical functionality that possibly allows an attacker to execute his injected and therefore potentially dangerous code. In the following, we will distinct between two types of sinks – those that allow direct execution of code and those that require some form of user interaction to trigger events.

Direct execution sinks

Usually, the aim of an attacker is to have the victim execute his code inside the vulnerable web application. JavaScript offers several functions which allow given strings to be executed and interpreted. In this thesis, we will refer to such functions as execution sinks. As discussed in Section 2.2, JavaScript provides a function called eval. This function takes one argument and interprets this argument as JavaScript code. The eval function is commonly used to execute code which was dynamically generated during runtime. If the data provided by the attacker is not filtered or encoded and is inserted into an eval statement, the attacker has complete control over the JavaScript program.

Alongside the straight-forward eval function, the DOM API provides two functions which allow JavaScript to be run using timeouts. The first function is setTimeout, which takes two parameters – the script code and a timeout. This function run the script code provided as the first parameter after a given timeout exactly once. While setInterval works in a similar manner, the execution of the code is repeated every timeout milliseconds.

As we stated earlier in this chapter, JavaScript supports the definition of new functions. This can either be done by writing the function and its body directly into the script source or by calling the Function function. This function takes one or more arguments, whereas the last argument of the call is treated as the source for the function which is returned by the call. Of course, this is only a direct execution sink, if the so generated function was to be called inside the program anyway.

Active content in HTML is enclosed in script elements. These elements have properties which can be modified during runtime, namely src, text, innerText and textContent. The src property defines the location of a file which contains the script content. If an attacker has write-access to this property, he can force the victim’s browser to download arbitrary script code from a server under the attackers control. The other three properties provide direct access to the script elements source code and must thus also be seen as potentially dangerous execution sinks.

If a malicious and well-crafted string was to be entered in the context described above, the attacker would be able take over control of the victim’s browser without any further interaction from the user as all of the functions directly execute the provided code.
Besides these direct ways of execution code, an attacker has a multitude of other means to execute his malicious code – using indirect sinks.

**Indirect execution sinks** First in the list of indirect execution sinks are the event-driven properties of any HTML element. As discussed in Section 2.3, the Events DOM API provides multiple means of reacting to certain events. Among these events are those which are triggered when an element is loaded, unloaded, hovered, focused or clicked. For example, a commonly used event in conjunction with Cross-Site Scripting is the `onError` even which is depicted in Listing 2.8. Once the download of the non-existent image file fails, the event is triggered which in turn makes the JavaScript engine interpret the value of the `onError` property. Thus, any `onEvent` properties must be considered indirect execution sinks, as they need some form of interaction to trigger code execution.

Another group of indirect execution sinks are those that modify the content of the HTML page displayed in the browser. First and foremost, these are `document.write` and `document.writeln`, whereas the latter is a variant of the first, which appends a line break to the passed argument. These functions directly write into the source code of the HTML page – just after the `script` block they are enclosed in. Both allow an attacker to write arbitrary HTML code into the vulnerable application. This attack vector enables the attacker to directly inject a new `script` block into the document which is subsequently executed.

Other possible sinks include the `innerHTML` and `outerHTML` property of any element as well as setting certain declaration in Cascading Style Sheets. The value of the `innerHTML` property is set to all HTML text which is contained in the node – namely its text content and all child nodes. The `outerHTML` property contains the HTML declaration of the node itself as well as the content of `innerHTML`. However, these functions cannot be used to insert and execute a new `script` block as the interpretation of this newly generated block is skipped by browsers. Nevertheless, using the technique described in the paragraph above allows the execution of attacker-controlled code using events.

**Other sinks** Another functionality that can be seen as a sink in DOM-based Cross-Site Scripting is the assignment to the `location` object. This can either be abused in code overwriting any of the `location` objects properties or using the `replace` and `assign` functions provided by the object [19]. This way, an attacker can either redirect the victim to a site of his choosing or can open a JavaScript URL. A JavaScript URL basically is a URL which has the protocol set to `javascript` and the remaining part set to arbitrary JavaScript code. When this kind of URL is opened, the given code is executed.
2.8 Taint tracking

In this section, we will discuss the concept of taint tracking during runtime of a given program. As discussed in the previous section, a program working on any data will always have sources and sinks. The data retrieved from a sink flows through the program in a given way and ends up in a sink. As we explained earlier, certain data values that are provided by the user can be used to exploit vulnerabilities in susceptible web applications. To find such vulnerabilities in web applications, we need to determine where user-provided input might be used in the program. Thus, we need some form of recognizing that a certain piece of data is actually coming from the user and is not hard-coded into the program’s source code. Taint Tracking describes the concept of attaching information about the source of a given piece of data to the data itself, forwarding this information through the use of the data in the program and detecting a flow to a sink [46]. The marking of data is also called tainting. Throughout the execution of a given program, this taint can be evaluated to gather information on where the data originated from.

Thus, at any given point in the execution of a program, the data used in the computation can be checked to see if it is tainted. Based on the result of this check, different actions might be taken. As discussed by Vogt et al. [34], in the context of protecting sensitive information, taint tracking can be used to check data which is about to be somehow leave the protected encapsulation by, for example, being send to a remote server. In the case where the attacker has control over the script content running in the victim’s browser, the malicious code could leak information about the cookie to the attacker. A scenario depicting the impact of such capabilities was described in Section 2.7. Vogt et al. propose that before the data is sent to the attacker – for example by making an XmlHttpRequest from JavaScript – the tainted value is detected and thus the request is stopped.

Similar to the approach discussed above, taint tracking can also be used to mark untrusted, user-provided data. The taint flag can then be checked before any data is provided to security critical functions. Within the scope of this thesis, a security critical function is any of the sinks outlined in Section 2.7.3.

Summary

In this chapter, we gave a detailed technical background to allow the reader full comprehension of the thesis. Firstly, we explained the basics concepts behind HTML and gave a comprehensive description on the design of JavaScript including its take on inheritance – prototyping. Next, we outlined the Document Object Model which allows JavaScript to interact with the HTML tree and several other document-related properties. In Section 2.4 the idea of the Same-Origin Policy and its impact on web security was given, followed
2 Technical background

by the DNS Rebinding attack aimed on circumventing the SOP. After this, Cross-Site Scripting in its different forms was depicted along with a detailed outline on DOM-based Cross-Site Scripting in particular. The chapter was concluded with an introduction into the concept of taint tracking in the context of this thesis.
3 Layout of the Chromium web browser

The basis for the implementations of the concepts presented in this thesis is the web browser Chromium. Chromium is the open-source counterpart of the well-known Google Chrome browser. We chose this browser to show that our proposed concepts can be implemented into a modern web browser without disrupting the designed functionality. Chromium uses a slightly customized version of the WebKit rendering engine to display pages and Google’s own JavaScript implementation, namely V8. The components are linked via the Chromium Core, which also implements functionality like HTTP communication. WebKit, as the rendering engine, also provides the Document Object Model and different APIs (also called bindings) for interaction with JavaScript. The layout of the components relevant for this thesis are depicted in Figure 3.1. On the left, the Document Object Model which contains part of the original WebKit code and Google’s custom implementations, is depicted. On the right side, the layout of the V8 engine is indicated – containing both runtime code written in C++ and JavaScript as well as generated code written directly in Assembler. The basics on the V8 JavaScript engine are detailed in Section 3.1 whereas the relevant parts of WebKit are explained in Section 3.2.

3.1 V8 JavaScript engine

The JavaScript engine that is used in Chromium is Google’s own implementation called V8. It was specifically developed to be used in conjunction with Google Chromium, and can also be used as inside any C++ application to allow execution of JavaScript. Development of V8 was started in 2006 to tackle problems JavaScript had at that time like overhead by interpreting the code.

Hidden Classes In static languages, the structure of any object is known at compile time, therefore access to any method or value can be implemented by looking up the offset of the property in respect to the object being accessed. This offset then will either contain the value directly if it is e.g. a boolean value or store a pointer to the actual value or function being called. Therefore, the lookup process is very fast and basically only an array access.

However, in dynamic languages like JavaScript, the structure of an object is not known at compile time and also may be changed during runtime. Thus, on access to any
property, the executing engine must first determine if a property exists and then lookup the address of the actual required value. Therefore, in a worst-case scenario using a simple form of lookup table like a linked-list where a property is not contained inside the object, every entry of the lookup table has to be examined. In heavy-weight web applications like Google Maps or Google Mail, this is not feasible as this would severely slow down the execution. The process can of course be sped up by using more intelligent implementations of the lookup table like a hash map. However, we still need to execute a lookup on the table on every access.

In order to optimize the access time, V8 implements so-called Hidden Classes. A hidden class is an object-oriented representation of the dynamic object created by JavaScript. Generation of a hidden class works in several steps. The Listing 3.1 shows a simple example of what a JavaScript object might look like.

In the example, a naive approach would be to first allocate an object \( p \) with two properties, namely \( x \) and \( y \). Afterwards, the values 1 and 2 would be stored in memory and their respective addresses stored in the property lookup table for the values \( x \) and \( y \). Secondly, the process would repeated for the object \( q \). Access to the property \( x \) of object \( p \) would now require two lookups - first a lookup of the object in the global object table and second a lookup in the map of properties for the now found object. Although the objects \( p \) and \( q \) have the same structure, they are independent objects. Therefore, access to \( q.x \) must do a separate search inside \( q \)'s property lookup table.
In object-oriented programming languages, \texttt{p} and \texttt{q} would have the same class and therefore access to \texttt{q.x} could utilize the known offset to property \texttt{x}. To allow exploitation of this benefit of object-orientation, V8 internally implements hidden classes that represent the dynamic objects inside JavaScript. A hidden class is generated in the following steps.

- On creation of a new object, the engine checks to see if there is an empty hidden class \texttt{c} in existence. If not, such a class is created.

- Next, the properties of the newly created object are examined. For the first property \texttt{p1}, a new hidden class \texttt{c1} is created containing only this property. After creation of \texttt{c1}, a reference to it is stored inside a lookup table of class \texttt{c} created in the first step. The lookup key for this class is the name of property \texttt{p1}. Essentially, the empty class stores a reference to the class which is created by adding \texttt{p1}.

- After this, the second property \texttt{p2} of a class is examined and yet another new class created – containing both \texttt{p1} and \texttt{p2}. Now, the lookup table in \texttt{c1} is created and stores the reference to \texttt{c2} with the key \texttt{p2}. Additional properties are added in such fashion recursively.

- If a new object with the property \texttt{p1} is created, this key can now be found in the lookup table of \texttt{c1}. The runtime engine now follows the reference to the previously created class \texttt{c1}. If an object containing only a previously unknown property \texttt{p3} is created, a new hidden class is generated and the lookup table of the empty class is updated accordingly.

This process of using hidden classes enables the V8 engine to enhance performance if objects of the same type are used more often.

**Inline Caching** Another optimization is access through inline caching. Inline caching is used to optimize performance when accessing objects and their properties. As discussed above, V8 uses hidden classes to model JavaScript objects, each of which has a unique identifier. Whenever an object is accessed, the corresponding hidden class must be
determined. To speed up lookup performance, V8 implements a cache which stores a stub of machine generated code. This stub checks whether the current object has the same identifier as the one used in the previous access. If the identifiers match, the runtime engine jumps directly into the corresponding piece of code. If the identifiers do not match, a normal lookup is performed and a new stub is generated and stored in the inline cache. This way and due to the nature of large-scale web applications, which often use multiple objects of the same type, this increases performance once the cache is filled.

**Generated code**  In this paragraph, we discuss an optimization in V8 which uses generated code rather than compiled code. The term *generated code* hereby refers to machine code which is written in an abstract form by the programmer, whereas *compiled code* refers to standard C++ code which is translated into machine by the compiler. The idea of using small stubs of generated machine code for trivial tasks and only using complex, compiled code in edge cases can be found throughout the source of V8. One example for this is the allocation of strings inside V8. In normal operation, a new string is allocated by calling a function consisting only of a few assembler instructions which mark a certain memory area as used. If this allocation fails, for example because there is no space left in the virtual page, the engine falls back to the compiled code to handle the exception. Please note that the term *virtual page* does not adhere to a operating system management memory page but to an area of memory governed by V8’s own memory management system.

**Implementation of JavaScript functionality**  As described earlier, V8 utilizes both generated and compiled code to implement runtime functionality. Next to these two types of implementation, some functions are also written directly in JavaScript. These routines in JavaScript are intermixed with calls to runtime functionality written in C++ and Assembler for performance reasons. This mixture of different programming languages makes it harder to implement new features as usually parts in every of the three code types have to be modified and enhanced.

### 3.2 WebKit rendering engine

Chromium uses the well-known WebKit rendering engine to display websites in the browser. WebKit is therefore also responsible for parsing the HTML source as well as implementing the Document Object Model API which allows interaction of scripted content with the document. In the following, we will describe the details of the original implementation of these components.
3 Layout of the Chromium web browser

3.2.1 Implementation of the HTML parser

As described in the previous chapters, HTML is a markup language represented in plain text. In a browser, this text is rendered according to the rules specified by the W3C. To allow the rendering engine to properly display a given HTML source, this source must first be parsed and stored as an HTML tree. For HTML and all other markup languages which are supported, WebKit implements a class responsible for parsing the source and generating the corresponding tree.

This functionality is built as a state machine in WebKit’s HTMLTokenizer class. The method nextToken takes two arguments – the source code currently being parsed and a reference to a HTML token to be filled. The method then parses the provided source character by character and returns when the next valid token – i.e. the next HTML element – has been completely extracted. In doing so, it supports correction of small syntactically errors in the HTML source. After creation of a new token, it is placed in the HTML tree. Thus, once the tokenization has been completed, the browser has a syntactically correct HTML tree, which can be used to display the requested document.

3.2.2 Implementation of the Document Object Model API

The Document Object Model API provides a well-defined interface for scripting content like JavaScript. It provides different methods of accessing and modifying nodes in the HTML tree as well as additional data like cookies.

In the following, we will provide a detailed overview of how the properties accessible via the DOM API are implemented inside WebKit, which can be group into three different categories.

**HTML elements** The parent class to every class representing an HTML element is WebCore::HTMLElement. This class provides methods allowing access to attributes commonly shared among all HTML elements, like the name (or HTML tag) of the element or event attributes. Child classes of HTMLElement in turn implement functionality only necessary for the corresponding type of element.

**Location** As the location object inside the Document Object Model is not part of the HTML source, it is implemented separately. In Chromium, Google uses their own customized implementation of the location class, located in WebCore/page/Location.cpp. Since the location describes a URL at which the currently viewed document is located, it is stored in the form of a KURL object. The class KURL implements all functionality necessary to retrieve any part of the URL, as depicted in Figure 2.5, as well as methods for modification of the aforementioned path.
Cookies The third category of data stored in the Document Object Model are cookies. Cookies are stored in a class called `CookieJarChromium` (located in WebCore/platform/network/chromium), which is also not part of the original WebKit code but was added by Google’s developers. It provides multiple methods to interact with cookies, such as adding, retrieving and deleting them.

Well-defined interfaces In order to provide well-defined interfaces to scripted content, WebKit employs the usage of Interface Definition Language (IDL) files. Listing 3.2 shows exemplary code taken from the IDL file `HTMLElement.idl`. The excerpts shows the definition of four attributes which allow both read and write access. On compilation, these Interface Definition Language are parsed by a perl script inside the WebKit folder. Based on the information derived from the IDL files, this script automatically generates the interface functions, which are provided in the V8 DOM Bindings as denoted in Figure 3.1. If the keywords `CustomGetter`, `CustomSetter` or `Custom` are attached to a definition (as shown in square brackets with the keyword `TreatNullAs` in the Listing), the script looks up the corresponding accessor function in the folder containing custom bindings (WebCore/bindings/v8/custom/). An accessor function is a manually programmed function which is to be called when the corresponding property is accessed. As an example, when the `CustomGetter` option is set for `document.URL`, the script looks up the function `V8Document::URLAccessorGetter` in `V8DocumentCustom.cpp` which is located in the folder for custom bindings. This function is then used during runtime to manage the read access to said property, whereas the automatically generated setter function is still used to provide write access.

After the script has generated the code for non-customized properties and gathered the code for the customized accessor functions, WebKit can be compiled. Thus, in order to add customized functionalty to any given HTML property access, the programmer needs to change the IDL file and implement the necessary functionality in the custom binding folder. The code must be implemented inside the file `V8NameCustom.cpp`, whereas `Name` denotes the name of the interface the method is used by.
Summary

In this chapter, we presented the components of Chromium which are relevant in conjunction with this thesis. First, we extensively described the inner workings of the JavaScript engine V8, followed by an overview over the key components of the WebKit rendering engine. Both parts present the technical basis for the approaches we will present in the next chapters.
4 Extending the Same-Origin Policy

The Same-Origin Policy is responsible for the separation of documents of a different origin. However, there are two ways for an attacker to circumvent this policy – either using techniques like DNS rebinding or by employing Cross-Site Scripting attacks. In this chapter, we propose an extended version of the Same-Origin Policy which is immune to DNS rebinding.

4.1 Motivation

As we discussed in Section 2.4, the Same-Origin Policy is the basic policy governing the separating of mutually distrusting web applications such that malicious access to each others resources is prohibited. However, in the past there have been multiple ways of circumventing the Same-Origin Policy, first and foremost different manifestations of DNS Rebinding. DNS Rebinding first emerged in 1996 when Princeton’s Secure Internet Programming group explained the attack using Java Applets in the browser. [38] In 2002, Adam Megacz [39] proposed a different attack that enabled DNS Rebinding using JavaScript. In reaction to this publication, Netscape modified their browser such that a DNS entry was pinned. Fundamentally, DNS Pinning caches the answer to the first DNS request for a domain even if the time-to-live is low. This effectively disables DNS Rebinding capabilities but provides other issues. Large-scale web applications often make use of so-called Content Distribution Networks (CDNs). These CDNs enable load-balancing by returning multiple IP addresses for a DNS request. Each time a request to the CDN’s domain is made, another IP address is to be used to enable distribution of the load on the servers. Obviously, DNS Pinning causes issues as only a single IP address is used for the connection to the CDN’s server and thus the load cannot be distributed and balanced [41].

In 2006, Martin Johns described a way of undermining DNS Pinning [40]. He proposed that an attacker lures the victim to the website example.org which is under his control. This website contains JavaScript content, which uses a timed event like document.setTimeout to load additional resources from example.org. However, after the initial connection is finished, the attacker-controlled web server drops all incoming connections from the victim’s host. The browsers he tested at the time then deleted their pinned DNS entries to re-resolve the domain name. The name server could then answer with a new IP address which effectively allowed the attacker-desired intranet access.
Apart from these previously described attempts at circumvention of the Same-Origin Policy, there is a not yet published new form of attack using the HTML5 Application Cache. However, this attack is not in the focus of this thesis and is therefore not described here.

All these discovered flaws show that there is constant arms race between the attacker and browser vendors trying to correct the flaws. The concept of DNS Pinning was introduced to work as a bandaid for the somewhat flawed Same-Origin Policy. Therefore, we try to redefine an extended Same-Origin Policy which tackles the problems caused by its original counterpart.

### 4.2 Concept of the extended Same-Origin Policy

As depicted above, DNS Pinning can only be considered a bandaid to the flawed concept of the currently used Same-Origin Policy. Therefore, we propose an extended Same-Origin Policy (eSOP) which improves the resilience of web applications against attacks like DNS Rebinding.

In our notion, the Same-Origin Policy is a policy which governs the separation of web applications on the client side. Thus, any extension to the Same-Origin Policy must also be implemented on the client side and must not rely on a server behaving in a certain manner. However, our approach proposes the use of new server-side HTTP headers. In order to be downwards compatible, our proposed implementation must be able to function properly without these special headers.

Currently, the Same-Origin Policy relies on three pieces of information to make its decision to grant or deny access to a certain resource. These are, as discussed in Section 2.4 the protocol, the hostname and the port. As explained earlier, this is not sufficient in regards to attack like DNS Rebinding. The problem inherently lies within the principals involved in the communication. On first look, it would appear that the principals involved in an HTTP communication are the web server $W$ and the web client or browser $C$. However, the decision to allow or disallow access to a resource is made by $C$ not based on information from principal $W$ but rather a third involved principal, the DNS server $D$. An origin in the sense of the Same-Origin Policy is a form of trust boundary between applications. Therefore, we must make sure that the two principals who rely on this trust boundary are involved in the decision-making process.

Our approach aims at taking information coming from the web server $W$ into consideration rather than information provided the DNS server $D$. Currently, a web server does not provide data intended to aid the client $C$ in deciding whether two or more resources are of the same origin or not. Hence, we propose that the server must convey additional information to $C$ before trust between the involved resources is established.

We extend the Same-Origin Policy to include a fourth piece of information which is to be considered – the server-origin. This value is intended to be sent by the server and
can contain one or more domains for which the server allows interaction. In the case where a server handles only one domain, say intranet.corp, it indicates being responsible for said domain. A client opening the web site gains this information and can very that the server-origin matches the hostname of the URL which is currently viewed. Thus, the eSOP is satisfied and access to resources on this site granted. We now consider the attack scenario from Section 2.6, an attacker using example.org for his rebinding attack and the aforementioned intranet server W at 10.0.0.1. The attacker firstly lures the victim to his site and then starts his rebinding attack. The DNS server D replies with the IP address of the internal server. The client C now starts verification using the original Same-Origin Policy. As the victim is still interacting with example.org, the original SOP is satisfied. However, when checking the server-origin intranet.corp provided by W against the active domain example.org, the eSOP is not satisfied and therefore access is blocked.

In a similar case, where the web server W handles multiple domains, it will communicate a complete list of domains which it is responsible for. We propose that W may send a comma-separated list of domains to the client C. The client then verifies whether the currently active domain is contained in the list and decides accordingly.

Definition of the extended Same-Origin Policy To sum up, we give a precise definition of the proposed extended Same-Origin Policy: When a script with origin \{protocol1, domain1, port1, server-origin1\} attempts to access a resource with origin \{protocol2, domain2, port2, server-origin2\}, the eSOP is satisfied only if

- \{protocol1, domain1, port1\} == \{protocol2, domain2, port2\}
- and \(domain1 \in server-origin2\)

To be backwards compatible, the second condition must evaluated to true if server-origin2 is not set.

4.3 Implementation

The final missing puzzle piece is the exact method, how the server communicates the server-origin property of his resources to the browser. We propose to introduce a dedicated HTTP response header, \texttt{X-Server-Origin}, that carries the server-origin property in the form of a comma-separated list.

Choosing this approach has several advantages: Foremost, it is compatible with the caching behavior of Web browsers. Web browsers are already required to cache HTTP response headers along with the actual resources, as they otherwise would not be able to properly interpret the cached content after retrieving it from storage. Also, unlike DNS or IP-based protection schemes, properties communicated via HTTP response headers
are preserved when the browser accesses the network via a Web proxy. Finally, adding features using new response headers is non-disruptive, as older browsers simply ignore unknown response headers.

In order to test our implementation, we used PHP’s header function [23] to manually set the X-Server-Origin header. If the approach was to be implemented in browsers, appropriate modules for web servers would have to be implemented as well. However, as this is just a proof-of-concept, we did not engineer any such module.

We conducted a practical implementation of our protection mechanism, to experimentally validate the feasibility, security and functionality properties of the eSOP. The technical basis for our implementation is the Chromium web browser. In order to implement our approach, we had to change only a small amount of code, the details of which are provided in Section 4.3.3. WebKit implements the Same-Origin Policy in its class SecurityOrigin which normally stores the triple protocol, domain and port. In this class, we added the aforementioned component for the server-origin. This component is set by extracting the value of the HTTP header X-Server-Origin.

Our implementation allows for three different types of values for the server-origin:

- **Empty or global wildcard**: The value is either set to an empty string, "*" or the header is not set at all. In this case, we assume that the server either does not implement our approach or wants to allow access via any domain name. Accordingly, we allow access regardless of the requested domain name.

- **Empty or missing**: The value is either set to an empty string or the header is not set at all. In this case, we assume that the server either does not implement our approach or wants to opt-out of the protection mechanism. Accordingly, we allow access regardless of the acting domain value.

- **Domain(s)**: A list of comma-separated domains without any wildcards. Access is only allowed for a complete match to any of the domains.

- **Domain(s) with wildcard**: A list of comma-separated domains with wildcards for subdomains

At this point, we need to distinguish between XmlHttpRequests (XHRs) and script access to a viewport, such as frames or popup windows. XmlHttpRequest is an API that provides script code with functionality to communicate between the client and the server as defined by the W3C [16]. In the following, we show how we implemented the two different paths.

### 4.3.1 Script access to a viewport

For a viewport, we want to align our implementation to how browsers should handle cross-origin requests, thus allowing a popup or frame from any resource to be rendered
4 Extending the Same-Origin Policy

but to deny script access if the origins do not match. This is also important towards being downwards compatible. In the current implementation of Chromium, script access to the response is governed by a method called `BindingSecurityBase::canAccess` which checks the `SecurityOrigin` objects from the resources involved in the access. We extended this method to verify the server-origin as well as the protocol, host and port. If a web application does not implement our suggested extended same-origin policy, the browser will fall back to the normal SOP validation and render the page properly.

4.3.2 XmlHttpRequests

For XHRs, we decided not to distinguish between simple and complex requests as specified in CORS (see Section 2.5). For CORS, a complex request is a cross-origin request that uses headers other than the small number of whitelisted headers in the specification or that sends credentials. The idea behind this differentiation is that a complex request might be state-changing whereas a simple request can be easily constructed by using an `img` tag.

Access to a resource requested by an XHR is not governed by the aforementioned method in `SecurityOrigin` but handled separately in `DocumentThreadableLoader::didReceiveResponse`. The current implementation in Chromium distinguishes between same-origin, simple cross-origin and complex cross-origin requests. We therefore changed the path of the same-origin request to parse the header as described earlier and to allow or disallow access to the resource accordingly.

To be fully interoperable with the browser’s XHR object, we had to ensure compatibility with its recently introduced cross-origin capabilities: To allow XHRs to access cross-origin resources, the W3C specified cross-origin resource sharing (CORS) [16]. CORS allows the initiation of `simple` requests to a cross-origin resource and only checks the right to access the response after the request has been completed. In the context of CORS a request is considered to be `simple` if it also would be possible to create an equivalent request with other means, such as `IMG`-tags or HTML forms. For `complex` requests, CORS requires that the browser sends a preflight request to the server to retrieve the CORS-relevant headers. Only if the retrieved headers allow access to the resource, the complex request is sent to the server to ensure that state-changing operations are only performed if explicitly allowed by the application.

In a naive notion, requests to a rebound domain are cross-origin requests as they are not using the same origin in respect to the definition of the extended Same-Origin Policy. Thus, we can allow simple requests to be sent directly but need to verify that the server-origin matches before allowing access to the resource. For a complex request, however, we need to check the preflight response and only allow the actual request to be sent if the server-origin matches. However, using the preflight functionality from CORS would break downwards compatibility. If - for example - the client requests a same-domain resource on a server that does not implement CORS, the CORS headers would not be
set and the check would fail. Hence, we choose not to distinguish between simple and complex requests in the sense of CORS but only check the headers after the request has been sent.

The flow chart in Fig. 4.1 shows the resulting implementation logic of the XHR object. Our addition to the implementation is positioned on the lower left of the chart, whereas the right part of the figure depicts the original logic as implemented by Chromium. Note that as an XHR is not rendered by the browser, we can directly block access upon receiving the response from the server.

4.3.3 Modifications and performance

In total, we modified 34 lines of code in Chromium. As discussed earlier, the implementation manifests itself only as parsing and extraction of the HTTP headers, the allocation
of a little amount of memory to store the server-origin and a string comparison of the
domain and the stored value. The parsing of HTTP headers is executed for any request,
thus the performance impact is reduced to just one more array access. We explicitly
disallow regular expressions to enhance performance and to remove potential margin for
error when formulating the regular expression. Thus, in our tests we had no significant
overhead when accessing a web application.

Summary
In this section we discussed an extended variant of the Same-Origin Policy, which involves
the web server in the decision-making process in regards to verifying matching origins.
Our approach is easy to implement which we showed by building it into the Chromium
web browser, changing only a very small amount of code.
5 DOM-based Cross-Site Scripting
detection using taint-tracking

This chapter describes the technical basis for our implementation, namely the Chromium web browser as well as the concept of using taint-tracking to detect DOM-based Cross-Site Scripting. In the last part of the chapter, we will outline three approaches of implementing the taint-tracking and finally will evaluate the detection rate of the implementation.

As discussed in Section 2.2, JavaScript is a highly dynamic language, allowing objects to change functionality stored in their properties at runtime. Also, due to prototyping, a given object does not necessarily need to implement a certain property as long as it is implemented its prototype chain. Apart from this, JavaScript allows the use of `eval` to evaluate code which may be generated during runtime. All these factors impede the usage of static code analysis. In order to detect DOM-based Cross-Site Scripting reliably, we decided to use dynamic analysis tracking the flow of untrusted data from source to potentially harmful sinks.

5.1 Basic concept

In this section, we will describe the basic concept behind our implementation and introduce the notion of propagators, which is necessary to understanding the implementation details.

Follow data from source to sink  As laid out in Section 2.8, data flows through a program from source to sink. On the way, the data may be altered in many forms, for example by extracting a substring or by concatenating two strings to form a new string. There are several functions that take a string as input and return a new string as output – we call these types of functions propagators.

If data is tainted in the sense we described earlier, the taint information needs to be passed on when the information flows through any of these propagators. Also, data may be stored in a temporary sink which later becomes a source – an example of this is storing a tainted piece of data in a new variable which is subsequently accessed.

Figure 5.1 shows an example of how the components in the implementation must function. Firstly, the JavaScript engine creates the variable \( x \). It then accesses the DOM implementation of WebKit to the left via the V8 Bindings. As the requested piece of data is
DOM-based Cross-Site Scripting detection using taint-tracking

Figure 5.1: Overview of exemplary data flow

coming from a known DOM-based Cross-Site Scripting source `document.location.hash`, the value must be tainted by the DOM implementation. It is necessary to do this inside WebKits DOM because once the strings leaves WebKit, it is only referred to as the variable `x` inside V8. During the execution of the program, possible propagators must convey the taint when working on the variable `x`. In the example, the tainted piece of data is not stored into temporary sink. Although `document.title` can normally not be controlled by an attacker and therefore is not considered an untrusted source in the sense of DOM-based XSS, it becomes an untrusted source once tainted data flows into it. Thus, WebKit must now extract the taint information for the passed input and store it for `document.title`. In a next step, the content of `document.title` is requested and written into `y`. Although `document.title` is not by default an untrusted source, the value is tainted as it originated from the untrusted DOM-based XSS source `document.location.hash`. In the final step, the data is input into a potentially dangerous function, namely `document.write`, as explained in Section 2.7.3. At this point, using our proposed implementation, we are able to determine that the website running this script code is prone to DOM-based Cross-Site Scripting.
5.2 Implementation

The implementation phase consists of two separate parts: implementation of storage for the taint information and adaptation of existing string operations to allow them to become taint propagators, i.e. changing them such that they convey the taint information whenever a tainted piece of data is passed through. String operations like concatenation only happen inside the V8 JavaScript engine whereas storing of the taint information is necessary both in V8 and WebKit’s DOM. The reason for this, which we discussed in the previous section is that almost any attribute accessible via the DOM can become a temporary sink of untrusted data. In the following, we describe two ideas on how to store information on the taint status which did not work out and reason why. Afterwards, we present the final solution we used to tackle this problem.

The aim of this thesis was to engineer a tool that is able to detect DOM-based Cross-Site Scripting in web applications. However, to ensure optimal results, the tool should not only be able to determine that a certain piece of data came from an untrusted and thus potentially harmful source, but also to determine from which source it came. Alongside this, the information that a string contains tainted data is not sufficient. Another important fact is the position and length of the tainted data inside the string. Thus, we decided to design our taint-tracking approach in such a manner that additionally to the information that a string contains tainted information, we are also able to gather detailed knowledge of the source of every character in such a tainted string.

Strings inside V8  
V8 implements its own classes for string handling although C++ provides implementations for both one- and two-byte strings. All types of strings are children of the `v8::internal::String` class. A string object itself only consists of header which also contains a reference to the memory address where the actual string is stored. Inside the header of any kind of object, V8 stores additional information in the so-called `map`. The map stores different values, among which are for example boolean values telling whether an object is shared or the number of properties for the given object. Incidentally, this map leaves additional space to store boolean values. Our implementation takes advantages of this free space to store whether a string contains any tainted values at all. Thus, when using a string inside a propagator function or in a sink, information on whether a string is tainted can be quickly accessed.

V8 utilizes two classes for handling sequential strings. A sequential string is the notion of V8 is a string which uses one continuous part of memory to store the entire string. In contrast to this representation, V8 also provides classes for sliced strings. A sliced string is a string consisting only of parts of another string and therefore does not allocate a distinct area in memory. V8 also supports external strings, which are used in conjunction with APIs like the Document Object Model. These types of strings, not unlike sliced strings, contain a reference to a memory area inside the external API’s memory and thus do not need allocation upon creation. Also, to optimize string concatenation, the
V8 JavaScript engine provides a class for consecutive strings. A consecutive string is a string comprised of a left and a right part, whereas either part can be either a sequential, external, sliced or consecutive string.

Strings inside WebKit Apart from changes to the V8 JavaScript engine, our taint-tracking approach also required us to make changes to the WebKit Document Object Model implementation. As depicted in Figure 5.1, the necessity to store taint information for strings residing outside V8 is arises whenever tainted data flows to a temporary sink/source. In contrast to V8, WebKit only uses one type of string – WTF::String. The WTF::String is a wrapper class around a WTF::StringImpl class. The latter provides the functionality to allocate memory and store character data inside whereas WTF::String provides functionality to return manipulated forms of the string. Manipulation in the sense of this class are translations to either upper or lower case, whitespace removal as well as conversion to standard C++ strings.

5.2.1 First approach: code representation

At first look, a feasible solution towards solving the problem of tainting a string, is to represent it in the form of JavaScript code which would generate such a string. We consider the example given in Listing 5.1. The first variable \( x \) only consists of document.location.hash, thus the corresponding JavaScript expression would be

\[
\text{document.location.hash}
\]

For the second variable, namely \( y \), a valid expression generating the string would be

\[
\text{document.location.hash + document.URL}
\]

One possibility to store this resulting code would be to add a second string \( s_{\text{code}} \) for every string \( s_t \) we want to tag. In this \( s_{\text{code}} \), the code producing \( s_t \) could be stored and analyzed whenever the tainted string \( s_t \) is used in a potentially dangerous sink.

Determining the access path In order to gain information necessary for generating \( s_{\text{code}} \), we need to be informed about the access path taken to a certain value. In our notion, the access path is the sequence of objects which were passed and whose properties were accessed to retrieve the value. During execution, V8 will process instructions left to right. If we consider an access to document.location.hash as depicted above, V8

```javascript
1 x = document.location.hash;
2 y = x + document.URL;
```

Listing 5.1: Code example illustrating the code representation approach
first tries to look up the object `document` inside its global object lookup table. Once the object has been retrieved, V8 calls `JSObject::GetPropertyWithCallback` on the object, passing the name `location` as the parameter. This call in turn returns a new object, on which `GetPropertyWithCallback` is called again, this time passing the name `hash` as the parameter. This process is depicted in Figure 5.2.

This method of recursively calling `GetPropertyWithCallback` can be exploited to achieve our approach. Each time the method returns a new object, both the name of the currently accessed property as well as the parent object are known. The access path can therefore be computed as

\[
\text{new path} = \text{parent path} + \text{name}
\]

Once the interpreter reaches the end of the call chain, the resulting access path can be stored.

![Diagram of access path](image)

**Figure 5.2: Flow chart showing the lookup of document.location.hash**

**Utilizing an external lookup service**

In the manner described in the previous section, we are able to determine the access path taken to value. With the additional information that a value came from an untrusted
source, this is sufficient to detect potentially dangerous flows. This information can be provided by DOM bindings, which we patched accordingly to mark a string coming from a DOM-based Cross-Site Scripting source as tainted.

However, in large-scale applications the chance of two strings coming from the same source are high. Hence, storing this access path directly for every string would cause redundancy. Due to this, we decided to implement an external lookup service to match a string to its source. Instead of allocating space inside V8’s memory to store the access path, only an identifier is stored within the structure of the object. Due to alignment issues, the header of objects in V8 allocated an additional 30 bits of memory which are not used (revision 19.0.1084.52). In order to keep the changes to the existing code small and mostly non-intrusive, we decided to store the numerical identifier in these unused bits. This identifier can be resolved back to the aforementioned $s_{code}$ via the lookup service.

Along with being able to store the access path to certain objects, the previously proposed taint-tracking approach must also support storing of function calls and array accesses. Thus, we designed the lookup service such that it supports multiple types of JavaScript expressions. As the first attempts to build the lookup service into Chromium itself failed due to problems with the garbage collection, it was implemented as an external stand-alone program communicating with the main V8 process via Interprocess Communication (IPC). For stability reasons, we used the well-established Boost C++ Interprocess Library [42]. The stand-alone server was implemented such that it uses a barrier to allow only one synchronized access to the lookup table and do remove the possibility of race conditions. Thus, the server was design to run in and endless while loop, waiting for new clients. On the other hand, V8 was patched to call the appropriate IPC method when either retrieving an object via $\text{GetPropertyWithCallback}$ or calling a propagator function. We utilized the concept of a barrier, which allows execution of a region of code only if exactly two entities – namely the IPC server and one instance of V8 – are waiting, because issues arose when running without synchronization.

Table 5.1 shows the different methods built into the external lookup service. The first method, identified by its code 0, is the normal property access depicted earlier, for example $\text{document.location}$. Taking two parameters – the parent object’s identifier and the name of the accessed property – it returns the identifier to be attached to the object resulting from the property access. The method for code 1 works in a similar fashion, but is used to represent array access, for example $\text{array[0]}$. Note, that although JavaScript allows to access arbitrary properties in this manner – for example $\text{document["location"]}$ – this is internally represented as a normal property access denoted by code 0. IPC code 2 was only used during development for debug reason. Taking an identifier as parameter, this call returns the matching $s_{code}$. The method with code 3 is meant for production use and takes two parameters – the object identifier of the value flowing to a given sink and the identifier of the sink itself. As access to a sink – if inside the DOM tree – is achieved by first looking up the property using
the `GetPropertyWithCallback` function, the object representing the sink is also tagged with its corresponding path.

IPC codes 4 and 5 are used in conjunction with propagator functions, whereas code 4 denotes a call in the form `obj.toString()` and 5 a call in the form of `unescape(obj)`. Finally, the method identified by code 6 adheres to a simple string concatenation–taking one identifier for each string involved and returning a new identifier. Note that internally, although JavaScript allows for more than two strings being concatenated in one expression, V8 calls the corresponding method multiple times, always taking exactly two strings as arguments.

During the implementation and testing phase, different problems with this approach arose. First and foremost were problems with rivaling access path to the same object. Due to its efficient memory management system, V8 stores exactly one object for one property accessed in the Document Object Model. An example of code leading to rivaling access paths is shown in Listing 5.2. In the first line of code, the value of `document.location` is retrieved in the aforementioned manner and stored into the variable. Subsequently, variable `loc1` is tagged with the identifier corresponding to `document.location`. The expression in line 2 is used to retrieve the first element with the tag `html`. As defined by the HTML standard, this element must be the first in the HTML tree – accordingly, its parentNode is the `document` itself. Thus, the code provided in line 3 references the same property as the one in line 1. As mentioned before, the optimization in V8 leads to the object which is pointed to by `loc1` to be the same as the one now referenced by `loc2`. Hence, the access path is overwritten.

To tackle this issue, we tried to implement our tagging such that a already tagged object would not be re-tagged – i.e. only the first access path was stored. However, this did not lead to feasible results either. Also, issues arose when adding properties due to the changes in the hidden classes outlined in 3.1. Apparently, due to the changed class in the back-storing object, the tag information was lost. Apart from these problems, we had to disable Chromium’s sandbox mode to allow the IPC implementation to work. Although our goal was only to implement a prototype to be used in a secure environment,
this was another insurmountable issue. Thus, the after extensive attempt to follow this approach, we had to drop it in favor of the implementation presented in the next section. Since we changed the direction of the thesis after the aforementioned problems, we never implemented the storing of taint information in WebKit according to the previously presented concept.

```javascript
1 loc1 = document.location;
2 tmp = document.getElementsByTagName("html")[0]
3 loc2 = tmp.parentNode.location;
```

Listing 5.2: JavaScript code example leading to rivaling access paths

### 5.2.2 Second approach: character-wise tagging

For the aforementioned reasons, the first approach we tried to implement did not work out. In order to achieve our goal of flow detection from untrusted sources to potentially harmful sinks, we needed to make more intrusive changes to the source code of Chromium’s V8 and WebKit components. The concept and the details on the implementation are presented in the following.

**Storing shadow data** The concept of the second approach we followed is not to compute code which represent the value of a string, but to use shadow data. In our notion, *shadow data* refers to information on the exact source of each character in a string. To assure the forwarding of taint information inside propagator functions, we needed to change all of them to ensure that the shadow data was properly copied. Details and limitations on propagator functions are outlined later in this chapter.

**Identifying sources**

Obviously, the storing of source information for every character requires additional memory. To keep this additional need for memory as small as possible, we decided to assign a unique identifier to each of the well-known DOM-based XSS sources. This way, we were able to store the shadow information in just one additional byte, denoting for each byte from source it originated from. Table 5.2 lists all relevant sources and their corresponding identifier. Note that although `location` as such is also listed as an untrusted source, both `location` and `location.href` return the same value, generated by the same function in WebKit’s DOM implementation. Thus, a distinction when tracking the source is neither possible nor necessary.
Table 5.2: Initial list of source identifiers used for our taint-tracking approach

<table>
<thead>
<tr>
<th>ID</th>
<th>Source</th>
<th>note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>benign</td>
<td>hard-coded strings or trusted sources</td>
</tr>
<tr>
<td>1</td>
<td>location.hash</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>location.pathname</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>location.search</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>location.href</td>
<td>equivalent to location</td>
</tr>
<tr>
<td>5</td>
<td>document.URL</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>document.documentURI</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>document.baseURI</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>document.cookie</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>document.referrer</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>window.name</td>
<td></td>
</tr>
<tr>
<td>255</td>
<td>unknown</td>
<td>used in tokenizer</td>
</tr>
</tbody>
</table>

Implementation inside V8

As depicted earlier in this chapter, the V8 JavaScript engine is a highly complex software project, optimized towards fast execution of JavaScript code. As our approach implements functionality which is not intended as such by the ECMAScript standard, we had to make a large numbers of changes to V8’s code, the details of which we will present in the following.

As discussed earlier in this chapter, there are different types of strings. In the following paragraphs, we will explain the changes necessary to allow the approach to work.

Sequential strings  V8 implements two classes responsible for handling sequential strings – SeqAsciiString and SeqTwoByteString. SeqAsciiString is used whenever a string compromised only of ASCII characters is used and thus allocates one byte per character in the string it stores. In contrast, SeqTwoByteString stores strings using two bytes for one character. Upon creation of a new string object, V8 checks to see if all character in said string can be represented by one byte. If this is the case, SeqAsciiString is used whereas SeqTwoByteString must be used if even just one character cannot be represented as ASCII.

To keep the changes as small as possible, we decided to store the shadow information for sequential strings in the memory directly behind the actual character data. Figure 5.3 depicts this idea. On the top, the original string implementation is outlined, consisting only of the header and memory allocate to store the character. On the lower part of the figure, our concept is illustrated. Here, the green part denotes where the shadow information is stored.
Inside V8, there are different functions responsible for creation and allocation of a new string. To allocate a new ASCII string, the function `Heap::AllocateRawAsciiString` is called, whereas `Heap::AllocateRawTwoByteString` is responsible for allocation of their two-byte counterparts. Both functions rely on the `sizeFor` methods provided by the corresponding classes which calculates the amount of memory needed to allocate a string of a given length. Our first instinct was to change these methods such that they would return larger values as needed for our approach to work. After modifying these functions such that V8 would allocate an additional byte per character, we ran our first tests. However, these tests crashed due to reasons we did not fully understand at first glance.

After extensive investigation of the possible causes for our problems, we noticed that allocation did not only happen inside the compiled C++ runtime code, but also inside the generated Assembler code. After changing the Assembler instructions to match the functionality implemented in the runtime code, we could successfully compile and run our prototype.

To read and write the shadow data, we now needed to implement additional functions providing access to the corresponding memory addresses. V8 already implements functions allowing access to single characters in a string for both types of strings. We adopted these functions, which use pointers relative to the address of the object, to access the shadow data. Listing 5.3 shows the added functions which are responsible for storing and retrieving the tag for a given offset in the string. To achieve this, the functions calculate the offset relative to the address of the object as $kHeaderSize + length \times bpc + offset$, whereas $bpc$ denotes the amount of bytes needed to store one character.

**Consecutive strings** As mentioned earlier, there are types of strings which consist of either multiple strings or ranges of one string, namely consecutive and sliced strings. As neither of these strings allocate own memory to store the character data, access to the taint information must be allowed via the underlying string resources. Essentially, when for example handling consecutive strings, which are represented in a tree-like structure,
we first need to determine which is the corresponding string resource for a given offset. Listing 5.4 shows the code implemented to achieve this. Firstly, the length of the second half of the string is checked. If this string is empty, we can fall back to directly accessing the first string. If, however, the string is comprised of two non-empty parts, we need to walk the tree-like down until the requested offset is found. This is accomplished using a while loop and iteratively checking for new consecutive strings in the result.

**External strings**  
External strings, as the name suggests, are strings for which the backing storage of character data is outside the memory area the V8 JavaScript engine. These are mostly used in conjunction with strings coming directly from the Document Object Model implemented by WebKit. V8 uses its class `ExternalStringResource` to represent such external strings, whereas all external components have a separate class which extends this internal class. Thus, we modified both the parent `ExternalStringResource` as well as all used child classes to gain functionality to retrieve tags. Note, that external strings in V8 are not writable from V8 to ensure integrity of the external component. Thus, a need to write taint tags is not required. Later in this chapter, we will outline how the implementation of WebKit was extended to allow our taint-tracking approach to work.

**Propagator functions**  
As defined earlier, a propagator function is a function which works on tainted data, returning a new set of tainted data. In the following, we will present which functions we modified in order to allow propagating of the taint information.

- **String concatenation**: In the sense of taint-tracking, a string which is comprised of at least one tainted string must also become tainted. As we discussed earlier, every string stores a boolean value indicating whether is is tainted. Thus, in order
```c
uint8_t ConsString::ConsStringGetTag(int index) {
    ASSERT(index >= 0 && index < this->length());

    if (second()->length() == 0) {
        String* left = first();
        if (left->IsTagged())
            return left->GetTag(index);
        else
            return 0;
    }

    String* string = String::cast(this);

    while (true) {
        if (StringShape(string).IsCons()) {
            ConsStrings* cons_string = ConsString::cast(string);
            String* left = cons_string->first();
            if (left->length() > index) {
                string = left;
            } else {
                index -= left->length();
                string = cons_string->second();
            }
        } else {
            if (string->IsTagged())
                return string->GetTag(index);
            else
                return 0;
        }
    }

    return 0;
}
```

Listing 5.4: Code excerpt showing the function for retrieving the taint information in consecutive strings
5 DOM-based Cross-Site Scripting detection using taint-tracking

```c
void String::WriteTagToFlat(String* src,
                          String* sink,
                          int f,
                          int t,
                          int offset) {
    if (src->IsTagged()) {
        for (int i = 0; i < t-f; i++) {
            sink->SetTag(offset+i, src->GetTag(i+f));
        }
    } else {
        for (int i = 0; i < t-f; i++) {
            sink->SetTag(offset+i, 0);
        }
    }
}
```

Listing 5.5: Function implementing the tag copy process

to check if the resulting string needs to be tainted, we first examine these boolean values for both strings involved in the concatenation. If either of them is marked as tainted, the result’s taint flag is set. Hereafter, the source information from both strings is extracted and copied to the new string. Listing 5.5 shows the corresponding code, which extracts the taint information from the source string (src) and stores it in the resulting string (sink). Note that as this is automatically called for both strings involved in the concatenation, this function first checks whether the source string is tainted. If it is not tainted, all values in the resulting shadow data is set to 0, as seen in the block starting at line 11.

- **Substrings**: The counterpart to string concatenation is the usage of substrings. If a substring is extracted from a tainted string, the resulting string is not necessarily tainted. We consider a string comprised of 20 characters, the first 10 stemming from a trusted source and the last 10 from an untrusted source. In this scenario, a substring from offset 0 to 5 does not contain any tainted data and thus should not be marked as tainted. On the other hand, a substring from offset 8 to 12 contains both trusted and untrusted data and thus must be tainted accordingly. We implemented this mechanism using a loop and a temporary buffer to the taint information. For each character that is copied to the newly allocated substring, the taint tag is examined and copied to the temporary buffer. If, for any of the characters, a tag denotes its source as untrusted, the substring is marked as tainted. Subsequently, after the loop has been completed, the taint information is extracted from the temporary buffer and written to the resulting string’s shadow store.

- **Array representation of strings**: JavaScript provides functionality which splits
DOM-based Cross-Site Scripting detection using taint-tracking

x = "h";
y = document.URL[0];

Listing 5.6: Example code causing issues with central string storage

a given string into an array and create strings from given character arrays. In order to allow our taint-tracking to function properly, we needed to modify both functions to retain the details on the origin of each character. An issue arose out of this need in special cases when strings are split into arrays in which each element only consists of one or two characters. To optimize memory usage, the V8 JavaScript engine employs a central storage for strings consisting of only one or two characters. However, if a string in this central storage was to be tainted, the taint could be falsely attached to unrelated strings. Listing 5.6 shows an example of code causing this issue. Firstly, we assign a value of length 1 to the variable x. In the second step, y is assigned the first character of document.URL. We consider an example of a website on a remote host accessible via HTTP, thus the URL starts with http://. Hence, the value of y becomes the same as the value of x.

In case the central storage for one- and two-character strings is used, the string "h" which is stored inside the central repository now is marked as originating from document.URL. This leads to a false-positive when accessing x later in the program. Although this example might not be found in real-world applications, strings are frequently split into arrays as proposed before. To ensure that the aforementioned issues do not arise, we decided to remove the optimization using the central string storage.

- **Case conversion:** The V8 JavaScript engine implements functionality to convert the case of a given string in both directions. In our notion, even though the data might be slightly changed when converting cases, this function must act as a taint propagator. Therefore, we extended the existing function such that the resulting string is tainted if the passed string was tainted. If a string consisting only of lower case characters is passed to the function responsible for converting the string to lower case, the original string can be returned and no additional taint propagation is necessary. The same holds true for the opposite case, converting an uppercase string to upper case.

- **Encoding functions:** In JavaScript, as in any other given language, there is a defined syntax which allows the interpreter or compiler to distinguish between expressions and data. Thus, when using user-provided data, special care must be taken to ensure that this data does not cause issues during program execution. An example of code which might cause harmful problems is given in Listing 5.7. The variable username obviously is provided by the user, i.e. when browsing to a URL
Listing 5.7: Example code demonstrating pitfalls in use of unencoded data

like http://example.org#test. The code in the second line is intended to create a variable message and set its value to be comprised of a given string and the user-provided input. However, an issue arises when the user browses to a URL in form of http://example.org#test'. The resulting string which is passed to eval now is var message = 'Hello test'.';. The single quote at the end of the input leads to a syntax error in the eval statement since it is used for string termination.

To nevertheless allow usage of arbitrary data in any context, JavaScript employs functions for encoding data. These functions are called escape, encodeURIComponent and are responsible for encoding data such that it can be safely used in a given context. Note, that these functions differ slightly. escape is used mainly to ensure syntactical correctness in the context of JavaScript instructions, whereas the other two functions are allow correct syntax in the context of HTML.

As we previously discussed, the V8 JavaScript engine employs different programming languages to implement the functionality necessary for executing JavaScript. Some of these functions are implemented in C++, whereas other parts are implemented in JavaScript itself. The latter also holds true for the aforementioned encoding functions. In the original code, this does not present a problem, as string access is abstracted by the C++ runtime code. However, for our purposes, additional functionality was added to allow examination and assignment of the taint information. Thus, we implemented an interface between the JavaScript code responsible for executing the encoding functionality and the runtime code written in C++.

The encoding functions examine each character in the string. If a character does not match a given predicate – i.e. is a reserved character in regards to this encoding function – it is represented in the form of %XX, whereas XX represents the characters ASCII value in hexadecimal. In cases where a two-byte string is encoded, it is stored in the form of %uXXXX, with XXXX representing the corresponding two-byte value. Thus, whenever a character is encoded, the taint tag from the original character is attached to all characters in the encoded result. Decoding works in a similar fashion, replacing the aforementioned hexadecimal representations with the corresponding characters. In this case, we extract the taint information on the first character % and attach it to the resulting character. This is a known
5 DOM-based Cross-Site Scripting detection using taint-tracking

```c
static unsigned addFuncTag(unsigned oldtag, int pos) {
    unsigned newtag;
    if (oldtag != 255 && oldtag != 0) {
        unsigned functag = oldtag >> 4;
        if (((functag >> (pos - 1)) & 1) == 0) {
            functag += (1 << (pos - 1));
            newtag = (oldtag & 15) + (functag << 4);
            return newtag;
        }
    }
    return oldtag;
}
```

Listing 5.8: C++ code implemented to allow adding of function tags

limitation of our approach when decoding tainted values.

In the context of Cross-Site Scripting as such, input encoding is a key component, since the right encoding function must be chosen depending on the context (JavaScript expressions or HTML). Thus, Cross-Site Scripting vulnerabilities might exist even if input encoding is used if the author of code does not use the encoding function required for the given context. Hence, we decided to mark characters returned by any of the aforementioned encoding functions such that on access to a sink, not only the original source of character could be recovered, but also the list of encoding functions executed on the data. Since a character might have been accessed and modified by different encoding functions, assigning a sequential numerical identifier as used in identification of the sources is not sufficient. Therefore, we decided to model the representation of accessed functions using one bit per function. In the following, we will refer to this representation as the *function tag*. Table 5.3 shows the bit sequences representing the different encoding functions. If two or more of these functions are applied to the same character, the bit sequence identifiers can be added. Hence, if a character was passed through both `escape` and `encodeURI`, the resulting bit sequence would be 011. Based on this approach, on access to a sink, all previously called functions can be detected. Since every of the aforementioned functions has a counterpart reversing the encoding – i.e. `unescape`, `decodeURI` and `decodeURIComponent` – we patched these functions to remove the function tag accordingly. The code for the functionality to add and clear function tags is shown in Listings 5.8 and 5.9, respectively.

As stated earlier, our implementation uses one byte per character to store the source information. However, only integers up to 10 are used to identify DOM-based Cross-Site Scripting sources as shown in Table 5.2. Hence, we only need 5
5 DOM-based Cross-Site Scripting detection using taint-tracking

```cpp
static unsigned clearFuncTag(unsigned oldtag, int pos) {
    unsigned newtag;
    if (oldtag != 255 && oldtag != 0) {
        unsigned functag = oldtag >> 4;
        if (((functag >> (pos - 1)) & 1 == 1) {
            // sub 2^pos from the original function tag
            functag -= (1 << (pos - 1));
            // now, take the least 4 bits + the new functag
            newtag = (oldtag & 15) + (functag << 4);
            return newtag;
        }
    }
    return oldtag;
}
```

Listing 5.9: C++ code implemented to allow clearing of function tags

<table>
<thead>
<tr>
<th>Bit sequence</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>escape</td>
</tr>
<tr>
<td>010</td>
<td>encodeURI</td>
</tr>
<tr>
<td>100</td>
<td>encodeURIComponent</td>
</tr>
</tbody>
</table>

Table 5.3: Bit sequence used to represent encoding functions

bits to stored the source information. This allows us to use the remaining 3 bits to mark a string as having been encoded by the functions mentioned above. Figure 5.4 illustrates how the necessary information is stored. The 3 most significant bits – marked in green – are used to store the encoding functions by which the string was encoded. The orange part, denoting the 5 least significant bits, in turn stores the source the corresponding character. In the example given, examination of the presented value shows that the corresponding character came from the untrusted source `location.hash` and was encoded using `escape`.

![Figure 5.4: Illustration of the shadow information layout](image-url)
Taint storage and propagation inside the WebKit DOM API

As discussed earlier in this chapter, WebKit relies on its own implementation of strings. Another distinct difference between V8 and WebKit is the fact that V8 – in regard to strings – does not use standard object orientation such as member variables. Instead, V8 uses header fields which are only accessed using pre-defined offsets relative to the beginning of the object’s memory address. This fact made the implementation of taint-tracking inside V8 much harder. In WebKit however, classes use member variables to store data. In contrast to V8, where we stored the source information ”behind” the actual data and accessed it via memory offsets, we were able to use a vector to store the information.

Whenever a V8 string enters WebKit, it is copied to a new string in the aforementioned WTF::String format. This is achieved in WebKit by allocating a buffer large enough to store the character data of V8’s string. Once this buffer is allocated, its reference is passed to the function v8::String::Write, located in the V8 API. This function, which works on V8’s string, copies the characters to the passed memory area. Similar to this character copying, we implemented a function responsible for the transfer of our added taint information. The code for this function is depicted in Listing 5.10. Note that although we use vectors inside WebKit to store the taint data, this data type is not known inside V8. Thus, we use a temporary buffer to store the taint information. After the call to String::WriteTag is finished, the vector is filled with the values extract from the temporary buffer. Also, we use a boolean member variable named m_tagged to indicate whether a string is tainted.

As previously noted, strings originating from the Document Object Model API of WebKit can be used directly in V8 as so-called external strings. Internally, these strings are represented by the class WebCoreStringResource which is an extension of the aforementioned ExternalStringResource class provided by V8. When an external string stemming from WebKit is used in a propagator function inside V8, the method isTagged is called on the corresponding WebCoreStringResource. This function then accesses the m_tagged variable mentioned above to determine whether the string is tagged. In a similar manner, the external string resource implements a method GetTag, which provides access to the tag for the offset passed as its parameter. This way, V8 is able to access all relevant data without the need to copy WebKit’s string into its own memory.

As detailed in Section 3.2.2, we distinguish between three categories of properties inside the Document Object Model. In the following, we will present the changes implemented to achieve our goal of extending Chromium with taint-tracking capabilities.

**HTML elements** An HTML element in the sense of WebKit is an object of any of the child classes of HTMLElement, which consists of a name, a number of different attributes and possible child nodes. In our notion, any attribute may serve as a temporary sink. Inside WebKit, such an attribute is stored as an object of the class Attribute (located...
bool String::WriteTag(uint8_t* tagbuffer, int start, int length) const {
    i::Isolate* isolate = Utils::OpenHandle(this)->GetIsolate();
    LOG_API(isolate, "String::WriteTag");
    ENTER_V8(isolate);
    ASSERT(start >= 0 && length >= -1);
    i::Handle<i::String> str = Utils::OpenHandle(this);
    if (!str->IsTagged()) {
        return false;
    }
    for (int x = start; x < start+length; x++) {
        tagbuffer[x] = str->GetTag(x);
    }
    return true;
}

Listing 5.10: WriteTag function responsible for transfer of the taint information from V8 to WebKit

in WebCore/dom/Attribute.h). This class stores both name and value of the attribute in the form of the aforementioned string type WTF::String. As explained earlier in this section, V8’s strings entering the DOM are converted to this WebKit-specific string type, retaining the taint information. Therefore, when storing attributes or their names, due to the use of the already patched string implementation, no additional changes were needed.

However, there is an additional issue with specific attributes. As explained earlier, the browser will parse and tokenize HTML content upon loading of a page. The same process is undertaken when either innerHTML or outerHTML is changed for a given element. For a better understanding of this, we consider the code example in given Listing 5.11. The first instruction in the JavaScript code stores the reference to the first p tag. Subsequently, the next line assigns a fragment of HTML to this element’s innerHTML property. Obviously, by setting innerHTML in the way depicted in the example, a new element is added to the HTML tree. Thus, internally WebKit now starts the tokenization process described earlier. This process leads to a new, well-formed tree as depicted in Figure 5.5. In the example, innerHTML is assigned a tainted value. However, due to the tokenization taking place, the assigned value is not stored as a simple string, but merely parsed and stored as both a node name as well as additionally as a text-node beneath this newly created node. Hence, to ensure that in assigning the innerHTML of a given element, the taint information is not lost, we had to make changes to the tokenizer. As we stated earlier, the tokenizer parses the passed string character by character. In doing so, depending on the state, it either assigns a name to a newly created element node, adds an attribute and corresponding value or add another child node. Therefore, apart from the aforementioned changes made to class representing attributes, we also added

61
Listing 5.11: Code example to explain tokenization process

```html
1 <html>
2 <body>
3 <p></p>
4 <script>
5 p = document.getElementsByTagName("p") [0];
6 p.innerHTML = "+document.location.hash+";
7 document.write(p.innerHTML);
8 </script>
9 </body>
10 </html>
```

With these changes, we were able to propagate taint information to element’s name, attribute names and values as well as text-nodes during the tokenization process. This is a key factor to allow our taint-tracking approach to work. After patching the tokenizer, we examined the functionality of the method responsible for returning `innerHTML`. As we discussed, an HTML document is not stored in its textual representation, but as a tree structure. Hence, calling `innerHTML` on the `p` element used in our example, WebKit must first compute the corresponding innerHTML – which is comprised of the `b` node and its first child, the text node containing our tainted value. This task is executed by WebKit’s `MarkupAccumulator` class, specifically its function `serializeNodes`. As an argument, this function takes any node `n` in the HTML tree and generates a sub-tree `tree_n`. The tree, whose root is node `n`, contains all child nodes of `n` and their child nodes, respectively. Once this tree is generated, the function walks down `tree_n` and generates HTML code representing the sub-tree. To achieve this, the serializer first allocates a string buffer. As the HTML standard defines, an element node in HTML is indicated by an opening angle bracket directly following by the node’s name. Hence, when serializing a HTML element node, the function first stores an opening angle bracket in the string buffer. However, in the case of an element that was created by parsing a tainted string (as shown in the example), we cannot know whether the angle bracket was also part of the tainted string. Thus, since we are not able to determine the origin of this character in the resulting string, we add a taint value of 255 to it. This denotes the fact that at this point in the program, we cannot determine with certainty whether this character came from an untrusted source or not. The same holds true for white spaces, which are inserted by the serializer between attributes. Figure 5.6 denotes this concept. It shows the sub-tree as generated for the call in line 7 of the example code in Listing 5.11. The colored parts indicate characters in the resulting string, for which the origin can be determined accurately. As denoted in black, the sources for the opening and closing angle brackets cannot be determined and thus are set to the value for ”unknown source”, 255.
Figure 5.5: Illustration of the HTML tree before and after executing the code in Listing 5.11

Figure 5.6: Illustration of taint information assignment
Tainting sources in WebKit

Up until now, we have described how we implemented taint storage and taint propagation in both WebKit’s DOM implementation and the V8 JavaScript engine. However, apart from these changes, we obviously also had to add functionality which allows us to mark a string as tainted.

As we depicted earlier in this chapter, we changed the implementation of strings inside WebKit to allow for taint storage. We also already discussed the concept of accessor functions inside the DOM API. In our implementation, we added custom accessors for all of the aforementioned sources relevant to DOM-based Cross-Site Scripting (see Table 5.2). Inside these accessors, we added functionality to taint all data leaving the DOM API through the accessors to be tainted accordingly using the newly created function `StringImpl::SetAllTag`. The code for this function is shown in Listing 5.12. This function is called on the string resource to be returned and takes one numerical argument, denoting the source identifier.

Detecting dangerous flows

The only missing piece in the puzzle now is the detection of untrusted and potentially harmful data entering the sinks. As presented in Section 2.7.3, sinks for DOM-based Cross-Site Scripting are located both inside the JavaScript engine itself and in certain properties inside the DOM. In this part of the thesis, we discuss the different sinks that were patched to detect dangerous flows and what kind of additional information can be provided on sink access.

**eval and Function** The first two functions we patched were `eval` and `Function`, which allow direct execution of code and are located inside the V8 JavaScript engine. As described earlier, V8 uses a mixture of C++, JavaScript and Assembler code to execute JavaScript code. `eval` and `Function` are defined in JavaScript code and make calls to the C++ runtime code. Therefore, inside the C++ runtime code, it is not possible to determine the exact position in the JavaScript code from which either of the functions

```c++
void StringImpl::SetAllTag(unsigned int tag) {
    ASSERT(tag != 0);
    m_tagged = true;
    m_shadow.clear();
    for (unsigned i = 0; i < m_length; i++) {
        m_shadow.append(tag);
    }
}
```

Listing 5.12: Source code for `StringImpl::SetAllTag`
was invoked. Hence, we only implemented logging functionality which warns the user about a dangerous flow to `eval` or `Function`.

**setTimeout and setInterval** The next two functions we patched and which allow direct code execution are `setTimeout` and `setInterval`. These two functions are not part of the JavaScript engine, but are located inside the Document Object Model API of WebKit. Since they are called via the API, the exact code position which triggered the call can also not be determined. Similar to the direct execution sinks located in V8, our implementation only logs the potentially dangerous flow on the command line.

**Arbitrary HTML element property access** In Section 2.7.3, we also discussed several attributes of different HTML elements which are potentially harmful sinks in the sense of DOM-based Cross-Site Scripting. Among them are different attributes responsible for handling events as well as attributes linking to remote resources, such as `script.src`.

Due to the tree structure in which HTML is stored inside WebKit, our implementation can give precise details on the exact position of the element in the HTML tree. To determine the position inside the tree, the function simply walks up the tree, logging each of the traversed elements. This, along with the attribute which was written is printed on the command line.

**write and writeln** Both `document.write` and `document.writeln` allow JavaScript to directly write content to the document. This content – which can either be text-content or HTML source – is added at the position just behind the `script` element which contains the corresponding call to either of the functions. To ensure the proper placing of this newly created content, WebKit must store a pointer to the current offset in the HTML source. As discussed earlier, upon loading a page, the tokenizer generates the HTML tree, which can be transformed back to HTML by the serializer. However, due to automatic formating and the tokenizer error correction feature, the resulting HTML text is not necessarily the same HTML source passed to the tokenizer. Hence, although when `document.write` is executed, the exact position (line and offset in that line) can be determined, this position information only relates to the well-formed and automatically formatted HTML produced by the serializer and not to the original HTML source. Nevertheless, our implementation prints out the positional information alongside the warning that a flow has been detected. This way, when inspecting the well-formed source as generated by the serializer, the script block containing the flawed code can be identified.

**Location assignment** The last of the sinks detailed in Section 2.7.3 is the `location` object, namely direct assignment to it or invocation of the `replace` and `assign` methods. Analogous to the previously employed methods, these functions were changed to log flows
DOM-based Cross-Site Scripting detection using taint-tracking

of tainted data to them. Since all methods are also located in the DOM API, the exact location in the code can not be recovered.

In this section, we outlined the modifications to both WebKit and V8 to enable the detection of untrusted data entering security-critical sinks. As we discussed, the taint information is stored in one byte per character. To allow human-readability of this information, we implemented a function called dumpTag. This function, which works on the string resource, iterates over all characters in the string and produces a report, stating which offsets in the string originated from which source. An example of JavaScript code and the resulting log message is given in the next section.

5.2.3 Evaluation

To evaluate our implementation, we built several test cases. Apart from testing each source and sink, we also developed test code utilizing all of the propagator functions we listed in Section 5.2.2. Listing 5.13 shows a complex test case. Firstly, an new type of object is created in line 6. In line 7, its prototype is changed to window. This is later used in line 18 to call eval. Although this property does not exist inside the object obj created in line 6, due to prototyping chaining, window.eval is called. In lines 10 and 12, the functionality for adding and clearing function tags is tested, whereas the code in line 11 is used to test the cloning and retaining of taint information when a string is stored in a temporary sink. In order to test the propagation in conjunction with substrings and concatenation, line 14 invokes both these functions. To assure that the taint information has been properly propagated throughout the program execution, and can be detected properly both inside V8 and WebKit, the string which is stored in the variable msg is stored in innerHTML of the given p element as well as being passed as an argument to eval. The resulting log output on the console is shown in Listing 5.14. As the log clearly indicates, all source and function call information has been properly transferred through both V8 and WebKit.

To ensure that our addition to the code does not break existing functionality, we frequently browsed complex web applications like Google Maps, Google Mail and Facebook. At no time did our prototype crash when browsing these sites. Also, the obvious functionality in these web applications was tested. In the case of Google Maps, we used features like route planning, different overlays (traffic, satellite image) and zooming. All of these features worked without any problems. This suggests that our implementation did not in any way remove or destroy functionality which was built into the original code.

Performance overhead and limitations

As we detailed in this section, we needed to implement a number of changes to allow for our taint-tracking approach to work. The most intrusive changes needed to made to the
DOM-based Cross-Site Scripting detection using taint-tracking

Listing 5.13: Complex test case to verify the functionality of our taint-tracking

```
html>
<head><title>Complex test case</title></head>
<body>
    <p></p>
    <script type="text/javascript">
        proto_demo = function() { }
        proto_demo.prototype = window;
        obj = new proto_demo();
        escaped = encodeURI(document.location.href);
        document.title = escaped;
        unescaped = decodeURI(document.title);
        msg = "This string is tainted from two sources: "+escaped+" / "+
             unescaped.substr(0,5);
        p = document.getElementsByTagName("p")[0]
        p.innerHTML = msg;
        obj.eval(msg);
    </script>
</body>
</html>
```

Listing 5.14: Resulting log output when executing the test case code shown in Listing 5.13

WARNING, tainted flow to setInnerHTML detected.
Node: html -> body -> p
Offsets 0 to 41 tagged as benign
Offsets 41 to 79 tagged as document.location.href (filtered by encodeURI)
Offsets 80 to 82 tagged as benign
Offsets 83 to 87 tagged as document.location.href
WARNING, tainted flow to eval detected.
Offsets 0 to 41 tagged as benign
Offsets 41 to 79 tagged as document.location.href (filtered by encodeURI)
Offsets 80 to 82 tagged as benign
Offsets 83 to 87 tagged as document.location.href
highly optimized V8 engine. To allow the storage of source information, roughly twice
the amount of memory had to be allocated for each string used inside V8. Also, some
optimizations in the code – like the central storage for one- and two-character strings –
had to be partly removed to ensure strings were not falsely marked. Apart from this, the
functionality responsible for propagating the taint information only adds little computing
time. When an untainted string is passed to such a function, the overhead consists only
of a simple check of a boolean value. The added computation effort increases when
tainted strings are used, since the taint information is copied byte by byte. However, in
our tests with large-scale applications like Google Maps, no noticeable loss of performance
was detected.

One limitation of our approach stems the fact that not all sources relevant to DOM-
based Cross-Site Scripting necessarily contain a value. Some sources, like the URL
fragment accessible via location.hash or the cookie, are not set initially. Thus, if a
JavaScript code uses these properties in a possibly harmful manner, it might not be
detected by our prototype. However, this can be fixed by employing user-scripts. In
our notion, a user-script is a fragment of JavaScript code which is executed automat-
ically by the browser when loading a document. In this user-script, those properties,
which initially might not contain values, can be set before the document’s JavaScript
is executed. This ensures that false-negatives cannot be caused by the aforementioned
properties being empty.

Since Cross-Site Scripting relies on the attacker being able to inject JavaScript or
HTML code, our taint-tracking only works on strings. However, JavaScript provides
methods to convert a string into an array of numbers and vice-versa. Hence, another
limitation of our implementation is the fact that if a string is converted in such a way,
the taint information is lost.
6 Conclusion

This chapter, which concludes the thesis, gives a summary of the work performed in creation of this thesis, followed by the key contributions provided by it. In the last section of this chapter, we will provide an outlook on future work on the basis of the concepts we discussed and implemented in this thesis.

6.1 Summary

In the first chapter of this thesis, we presented the motivation towards implementing security-enhancing functionality in a modern web browser and presented related work in regard to enhanced versions of the Same-Origin Policy and relevant publications on the topics of Cross-Site Scripting detection and dynamic taint-tracking. Chapter 2 discussed the technical background relevant to this thesis, explaining the basics on HTML, JavaScript, the Document Object Model and the Same-Origin Policy as well as DNS Rebinding and Cross-Site Scripting attacks. Finally, an introduction into the notion of taint-tracking was given. In Chapter 3, we outlined the layout of the Chromium web browser, the technical basis for the implementation of the concepts pursued in this thesis. In the next chapter, we presented our approach of the extended Same-Origin Policy which is immune against DNS Rebinding attacks. Chapter 5 explained our concept of employing taint-tracking to detect DOM-based Cross-Site Scripting vulnerabilities. This chapter also gave an extensive outline into the implementation details for the functionality to store, forward and analyze taint information in both the V8 JavaScript engine and WebKit’s DOM API. We concluded this chapter by depicting the results of our evaluation of the implemented approach.

6.2 Contributions

The goal of this thesis was to implement security-enhancing functionality into a prototype browser. In this section, we will outline the key contributions provided by this thesis.

6.2.1 The extended Same-Origin Policy

The first contribution of this thesis is the extended Same-Origin Policy (eSOP). The current implementation of the Same-Origin Policy involves two principles in the decision-
6 Conclusion

making process regarding the origin of a document – the browser and the DNS server. However, the third involved principal, namely the web server, is not involved at all. We proposed an revised approach to the Same-Origin Policy, extending the basis for the decision-making process to include the server-origin which is provided by the web server. In this way, the client can make an informed decision as to the true origin of a given document. This extended Same-Origin Policy effectively stops DNS Rebinding attacks, which relied on the fact that the web server was involved in the origin matching. To verify that our approach can be easily implemented into a browser, we added the program logic necessary to verifying origins in the proposed manner to the Chromium web browser. In total, we only changed 34 lines of code which proves the feasibility of our concept.

6.2.2 Detecting DOM-based Cross-Site Scripting vulnerabilities

In the course of this thesis, we developed a prototyped browser capable of detecting DOM-based Cross-Site Scripting vulnerabilities in web applications. To achieve this goal, we added taint-tracking functionality to the V8 JavaScript engine and the DOM API inside WebKit. In doing so, we made major changes to string implementations in both V8 and WebKit to allow for character-based taint information storage. To allow proper forwarding of the taint information on access to propagator functionality like string concatenation and substrings, we modified the implementation of the corresponding functions inside V8 accordingly. Also, to allow the propagation inside HTML content written to the DOM using functions like innerHTML, we made changes to the HTML document parser inside WebKit. We then verified the implementation’s ability to detect flows from untrusted sources to potentially harmful sinks. To achieve verification, we developed test cases using all of the relevant sources and sinks as well as test cases invoking all of propagator functions and using different temporary sinks. For all of these tests, we could successfully detect flows to be considered harmful in respect to DOM-based Cross-Site Scripting.

6.3 Outlook

This section gives an outlook on the possible future work to be done in conjunction with our proposed extension to the Same-Origin Policy as well as on the basis of our prototype implemented to allow for DOM-based Cross-Site Scripting detection.

The extended Same-Origin Policy The proposed extended Same-Origin Policy is currently only built into our prototype and still lacks proper support on the server-side. The concept could therefore be implemented in other browsers as well as in the server-side counterparts, to end the threat posed by DNS Rebinding attacks.
Adding reporting functionality  With the functionality implemented in this thesis, we extended the JavaScript engine such that it supports taint-tracking for all strings. Currently, our prototype only reports flows from untrusted source to potentially dangerous sinks on the command line and can only be run in Linux. In a next step, the prototype could be enhanced by either adding a graphical user interface to gather additional information on a given flow and to generate extensive reports on the involved sources, sinks and information on the location of the involved calls in the code.

Large-scale empirical study  To further validate the functionality of our approach and to gather information on the current state of the web in regard to DOM-based Cross-Site Scripting, a large-scale empirical analysis could be executed. To do this, our prototype would have to be extended to allow automated crawling of a given set of websites. Such an empirical analysis could be executed.

Enhancing the Cross-Site Scripting filter  Chromium currently employs an automatic filter which detects and blocks Reflected Cross-Site Scripting attacks. It works based on string matching performed against the URL. In order to enhance the Cross-Site Scripting filter to also detect and block DOM-based XSS attacks, any tainted string entering a potentially harmful sink could be evaluated by this filter function as well. This would provide an additional layer of security when surfing the web.
Bibliography


Bibliography


Bibliography


Bibliography
