IceShield: Detection and Mitigation of Malicious Websites with a Frozen DOM

Mario Heiderich, Tilman Frosch, and Thorsten Holz

Chair for Network and Data Security
Ruhr-University Bochum
{mario.heiderich|tilman.frosch|thorsten.holz}@rub.de

Abstract. Due to its flexibility and dynamic character, JavaScript has become an important tool for attackers. The widespread scripting language often helps them to perform a broad variety of malicious activities, for example to initiate drive-by download exploits or to execute clickjacking attacks. Current defense mechanisms as well as reactive analysis and forensic approaches are often slow or complicated to set up and conduct since an attacker can use many different ways to obfuscate the code or make it hard to obtain a copy of the code in the first place.
In this paper, we introduce a novel approach to analyze this class of attacks by demonstrating how dynamic analysis of websites can be accomplished *directly in the browser*. We present IceShield, a JavaScript based tool that enables in-line dynamic code analysis as well as de-obfuscation, and a set of heuristics to detect attempts of attacking either a website or the user accessing its contents. Special care needs to be taken to implement the instrumentation in a robust and tamper resistant way since an attacker should not be able to bypass our detection process. We show how features of ECMA Script 5 can be used to *freeze* object properties, so they cannot be modified during runtime. We implemented a prototype version of IceShield and demonstrate that it detects malicious websites with a small overhead even on devices with limited computing power such as smartphones. Furthermore, IceShield can mitigate detected attacks by changing suspicious elements, so they do not cause harm anymore, thus actually protecting users from such attacks.

1 Introduction

During the last few years, we observed a shift in attacks against end-users: instead of attacking network services, many of today's attacks focus on vulnerabilities in client applications. Especially the web browser is a popular target for attackers. There are many different kinds of threats and attack vectors against current browsers, such as for example:

– *Drive-by download attacks* in which a vulnerability in the web browser or one of its components/extensions (e.g., Acrobat Reader or Flash plugins) is exploited to execute code of the attacker's choice [1].

– *Cross-Site Scripting (XSS) vulnerabilities* that enable an attacker to inject arbitrary client-side scripts into web pages [2,3,4].
– *Clickjacking* (also known as *UI redressing*) is a technique in which an attacker tricks a web site visitor into clicking on an element of a different page that is only barely (or not at all) visible [5].

These and similar attack techniques target different vulnerabilities within a browser or one of its components. The root cause of this problem is the fact that an attacker can compromise the integrity of almost all DOM properties of a website by injecting malicious JavaScript code into the website's source code. Several techniques attempting to address this problem have been proposed. On the one hand, there are analysis frameworks such as Wepawet [6], performing an offline analysis of a given page in order to detect drive-by download attacks. Cujo [7] performs on online analysis, but introduces an overhead of more than 1.5 seconds on JavaScript-heavy sites such as Facebook, which negatively impacts the user experience. On the other hand, there is a huge body of work in which different techniques are proposed to avoid attacks in the first place [8,9,10]. Approaches such as Gatekeeper [8] or *Google Caja* [9] attempt to find a way to execute arbitrary JavaScript in a secure environment. Such attempts typically require working on a subset of the complete JavaScript specification, e.g., Gatekeeper removes language constructs such as `eval()` and `document.write()` from the JavaScript specification for their analysis. Complementary to these approaches are novel browser designs, such as Gazelle [10], constructed to address these problems from the ground up. However, as such approaches tend to focus on a limited range of attack vectors or lack compatibility with the current infrastructure, many do not effectively mitigate current threats for the user.

In this paper, we introduce IceShield, a novel approach to perform light-weight instrumentation of JavaScript, detecting a diverse set of attacks against the DOM tree, and protecting users against such attacks. The instrumentation is light-weight in the sense that IceShield runs directly *within the context* of the browser, as it is implemented solely in JavaScript. Thus, the runtime overhead is low, and IceShield even works on embedded browsers used, for example, in modern smartphones. By performing dynamic analysis, we do not need to worry about obfuscation since we can inspect the attack attempt during runtime, exactly at the point where the payload is being decoded and available in plain-text. Furthermore, our approach is (almost) independent of the actual browser since the detection is implemented in JavaScript, and thus portable across browsers and platforms.

Special care needs to be taken to implement the instrumentation in a robust and tamper resistant way: since the tool is implemented in JavaScript, an attacker could try to overwrite our analysis functions during runtime. We demonstrate how an instrumentation can be rendered tamper resistant.

By performing the analysis directly in the browser, IceShield can also mitigate attacks and protect the user and websites utilizing the tool. We are able to identify which parts of the page contain suspicious elements and change them accordingly. To have a minimal impact in case of false positives, we use padding to destroy the payload of the potential exploit, but avoid visible impact on the rendered website. This enables us to actually protect users from attacks, with only a very low perceivable percentage of false positives.

We have implemented a prototype version of IceShield and evaluated the tool on a high-end workstation, a netbook, and a smartphone. The runtime overhead of IceShield is on average below 12 ms for the workstation and 80 ms on a smartphone, and we were able to achieve a detection accuracy of 98% using live malicious websites. Furthermore, we also successfully detected three exploits that the tool had never seen before and demonstrate how attacks can be mitigated successfully.
In summary, we make the following three contributions in this paper:

– We introduce a new way for tamper resistant meta programming in modern browsers, based on safely overwriting JavaScript core methods and DOM properties with a minimal performance overhead. This approach works on all modern browsers supporting ES5.
– We show how specific properties and methods can be overwritten with (almost) no footprint by recursivly modifiying the affected `toString()` and `toSource()` methods. This enables the implementation of a robust analysis framework that an attacker cannot easily detect or affect.
– We implemented a system called IceShield capable of runtime based deobfuscation of known and unknown obfuscation techniques based on the fact that overwriting core methods allows parameter inspection at call time. IceShield can be used as a framework for detecting and analyzing web based attacks in real-time with the possibility to defuse malicious payloads before actual execution.

2 Design Overview

2.1 Motivation and Basic Idea

We assume that almost every JavaScript based attack will have to use native methods at some point in order to prepare necessary data structures (e.g., to store the shellcode on the heap or stack) and afterwards perform the actual exploit by triggering a vulnerable function. This is true for heap and JIT spraying attacks, exploits against vulnerabilities in a browser plug-in or the user agent itself, as well as security issues in particular websites. The data set of malicious code samples we assembled during the testing phase of IceShield showed that most malicious scripts use native JavaScript methods such as `concat()`, `unescape()`, `substring()`, and similar string functions [11] during preparation and deployment of their malicious payload. The exploit code utilizing these functions is usually heavily obfuscated, making static code analysis and detection cumbersome and difficult. The four JavaScript code examples shown in Listing 1.1 illustrate several novel obfuscation techniques introduced and discussed on `sla.ckers.org`. These code snippets are meant to be a proof-of-concept, thus performing nothing more than a simple call to `alert(/* some data */)`.

Listing 1.1. Obfuscated JavaScript code samples executing the alert() method

```
1) ({0:#0=alert/#0#/#0#(1)});
2) (1..__proto__.e0=alert)(1.e0);
3) a=a setter=alert;
4) _=[[$,__,,$$,,_$,$_,_$_,,,$_$]=!''+[!{}]+{}]
     [_$_+$_$+__+$],_()[_$+$_+$$+__+$](-~$)
```

Especially the last of the four examples in Listing 1.1 is hard to analyze since it takes advantage of non alpha-numeric characters. This demonstrates the enormous versatility and flexibility of JavaScript and underlines the difficulty of static JavaScript code analysis. Furthermore, JavaScript allows an attacker to create morphing code, a fact that has recently been demonstrated by Heyes et al. [12]. This suggests that an attacker can render any signature based malware detection lacking advanced de-obfuscation routines useless, similar to the limitations of signature based shellcode [13] and malware [14] detection. In addition, filtering mechanisms working on a layer different than the layer to actually protect against attacks are not capable of detecting obfuscated code as for example demonstrated by the large amount of bypasses against the Webkit XSS Auditor [15] and the Internet Explorer 8 XSS filter [16].

With IceShield, we introduce a new approach to detect and mitigate attacks against web browsers and to protect the integrity of the DOM. We do not rely on any form of static code analysis, but rather the creation of an alternative and light-weight execution context that can be deployed as a script on arbitrary websites or as a browser extension. We use inline code analysis such that we do not need to worry about obfuscation: we can perform the analysis after the de-obfuscation has taken place and can analyze the exploit attempt in clear text. The analysis itself is based on detecting attack patterns of suspicious behavior. We describe these patterns in heuristics similar to the ones proposed by Wepawet [6] and Cujo [7], but we demonstrate how such features can be extended to cover other attack vectors and be used in a live analysis rather than in an offline setting. IceShield can be run in a low prioritized execution context such as being included on a website protecting the user of this website from attacks embedded in the website (e.g. via banner advertisements). The tool can also be deployed as a browser extension or injected via a proxy to provide a better protection range and independence from the individual websites potentially including IceShield. Our approach aims to have minimal footprint and overhead, and we propose a novel way of JavaScript property mimicking which we discuss in detail in Section 3.

2.2 Dynamic Detection and Protection Framework

IceShield attempts to accomplish several different goals. The first and most important is to provide the possibility to analyze drive-by download attempts at the time a malicious websites tries to execute code in the context of the victim's browser. By performing this analysis *within the context* of an actual browser, we are able to analyze the code dynamically. Thus, IceShield is not affected by any level of code obfuscation since it can analyze the code *after* the decoding/decrypting has finished. This is achieved by dynamically instrumenting objects and functions, and providing an execution context in which we can analyze their behavior. The instrumentation enables us to perform parameter analysis allowing inspection of the called methods and their parameters during runtime. With a set of heuristics and a scoring based attestation trained with data mining techniques, IceShield can determine, if the combination of method call and parameter setup indicates malicious intent. To illustrate the expressiveness of the approach, we use a set of heuristics to detect different kinds of attacks. Besides new features, we use several heuristics similar to the ones implemented in Wepawet. The set of heuristics can easily be extended to enhance IceShield's detection features in case completely novel attack vectors become known.

Second, we aim at protecting users against malicious websites: once IceShield has detected an exploitation attempt, we are able to manipulate potentially malicious code before an attack takes place. This can, for example, be achieved by modifying or removing malicious content from the DOM tree. This enables us to protect the victim from the full consequences of an attack and provide detailed information on the attack technique itself. Preliminary results suggest that this approach is effective in practice and enables us to effectively mitigate attacks.

The third goal is to implement the instrumentation in a light-weight and tamper resistant manner. On the one hand, the overhead of our analysis framework should be low such that the temporal impact is small and hardly noticeable by a user. On the other hand, an attacker should not be able to remove our instrumentation since this would enable a way to bypass our system. We achieve these two objectives by implementing our instrumentation in JavaScript and introducing a novel way to use latest features of ES5. If the browser correctly implements ES5 functionality, it is hard for an attacker to bypass the system.

In empirical measurements, we show that the overhead is small: on average, our instrumentation has an overhead of a few tens of milliseconds even on low-end systems, which is significantly less compared to the loading time of a web page. The framework can be used on different browsers and it is portable since IceShield does not depend on specific features or proprietary extensions.

We successfully tested IceShield with all modern major browsers such as Firefox 4, Chrome 6-10, Safari 5, and Internet Explorer 9. This enables a deployment of IceShield on many different devices in diversity and number. For each page a user visits, IceShield monitors the behavior of this site by dynamically analyzing the code that was supposed to be executed.

3 System Implementation

In this section, we provide a detailed overview of the dynamic instrumentation and detection techniques used by IceShield. We discuss how such an instrumentation can be implemented in a robust way and present the different components and analysis techniques used by the tool.

3.1 Heuristics to Identify Suspicious Sites

The set of heuristics and rules can be comparably slim, since the parameters inspected are usually being de-obfuscated by the executing script before hitting the rules. This significantly reduces overhead and enables further and more detailed analysis on potentially malicious code. Our heuristics are based on a manual analysis of current attacks, and we tried to generalize the heuristics such that they are capable of detecting a wide variety of attacks. Some heuristics are used in a similar way by WEPAWET [6], and we extended the coverage by taking features such as the creation of potentially dangerous elements into account. Note that these heuristics serve as a proof-of-concept and new heuristics can be easily added to the system. We found in our empirical tests that our features already cover all relevant and current attack vectors, and the heuristics can still be refined if the need arises. The following list describes the heuristics currently used by our prototype:

1. *External domain injection*: A script injects an external domain into an existing HTML element which can indicate malicious activity, for example, link or form hijacking. We distinguish between injection of `<embed>`, `<object>`, `<applet>`, and `<script>` tags, as well as, `<iframe>` injections.
2. *Dangerous MIME type injection*: A script applies a MIME type that is potentially dangerous to an existing DOM object such as `application/java-deployment-toolkit`.
3. *Suspicious Unicode characters*: A string used as argument for a native method containing characters indicating a code execution attempt such as `%u0b0c` or `%u0c0c`.
4. *Suspicious decoding results*: Decoding functions like `unescape()` or `decodeURIComponent()` that contain suspicious characters indicating code execution attempts.
5. *Overlong decoding results*: A decoding function like mentioned above receives an overlong argument. For now, we use a threshold of 4096 characters based on our empirical evaluation of current attacks and benign sites.
6. *Dangerous element creation*: A script attempts to create an element that is often used in malicious contexts for example, `<iframe>`, `<script>`, `<applet>` or similar elements. We distinguish between elements being created with and without an explicit namespace context.
7. *URI/CLSID pattern in attribute setter*: An element attribute is being applied with an external URI, data/JavaScript URI or a Class ID (CLSID) string.
8. *Dangerous tag injection via the* `innerHTML` *property*: A script attempts to set an existing element's value with a string containing dangerous HTML elements such as `<iframe>`, `<object>`, `<script>`, or `<applet>`.

3.2 Dynamic Instrumentation and Detection

We use inline code overwriting and hooking as the basic techniques to perform the instrumentation such that we can check for the heuristics introduced above.

We overwrite and wrap the native JavaScript methods into a context that allows us to dynamically inspect the name of the called function and its parameters during runtime. The original, overwritten method is being stored inside IceShield's scope in case we want to call it later on. This kind of overwriting is also successfully used in other contexts, for example, to perform binary analysis [17, 18].

In case the heuristic analysis does not indicate an ongoing attack attempt, the stored original method will be called with the unmodified set of parameters to preserve the intended code flow. In case a particular threshold defined by the internal scoring mechanisms of IceShield has been reached after the analysis, the method call can either be blocked completely or the set of arguments can be modified to keep the code flow intact, but prevent the attack. As an example for mitigating attacks, imagine a long string of shellcode being nulled before being used as a parameter for the original version of the JavaScript method `unescape()`. This approach enables us to generate complete maps, illustrating the actual code flow of JavaScript code.

IceShield utilizes an ES5 feature called `Object.defineProperty()` [19] to implement the instrumentation in a robust way. This method allows us to define new (and re-define existing) object properties, including methods and native DOM properties. Furthermore, the method allows us to pass a descriptor literal specifying the options applying for the defined property.

The most relevant descriptor for IceShield is *configurable* and the possibility to set it to `false`, thereby *freezing* the property state. Freezing means that no other script can change the property or any of its child properties again. Even a `delete` operation will not affect the property value or any of the descriptor flags. This renders our approach tamper resistant against attackers trying to change or reset the overwritten methods or access the original native methods to bypass the inspection and detection process. The same is true for property retrieval tricks working on Gecko based browsers such as `Components.lookupMethod(top, 'alert')` - an attacker cannot use this technique to bypass the freezing we used in IceShield either.

The object freezing can also be accomplished by using the method `Object.freeze()`. Batch processing of several objects to be frozen at once can be accomplished by using `Object.defineProperties()` [20].

All modern user agents such as Firefox 4, Chrome 6-10, and Internet Explorer 9 support object freezing. However, older or obscure browsers that do not fully support ES5 will not provide reliable tamper resistance for IceShield, which means that an attacker can potentially bypass the system. We performed several tests to verify the degree to which browsers support the standard. Some of the tested user agents such as Safari 5 7533.16 allows to overwrite a frozen object property. These artifacts can be considered to be software bugs: we tested later versions of the Webkit engine noticing the problem does not exist anymore.

Our tool will not attempt to modify the user agent protected `location` object [21]. Most modern browsers forbid getter access to this object and its child nodes for the sake of user privacy and avoiding security problems. JavaScript executed via direct location object access – for example, via the vector `location=name` or `location.href='javascript:alert(1)'` – will be executed in the scope we control, so no additional protection mechanisms need to be applied. This is the same for location methods like `replace()`, `apply()` or the `document.URL` property [22].

To make sure that IceShield will notice even more exotic code execution attempts, it turned out to be not sufficient to just intercept calls to native methods relating to `window` and `window.document`, but also monitor read and write access for several DOM properties as well as the dynamic creation and manipulation of HTML elements and tags. Thus, we overwrite the setter and getter methods of several HTML element prototypes, such as for example, `HTMLScript.prototype.src` or any given HTML element prototypes `innerHTML` and `outerHTML` properties. We also overwrite and seal `document` methods capable of creating new HTML elements, such as `document.createElement()` and `document.createElementNS()`. Malicious code often creates new DOM elements, applies the necessary attributes, and then attaches the element to the DOM to execute the payload.

3.3 Scoring Metric

We use techniques from the area of machine learning to decide whether or not a given site is malicious. Specifically, we use the features discussed in Section 3.1 as input for a decision function F. We treat these heuristics observed by IceShield when visiting the site as vector x of the form $(f_1, f_2, \ldots, f_n)$ and define a linear decision function $F(x)$ using a weight vector w and a bias term b as

$$F(x) = \begin{cases} w^T x - b > 0 & \text{if } x \text{ is a malicious site} \\ w^T x - b \leq 0 & \text{if } x \text{ is a benign site} \end{cases}$$

The decision surface underlying F is the hyperplane $w^T x + b = 0$, which also induces a way to distinguish between instances of benign and malicious sites based on the behavior observed by IceShield. In our proof-of-concept implementation we use Linear Discriminant Analysis (LDA [23]) to find a linear combination of weights that separate the two classes, but other machine learning algorithms could be used as well. To find the optimal weights w and bias term b, we use a corpus of labeled benign and malicious sites as our training set (see Section 4).

The decision function $F(x)$ induces a scoring metric $f(x)$ that we can use to actually detect malicious sites. The scoring metric is defined as $f(x) = w^T x$ and $f(x) > b$ indicates an instance of a malicious site, while $f(x) \leq b$ denotes a benign site. We can also use the scoring metric as some kind of *ranking*: higher values of $f(x)$ indicate a site that tries to exploit multiple vulnerabilities of a visiting browser. As noted above, other scoring metrics can be integrated into IceShield, we just chose LDA due to its simplicity and to demonstrate how an actual metric and data mining algorithm can be incorporated into the tool.

3.4 User Protection

IceShield is also capable of changing the parameters passed to native methods in case the heuristic analysis indicates a malicious attempt. The easiest way to do so is to just overwrite the suspicious argument with an empty string or add randomly dimensioned padding to maliciously looking strings before passing them to the actual method. To avoid interference with the user experience, we null the payload of the possible exploit, which mitigates the danger to the user, but in most cases has no visible impact. The IceShield prototype currently defuses a possible exploit payload in case the heuristics indicate any form of overflow or heap spray. This means that strings longer than 4096 bytes containing suspicious characters, as well as, suspicious MIME types and CLSID strings assigned to new and existing DOM elements, are being modified.

Unlike approaches either completely allowing or disallowing JavaScript execution such as NoScript or the Internet Explorer XSS Filter, IceShield has minimal impact on the user experience since only the critical function call is being defused, whereas the rest of the (possibly benign) JavaScript codeflow is not affected at all. This also minimizes the negative effects of false positives our tool might have in practice.

3.5 Implementation as Browser Extension

The purely JavaScript based approach that we introduced so far has a few limitations which we discuss next. We found several ways to circumvent and attack our own tool while testing our approach, but we also came up with new techniques to be able to harden it against those detection bypasses. In the following, we first discuss several limitations, before we present a robust design of the general approach as a browser extension. Note that this reduces the portability since IceShield needs to be customized for each browser, but the tool is better hardened against tampering attempts against our instrumentation. While the extension is browser-specific, each extension is still portable across operating systems and hardware platform. Furthermore, the core technology of our approach remains the same for each browser.

Iframes One of the biggest challenge for our JavaScript approach and comparable tools are `<iframe>` tags pointing to JavaScript URIs [24] or resources using the `data` protocol handler (so called *data URIs* as defined in RFC 1998 [25]). An iframe containing a `src` attribute pointing to such an URL executes the JavaScript or similar code contained in the URL as soon as the user agent's parser has reached this position in the DOM tree. The JavaScript is not being executed in the window context we can control with our tool, but in an implicitly created fresh context. This of course renders our approach useless since there is no way we are able to monitor the execution in the previously described manner. Listing 1.2 illustrates this problem, and we verified this behavior in all major browsers.

Listing 1.2. Iframe and object tag setup to bypass analysis

```
<iframe src="javascript:evil()"></iframe>
<object data="data:x,%3cscript>evil()%3c/script>"></object>
```

The same effect can be observed for `<object>` tags since most user agents have them behave similarly to `<iframe>` tags depending on what source they point to. The example in Listing 1.2 also shows how an object tag using a data attribute acts equivalently to an `<iframe>` with a `src` attribute.

Links Similar to the previously described iframe problem, a `<a>` tag applied with a target attribute either set as `_blank`, `_top`, or just a bogus value and a JavaScript or data URI as `href` attribute value will have the given code be executed in a new window context. This again bypasses the detection mechanism and renders an implementation in pure JavaScript bypassable. The target attribute is usually used to specify if a link should open in the same or rather a new window. The target attribute can also be used to open a link in a specifically named window context.

This feature is necessary for websites making heavy use of frame sets, frames, and pop-up windows. In case the user agent receives a target attribute value that does not exist in the currently existing scope, the link will open in the same window, but a new window context.

META Redirects Many user agents provide the possibility to emulate HTTP header information in-line by using `<meta>` tags combined with the `http-equiv` and the `content` attributes. An attacker can abuse this feature by forcing the user agent to perform a redirect after a given amount of time ranging from 0 to n seconds as shown in Listing 1.3.

Listing 1.3. META refresh example bypassing analysis

```
<meta http-equiv="refresh" content="0;url=javascript:x()" />
```

Again, JavaScript and data URIs are being used to execute script code. It strongly depends on the user agent in how far this kind of attack is capable of bypassing our approach. Browsers based on the Gecko layout engine [26] do not allow META redirects to JavaScript URIs anymore, but they still support data URIs to be used instead. All other tested browsers such as Chrome, Opera and Internet Explorer still support JavaScript URIs in this use case. While some of them execute the JavaScript code in the scope our tool controls, all browsers supporting data URIs can use those as a working bypass.

DOM Element Surveillance The solution to the problems discussed above can be found in scanning and analyzing the website's markup during parsing of the DOM tree. This can be accomplished by using two user agent features: the DOM event `DOMContentLoaded` and the possibility to select all existing DOM elements with the query `document.getElementsByTagName('*')` [27]. Before the document is actually loaded and rendered, the script can loop over the existing DOM elements and check assorted tag attribute combinations such as `<iframe>` and `src` or `<a>` and `href` or the mentioned `<meta>` and `content`. Listing 1.4 illustrates how this pre-evaluation of JavaScript code can be implemented.

Listing 1.4. Example for markup analysis before execution

```
document.addEventListener("DOMContentLoaded", function(){
    var elements = document.getElementsByTagName('*');
    for(var i in elements) {analyze(elements[i].src);}
}, false);
```

In case the protocol handlers `javascript:` or `data:` appear at the very beginning of the strings to check, a pre-evaluation can take place: the code can be executed in an environment again controlled by our tool. Most user agents allow line-breaks, tabs and several more control characters merged into the protocol handler so a pre-filtering is mandatory.

To avoid interferences with the website's functionality and user experience, this can be done in a cloned version of the existing DOM. After evaluation and analysis, the results can be channeled back to the tool's logging components and be merged with the already existing scoring. Tests have shown that this approach works very well in practice already with most passive attack vectors requiring user interaction. Active JavaScript execution via `<iframe>` and `src` combinations can be intercepted too, but most user agents besides Chrome add unnecessary limitations. Note that such an approach is not affected by heavy obfuscation either since the relevant data is being taken and analyzed directly from the already existing DOM tree and not the raw markup itself. The script accesses the code that has already been de-obfuscated and normalized by the user agent itself.

Nava demonstrated with *Active Content Signatures* (ACS) [28] how a `<plaintext>` tag can be used to render all markup following after an arbitrary branch in the DOM tree can be rendered inactive for thorough inspection, modification, and sanitization before being inserted in the DOM tree again. This approach can be used to effectively deal with the mentioned problems around `<iframe>`, `<object>` and similar tags. This way, no race conditions can appear since the plaintext tag is turning every element into a single passive text-only DOM element providing unlimited amount of time for analysis and removal of malicious code.

Browser Extensions Phung et al. [29] showed how similar approaches can be used to protect specific websites and applications against JavaScript based attacks such as XSS, CSRF and other attacks targeting the users of the attacked website or application [30]. Their approach encapsulates the native JavaScript methods and properties with an Aspect Oriented Programming (AOP) related approach based on a specific policy tailored to the website's features and specifics [31]. We suggest to move further and create browser-specific extensions such as a Firefox plug-in or an Internet Explorer Browser Helper Object (BHO) to provide more generic protection as well as gain better hardening against tampering attempts against our solution by attacker-provided code.

Extensions for Google Chrome are easy to create, but do not provide the amount of flexibility necessary for our tool to work. This is due to the technique of using *isolated worlds*, meaning a read-only mirroring for important and security critical DOM properties [32]. Our approach requires the ability to overwrite DOM elements of the website to protect users against attacks. An extension for Gecko based browsers fulfills all requirements necessary to make our approach work from within the browser as well as BHOs for the Internet Explorer. Besides the described JavaScript based version of IceShield, we have also implemented a Greasemonkey user script and a browser extension for Firefox that performs basically the same task.

3.6 Fingerprinting

IceShield is designed to be hard to detect by an attacker. We consider this to be important since many drive-by download attacks we observed fingerprinted the visiting user agent and deployed their payload conditionally. The same behavior is shown by several current exploit kits [33]. As a first step to be stealth, our tool consists exclusively of JavaScript code and does not make use of any external resources such as style sheets or images. Thus, an attacker has no possibility to read style sheet information via `window.getComputedStyles()` or utilize image tags and error handlers to find out about the existence of our tool. IceShield also does not pollute the global scope such as the OWASP ESAPI tool [34] or other comparable libraries. Instead, we use an architecture wrapped in an anonymous function. Any declared variable will reside inside this function scope, and thus does not leak into the global scope.

Since the tool is making heavy use of overwritten native methods, an attacker could easily find out about its existence via several child properties of those methods if no further precautions are met. Let `window.alert` be overwritten by a custom function. An attacker can call the `toString()` or `valueOf()` method of `window.alert` which will result in leaking the source code of the overwriting function, instead of the string `function alert() { [native code] }`.

The solution to avoid leakage via `toString` and its child nodes, is to overwrite the `window.alert.toString.toString` with its parent method `window.alert.toString`. The attacker will not be able to detect the presence of our tool by using these two methods or a combination thereof. This approach works well in all tested browsers. Note that an adversary capable of executing arbitrary JavaScript in the attacked DOM might always find ways to detect the presence of IceShield. Thus the tamper resistance established via the ES5 object capabilities is of immane importance for our approach.

A major aspect of fingerprinting are *timing attacks*, which are in general a very hard problem to deal with. This aspect can be considered as a limitation of IceShield that we have so far not managed to get around: an attacker can make use of the fact that functional string concatenation and operator based string concatenation will have a completely different code flow as soon as the

`String.concat()` method has been overwritten. An attacker can thus perform two concatenation operations: if the timing value for the first one (i.e., done functionally with `concat()`) differs significantly from the second one (e.g., performed with the + operator), then a method modification must have taken place. This could cause the attacker to not deploy the payload to avoid detection, and thus waste precious attack code, possibly containing exploits against unreported vulnerabilities.

4 Evaluation

In this section, we describe the settings and datasets we used to evaluate the prototype version of IceShield. We also present an overview of the detection and performance results obtained during several experiments.

4.1 Evaluation Environment

We compiled two datasets for the evaluation of IceShield: Our *known-good dataset* consists of the top 61,554 websites chosen from the top list of the Alexa traffic ranking [35]. To minimize the possibility that malicious sites exist in this set, we checked all URLs against the `malwaredomainlist.com` (MDL) block-list [36], which lists currently active malicious sites. The *known-bad dataset* is composed of 81 URLs selected from MDL [36]. While the number of URLs may seem to be small, all URLs in this dataset point to *exploit kits* like for example *Phoenix*, *Neosploit*, or *Eleonore*. An exploit kit is a framework to serve a variety of pre-built exploits to the unsuspecting user to initiate a drive-by attack [1]. We chose to focus on exploit kits as each instance of an exploit kit represents a whole class of exploits, and Curtsinger et al. showed that such a set is representative for current attacks [37]. Given this result, we can use a smaller known-bad set to test for a much larger amount of actual malicious sites.

To demonstrate the versatility of our approach, we evaluated IceShield on three different devices:

– High-end workstation equipped with an Intel Core i7-870 processor and 8 GB RAM, running Ubuntu 10.04 Linux and Firefox 3.6.8
– As an example of a typical mid-range system, we used a netbook ASUS EeePC 1000H with an Intel Atom N270 and 1 GB RAM, running Ubuntu 10 Linux distribution and Firefox 3.6.12.
– To evaluated the performance of our tool on a low-end device, we performed tests on a Nokia n900 smartphone with a 600 MHz ARM7 Cortex-A8 processor and 256 MB RAM, running a Maemo Linux distribution and Firefox 3.5 Maemo Browser 1.5.6 RX-51

We performed tests on all three devices and did not have to adjust IceShield for any of them: as long as the browser on the device supports the features we require, the underlying platform is not relevant.

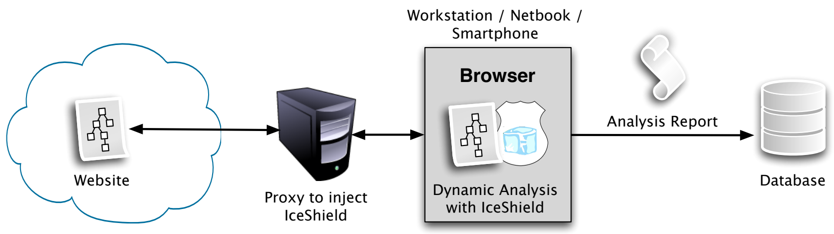
Workstation / Netbook / Smartphone
Browser
Website
Proxy to inject IceShield
Dynamic Analysis with IceShield
Analysis Report
Database

Fig. 1. Evaluation setup for IceShield: We inject the instrumentation code via a proxy and send the result to a database.

The evaluation environment is completed by a proxy server to inject IceShield into the HTML context of the visited pages, and a logging infrastructure, as depicted in Figure 1. Once a website has been successfully loaded in the browser, we log the following data points: the URL visited, execution time of IceShield and *onload* time of the respective page as well as the features observed in this website as discussed in the previous section. Furthermore, we log whether the URL belongs to the malicious or the benign set.

4.2 Classification Results

For the proof-of-concept implementation, we developed heuristics for 16 features that are computed for a given website, as described in Section 3.1. To determine whether a website is benign or malicious, we use Linear Discriminant Analysis (LDA) as described in Section 3.3. To instantiate the parameters for our data mining algorithm, we used the following training data: the complete training set consists of the top 50 sites from the Alexa traffic ranking and 30 malicious sites we randomly chose from the known-bad dataset. The test set consists of the 61,504 sites ranked below the top 50 sites we used in our training set and the remaining 51 exploit kit instances from the known-bad dataset.

Using the model computed from the training set, we were able to detect 50 of the 51 malicious sites in our known-bad test set, while achieving a false positive rate of 2.17%. We manually investigated the malicious sample that went undetected and found that this particular exploit relied on a DOM variable for execution, which was not set by the JavaScript code, but by a Java file (`.jar` file) loaded from within the site's context. As we do not currently execute Java in our test environment, the de-obfuscation routine lacked said variable. Hence the execution stopped, and we were unable to observe any relevant feature, except that the site accessed `document.cookie` twice. However, a successful attack would require the execution of the Java applet, and this would enable us to actually observe the behavior (and a feature vector) indicating a malicious site. We re-tested this site with a browser that had Java enabled and could indeed detect this particular exploit successfully.

The false positive rate of 2.17% might sound high. However, to protect the user, IceShield does not need to block access to a site that triggers an alert.

Instead, the tool can remove the elements in question from the DOM tree. Since our solution is capable of determining in which method call the possible attack takes place and which external resources are necessary to conduct and deploy the attack, we can strip this data from the site, and thus mitigate the attack. Even if we have a false positive, the user will likely not notice this since only certain elements are lacking from the DOM tree. We manually evaluated a 10% sample set (134 sites) randomly chosen from the false positives to confirm that the majority of pages remain usable even with parts of the DOM removed. The removal of the DOM elements was not noticeable by the human test user in 82.9% of the sites and 9.6% of the websites were partially usable (e.g., banner ads were not displayed correctly). Only 7.5% of the false positives were rendered unusable through the removal of the DOM elements. This means that the *effective false positive rate*, where the presence of the tool is noticed by the user in a negative fashion, is roughly only 0.37%.

4.3 Detecting Unknown Exploits

Besides testing our tool against exploit kits and the known-bad dataset, we also examined if IceShield is capable of detecting attack vectors which it had never seen before. To perform this test, we manually searched for websites serving individual exploits like an Internet Explorer exploit (CVE 2010-3962) and sites exploiting a memory corruption flaw in Apple Quicktime's QTPlugin.ocx ActiveX control(CVE 2010-1818). We manually confirmed that both exploits were not included in our known-bad dataset. We tested IceShield against these exploits and both attack vectors were labeled as malicious using our heuristics and model, which underlines the flexibility of our approach to detect both very recent and older, more widespread threats. Furthermore, we also verified that both exploits are effectively mitigated, as the respective payload is not executed since it was removed from the DOM tree.

Similarly positive results were obtained when testing against an exploit delivered via MHTML (CVE-2011-0096). This way of payload deployment is known to bypass most existing filter mechanisms since the subset of necessary characters to execute JavaScript is very small and does not include quotes or parenthesis. The payload was delivered in Base64 encoding, but had to use a set of native functions monitored by IceShield during the user agent's decoding and execution process. These results suggest that IceShield is also capable of detecting novel attacks that were unknown to the system in advance.

4.4 Performance Results

Under the aspects of usability on the one hand and stealthiness on the other, it is important to keep the execution time of IceShield low. As *execution time*, we log the time difference between the execution of the first line of code and the time immediately after we have overwritten and wrapped all required methods and objects. This is accurate since the first line that is executed is `var timestamp = Date.now();`, as IceShield is injected such that it is executed first in the browser. We measure the *onload time* as the difference between the execution of the first line of code and the moment when the process of rewriting the document is finished, i.e., the DOM is ready. We define the overhead as the percentage of the onload time that is needed to execute IceShield.

We recorded all times on the high-end workstation. Analyzing the Alexa data set, we found that the execution time ranges from 2 ms to 760 ms. While the maximum execution time seems high, the average execution time measured over all samples is 11.6 ms, which corresponds to an average overhead of 6.27%. The 99.5th percentile is 25 ms. In summary, these results indicate that the execution time and overhead is very low for the vast majority of websites and hardly noticeable by the user in practice given the typical time it requires to load a web page.

We also evaluated the performance of IceShield against several common JavaScript benchmarks such as SunSpider, Google's V8 Benchmark, and the SlickSpeedbenchmark. Only the V8 benchmark showed a significant performance loss due to its excessive use of native functions: the benchmark result on the tested workstation changed from 376 points without using IceShield to 222 points with having the tool observing the DOM. However, we believe that this is not very relevant in practice, since the V8 benchmark focuses on rendering and number crunching tasks, rather than representing real life web application test scenarios. SlickTest did not show any noticeable performance changes while the confidence interval displayed in the SunSpider results insignificantly changed from 2.7% to 4.4% when having IceShield active and running.

Fast execution and a low overhead is even more relevant on devices that rely on battery power. Thus, we conducted performance tests on a netbook and a smartphone (and again on a high-end workstation for comparison). As test cases, we selected seven interactive, high-profile websites. We accessed each URL ten times with each device and present the average over all runs in Table 1. Even on limited hardware, IceShield manages to perform reasonably fast. The execution time exceeds 100ms only on `twitter.com` and stays below in all other test cases. On average, our tool executed in 8.7 ms on a high-end workstation, in 50.4 ms on a netbook, and in 89.3 ms on a smartphone.

Table 1. Execution times on different platforms

Site (#DOM nodes)	High-End PC	Netbook	Smartphone
Google.com (113)	8.2 ms	48.9 ms	80.9 ms
Google Maps (436)	8.0 ms	50.1 ms	93.4 ms
Twitter.com (1032)	8.1 ms	49.4 ms	102.4 ms
Facebook (195)	11.6 ms	56.3 ms	92.6 ms
Yahoo! (818)	8.4 ms	48.5 ms	92.4 ms
Youtube (745)	7.9 ms	50.7 ms	79.8
Baidu (52)	8.4 ms	48.7 ms	83.6 ms
Average	8.7 ms	50.4 ms	89.3 ms

In recent months we have observed a huge improvement in the performance of JavaScript engines in the different browsers. If this trend continues, we can expect that the performance of IceShield even increases in the future.

5 Limitations

There are several limitations IceShield is faced with in its current proof-of-concept state. In case an attacker deploys a malicious PDF, Java Applet, or Flash file without using any native DOM methods to create the necessary tags and attributes, the heuristics used by IceShield might not collect enough information to deliver an adequate score. A malicious website containing no more than `<embed src="evil.pdf"/>` and avoiding utilization of native DOM methods will still be able to deploy and execute its payload.

Another limitation of the current prototype is the lack of heuristic coverage on ActiveX based attacks. This is merely due to the fact that legacy versions of Internet Explorer are not capable of executing the IceShield code. These problems do not apply for the Internet Explorer 9 Beta we tested on. Note that this limitation is merely a matter of implementation and not a substantial problem of scope such as the aforementioned issue. Another limitation of IceShield, deployed in the JavaScript version by a website, is given by the *Same Origin Policy* (SOP). In an attack scenario, where an exploit will be deployed *after* redirecting the victim to another domain, a new window context will be loaded and the protective mechanisms of our approach cannot work anymore: IceShield cannot “stick” to the users window context since the domain borders have been crossed. To mitigate this limitation, we can run the tool on a higher level of execution privileges than the usual website context, for example, with a Firefox extension or a user script running on Greasemonkey. The Firefox extension we created successfully addresses this limitation. The Greasemonkey user script we created is also not affected by this.

The lack of tamper resistance support for older user agents such as Firefox 3, Internet Explorer 8 and Opera 10 is another limitation. These older browsers do not support features such as `Object.defineProperty()`, and need workarounds like `obj.__noSuchMethod__`. The features necessary for making our approach work safe and successfully have been implemented in the new versions of these user agents, which support the latest ECMA Script specification as discussed in Section 3.

The heuristics we used to detect attacks as introduced in Section 3.1 already cover a diverse set of possible attacks, as also illustrated by the fact that we detected three attacks with IceShield that the tool had not seen before. The heuristics are not complete in a sense of them covering each possible attack vector. Depending on the actual exploit, our heuristics might be bypassed and allow sophisticated attackers to deploy their payload. However, IceShield can be easily extended to include more heuristics that then cover more attack vectors.

6 Related Work

We are not the first to propose techniques to address the problem of malicious code on the web. We briefly discuss related work in this section and compare the different approaches to the one we presented in this paper.

In the last few years, several different kinds of low- or high-interaction honeyclients were introduced such as for example HoneyMonkey [38], Capture-HPC, SpyProxy, Monkey-Spider, or PhoneyC. All of them can only be used in an (offline) analysis setting and are not capable of actually protecting end-users due to their high runtime overhead and the complexity involved when using them.

Wepawet/JSAND [6] and Cujo [7] are closely related to our approach. Wepawet is a framework to detect and analyze malicious JavaScript code in an offline setting. The tool combines anomaly detection techniques and dynamic emulation to analyze a given piece of code. Cujo uses similar heuristics to detect drive-by download, but performs the analysis on a web proxy. This approach introduces on average an analysis overhead of 500 ms and JavaScript-heavy sites such as Facebook might even introduce an overhead of more than 1.5 seconds.

Compared to these two tools, we use a similar set of detection heuristics, but IceShield can analyze the actual DOM tree within the browser and thus perform a more fine-grained analysis. Furthermore, the overhead is an order of magnitude lower compared to Cujo. In addition, our tool protects users from attacks since we can modify parameters passed to native methods to mitigate potential attacks.

An advantage of our approach compared to recent proposals such as Zozzle [37] is the light-weight implementation and the portability. However, our current prototype has a higher false-positive rate which could be lowered by using more elaborated machine learning techniques.

7 Conclusion

In this paper, we presented IceShield, a tool to perform light-weight dynamic analysis of JavaScript code *directly* in the context of a browser in order to detect and prevent attacks. This is achieved by inline code analysis and hooking to wrap native JavaScript methods into a context that enables us to dynamically analyze the behavior of these methods. We use techniques from the area of machine learning to compute a model of malicious behavior and can efficiently apply this model during runtime. Special care is taken to implement the instrumentation in a robust way such that an attacker cannot overwrite or infere with our analysis code. To this end, we introduced a novel technique to use features of the new ECMA Script 5 standard which allows us to *freeze* object properties. In an empirical evaluation, we achieved a detection accuracy of 98% and were able to detect three previously unknown attacks. The performance overhead of IceShield is low, even on small devices such as smartphones or netbooks.

7.1 Acknowledgement

This work has been supported by the the Ministry of Economic Affairs and Energy of the State of North Rhine-Westphalia (Grants 315-43-02/2-005-WFBO-009, 290077302) and the Federal Ministry of Education and Research (Grant 01BY1020 – MobWorm). We also thank the anonymous reviewers for their valuable insights and comments.

References

1. Provos, N., Mavrommatis, P., Rajab, M.A., Monrose, F.: All your iFRAMEs point to us. In: USENIX Security Symposium. (2008)
2. Kirda, E., Kruegel, C., Vigna, G., Jovanovic, N.: Noxes: A client-side solution for mitigating Cross-Site scripting attacks. In: ACM Symposium on Applied Computing (SAC). (2006)
3. Martin, M., Lam, M.S.: Automatic generation of XSS and SQL injection attacks with Goal-Directed model checking. In: USENIX Security Symposium. (2008)
4. Wassermann, G., Su, Z.: Static detection of Cross-Site scripting vulnerabilities. In: International Conference on Software Engineering (ICSE). (2008)
5. Balduzzi, M.: New insights into clickjacking. In: OWASP AppSec Research. (2010)
6. Cova, M., Kruegel, C., Vigna, G.: Detection and analysis of drive-by-download attacks and malicious JavaScript code. In: 19th international conference on World Wide Web. (2010)
7. Rieck, K., Krueger, T., Dewald, A.: Cujo: Efficient Detection and Prevention of Drive-by-Download Attacks. In: Annual Computer Security Applications Conference (ACSAC). (2010)
8. Guarnieri, S., Livshits, B.: GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In: USENIX Security Symposium. (2009)
9. Miller, M.S., Samuel, M., Laurie, B., Awad, I., Stay, M.: Caja - safe active content in sanitized javascript (2007) `http://code.google.com/p/google-caja/`.
10. Wang, H.J., Grier, C., Moshchuk, A., King, S.T., Choudhury, P., Venter, H.: The Multi-Principal OS Construction of the Gazelle Web Browser. In: USENIX Security Symposium. (2009)
11. Mozilla: String - MDC (2011) `https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Global_Objects/String#Methods_2`.
12. Heyes, G.: Polymorphic javascript (2010) `http://www.thespanner.co.uk/2008/02/27/polymorphic-javascript/`.
13. Song, Y., Locasto, M.E., Stavrou, A., Keromytis, A.D., Stolfo, S.J.: On the infeasibility of modeling polymorphic shellcode. Mach. Learn. **81** (2010)
14. Oberheide, J., Cooke, E., Jahanian, F.: CloudAV: N-Version Antivirus in the Network Cloud. In: USENIX Security Symposium. (2008)
15. Barth, A.: Bug 29278 XSSAuditor bypasses from sla.ckers.org (2009) `https://bugs.webkit.org/show_bug.cgi?id=29278`.
16. Kouzemchenko, A.: Examining and bypassing the IE8 XSS filter (2009) `http://www.slideshare.net/kuza55/examining-the-ie8-xss-filter`.

17. Father, H.: Hooking Windows API - Technics of Hooking API functions on Windows. The CodeBreakers Journal **1** (2004)
18. Willems, C., Holz, T., Freiling, F.: CWSandbox: Towards Automated Dynamic Binary Analysis. IEEE Security and Privacy **5** (2007)
19. Mozilla: defineProperty - MDC (2011) `https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/defineProperty`.
20. Mozilla: defineProperties - MDC (2011) `https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/defineProperties`.
21. Mozilla: window.location - MDC (2011) `https://developer.mozilla.org/en/window.location`.
22. Mozilla: document.URL - MDC (2010) `https://developer.mozilla.org/en/document.URL`.
23. Hastie, T., Tibshirani, R., Friedman, R.: Linear discriminant analysis. In: The Elements of Statistical Learning. Springer (2001) 84 ff
24. W3C: Client-side scripting techniques for WCAG 2.0 (2004) `http://www.w3.org/TR/2004/WD-WCAG20-SCRIPT-TECHS-20041119/`.
25. Masinter, L.: RFC 2397 - the "data" URL scheme (1998)
26. Mozilla: Gecko - MDC (2011) `https://developer.mozilla.org/en/Gecko`.
27. Mozilla: Gecko-Specific DOM events - MDC (2011) `https://developer.mozilla.org/en/Gecko-Specific_DOM_Events`.
28. Nava, E.V.: ACS - active content signatures. PST_WEBZINE_0X04 (2006)
29. Phung, P.H., Sands, D., Chudnov, A.: Lightweight Self-Protecting javascript. In: ACM Symposium on Information, Computer and Communications Security (ASIACCS). Volume March 2009. (2009)
30. Johns, M.: Code Injection Vulnerabilities in Web Applications - Exemplified at Cross-site Scripting. PhD thesis, University of Passau, Passau (2009)
31. Deiters, M.: Aspect-Oriented programming (2010) `http://msdn.microsoft.com/en-us/library/aa288717(VS.71).aspx`.
32. Barth, A., Felt, A.P., Saxena, P., Boodman, A.: Protecting browsers from extension vulnerabilities. In: Proc. of the 17th Network and Distributed System Security Symposium. (2009) `http://www.adambarth.com/papers/2010/barth-felt-saxena-boodman.pdf`.
33. Naraine, R.: Drive-by downloads. the web under siege - securelist (2009) `http://www.securelist.com/en/analysis?pubid=204792056`.
34. OWASP: Enterprise security API (2011) `http://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API`.
35. Alexa, the Web Information Company: Top 1,000,000 Sites (2010) `http://www.alexa.com/topsites`.
36. Malware Domain List: (2010) `http://www.malwaredomainlist.com/mdlcsv.php`.
37. Curtsinger, C., Livshits, B., Zorn, B., Seifert, C.: Zozzle: Fast and Precise In-Browser JavaScript Malware Detection. In: USENIX Security Symposium. (2011)
38. Wang, Y.M., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., King, S.T.: Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In: Network and Distributed System Security Symposium (NDSS). (2006)