PMForce: Systematically Analyzing postMessage Handlers at Scale

Marius Steffens and Ben Stock
CISPA Helmholtz Center for Information Security
{marius.steffens,stock}@cispa.saarland

ABSTRACT

The Web has become a platform in which sites rely on intricate interactions that span across the boundaries of origins. While the Same-Origin Policy prevents direct data exchange with documents from other origins, the postMessage API offers one relaxation that allows developers to exchange data across these boundaries. While prior manual analysis could show the presence of issues within postMessage handlers, unfortunately, a steep increase in postMessage usage makes any manual approach intractable. To deal with this increased work load, we set out to automatically find issues in postMessage handlers that allow an attacker to execute code in the vulnerable sites, alter client-side state, or leak sensitive information.

To achieve this goal, we present an automated analysis framework running inside the browser, which uses selective forced execution paired with lightweight dynamic taint tracking to find traces in the analyzed handlers that end in sinks allowing for code-execution or state alterations. We use path constraints extracted from the program traces and augment them with Exploit Templates, i.e., additional constraints, ascertaining that a valid assignment that solves all these constraints produces a code-invoking or state-manipulating behavior. Based on these constraints, we use Z3 to generate postMessages aimed at triggering the insecure functionality to prove exploitability, and validate our findings at scale.

We use this framework to conduct the most comprehensive experiment studying the security issues of postMessage handlers found throughout the top 100,000 most influential sites yet, which allows us to find potentially exploitable data flows in 252 unique handlers out of which 111 were automatically exploitable.

CCS CONCEPTS

• **Security and privacy → Web application security**.

KEYWORDS

taint analysis; forced execution; large-scale analysis; XSS

ACM Reference Format:
Marius Steffens and Ben Stock. 2020. PMForce: Systematically Analyzing postMessage Handlers at Scale. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3372297.3417267

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3417267

1 INTRODUCTION

The web that we know today heavily relies on the intricate interplay of various services that jointly contribute to the plethora of feature-rich applications that we have grown fond of over the years. This interconnectivity, however, would not be possible without controlled relaxations of the web's fundamental security principle, i.e., the *Same-Origin Policy* (SOP). The SOP sets a clear security boundary that protects the integrity of web sites by restricting how content from different origins (i.e., the tuple of protocol, host, and port) may interact with one another. That is, an attacker's page cannot read or modify the content of a banking application by accessing the frame or read content from other origins. Sharing data across origins can be very beneficial to users, e.g., having one central billing provider which can be used on various online shops removes the need to provide payment information to all the parties. The shop can then share the invoice with the service provider, which handles confirmation of the user and billing, and then notifies the shop that the transaction was successful. The shop can then continue with the shipping process and provide the user with real-time information about the checkout process.

One such mechanism introduced to allow for the sharing of data across origin boundaries is the postMessage API. It allows sending serializable JavaScript objects from one frame to another by making use of the `postMessage` function that is accessible cross-origin on any frame. The receiving frame can register JavaScript callback functions that are invoked when a postMessage is dispatched to the current frame. The postMessage API provides the means to ascertain the integrity and confidentiality of messages. However, these mechanisms are purely optional. As Son and Shmatikov [24] could show back in 2013 via manual analysis, postMessage handlers are frequently exposing security-critical functionality while not checking the integrity of messages at all or doing so incorrectly. In various cases, these handlers could be abused to achieve Cross-Site Scripting (XSS), which allows an attacker to exfiltrate data and perform actions on behalf of the user. Besides introducing XSS, postMessage handlers can be abused to manipulate client-side state (such as cookies and localStorage) or leak the very same state to attackers. As shown in a recent study by Stock et al. [26], the amount of sites making use of cross-origin communication using the postMessage API has increased between 2013 and 2016 by more than 20% among the top 500 sites. In fact, our crawls show that among the top 100,000 sites we can find over 27,000 hash-unique handlers. Given this widespread usage of postMessages, we can no longer meaningfully rely on manual efforts to reason about the state of postMessage handler security.

To tackle this issue, we present the first automated pipeline, which allows us to analyze the security- and privacy-sensitive behavior of postMessage handlers across the web. We leverage forced

execution with lightweight dynamic taint tracking techniques to explore the complete behavior of handler functions and extract potentially vulnerable program traces. By using an SMT solver on the path constraints extracted from these traces, we can generate postMessages that trigger the dangerous behavior. Furthermore, we augment our traces by encoding exploitation criteria on the data flows, allowing us to generate exploit candidates, which we validate automatically. Contrary to previous approaches, which had to rely on patched browsers or reduced JavaScript language features, our complete pipeline runs as an in-browser solution, without the need to change the underlying JavaScript engine.

We report on a study of the postMessage handlers of the top 100,000 sites, according to Tranco [19]. We find 111 unique handlers with validated flaws, affecting 379 sites. Of those, due to insufficient origin checks, 219 sites can be trivially exploited by any attacker.

To sum up, our work makes the following contributions:

- We present a dynamic execution framework for JavaScript, augmented with forced execution and taint tracking, that automatically collects security- and privacy-relevant program traces from postMessage handlers in Section 3.
- Based on these traces, we show the feasibility of encoding exploitability constraints in Section 4, which allow us to automatically generate payloads that trigger malicious functionality using a state-of-the-art SMT solver.
- With this pipeline in place, we report on the most comprehensive study of the threats of postMessage handlers as of today in Section 5. We analyze the top 100,000 web sites, uncovering abusable security issues on 379 sites from which 219 are trivially exploitable.

2 BACKGROUND AND RELATED WORK

In this section, we provide the necessary background information for our paper. In particular, we briefly discuss the underlying mechanism of postMessages and which attacker models we consider for our analyses and how these might interact with the handler functions of the target site. Subsequently, we outline how our work relates to prior work.

2.1 Background and Attacker Models

The fundamental security boundary of the web is the so-called *Same-Origin Policy* (SOP). The SOP restricts interaction among resources that do not share the same origin by default, e.g., two frames cannot access one another if their origin differs. However, such two frames might need to exchange data to allow users a seamless integration of services, e.g., a website using a third-party payment provider that handles the billing of the customer. To enable such use cases, the postMessage API was introduced to allow for controlled relaxations of the SOP. In the setting of postMessages, one frame sends a postMessage containing arbitrary, serializable data to another frame by calling the postMessage functionality on a handle to the other frame. While the SOP disallows access to functions on cross-origin frames, the postMessage function is an exemption by design to enable cross-origin communication. If the receiving frame has a postMessage handler registered, it will be called with an event that contains the sent data (`event.data`), alongside the origin (`event.origin`) of the sender frame and a handle to the sender frame (`event.source`), as shown in Figure 1.

```
// running at https://foo.com
function handler(event){
    if(event.origin == 'https://bar.org' && event.data == 'Ping')
        event.source.postMessage('Pong','https://bar.org')
}
window.addEventListener('message', handler);

// running at https://bar.org
foo_window.postMessage('Ping', 'https://foo.com')
```

Figure 1: Simple postMessage example

While the postMessage API allows for both the enforcement of integrity and confidentiality of messages, these guarantees are not provided by default. In our example, integrity is enforced by checking that the message originates from `https://bar.org` before executing our intended functionality. Confidentiality is achieved by fixing the second parameter of the postMessage call to the desired destination origin. In the example, this is set to `https://foo.com` as the browsers enforces that the postMessage is only sent when the origin of the frame matches the supplied origin. That is, should the frame have been navigated away for some reason, the message will not be leaked to another origin.

Our work aims at automatically finding security- and privacy-sensitive functionalities inside such handlers, which can be exploited using standard attacker models, i.e., the web attacker. Once the victim visits the attacker's site, the attacker's JavaScript can get a handle to the target page, either via iframes, popups, or newly created tabs, and then send postMessages to the target page. Depending on whether a registered handler performs checks on the origin, the attacker might need to control specific domains. In our example above, an attacker needs to be able to send a postMessage from the origin of `https://bar.com` to trigger functionality.

We are interested in understanding how many handlers conduct security- and privacy-sensitive behavior that can be used across origins. In particular, out of this set of sites, we want to investigate how many of those could be abused by an attacker, e.g., because they lack proper integrity checks, to compromise the site. For our work, we set out to find four types of security- and privacy-related issues, as depicted in Figure 2:

Cross-Site Scripting (XSS). If a postMessage handler uses data sent via a postMessage in the context of a native function that performs a string-to-code conversion, such as `document.write` or `eval`, the attacker can send a message containing a payload that they want to have executed within the vulnerable site. An attacker might leverage this to steal confidential information or perform actions on behalf of the user.

State Manipulation. When the sent data is used in an assignment to `document.cookie` or a localStorage value, the attacker can tamper with the client-side state. Such state is frequently used inside the site, bearing the risk of introducing persistent client-side XSS [25] or undermining the efficacy of security mechanisms deployed by the site. More concretely, if an attacker can arbitrarily change cookies for a given site, they may change the value of a Double Submit cookie [17] and thus be able to circumvent Cross-Site Request Forgery protections or perform Session Fixation attacks [18].

```
function handler(event){
    switch(event.data.mode){
        case 'xss':
            eval(event.data.xss)
        case 'state':
            document.cookie = event.data.cookie
            localStorage[event.data.key] = event.data.value
        case 'launder':
            let frame = document.getElementById('other_window');
            frame.contentWindow.postMessage(event.data, '*')
        case 'leak':
            event.source.postMessage(document.cookie, '*')
    }
}
```

Figure 2: Four types of PM handler vulnerabilities

PM Origin Laundering. Given a postMessage handler that relays received postMessages, an attacker can leverage this functionality to launder their origin and use the origin of the vulnerable handler to circumvent otherwise secure origin checks. In this scenario, the attacker needs to position the frame, which should receive the laundered message relative to where the vulnerable frame relays this message. In the example depicted in Figure 2, the attacker can redirect the frame that is fetched from the DOM as discussed by Barth et al. [3], by using the `window.frames` property.

Privacy Leaks. When a postMessage handler sends out private information, such as user preferences or session information, to another frame when requested to do so, an attacker can potentially trigger this action and leak sensitive information. Such information may be fetched from cookies or localStorage and then sent to the frame, which sent the original message as depicted in Figure 2.

2.2 Related Work

Our related work is mainly distributed among two axes, the first one being the feasibility of applying advanced program analysis techniques such as forced execution and symbolic/concolic execution to web sites, whereas the second area is the identification of security and privacy threats on the web at scale. Naturally, the underlying techniques have been applied in various other domains, e.g., binaries [22] or LLVM IR [5], however, we mainly focus on the area of the web as it introduces its own set of challenges.

Advanced Program Analysis Techniques for Web Security. Saxena et al. [20] used symbolic execution, which was patched into the WebKit engine, to find injection vulnerabilities in web sites automatically. They utilized their engine to generate test cases, which were then further analyzed using dynamic taint tracking approaches combined with fuzzing techniques to find XSS. We purposely chose forced execution over symbolic execution, as our approach only needs to conduct the costly constraint solving step when we have found an interesting trace in the program, as we are only interested in a small subset of all behavior constituting a normal postMessage handler. Similarly to the aforementioned work, Li et al. [11] built a symbolic execution engine with added event exploration mechanisms guided by a taint analysis. In contrast, Kolbitsch et al. [9] picked up on the idea of symbolic execution for malware detection and added what they dubbed Multi Execution, which allows covering multiple symbolic paths in one execution simultaneously. Since postMessage handlers found in the wild are not actively trying to subvert analysis, e.g., by provoking path explosions, the number of paths that need to be explored are typically reasonable, which lessens the need for a Multi Execution framework. Additionally, choosing such an approach would incur changes to the underlying JavaScript engine. Hu et al. [7] built forced execution atop the Webkit engine to find malicious Javascript code. Contrary, to their approach, we opt to implement a selective forced execution framework, which allows us to only forcefully execute PM handlers, while the rest of the code runs normally. Kim et al. [8] presented a crash-free forced execution engine built atop Webkit with a similar goal of uncovering malicious Javascript. Among other things, their engine handles missing DOM elements and exceptions to advance further into the malicious parts of the code. Since our attacker model cannot influence the presence of DOM elements by itself, we are restricted with the environment that is currently present in the page. Therefore, artificially handling such cases would produce infeasible paths that cannot be exploited. Modeling real-world behavior of modern web sites requires extensive support for string operations and regular expressions in the logic of the constraint solver, which has been an ongoing line of research [12, 20, 27, 28] for several years.

While our work shares the common theme of applying advanced programming techniques to the web, we show the feasibility of moving most of the building blocks, i.e., forced execution, dynamic taint analysis, and exploit/test generation, to an in-browser solution. We apply all these building blocks to the domain of postMessage handlers and show the feasibility of modeling in-the-wild behavior in a state-of-the-art SMT solver and discuss how we can augment collected constraints to find exploitable handlers automatically. This enables our approach to be seamlessly migrated to newer browsers with added features, allowing easy extension to new APIs if needed.

Large-Scale Analyses on the Web's Security and Privacy. We now turn towards discussing related works concerning the security and privacy threats on the web at scale. In 2013, Son and Shmatikov [24] presented the first systematic security and privacy analysis of postMessage handlers showcasing real-world vulnerabilities in 84 of the top 10,000 sites by manually analyzing 136 handler functions. With respect to XSS on the client, plenty of research has been conducted on the feasibility of finding client-side XSS at scale [10, 14, 25] by using a browser engine with byte-level taint tracking and context-sensitive exploit generation schemes. Finally, the privacy implications of leaking browser state in the form of cookies was analyzed by Sivakorn et al. [23]. They concluded that a plethora of sites expose sensitive information via cookies.

Since the 2013 paper from Son and Shmatikov, the number of handlers has significantly increased, leaving us to analyze over 27,000 hash-unique handlers rather than 136. This necessitates the need for automated tools capable of analyzing the web at scale. We further show that the 2013 insight that even if postMessage handlers perform origin checks, most of them are faulty and circumventable by an attacker, are no longer valid. Instead, we show that for modern handlers the majority implement the origin checks correctly. Comparing ourselves to the work about client-side XSS detection, these works observe the data flows present within the pages and build exploit candidates purely on the observed values. In contrast, our approach precisely captures all operations performed

```
1  // running at example.com
2  (function(){
3  function isAllowedOrigin(origin){
4      return /example\.com/.test(event.origin);
5  }
6
7  function handler(event){
8      if(!isAllowdOrigin(event.origin))
9          return;
10     if(event.data && event.data.mode == 'eval')
11         eval(event.data.fn.split(',')[1])
12 }
13 window.addEventListener('message', handler);
14 })();
15 // running at example.com.attacker.com with vuln pointing to example.com
16 vuln.postMessage({mode: 'eval', fn:',alert(1)'}, '*')
```

Figure 3: Vulnerable postMessage handler

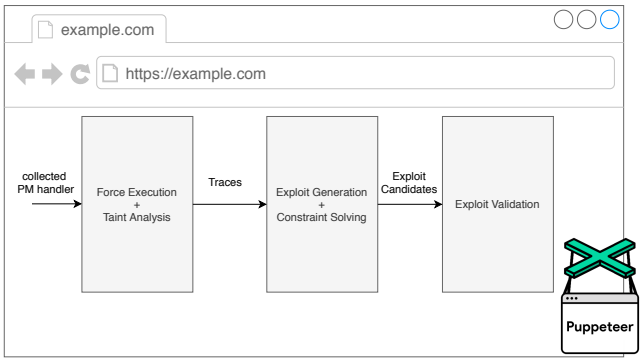

Figure 4: Overview of our approach

on single data flows and can generate a payload accordingly. Finally, while the work of Sivakorn et al. [23] on leaking cookies relied on unencrypted network traffic, transport encryption has become ubiquitous in recent years, making this vector less prevalent. We instead show the dangers associated with postMessage handlers that can leak client-side state, underlining that the threats to privacy can be extended to other threat models.

3 METHODOLOGY

In this section, we discuss our approach leveraging the concepts of forced execution and dynamic taint propagation to automatically extract security and privacy-related traces given a postMessage handler function as input. Furthermore, we explain how we leverage an SMT solver to automatically generate valid postMessages from these traces that trigger the observed functionality. Figure 3 depicts a vulnerable PM handler that serves as a running example throughout this section alongside an exploit that causes an alert to show. An attacker controlling a domain such as `example.com.attacker.com` can send a Javascript object which has the `fn` property set to their payload via a postMessage to the frame that has this handler registered to execute the payload in the vulnerable origin (see line 16).

3.1 PMForce Overview

PMForce consists of three distinct modules, as depicted in Figure 4, that are automatically injected into every frame that we visit using the Chrome Dev Tools protocol. We use the puppeteer Node.js framework to steer our instances of Chromium. All the modules, except for the constraint solving routine, are implemented in Javascript, which allows us to perform most of the necessary operation within the browser itself. As there exists no stable port of Z3 for Javascript, we implemented our constraint solving mechanism in python using `Z3Py`, which is exposed to the other modules via bindings through the Dev Tools protocol, thus accessible through the window object. Our implementation will be available to the general public [1] once this work is published.

In the first step, we use forced execution and taint tracking to find potential flows from the postMessage object into sensitive sinks such as `document.write`, localStorage, and other postMessages. Furthermore, we track flows that stem from all client-side storage mechanisms to check for leakage of privacy-sensitive information.

In the second step, we use these traces to construct JS objects that, when sent as a postMessage, will trigger the sensitive functionality and thus lead to code execution, manipulation of client-side state, or leak information about the client-side storages of the page. To that end, we introduce the concept of *Exploit Templates* and utilize those together with the path constraints found in the traces to generate exploit candidates using Z3 as an SMT solver.

As the last step, we validate that these candidate exploits indeed achieve our intended behavior by calling the unmodified handler code with our candidate exploit as input and checking whether the intended action (such as code execution) was successfully triggered.

3.2 Forced Execution

We leverage the concept of forced execution, in which the control flow of a program is forcefully altered to explore as much code of the program as possible. While other works are making use of symbolic execution for Javascript [12, 20], we only want to make use of the expensive step of constraint solving when we have found an interesting trace through the program. There exist various paths throughout one particular handler, which are not interesting from our point-of-view, which means that we also do not need to generate valid inputs that allow us to reach these points in the program.

To achieve this goal, we utilize the dynamic instrumentation framework Iroh [13] and extend its capabilities where necessary. Doing so allows us, among other things, to register callbacks that are triggered whenever conditionals are evaluated. More specifically, we can also change the results of any of the operations. Figure 5 represents a minimal code snippet that showcases how we can change the outcome of the conditional used within an If statement and thus can choose to either execute the consequence or the alternative. Similarly, we change the values of switch-case constructs to execute particular cases selectively. As a final control-flow altering step, we change the outcome of any expression that is lazily evaluated, i.e., if an OR is lazily evaluated, we change the value of the first expression to false and if an AND is lazily executed we change the value to true. This allows us to forcefully capture the full path constraints, which we need to solve later. In our concrete example of Figure 3 this means that we collect both the constraint that `event.data` must evaluate to true and that `event.data.mode` must be equal to the string `eval` as checked in line 10.

```
// functionCode is the string representation of the function to force execute
let stage = new Iroh.Stage(functionCode);
let IFListener = stage.addListener(Iroh.IF);
IFListener.on("test", (e) => {
        // shouldForceExecute returns true if this Basic Block should be
        ↪ forcefully executed in this program run
    e.value = shouldForceExecute(e.hash);
});
// isNotStale returns true for as long as we can find new code while forcefully
↪ executing the program
while(isNotStale()){
        eval(stage.script)
}
```

Figure 5: Using Iroh to forcefully execute a basic block

Selective Forced Execution. While the initially registered postMessage handlers serve as an entry point into the code portion handling incoming messages, such handler code frequently calls into other pieces of the code, e.g., functions accessible in the scope of the handler to perform origin checks or further process the message. Thus, whenever we forcefully execute a call to a function that is not a native browser function, we instrument this code on the fly and execute our instrumented version instead. Since our instrumentation step relies on Iroh's changes to the source code of the handler functions, the transformation loses all handles to variables defined in the scope where the initial function was defined. In our example, this means that once we have instrumented the initial handler function, any reference to `isAllowedOrigin` is lost, as this was only locally scoped inside the closure. To solve this issue, we execute our complete pipeline in the strict mode of Javascript, such that non-existing variables lead to exceptions. We can then handle these exceptions by fetching the appropriate values, be it basic types, objects, or functions, from the appropriate scope, using the `Debugger` and `Runtime` domain of the Chrome DevTools Protocol [6]. Importantly, the return value of any of our instrumented functions might be dependent on further constraints on the event that is passed to the handler function. Considering our example in Figure 3, we only return true if the origin matches a particular regex. However, there is only an implicit data flow from `event.origin` to the return value of the function. To solve this issue, we emit all path constraints of the called function once we return and append those to the path constraints of the calling function.

Side Effects. Naturally, forced execution of every possible path of the handler function will incur side effects to the page, e.g., change the DOM, add cookies, or change global variables. However, most of these side effects do not affect our further analysis, e.g., even if we change global values, they cannot prevent us from executing specific paths of the program as we are forcing path constraints anyway. Solely side effects that destroy the current execution context or remove elements from the DOM hinder our analysis. The most prominent example of such destructive behavior is a PM handler that is used for authentication, i.e., on a successful authentication, it sets a cookie and reloads the page. Reloading the page will terminate all ongoing JavaScript executions and thus interrupt our analysis. To prevent this, we implement a navigation lock on the currently visited page and abort every navigational request using the Chrome DevTools Protocol [6]. Since our crawlers do not click on any elements, all navigational requests after the initial document load are byproducts of our forced execution and can thus be aborted without changing otherwise benign functionality of the document. As for removing elements from the DOM, we could find handlers that remove certain elements that could be abused if they are still present, e.g., a document.write on the document of a same-origin frame. If this element was removed during our force execution, any subsequent validation attempt would fail. Therefore, while forcefully executing the handler, we substitute such function calls with no operations.

3.3 Taint Analysis

While the forced execution allows us to reach interesting parts of the handler functionality, we still need to discuss how we can leverage it to find traces that are relevant to the security or privacy of the site. To achieve this goal, while we are forcefully executing different paths throughout the handler, we supply the handler function with a Javascript `Proxy` object as input. Such proxy objects allow intercepting accesses to properties on the object. We utilize these traps to persistently capture all operations that the code performs on the proxied object. Together with the dynamic execution engine Iroh and these traps, this builds a lightweight taint-engine which does not rely on modifications of the browser as, e.g., the taint engines of Lekies et al. [10], Melicher et al. [14], Saxena et al. [21], and can selectively be applied to parts of the code. In the following, we discuss how different types of accesses on our Proxy objects need to be handled to ensure that we do not lose taint information and that we capture all necessary operations to allow for the automated generation of attack payloads.

Base Types. The basic case deals with accessed values that are basic types; these might be strings or further Javascript objects. Every proxy object maintains two internal structures, the first one being an identifier, which coming back to our example might be `event.data` or `event.data.mode`, and the operations that were executed on this specific element. This means that if we access a property, say `mode` on a proxy that represents `event.data`. We can simply create a new Proxy that represents `event.data.mode` and remember all operations that were executed on the parent element inside the new object. Naturally, since we start with no knowledge about the expected format of received postMessages for any handler, whenever we encounter properties that are not defined on a proxied object, we initialize those with empty objects. Additionally, we try to infer types of proxied properties based on the further usage throughout the program, e.g., if a string function is accessed on a proxied object, we correct our assignment from an empty object to a string and remember this typing information for later use when solving path constraints.

Functions. When accessing native functions on objects, we need to ascertain that we remove our proxy layer on the arguments before calling the function, as the native functions only work on the underlying wrapped values. After the function call returns, we re-proxy the returned value and note that this native function was called on the proxied object in the internal data structure of the proxied object. When a function is called on a specific object, we not only need to remove the proxy layer for the arguments but also

```
{
      "ops": [
        {
          "type": "ops_on_parent_element",
          "old_ops": [],
          "old_identifier": "event"
        },
        {
          "args": [
            0,
            8
          ],
          "type": "member_function",
          "function_name": "substring"
        },
        {
          "op": "===",
          "val": "https://",
          "side": "left",
          "type": "Binary"
        }
      ],
      "identifier": "event.origin"
}
```


Figure 6: Example output of taint analysis

for the underlying object. In particular, it might be the case that both the object on which the function is called and an argument are proxied values. Any non-native function will be instrumented on-the-fly and thus can handle our proxy objects as input.

Symbols. Symbols are a way to define, e.g., custom iterators on objects [15]. When the program logic iterates over our proxies, they are accessed with the Iterator Symbol as a property. We leverage such accesses, to infer further type information and return an iterator that consecutively outputs further proxied objects that represent accesses to the different indices on the underlying object. While we can leverage this pattern to accommodate any of the currently specified symbols, we could only find that the iterator symbol was of use for our investigated handlers.

Implicit Type Conversions. Similarly to Symbols, the native functions `toString` and `valueOf` need further considerations. These functions are commonly used to convert objects to the same type, which frequently happens when one of our proxied objects is part of a Binary Expression. Thus, we always return the underlying object when these functions are called and within our callbacks of Iroh discern whether the initial program issued this call; thus we need to add it to the operations of the proxy, or whether we caused it and it can, therefore, be omitted.

Provided with the means to handle all operations on such proxy objects, we still need to capture all those expressions in which proxy objects are used, e.g., an equality check to the string `eval` as is the case in our running example in Figure 3. For this, we resort to Iroh's callbacks, allowing us to hook, e.g., Unary and Binary Expression. We apply the corresponding operation to the underlying objects and return the updated proxy as result of the operation. We can then check whether we find any of our proxied objects as part of the conditionals of a control flow statement. Figure 6 shows a sample constraint extracted from a conditional, in which the handler function asserts that the origin is an HTTPS origin.

3.4 Solving Constraints

Our taint analysis allows us to precisely capture all accesses to the event object, thus, whenever we encounter a proxied object as part of a control flow-altering statement, we can add this object to the list of path constraints that would hypothetically need to be fulfilled for this execution path to execute without being forced. Once we encounter an access to a sink, e.g., `document.write` to find XSS, we generate a report containing all the collected path constraints (including negated constraints if we forced specific branches to be false) and the respective object that ended up in the sink and pass this information to our exploit generation engine.

The next step of PMForce consists of transforming our representation of function calls and expressions into Z3 clauses, these can then be attempted to solve and if successful will provide us with assignments to our collected identifiers that execute the intended functionality. Even though Javascript is a weakly typed language and is renowned for having various language quirks, we found that functionality used within real-world postMessage handlers can be reasonably well represented as Z3 clauses. In particular, a prime example of such hardships is that JavaScript allows for comparisons between arbitrary types. Fortunately, in handler functions, such implicit conversions are rarely part of the program logic.

We use our types inferred at runtime to instantiate Z3 variables with fixed types. For variables for which no type hints were recorded at runtime, we defer to treating them as strings. Further, we coerce types on the fly if we observe that two Z3 expressions appear to mismatch, e.g., when we guess that a variable is a string while it is actually compared against an integer, which can be done in JavaScript but lacks an implicit representation in Z3.

In the following, we discuss further considerations that allow us to represent common behavior using Z3 clauses.

Automated Conversion to Boolean. In Javascript, basically any value can be coerced to a boolean value on the fly. This pattern is regularly used to check for the existence of properties on objects, as is done in Figure 3 line 12. In Z3, however, clauses need to be real boolean values as there does not exist any implicit conversion (even though there exist explicit conversions such as `str.to.int`). To allow for the JavaScript shorthand to be representable in Z3, we introduce constraints on basic types that mimic the behavior of Javascript. As an example, the empty string in Javascript is treated as false, while a non-empty string is always treated as true. With these modifications to the clauses, we can emulate the behavior of the Javascript engine for conditionals. While this automated conversion works for most use cases where the values are used inside conditionals, it does not work when the resulting value is further processed. Line 2 in Figure 7 highlights the pattern that a value is assigned to the first object that evaluates to true, a common practice to allow for cross-browser compatibility. Since this value used inside a Binary expression in line 3, coercing it to a boolean value does not work. Since we assume by default that an OR expression produces a boolean, we perform the coercion directly and only later notice that these values are not used as booleans, e.g., when accessing further properties. However, once we use such a value outside of a conditional, we can correct this erroneous coercion. To that end, we introduce a helper variable that must be equal to either of the values and use this helper variable as a substitute for our wrongly coerced

value. Coming back to the example, we then compare this helper variable against `https://foo.com` and correctly enforce that either `event.origin` or `event.originalEvent.origin` must match it.

Regular Expressions. Even though Z3 supports the use of regular expressions, we need to transform Javascript regular expressions into Z3 clauses automatically. We leverage an open-source regex parser[4] and transform the abstract representation into Z3 clauses. Additionally, we emulate the common behavior of Javascript functions that use regular expressions in which the matched string can have arbitrary prefixes and suffixes as long as the regular expression does not force this explicitly using ^ and $ respectively.

String Functions. While Z3 supports various string operations due to work by Zheng et al. [28], functionality exhibited by postMessage handlers quickly exceeds the capabilities that Z3 offers natively. Therefore, we emulate the behavior of commonly used functionality, such as `split` or `search`. We use our collected handlers to find the functionality used in the wild. Since our underlying string solving logic does not incorporate all string functions that the JavaScript engine supports, we need to model some of the function calls with the underlying building blocks of the logic. As an example, the search function in Javascript takes as input a regular expression and checks whether the given string contains a substring matching the regular expression and returns the index of the matching string. To emulate this behavior we introduce a helper variable, asserting that this variable is part of the language spanned by the regular expression, using our regex conversions and Z3's `ReIn`, and then return the index of said helper string in the original string using Z3's `indexOf` operation on strings. While we were able to accommodate most of the behavior found in these handlers, some of the used functionality lacks an explicit representation in Z3. One of the prime examples of behavior that cannot be supported by the current logic of strings is replacement with regular expressions. Although Z3 supports functionality which checks whether or not a string is part of a regular language, and supports string replace on strings, there is no generic way to express string replace with regular expressions with these building blocks. While these are clear limitations of our instantiation of PMForce, which stem from choosing a specific SMT solver, the underlying logics could accommodate such behavior[27].

Non-existent Properties. We found that handler functions regularly check for the presence of objects which are not normally part of an incoming postMessage. Line 2 in Figure 7 shows such an example from the wild, where the `originalEvent` property is accessed, which is not standardized but rather added by libraries such as jQuery. However, some of the handlers are no longer registered via frameworks but rather directly added by using the `addEventListener` function; thus, the accessed property is merely an artifact of continuously evolving code. Naturally, properties other than `event.origin` and `event.data` cannot be abused by an attacker. Since our forced execution collects all constraints, i.e., also those that are part of lazy execution chains that would normally not be relevant, we end up with path constraints that incorporate clauses with identifiers that are not attacker-controllable. More specifically in the aforementioned example we would generate the constraint that either `event.origin` or `event.originalEvent.origin` must

```
1 function handler(event){
2     let origin = event.originalEvent.origin || event.origin;
3     if(origin === 'https://foo.com')
4         // ...
5 }
```

Figure 7: Example of non-existing property usage and lazy-evaluation to objects

pass the origin check. For every property on the event object that cannot be influenced by the attacker, we will thus emit additional constraints asserting them to be equal to the empty string. Doing so will enforce that these properties coerce to false once used inside conditionals on their own. In our example this means that we force the SMT solver to disregard the non-tamperable property and thus find a valid assignment in the `event.origin` property.

4 AUTOMATICALLY VALIDATING POSTMESSAGE SECURITY ISSUES

In this section, we discuss our exploit generation techniques. To that end, we first discuss how we use assignments from Z3 to reconstruct JavaScript objects, followed by our encoding of exploits as Z3 clauses. We then present how we automatically validate that the generated assignments exploit the handler functions to confirm the discovered vulnerabilities.

4.1 Translating Z3 Assignments to JavaScript

Since we use the access patterns as identifier for the Z3 string representation of our constraints, upon solving these constraints, we need to transform the mapping of identifiers to values back to the object that can be called with the handler functionality. For this, we recursively build up the object based on the access path of the identifier. Doing so might unveil imprecisions of our type inference/conversion from Javascript to Z3. If we come back to our initial example of Figure 3, we have the constraint that `event.data` must evaluate to a true value and that `event.data.mode` must be set to a specific string. Since we represent `event.data` as a string value, due to the lack of other options, our assignments incorporate a non-empty string assignment of `event.data`. We add the assigned string of the parent element as another property of the object. This allows us to correctly handle those cases where the assigned strings are necessary, e.g., a check on whether `event.data.toString()` contains a particular substring.

Similar to our taint analysis, which helps us to infer types of our identifiers, there exist cases in which additional typing information is part of our assignments. More concretely, we might have captured in our taint-analysis that `JSON.parse` was used on `event.data` prior to accessing further properties on the loaded object. In these cases, we emit further constraints that force the assignment of a variable representing the type of `event.data` to be JSON. When we encounter such further typing information once reassembling the assignments into a JavaScript object, we adjust the generated object to accommodate for this typing information, e.g., encode the subpart of the data object as JSON.

4.2 Exploit Templates

Until now, we have presented the complete pipeline, which allows us to collect and generate path constraints of security- and privacy-relevant program traces. This allows us to generate assignments that trigger said functionality but do not necessarily exploit them from an attacker's point of view.

To tackle this issue, we also collect the precise information of the operations applied to the proxy object that was called in a sink and encode our payload as further constraints on the underlying object. For this step, we introduce what we call `Exploit Templates`, which is an abstraction on the context in which a specific exploit might trigger. For example, the most basic Exploit Template could enforce that a string flowing into `eval` contains a payload, e.g., `alert(1)`. The constraint solver will then, along with other constraints that stem from the page, find an assignment that fulfills both the constraints of the handler as well as contains our payload. Such a basic template will most likely generate assignments that will not execute our payload, e.g., by generating syntactically incorrect JavaScript code that will then flow into eval. However, this simple example showcases a trade-off that our real templates need to balance; they must be as generic as possible to allow for as many constraints of the page as possible while ascertaining malicious behavior once successfully solved. The basic template is the most generic one there is, as we only ascertain that our payload is *contained* in the assignment, but, it fails to ensure exploitability.

Adding further constraints to the path constraints, however, means that chances that the exploit generation terminates in a reasonable amount of time diminishes. To allow for the analysis to finish without timeouts, we apply each template in a separate query to the SMT solver and refrain from using constraints that are difficult to solve, i.e., regular expressions, in the Exploit Templates. We restrict operations induced by the Exploit Templates to `startsWith` and `endsWith` constraints of fix strings and only enforce that origins must start with either `http://` or `https://` as there is no way to express a valid origin using these restrictions. This allows us to solve most of the path constraints found in the wild augmented with our Exploit Templates in less than 30 seconds. We defer the discussion of timed-out attempts to Section 6.1, where we also provide insights from the encountered timeouts.

Other approaches on finding client-side XSS [10, 14, 25] generate exploits in a manner that is only sensitive to the syntactic structure of the data passed to the sink, but not constraints to even reach the sink. As we observe in practice, though, path constraints regularly impose restrictions on the generated payload, leaving current techniques inapt. In the following, we discuss the considerations that lay the foundation of our different types of Exploit Templates, i.e., templates for XSS and those for client-side state manipulation.

XSS Templates. The overall goal of our XSS Templates is to impose restrictions on the object that ends up in a sink such that an attacker can execute arbitrary code in the page while allowing as many degrees of freedom as possible concerning the exact circumstances. In general, we distinguish two cases depending on whether the sink that is accessed is an HTML executing sink (e.g., innerHTML) or a Javascript sink (e.g., eval). Since HTML parsers are lenient in the way that they parse HTML and allow for various errors (e.g., auto-closing elements if end tags are not found, or parsing of broken tags) the former case can be solved relatively easy by resorting to so-called XSS polyglots [2]. These are payloads intended to break out of as many contexts as possible, before adding pieces of HTML code that then execute the XSS payload. In these cases, our very simple constraint that only enforces that the payload is contained in the string that ended up in the sink suffices. Contrarily for JS, parsers strictly check the syntax and incorrectly breaking out of the current context would violate the syntax. Therefore, we apply various Exploit Templates to capture as many contexts as possible. A common check enforced by sites is that the string that is used inside eval must contain an site-specific substring. A generic template that would capture such a context would essentially ascertain that the string starts with our payload, followed by a JavaScript comment. This template allows the constraint solver to add any arbitrary string at the end, and the comment asserts that anything appended does not tamper with the exploitability.

For more details about the exact Exploit Templates used, we refer the interested reader to Appendix A for an overview or to our codebase for the exact implementation.

State Manipulation Templates. The second goal of our attack scenario consists of the manipulation of the client-side state in the victim's browser. While there exist cases in which an attacker might be able to control keys or values of these stores partially, we specifically target those cases in which an attacker can arbitrarily control the values as these trivially lead to an infection vector for persistent client-side XSS [25] or can allow an attacker to circumvent defense mechanisms, e.g., when the site uses Double Submit cookies to protect against CSRF [17]. To achieve arbitrary control, we enforce in our Exploit Templates for state injections that the attacker can fully control both keys and values of localStorage or cookies.

4.3 Automated Validation

With the generated candidate exploit assignments and our automatic transformation to Javascript objects, we can now use these objects to call the un-instrumented handler functions directly. While directly calling functions with our prepared objects does not perfectly mimic the behavior of sending postMessages using the API across origin boundaries, we note that our exploit generation only sets the data and the origin attributes. We do not make use of properties that cannot be serialized using the structured clone algorithm [16]. Thus the data part of our constructed message is guaranteed to work the same whether or not we make use of the postMessage API. When origin checks are recorded in crawling, we generate origins that fulfill the required constraints. Note that these are not necessarily valid or existing origins, however, enforcing the correct structure of origins would incur an extensive regular expression check that would be difficult to solve using our SMT solver. We assume that whenever we can find an assignment for an origin even if it is incorrect, that there exists a valid origin that still passes the constraints on the origins. We verify that this assumption holds for our investigated handlers when manually analyzing origin checks found in the wild, as discussed in Section 5.2.

To validate the exploitability, we set our payload to either call a logging function (in case of XSS) or invoke storage access with randomized nonces, such that we can later check if the random key with random value has been successfully set. Only when we

Table 1: Overview of discovered handlers using dangerous sinks and prevalence of vulnerabilities. Table shows total number of handlers (by file hash), unique handlers (by structural hash), and vulnerable handlers. Additionally, outlines how many handlers had origin checks and how many *sites* were affected by the vulnerable handlers.

Sink	total number of handlers	number of unique handlers	vulnerable handlers		**with** origin check		**without** origin check	
			number	*sites*	number	*sites*	number	*sites*
eval	132	57	43	*166*	18	*110*	25	*56*
insertAdjacentHTML	38	4	4	*12*	1	*1*	3	*11*
innerHTML	37	37	16	*54*	4	*35*	12	*19*
document.write	26	4	3	*5*	2	*4*	1	*1*
scriptTextContent	4	4	1	*3*	0	*0*	1	*3*
jQuery .html	3	3	1	*1*	0	*0*	1	*1*
sum code execution	217	105	66	*240*	24	*149*	43	*91*
set cookie	108	101	18	*110*	2	*4*	16	*106*
localStorage	63	60	30	*31*	7	*8*	23	*23*
sum state manipulation	161	150	47	*140*	9	*12*	38	*128*
total sum	377	252	111	*379*	32	*160*	80	*219*

find evidence that our candidate indeed triggered the intended functionality we generate a report of a successful exploitation, meaning our analysis does not have any false positive cases.

4.4 Modeling PM Laundering and Leakage

PM laundering and PM leakage both capture similar flows, albeit with slightly different environmental constraints. In the case of PM laundering, the attacker wants to achieve that a postMessage handler relays (parts of) the message that the attacker sent to another frame. This is then received by the second frame with the origin of the relaying frame. Contrary, for PM leakage, the attacker wants to be the target of a postMessage carrying sensitive information such as localStorage entries or cookie values.

In both cases, the attacker needs to be able to control which document receives the postMessage. We can distinguish between two cases of how a target page might send postMessages, i.e., by fetching specific iframe elements from the DOM or by using relative frame handlers such as `top`, `opener` or `event.source`. Unfortunately, as described by Barth et al. [3], an attacker can navigate specific sub-frames of any target page using the `window.frames` property cross-origin, leaving the former trivially exploitable. As for the latter, exploitability strictly boils down to the attacker's capabilities of manipulating these properties, e.g., having a site frame another vulnerable application. Since there is no objective criterion which allows us to define the success of an attacker as these issues are context-specific, we resort to manual analysis in those cases where we find potentially dangerous patterns as output by our dynamic execution engine.

To also account for flows coming from either `document.cookie` and `localStorage`, we replace values fetched from either storage mechanism with our proxy values and capture operations on these as in our general case. This showcases the flexibility of our framework, as we can essentially replace any value with a proxy version to capture all operations performed on these objects.

5 RESULTS

In this section, we discuss the results of applying PMForce to the top 100,000 sites, according to Tranco[19] created on March 22, 2020. We visited each tranco link, and ten randomly selected same-site links found on the starting page and analyzed each handler that was registered by the pages, totalling 758,658 documents and 27,499 handler functions Our experiment was conducted March 23, 2020 and took around 24 hours using 130 parallel instances of our pipeline, using a timeout of 30 seconds per query to the SMT solver.

5.1 Vulnerability Analysis

Table 1 depicts the findings of our experiment on the Tranco top 100,000. The total number of handlers represents the amount of unique handlers per hash sum of the handler code, for which we could observe a tainted data flow into the respective sinks. By manually sampling our results we could find various handlers which use slightly differing layouts, as they were the same library but slightly adapted to the website, or had differing nonces across observed instances of the same handler. To paint a clear picture of how many different families of handlers we could observe to be vulnerable, we used a hash over the lexical structure, i.e., the representation as tokens, of the registered handlers and used this as a distinguishing factor. Overall, this resulted in 10,846 unique handlers that we encountered in our experiment. In total, we found 252 handler families with a data flow to any of our considered sinks, out of which we are unable to analyze 21 due to timeouts and another 21 due to unsupported behavior. We defer a detailed analysis of these issues to Section 6.1.

Naturally, not all of our forcefully found flows are abusable by an attacker, e.g., sanitized values for XSS or only partially controllable storage values. The number of abusable cases represents our automatically verified issues, which can then be further classified among handlers without any check and handlers with origin checks. Even though Son and Shmatikov [24] showed that most origin checks are faulty, we defer a thorough analysis of these checks to Section 5.2.

In terms of direct XSS, we find that `eval` is the most prominent sink, with 43 unique handlers that have an exploitable flow. Out of those, 25 do not perform any origin checks and thus can be exploited by a web attacker without any other pre-conditions. Similarly, 16 handlers use attacker-controllable data in an assignment to `innerHTML`, out of which twelve do not perform an origin check.

```
window.onmessage = function (event){
    if(event.data.type === 'foobar')
        eval(event.data)
}
```

Figure 8: Example of false positive of the taint analysis

Randomly sampling eight (~20%) handlers for which we could not automatically validate code execution flaws, we could find five cases in which exploitability relied on environmental constraints, e.g., the presence of certain DOM elements which were not present in the page. One handler, depicted in Figure 8, is unexploitable. In this handler, it is first checked that the property `type` of `event.data` exists, and subsequently `eval` is called with the entire `event.data` object. To exploit this as an attacker, we'd have to set `event.data` to, e.g., `alert(1)`. The surrounding code, however, expects the data property to be an object with the key `type`, i.e., there is no way to satisfy both constraints. The remaining two handlers ensure that only alphanumerical payloads can be used in the context of the sink, for which none of our Exploit Templates fulfill this criterion. While we cannot automatically validate such cases, the output of our taint analysis might be passed to a human expert to provide a final verdict on the exploitability using domain knowledge to, e.g., bypass custom sanitization or filter routines. However, we were still able to find 43 handlers that lead to a trivial code execution by any web attacker and overall 66 that might be abusable by an attacker if they could compromise a trusted host.

We note here that the sum of handlers with and without origin check amounts to 67. This is caused by the fact that we determine uniqueness on the structure of the directly registered handler, not all code that was used to handle an incoming message. An example of such a handler is shown in Figure 10 in the appendix, where the same dispatcher is used to invoke different functionality (once with and once without origin checks) for different sites. Even though we analyze all hash-unique handlers, the table shows the aggregate of structure-unique handlers, hence folding together cases where the registered handler matches, but the invoked functionality differs.

In terms of arbitrary storage manipulation, we could find that 30 handlers are susceptible to localStorage alterations, while 18 to cookie alterations. Again the vast majority does not perform any checks at all, leading to trivial manipulations by an attacker. Sampling another 20 handlers (~20%) where PMForce was unable to validate storage manipulations, uncovers 19 cases in which the handler only allows certain prefixes for the keys of storage alterations or even allows only a single fixed key. In the remaining handler, we could observe that the constraint solver runs into a timeout, even though an arbitrary storage manipulation was possible, which forms a false negative in our analysis. While alterations of specific key-value pairs might still suffice in a specific attack scenario, this does not capture the attack vector that we set out to investigate, i.e., full control of the client-side storage mechanism.

To conclude our results, we found that 43 handlers allowed for trivial XSS affecting 91 sites, as well as, 38 handlers allowing for storage manipulation affecting 128 sites. In total, an attacker can exploit 219 sites due to a complete lack of origin checks.

Even though an abundance of handler functions are performing non-critical operations, we can still find various handlers that do, and that can be abused by an attacker. This highlights the strengths of PMForce in contrast to manual efforts, which would no longer scale to the current corpus of handler functions.

5.2 Origin Checks

We now turn to analyze the correctness of the origin checks of the problematic handlers we discovered. Using our lexical uniqueness criterion, we captured a total of 32 unique handlers that have exploitable flows once the origin check can be bypassed by an attacker which would affect another 160 sites. Manually examining these checks shows that contrary to the results of Son and Shmatikov [24] from 2013, nowadays, 24 out of the 32 handlers perform strict origin checks that are not circumventable. With 19 out of these 24 handlers, the vast majority compares the origin to a set of fix origins. The remaining five implement checks that allow for arbitrary subdomains of a set of fixed eTLD+1, either via regular expressions or checking that the origin ends with the eTLD+1. The incorrect checks constitute of seven `indexOf` checks that an attacker can circumvent using an arbitrary domain with appropriate subdomains or registering a specifically crafted domain and one incorrect check using a broken regular expression. We can conclude that contrary to previous analyses, origin checks have shifted to being mostly correctly implemented with the exceptional odd-ones out.

5.3 PostMessage Relays

In this section, we set out to discuss the results of our manual investigation of handlers for which we could observe a flow from a received postMessage to another call to the postMessage function. We found a total of 45 unique handlers that exhibited any such flow, from which 25 use the data taken from the received postMessage and use it inside a fixed structure that is then sent further along, thus, not controllable by an attacker. Of the remaining 20 handlers, four reflect the message to the sender, thus cannot be used to relay a message to another frame reliably. In Chrome the `event.source` property will be set to null once the frame that originally sent the message was navigated, thus preventing an attacker from navigating the attack page before the postMessage is processed. Firefox and Safari, in contrast, do not have this protective measure in place, which introduces a race condition, in which the attacker tries to navigate the frame before the vulnerable handler echos the data back using the `event.source` property. While we were able to confirm these issues with toy examples, in which messages are reflected to the sender after 100ms, we discard these cases for our analysis as they are dependent on whether or not an attacker can delay the execution of the vulnerable handler in practice.

Overall, this leaves us with 16 handlers that relay messages that an attacker can abuse. For six, the message is relayed to the parent frame, and ten relay the message to another frame in the same document. As described by Barth et al. [3], frames can be navigated across origins, which allows an attacker to set the location of any target frame across origins, unless site make use of CSP's frame-src directive. In fact, in two of these ten cases, frame-src prevents an attacker from choosing arbitrary targets for the relay.

While the direct security implications of postMessage relays remain dependent on further postMessage handlers, which allow particular origins to execute sensitive functionalities, they unveil a more general issue that arises from the usage of the origin as an integrity check. The receiving frame cannot discern whether the message stems from the benign sender, an attacker, or even any other script that runs in the same origin as the intended sender (e.g., as a third-party script).

5.4 Privacy Leaks

In a separate crawl of the same dataset performed on March 25, 2020, we proxied all elements stemming from either cookie or localStorage and observed flows from these stores which are sent out via a postMessage as described in Section 4.4. We found eight unique handlers with such a flow, for which one was a false positive, and all other flows constituted privacy leaks. Four handlers leaked specific values to the sender, and three leaked arbitrary values that can be influenced via the received postMessage. Contrary to our other cases, these were exclusively found on a single site and were not part of library functionality found on multiple sites.

Naturally, this analysis comes with the inherent limitation that we do not have any means to log in to the sites. While this is a general limitation of a large-scale analysis, our framework could be used in a context where automatic logins are feasible, e.g., assisted by login information of the developer. This would allow us to uncover more functionality of the sites overall, but in particular, could unveil more handler functions which handle sensitive user data since these might only be present after the login.

5.5 Case Studies

In the following, we discuss two case studies that depict interesting vulnerabilities that we could find with PMForce.

Obfuscated Ad Frame. We found an XSS flaw in the obfuscated postMessage handler of an ad company (shown in Figure 9). Our dynamic analysis collected the corresponding values used in the conditionals, which are shown as comments in the source code. The postMessage format expected by the handler consists of four strings separated by the string `~@#bdf#@~`. The first string needs to be `Ad`, and the second string is the injection point. The third and fourth string are used for checks not directly related to the exploitable program trace, however, they need to be present to avoid a runtime error. The `setIfr` method calls `document.write` with our payload enclosed in HTML which is fixed by the page. We note that our approach was able to fully automatically find and validate the exploit; a task that would be extremely time-consuming for a manual analysis of this heavily obfuscated code snippet.

Bot Protection Service. We found that a widely used bot protection service was, once it suspected a browser of being operated automatically, delivering captcha interstitials, which had a vulnerable postMessage handler accepting messages from any origin and using sent data to set cookies. This pattern can be used by an attacker to set arbitrary cookies for all the sites that make use of this protection mechanism by first triggering the bot detection via frequent requests and then use the handler to set cookies. Investigating one of the vulnerable sites, an online real estate market

```
function receiver(a) {
    if (a[_$_8e7c[46]]) { // a['data']
        var s = _$_8e7c[1]; //
        try {
            var r = _$_8e7c[47]; // r = ~@#bdf#@~
            block = a[_$_8e7c[46]][_$_8e7c[48]](r)[3]; //
            ↪ event.data.split('~@#bdf#@~')[3]
            size = a[_$_8e7c[46]][_$_8e7c[48]](r)[2]; //
            ↪ event.data.split('~@#bdf#@~')[2]
            message = a[_$_8e7c[46]][_$_8e7c[48]](r)[1]; //
            ↪ event.data.split('~@#bdf#@~')[1] contains our payload
            s = a[_$_8e7c[46]][_$_8e7c[48]](r)[0] //
            ↪ event.data.split('~@#bdf#@~')[0]
        } catch (ex) {}
        ;if (s === _$_8e7c[49]) { // s == 'Ad'
            try {
                ad = message; // sets global ad value to our injected payload
                if (block == _$_8e7c[50]) { // block == 'true'
                    // ...
                } else {
                    // ...
                }
                ;setIfr(currentIframe, size[_$_8e7c[48]](_$_8e7c[57])[0],
                ↪ size[_$_8e7c[48]](_$_8e7c[57])[1]) // does document.write
                ↪ with ad variable on currentIframe
            } catch (ex) {
                // ...
            }
        }
        // ...
    }
}
// hosted on the attackers page with target pointing to the vulnerable frame
target.postMessage('Ad~@#bdf#@~<img src="foo"
↪ onerror="top.alert(document.domain)"><textarea>~@#bdf#@~a~@#bdf#@~true')
```

Figure 9: Obfuscated Ad handler

place, unveils the use of Double Submit cookies for CSRF prevention. While the Bot prevention also means that any further request is blocked unless a captcha was solved, the attacker can rely on the user to assist in this endeavor. Once the captcha is solved, the handler sets a cookie from the bot prevention service for the target domain indicating the success, thus allowing any subsequent requests until it reclassifies the behavior as suspicious. After the captcha was solved, the attacker can perform the cross-site request and circumvent the protection due to the previously planted cookie.

6 DISCUSSION

In this section, we discuss limitations of our prototype implementation and draw overarching conclusions from our work.

6.1 Limitations

While initial work from 2010 by Saxena et al. [20] showed promising results in using symbolic execution, taint analysis, and fuzzing to uncover XSS on a small scale, it remained an open problem to scale this approach to the web. We could show the feasibility of such a large-scale approach in an in-browser solution that does not rely on patching the underlying JavaScript engine. Furthermore, we remove the necessity of relying on fuzzing with constraint solving. Nevertheless, our approach obviously has limitations related to the applicability of existing tools to our problem space.

Our approach leverages the fact that *most* postMessage handlers in the wild make use of a subset of Javascript behavior that can be reasonably represented in current state of the art SMT solvers. In 21 handlers of our total 252 unique handlers with data flows to relevant sinks, we encountered two types of behavior that interfered

with the analysis. First off, certain behavior is not transferrable to the constraint language. In particular, Z3 only supports ASCII characters. In addition, JavaScript's usage of `bind`, `apply` or `call`, which cannot be handled in a generic fashion. Moreover, Z3 lacks support for backreferences and capture groups in regular expressions, and cannot reason about the length of arrays, as these are represented as functions within Z3. Second, our approach inherits limitations of the open-source software used in our prototype. As an example, the open-source lexer that we used raises errors for specific regular expressions encountered in the wild. Hence, we cannot analyze such handlers. We nevertheless believe this does not impair the conceptual applicability of our approach. Unfortunately, arbitrary programs might induce further issues. In particular, behavior, such as changes to prototypes, usage of implicit type conversions, complex objects such as Sets or Maps, need further modeling.

Additionally, relying on SMT solvers naturally comes with the limitation that satisfiability is NP-complete. We have seen in our analysis that a total of 21 constraints could not be solved due to timeouts. While this is a general limitation that any such analysis faces, there are nonetheless two conclusions we can draw from this limitation. First, if we think of applying PMForce in a development environment, any of the costly operations that lead to timeouts can be rewritten by developers to produce constraints that are easier to test for. For example, we have seen that performing further operations on strings split by a separator, e.g., when passing several values inside one string, quickly exceeds the capabilities that Z3 can solve in reasonable time. However, since postMessages allow for arbitrary serializable objects, those values could also be sent as an array. Furthermore, developer feedback might further be useful to steer the forced execution away from unsolvable paths in the program, e.g., paths that only work for legacy browsers.

Secondly, even in those cases where constraint solving fails, the output of our force execution paired with the taint analysis provides precise information about the constraints that need to be fulfilled to reach a critical path in the program. This information can then be used in further manual analysis to verify exploitability.

6.2 Security and Privacy Issues in postMessage Handlers are Still Prevalent

Son and Shmatikov [24] showed with their manual analysis of postMessage handlers back in 2013, that they are a prime target for XSS and even allow attackers to manipulate the state of the site. Compared to today, the number of sites making use of postMessage and in particular the sheer number of handlers has exploded. During our analysis of the top 100,000 sites we found 27,499 handlers and looking at the top 10,000 specifically we could find 7,599 hash-unique handlers. Comparing this to the amount of handlers found in 2013 this constitutes an increase by a factor of 55. While the number of handlers that are vulnerable did not increase accordingly and is slim compared to the overall corpus of all handlers, this highlights the need for an automated analysis.

Furthermore, our results beg the question of how we can better support developers in securing their sites. We think that providing tool support is a first step in helping developers understand the dangers associated with insecure postMessage handlers; however, we also believe that we should reconsider making the postMessage API secure by default. While an investigation of how we can adapt the postMessage API to make it secure and usable for developers in this work would not do it justice, we nonetheless want to highlight two observations that we hope to be influential for developers and standard authorities alike. First, most of the handlers that protect sensitive behavior implement origin checks correctly. Second, postMessage relays can undermine the security of correct origin checks. PMForce could be used by a first party to vet their trusted third-party scripts, not to include such a relay. Overall, we hope that our work raises awareness and re-opens the discussion about the security of the postMessage API.

6.3 Ethical Considerations

During our large-scale analysis, we try to impact the live versions of the web sites as little as possible, while allowing a thorough assessment of the threat landscape. We restrict our crawlers not to visit more than ten pages of any given site and produce similar resource consumptions as an average user visiting the page and clicking through 10 subpages. We notified the affected parties if we could find any contact on their sites or using well-established security reporting mechanisms (VRPs or `security.txt`). We have already heard back from some vendors and are in active discussions in assisting them in implementing appropriate fixes for the encountered vulnerabilities. We firmly believe that making PMForce available helps developers uncover postMessage handler related issues before being exposed to the public in production systems.

7 CONCLUSION

We showed that the amount of postMessage handlers has increased tremendously over the recent years, rendering any manual efforts to measure the security- and privacy-relevant behavior inept.

We tackle these issues by presenting an in-browser solution that can selectively apply forced execution, and dynamic taint analysis to postMessage handlers found while crawling the 100,000 most popular sites. We track data flows originating from the received postMessage into sensitive sinks such as `eval` for code-execution flaws and `document.cookie` for state-alteration flaws. Once we encounter such potentially dangerous flows, we utilize path constraints collected in our execution framework augmented with what we dubbed Exploit Templates and solve all these constraints using state-of-the-art SMT solvers. Doing so shows that most behavior exhibited by handler functions found in the wild can be represented in our chosen constraint language.

We use the assignments generated by the constraint solver to create exploit candidates that we validate automatically with the uninstrumented handler functions. Doing so allows us to automatically uncover abusable flaws in 111 handlers, which affect 379 sites, out of which 80, affecting 219 sites, do not perform any origin checks, such that a web attacker can trivially exploit those. Contrary to previous analyses, we show that the majority of origin checks protecting sensitive behavior are implemented correctly; thus, no longer allow an attacker to bypass them. Additionally, we report on an analysis of the threat of postMessage relays and privacy leaks via postMessage handlers showcasing how our framework can be further used to uncover flaws in real-world sites.

Table 2: Exploit Templates used

	regular expression	sample code context
T1	`/^alert\(1\)\/\*(.*)\*\/$/`	`if(event.data.indexOf('foobar') !== -1){ eval(event.data) }`
T2	`/^\(alert\(1\)\/\*(.*)\*\/\)$/`	`if(event.data.indexOf('foobar') !== -1){ eval('('+event.data+')') }`
T3	`/^\/\*(.*)\*\/alert\(1\)$/`	`if(event.data.indexOf('foobar') !== -1){ eval(event.data) }`
T4	`/\.toString\(\),alert\(1\)$/`	`eval('globalLib.' + event.data.fun)`
T5	`/=1,alert\(1\)$/`	`eval('foo=' + value)`
T6	`/\(function\(\){alert\(1\)}\)\(\);\/\/(.*)/`	`let fun = eval('function(){' + value + '}')`

```
// site 1, with origin check
function actual_functionality(e) {
  if (e.origin == 'https://foo.com') {
    eval(e.data);
  }
}

// generic handler on which we calculate structure uniqueness
function dispatcher(e) { actual_functionality(e) };
window.addEventListener("message", dispatcher);

// site 2, no origin check
function actual_functionality(e) {
  eval(e.data)
}

// generic handler on which we calculate structure uniqueness
function dispatcher(e) { actual_functionality(e) };
window.addEventListener("message", dispatcher);
```

Figure 10: Example of simple dispatcher functions

A EXPLOIT TEMPLATES

Table 2 represents examples of our templates that capture all contexts that we currently cover for JavaScript sinks. Note that while the template is represented as a regular expression, we used startsWith-/endsWith constraints to lessen the burden on the SMT solver.

REFERENCES

[1] 2020. PMForce Code. (2020). https://github.com/mariussteffens/pmforce

[2] Ahmed Elsobky. 2018. Unleashing an Ultimate XSS Polyglot . https://github.com/0xsobky/HackVault/wiki/Unleashing-an-Ultimate-XSS-Polyglot. (2018). [accessed 06-Apr-2020].

[3] Adam Barth, Collin Jackson, and John C Mitchell. 2009. Securing frame communication in browsers. *Commun. ACM* 52, 6 (2009).

[4] blukat29. 2020. regex-crossword-solver. https://github.com/blukat29/regex-crossword-solver. (2020). [accessed 06-Apr-2020].

[5] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*.

[6] Google. 2020. Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/. (2020). [accessed 06-Apr-2020].

[7] Xunchao Hu, Yao Cheng, Yue Duan, Andrew Henderson, and Heng Yin. 2017. Jsforce: A forced execution engine for malicious javascript detection. In *International Conference on Security and Privacy in Communication Systems*.

[8] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. 2017. J-force: Forced execution on javascript. In *WWW*.

[9] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2012. Rozzle: De-cloaking internet malware. In *IEEE Symposium on Security & Privacy*.

[10] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: Large-scale detection of DOM-based XSS. In *CCS*.

[11] Guodong Li, Esben Andreasen, and Indradeep Ghosh. 2014. SymJS: automatic symbolic testing of JavaScript web applications. In *FSE*.

[12] Blake Loring, Duncan Mitchell, and Johannes Kinder. 2017. ExpoSE: practical symbolic execution of standalone JavaScript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*.

[13] Felix Maier. 2020. Iroh. https://github.com/maierfelix/Iroh. (2020). [accessed 06-Apr-2020].

[14] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out domsday: Towards detecting and preventing dom cross-site scripting. In *NDSS*.

[15] Mozilla Developer Network. 2020. Symbol. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol. (2020). [accessed 06-Apr-2020].

[16] Mozilla Developer Network. 2020. The structured clone algorithm. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm. (2020). [accessed 06-Apr-2020].

[17] OWASP. 2020. Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet. https://owasp.org/www-project-cheat-sheets/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet#double-submit-cookie. (2020). [accessed 06-Apr-2020].

[18] OWASP. 2020. Session fixation. https://owasp.org/www-community/attacks/Session_fixation. (2020). [accessed 06-Apr-2020].

[19] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A research-oriented top sites ranking hardened against manipulation. *NDSS* (2019). https://tranco-list.eu/list/NZQW/100000

[20] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A symbolic execution framework for javascript. In *IEEE Symposium on Security & Privacy*.

[21] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. 2010. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *NDSS*.

[22] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security & Privacy*.

[23] Suphannee Sivakorn, Iasonas Polakis, and Angelos D Keromytis. 2016. The cracked cookie jar: HTTP cookie hijacking and the exposure of private information. In *IEEE Symposium on Security & Privacy*.

[24] Sooel Son and Vitaly Shmatikov. 2013. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *NDSS*.

[25] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. 2019. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild.. In *NDSS*.

[26] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. 2017. How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security. In *USENIX Security*.

[27] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A symbolic string solver for vulnerability detection in web applications. In *CCS*.

[28] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*.