Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Observing web sites from inside the browser – Instrumenting WebAPIs in Firefox

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Stefan Zabka
geboren am: 31.08.1998
geboren in: Berlin

Gutachter: Prof. Dr. rer. nat. Jens-Peter Redlich
Prof. Dr. rer. nat. Lars Grunske

eingereicht am: ..... verteidigt am: .....

Contents

Abstract

Observing websites behavior is crucial to understand the state of web privacy. To observe websites' behavior, researchers need to capture WebAPI function calls by either changing the websites code or manipulating the browser. Researchers have used either approach to write highly specialized tools that collect data to answer their specific research question. This leads to a lot of duplicated effort in the fundamental building blocks of website instrumentation. In this thesis I will present CallMonitor, a general purpose website observability component for the Firefox browser, which aims to make one-off tools obsolete. After explaining its current implementation and its design decisions, I will evaluate it against an existing general purpose WebAPI instrumentation tool, OpenWPM's JavaScript instrument, and discuss that while the current implementation isn't better than the JavaScript instrument, CallMonitor's design allows it to surpass the limitations inherent in the JavaScript instrument's design.

1 Introduction

1.1 Motivation

The World Wide Web is a diverse ecosystem that most people use in their day-to-day lives. The browser is a complex execution platform through which most people experience the web. The variety of websites that a user can visit range from personal websites, that have been written over a weekend to web portals created and operated by multi-billion dollar entities and everything in between. All of them aim to provide an experience and use the browser to provide that experience.

Through the use of client-side scripting, most commonly JavaScript, and rich browser APIs website operators are able to deliver a highly dynamic, tailored and engaging user experience. However, the big interaction surface between websites and browsers also allows for websites to collect data regarding the configuration of a user's browser and system, so-called browser fingerprinting [1].

To observe whether a website is engaging in browser fingerprinting privacy researchers need to be able to observe and analyze website's behaviors.

While there are existing tools in this space, most of them are purpose-built and non-reusable, which leads to duplicated effort and a lack of comparability between studies.

1.2 Terminology

The following section will define common terms and introduce less commonly known terms for the reader.

Document The Web Hypertext Application Technology Working Group (WHATWG) defines a document as a structured, markup-based file, that can be represented through a tree [2]. In the context of the World Wide Web, most documents are HTML documents, but other formats such as SVG or XML are also common.

Browser Browsers originally were applications that loaded and rendered documents loaded from the internet. Over time, they expanded their capabilities to not only display static documents, but also run scripts and display multimedia content such as pictures, audio and video. The latest browsers also offer the user to share information about themselves or their device, such as their location or attached devices, such as cameras, microphones or bluetooth connected devices.
The three most popular browsers are in order Google Chrome, Apple Safari and Mozilla Firefox [3].

ECMAScript ECMAScript is a language standard created by the ECMA organization that specifies a weakly typed and dynamic programming language with first class functions [4].
ECMAScript uses prototype based programming, meaning that instead of an object being of a certain type, which defines the valid methods and fields on it, objects can share behaviors and properties by having the same `prototype` object. When properties that are undefined on the current object, the property is looked up on its `prototype`. This process happens recursively until the property is either found or the `prototype` of an object is undefined.[5].
ECMAScript is implemented by the JavaScript interpreters of the browsers. Each major engine maintains its own JavaScript interpreter. Firefox's interpreter is called SpiderMonkey. Chrome's is named V8 and Safari has JavaScriptCore. The ECMAScript specification intentionally does not define facilities for receiving data or communicating computation results with anything outside the language environment, which is explicitly left to hosts, such as browsers.

WebAPI A WebAPI is an application programming interface (API) that is exposed to JavaScript code running the browser. Since ECMAScript doesn't specify a way for the JavaScript code of a website to interact with the rest of the browser, browser vendors created APIs that enable websites to interact the loaded document and the browser [6].
These APIs are specified by the World Wide Web Consortium (W3C) or the Web Hypertext Application Technology Working Group (WHATWG) through numerous WebAPI standards. WebAPIs provide a wide range of functionality including basics

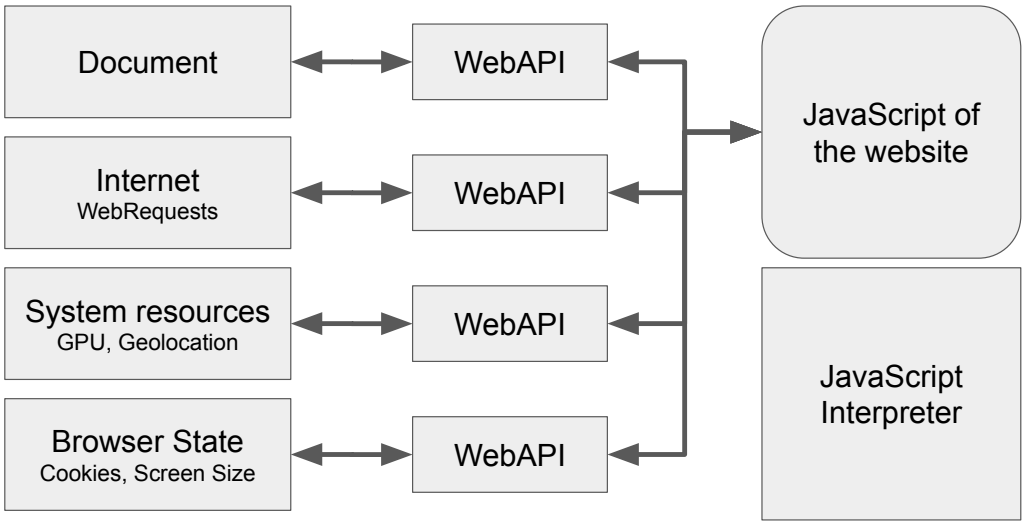

Figure 1: A visualization aid showing components of a browser (rectangles), the JavaScript code of a website (round box) and their possible interactions (arrows). The figure highlights that websites can only observe and interact with its own document or external resources through WebAPIs

such as DOM access and manipulation as well as more advanced tasks such as geolocation or audio-video-processing. They also allow querying for computer specific state such as the operating system, the current window size, the GPU model and driver or installed fonts.

DOM The Document Object Model (DOM) is the core abstraction of the Web. It connects web pages to scripts or programming languages by representing the structure of a document as a tree [7].

While it could theoretically be used in other languages or to represent other document types, it usually describes the programmatically exposed structure of an HTML document to JavaScript running on the same website.

The DOM standard is developed in conjunction with the HTML specification by the WHATWG.

Browser Fingerprinting Browser Fingerprinting, as defined by Laperdrix et. al., is the practice of collecting data regarding the configuration of a user's browser and system [1].

While most WebAPIs including the DOM provide information, that expose little or no entropy, meaning they return the same or almost the same result on any browser and system, some WebAPIs produce a wide variety of results that are (almost) unique to

an individual's browser and system, which allows websites to identify users across browsing sessions.
Some of these APIs are the graphics (WebGL) and audio WebAPIs as they expose access to the underlying hardware. The WebGL APIs also expose information about the system configuration, such as installed fonts or drivers.

1.3 Problem statement

It is imperative that Web privacy researchers are able to observe how a website uses JavaScript to interact with the browser, to detect whether it is fingerprinting the user and if so, which techniques it uses.
As JavaScript needs to use the WebAPIs to affect or observe state outside the interpreter (see Figure 1) and browser fingerprinting requires information outside the interpreter, it is sufficient for the researchers to be able to observe all calls through the WebAPIs to detect whether a website is engaging in malicious activities. Unfortunately there are no standardized interfaces for observing WebAPI function calls. This leaves researchers with two options:

1. Change the websites code to record calls before they reach the WebAPI.

2. Change the browser to record WebAPI function calls after they have been made.

Current research uses both of these approaches, however most commonly they only modify the browser in such a way that it answers their specific research question [8] [9]. This lack of reusability leads to a lot of duplicated effort and hinders combining and comparing the collected data. To mitigate this, a number of open source browser-based web crawlers have been published in the last years, such as DuckDuckGo's Tracker Radar Collector [10] or Kraaler [11].
One of the most commonly used crawlers is OpenWPM, a modular web crawler written in Python that uses the Firefox browser and the Selenium automation framework to crawl the web [12]. It provides facilities to detect a wide range of website activities as well as a platform that allows for parallelism and abstracts data storage. One of OpenWPM's tools for data collection is the JavaScript instrument which injects scripts into the website to record WebAPI function calls.
Due to the popularity of Chromium-based browsers (such as Google Chrome or Microsoft Edge), many browser-based crawlers (including the Tracker Radar Collector and Kraaler) use Chromium for data collection. For crawls using Firefox, OpenWPM is the leading option, but as it changes the website code to intercept calls to the WebAPI functions, it is inherently limited in performance and detectability by having to use JavaScript.
The lack of a browser-integrated, general purpose instrumentation tool that uses the Firefox browser will be addressed in this thesis. This tool should allow researchers to instrument arbitrary WebAPI function calls with sufficient metadata to be able to advance the state of web privacy research while being indistinguishable from an

uninstrumented browser and ensure that web privacy research doesn't only rely on data collected for a single engine.
This goal is achieved, if CallMonitor captures the same width of information as the JavaScript instrument currently does, without CallMonitor exhibiting the same shortcomings as the JavaScript based implementation.

1.4 Structure

In Section 2 I will explain the JavaScript instrument's architecture and examine its strengths and weaknesses.
After that, in Section 3, I will explain CallMonitor, my implementation of a component, that changes the way Firefox processes WebAPI function calls to record them. I will begin by defining a set of requirements, explain the Firefox process model, derive an architecture for the component, and discuss three implementation issues.
Finally, in Section 4, I will evaluate my current implementation against the JavaScript instrument and conclude by defining the steps necessary for my proof of concept to achieve feature parity with the existing implementation in Section 5.

2 OpenWPM's JavaScript Instrument

OpenWPM is an open source web privacy measurement platform, built on top of Firefox with parallelism, robustness and modularity in mind [12]. It provides tools for simulating user activity, through the use of the browser automation framework Selenium and recording a wide range of website behavior. Its modularity allows researchers to define new observation tools and crawling strategies, making it flexible enough to be used in over 76 studies. [13]
One of these observation tools is the JavaScript instrument. It allows researchers to specify which WebAPI function calls they are interested in and which kind of data should be captured for each call.
In this section I will outline the foundational knowledge required to understand the JavaScript instrument, describe its implementation and finally outline its strengths and weaknesses.

2.1 The Architecture

OpenWPM uses a custom add-on, that cotains its data collection utilities, which communicates with the execution platform, from where it receives commands and to which it sends the collected data.
This add-on, as all modern add-ons, uses the WebExtension standard, which will be described in this section. Additionally, this section will explain how the JavaScript uses the facilities provided by the WebExtension standard to record WebAPI function calls.

2.1.1 Architecture of a WebExtension

Extensions, also called add-ons, can modify and enhance the users experience of a browser. Common use cases include ad blockers, password managers or themes. Extensions in Firefox use the standardized WebExtension APIs. These APIs expose a wide variety of functionality to add-on developers, such as the clipboard, the users history or multiple points at which the add-on developer can intercept and manipulate requests made to the internet by websites (before the request is made, after it has been received, etc.). [14].
The standard also specifies a packaging format to declaratively describe a WebExtension. A WebExtension can contain the following artifacts:

- manifest.json
- Background scripts
- Content scripts
- Icons
- Sidebars, popups, and options pages

- Web-accessible resources

The manifest.json is mandatory as it contains extension metadata, including which other files are part of the extension and which APIs the add-on intends to use, so the user can be asked for those specific permissions when installing the it [15].

Background Scripts Background scripts maintain long-term state. They have access to the full WebExtension APIs, if they have requested the appropriate permissions.
E.g. background scripts can dynamically register new content scripts or maintain a connection to external servers.

Content Scripts Content scripts have access to and can manipulate web pages. They are loaded into the web page and run in the context of that page.
They have limited access to WebExtension APIs but have the ability to send messages to the background script which can then interact with the world on their behalf.
While content scripts are running in the context of the web page and can access variables exposed on the document of the website, they do not have the same view of the document as the JavaScript of the website, instead they see the document through so called Xray Wrappers.

Xray Wrappers To allow WebExtensions to safely access the contents of a web page, Firefox always exposes the native version of a WebAPI to content scripts. This means that even if e.g. a website were to override a property on an HTML element like so

```
1    document.getElementById("testElement").matches =
2     (s) => {return true;}
```

if the WebExtension were to call the `matches` function on `testElement`, it would still call the original implementation. This way the WebExtension can trust that it is able to safely navigate the DOM or receive truthful information when querying for website state [16].
However, this practice also works the other way around, so any modifications the content script does do its view of the document do not get reflected back into the websites view.
This is problematic when instrumenting WebAPIs with the JavaScript instrument, as the instrumentation code has to wrap the WebAPIs in order to capture calls from the website before they reach the WebAPI. The circumvention of this mechanism and its consequences are described in the next section.

2.1.2 JavaScript Instrument

The JavaScript instrument is part of the add-on, that OpenWPM installs in the Firefox browser before it visits a website, to collect data during a visit. In particular, the JavaScript instrument allows the user to specify a set of WebAPIs and whether they want to capture the callstack or the arguments of a WebAPI function call.

Figure 2 shows the process by which the JavaScript instrument gets set up and sends information back to the aggregator (an external server receiving and saving out captured data) once a WebAPI function gets called.

The diagram's left side should be understood as OpenWPM's automation platform, as it issues commands to and receives information from the browser and the add-on. The diagram starts off after the browser has been started, the extension has been installed, and the platform has detected that the background script is ready to receive information.

The first step is sending the config to the background script, which then proceeds to generate an appropriate content script, which it registers with the browser using WebExtension APIs. At this point the configuration is baked into the script and can no longer be changed.

Once the platform is informed that the registration is complete, it instructs the browser to visit a website. This results the browser creating a new WebContent environment, loading the content, injecting the previously registered content script into the scope of the website and executing it. This is the point where the content script needs to bypass the X-Ray wrappers to make sure that it isn't wrapping its own WebAPIs but the ones exposed to the web content. To do this the content script creates a `<script>` tag as the first element, letting it do the wrapping and then removing it again. As this happens before the web content starts executing, it is undetectable for the website code.

Wrapping is done in two different ways depending what is being instrumented. If it is a (nested) property of the global Window object, such as `window.screen.height`, the JavaScript instrument calls `Object.defineProperty` on the property to wrap the property in a getter. However, if the object being instrumented is one that does not exist by default, such as the `IntersectionObserver`, the JavaScript instrument overrides its prototype's properties instead. This way, any object that has `IntersectionObserver` as its prototype, will call instrumented methods.

As the wrapping code now is no longer considered part of the WebExtension code it doesn't have to deal with X-Ray wrappers but also has no access to the WebExtension APIs. To still communicate with the background script it has to use the content script as a relay. The wrapping code emits a custom event, which is propagated to the event handler of the content script, which then uses the `runtime.SendMessage` API to forward the message to the background script. To capture information about the caller of the API, the instrument creates a JavaScript `Error` object and accesses the Firefox-specific API `stack` which contains the callstack of an error as a string [17]. It also captures the arguments array by serializing them before the WebAPI function call.

This process is repeated for every call to a WebAPI function that has been instru-

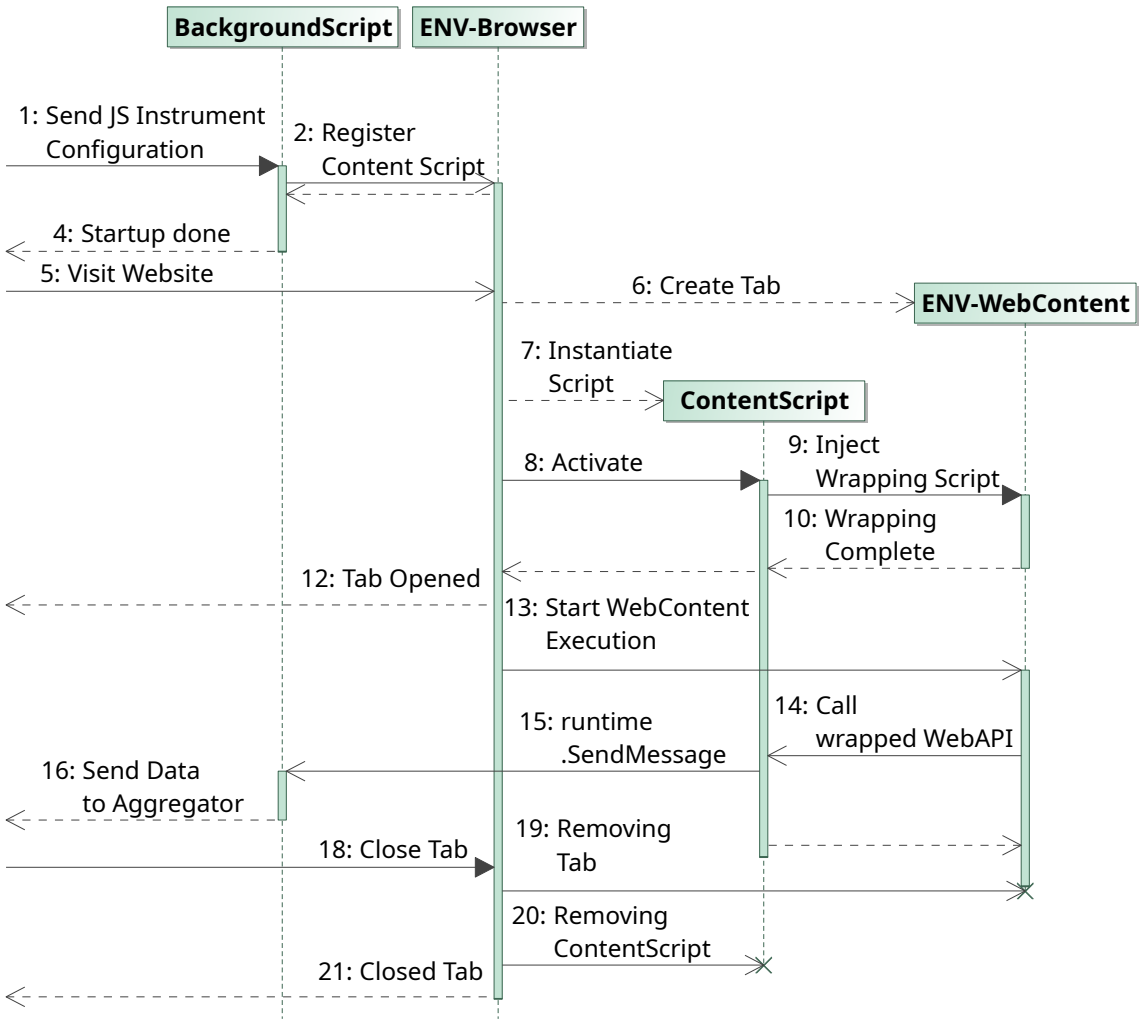

Figure 2: An UML sequence diagram showing the process of injecting instrumentation code into the web page and the propagation of events captured by the JavaScript instrument to OpenWPM's execution platform (at the left)

mented until the platform receives a command to close the tab and orders the browser to do so. The browser then proceeds to destroy the tab containing the web content environment and the content script.

2.2 Strengths and Weaknesses

The JavaScript instrument has some very clear strengths and less obvious weaknesses. Its strengths lay in its unrestricted access to JavaScript's introspection capabilities. Being able to access information about the caller of a WebAPI via the call stack and serialize the arguments using language level utilities is extremely powerful, as it allows it to emit rich events, that capture a lot of metadata. This allows researchers to later correlate these events and use them for complex analysis.

An example of a websites script and the respective captured event can be seen in Figure 3

Through WebExtension APIs exposed in the content script, OpenWPM is able to record the window, tab and frame id. This makes it possible to attribute the function call to a specific iFrame, an isolated document which websites can embed in themselves but don't control [18]. This information is crucial to understand, if the call was made by the website visited, a so called first-party call, or by another site that was embedded, which would be a third-party call.

In addition, by analyzing the callstack, they can observe the URL of all scripts involved in the call as well as the location of each function in their respective files. This information is useful to determine, the source of tracking scripts even when they are embedded in a first-party site.

However, the instrumentation also has one major vulnerability, the wrapping code which is running as part of the website. As it has disabled the Xray wrappers to have direct access to the WebAPIs, the browser treats it as part of the website. This allows a malicious website to detect whether a given API is instrumented by calling `toString` on the function. If it has been overwritten, `toString` will return the source code of the function, otherwise it will just return `[native code]`. This detection vector is used in Section 4 to evade the JS instrument.

In addition to malicious detection, the overwritten function could also be detected by benign, so-called polyfill scripts, that are used to implement newer features to older browser, by checking if the function exists and if it can not be detected, providing their own JavaScript implementation for the feature. Since the wrapped functions does not behave like a native implementation, the script might mistake them for missing or outdated functions and overwrite them.

```javascript
function func(data) {
  window.btoa(data);
}
function main() {
  func("abcd");
}
```

```json
{
  "id" : 1,
  "incognito" : 0,
  "browser_id" : 1812454466,
  "visit_id" : 7003191432381221,
  "extension_session_uuid" :
   ↪ "ac42fb19-c56f-4b30-9025-c69a1572adbb",
  "event_ordinal" : 6,
  "page_scoped_event_ordinal" : 0,
  "window_id" : 1,
  "tab_id" : 1,
  "frame_id" : 0,
  "script_url" : "http:\/\/0.0.0.0:8000\/evil.html",
  "script_line" : "6",
  "script_col" : "33",
  "func_name" : "func",
  "script_loc_eval" : "",
  "document_url" : "http:\/\/0.0.0.0:8000\/evil.html",
  "top_level_url" : "http:\/\/0.0.0.0:8000\/evil.html",
  "call_stack" : "func@http:\/\/0.0.0.0:8000\/evil.html:6:33
                  main@http:\/\/0.0.0.0:8000\/evil.html:13:11",
  "symbol" : "window.btoa",
  "operation" : "call",
  "value" : "",
  "arguments" : "[\"abcd\"]",
  "time_stamp" : "2022-01-22T12:01:17.608Z"
}
```

Figure 3: A piece of JavaScript code and a JSON representation of the event captured by the JavaScript instrument when the `window.btoa` function was called. The event is rich in metadata, capturing not only the arguments but also the exact caller location in the file and the callstack

3 CallMonitor

CallMonitor is a component for Firefox that designed to allow consumers to dynamically instrument arbitrary WebAPIs.
Consumers in this context are both researchers through a non-standard extension to the WebExtension API and browser developers through a Firefox internal API.
In this section I will cover the requirements for the component, explain the necessary concepts of Firefox to understand the design of CallMonitor, give a high level overview of the design and conclude by highlighting interesting implementation decisions.

3.1 Requirements

CallMonitor aims to replace OpenWPM's JavaScript instrument by being undetectable while maintaining the vast amount of information captured by the current instrumentation. To compete CallMonitor needs to capture the callstack of a WebAPI function call as well as information about the top level window and the iFrame from which the call originated. The experience of using it should not be worse than using the existing instrumentation.
In order for CallMonitor to be accepted into the Firefox codebase, it must be maintainable, tested, and easily extensible.
Since Firefox has been under active development for more than 20 years, it has built custom infrastructure for common problems in the codebase, such as abstractions for inter-language function calls or interprocess communication.
In order to achieve my maintainability requirement, I used these building blocks and designed my component around them.
In Section 5 I will compare the progress made as part of this thesis against these requirements.

3.2 Implementation

In this section I will explain the process layout of Firefox and the existing solutions for IPC and inter-language function calls to give context for the layout of the CallMonitor. I will then proceed to explain CallMonitor's current implementation leaving my design considerations for Section 3.3

3.2.1 Context

Multi Process Architecture All major browsers use a multi process architecture. This means they run different tasks on different processes, allowing them to use OS level security features to protect against malicious web code [19] [20].
Firefox, for example, uses a central process called main process, to maintain browser state, such as history or the password manager and other processes to deal with the different stages of processing a website, such as loading the website in the socket

process, parsing the HTML, CSS and JavaScript and executing their JavaScript in a web content process or drawing them in the render process [21].
This separation of responsibility and defining static interfaces between processes through actors, makes it possible to mitigate the impact of a single exploit. Should, for example, an attacker be able to achieve arbitrary code execution on the network process, they will still be unable to access cookies or draw to the screen.
The processes that are relevant in the context of this thesis are

- the web content process as the place, where CallMonitor collects the information about WebAPI function calls
- and the main process, where CallMonitor subscribers register to receive information

WebIDL WebIDL is an interface definition language, that is used by the WHATWG to specify WebAPIs [22]. The IDL is a typed language that specifies standardized interfaces and defines how these interfaces should be exposed to ECMAScript. By defining its semantics in terms of ECMAScript, the IDL is able to achieve consistent behavior, regarding things such as iteration order, for different WebAPIs and across browsers.
Firefox consumes these IDL files to generate binding code. This binding code gets called whenever a WebAPI function is invoked. It unwraps the JavaScript arguments, if they exist, checks if the invariants encoded in the IDL are upheld, and calls the C++ implementation of the function.
This will be discussed further in Section 3.3.1.

Actors To allow for secure communication across processes Firefox uses an actor model. In the traditional actor model, there is no global state instead actors are independent entities that are only able to perform three types of actions, changing their internal state, sending messages to other actors or creating new actors [23].
However unlike the traditionally defined actor semantics Firefox's actors can not reach out to arbitrary other actors or create arbitrary new actors. Instead, actors always exist as pairs, where one side is defined as a parent and the other as a child, and can only send statically defined messages to their counterpart. These actor pairs are split into two classes, top-level actors and managed actors. Top-level actor pairs initialize themselves by connecting with their counterpart over a message pipe. This connection is then used to communicate between the two endpoints. All processes in Firefox have a top-level actor pair to communicate with the main process, this way the main process can coordinate the different processes. The main process always acts as a parent in these relationships.
Managed actors do not initialize themselves but instead get created by their manager, when their manager receives a request to create them. Since a managed actor doesn't have its own message pipe, it has to go up the management chain to the top-level actor to use its message pipe to reach its counterpart on the other side. As managed

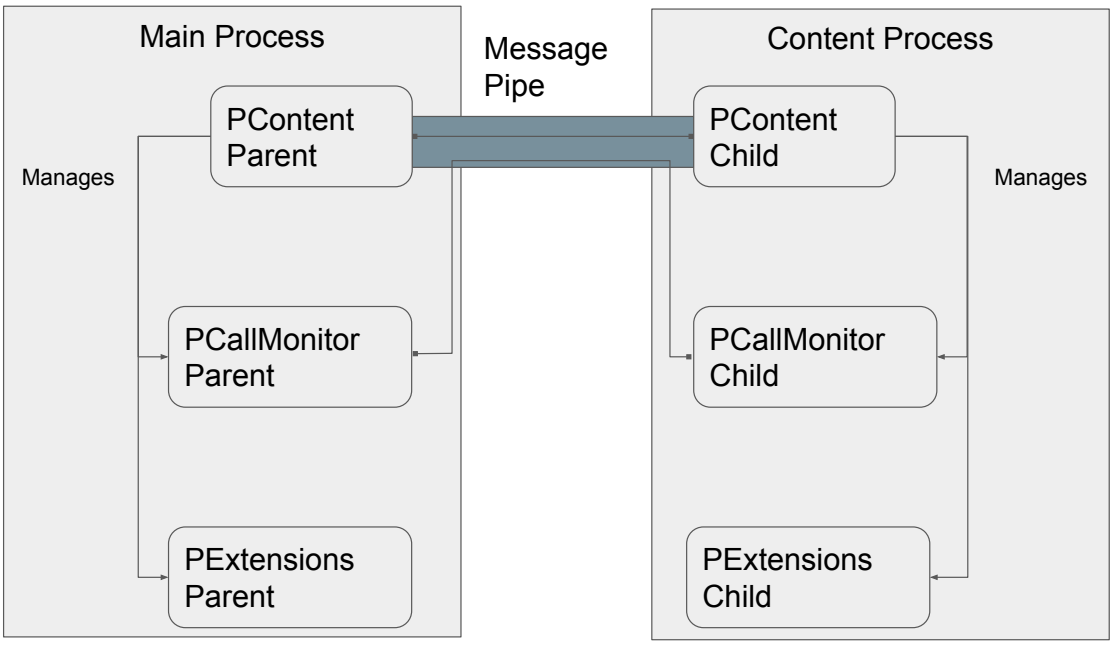

Figure 4: A visualization aid showing the relationship between top level actors and their managees

actors are subordinate to their manager, they will be automatically shut down when their manager is.

Figure 4 exemplifies this by showing a simplified actor setup for the relationship between a content process and the main process. When PCallMonitorChild wants to send a message to its parent, it does not have a direct connection but instead communicates with its counterpart through the message pipe of the top-level actor, which happens to be its direct manager.

Actor protocols in the Firefox code base are defined by IPDL, the interprocess communication definition language. Protocols have to use the prefix P. In the examplary figure the top-level protocol is called PContent, which manages the PExtensions and PCallMonitor protocols. Each protocol defines which types of messages can be sent, in which directions they can be sent and which values they contain.

XPCOM XPCOM, the Cross Platform Common Object Model, is a technology that allows sharing objects between different programming languages. In the context of Firefox XPCOM allows developers to write a component in a language of their choice and expose it to JavaScript, C++ and Rust.

XPCOM interfaces are defined in XPIDL, the XPCOM interface definition language and canonically are prefixed with nsI. XPIDL allows Firefox developers to typed interfaces with functions and properties. An object that implements an XPCOM interface is commonly referred to as an XPCOM component. References to XPCOM components can be acquired in a language-agnostic way through XPCOM services.

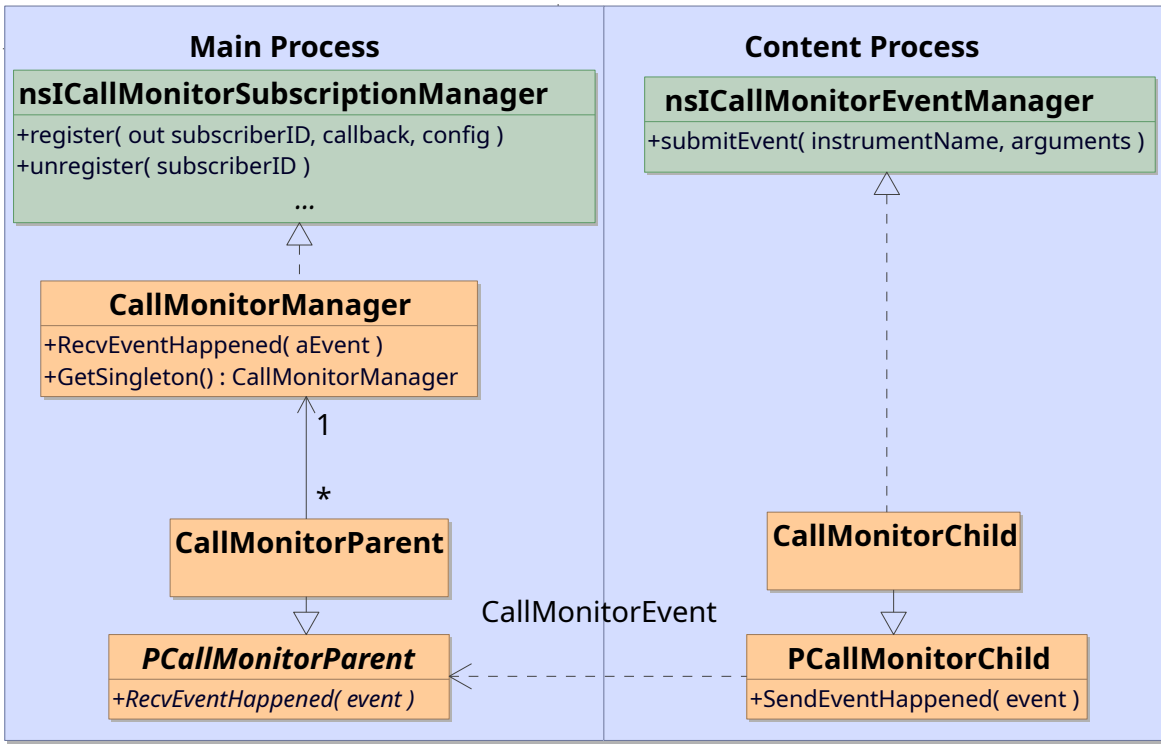

Figure 5: A UML class diagram of all C++ classes in the CallMonitor component

The CallMonitor implementation uses XPCOM in two places. On the event capturing side, `nsICallMonitorEventManager` is defined to be able to capture events in whatever language the WebAPI might be implemented in. On the consumer side, `nsICallMonitorSubscriptionManager` exposes the subscriber interface to any language that consumers want to use. Both of these interfaces will be described in more detail in the next section.

3.2.2 CallMonitor's Architecture

In this section I will present the current architecture of CallMonitor, by first outlining the high level architecture and then describing the implementation by following the process of registration and event capturing.

The current implementation is a proof of concept that only exposes its registration interface to privileged JavaScript code in the main process. It captures events for two WebAPI functions, namely `btoa` and `atob`, which en- and decode binary data to/from ASCII respectively. The captured events contain the function argument, but no information about the caller or any other context.

The code of the current implementation can be found under `https://phabricator.services.mozilla.com/D110681`.

High-Level CallMonitor has to interact with the rest of the Firefox codebase at two points. It has to collect data, when a WebAPI function call happens in a web content process and inform consumers in the main process that an event they registered for has occurred.
The interfaces for these two tasks are defined in `nsICallMonitorEventManager` and `nsICallMonitorSubscriptionManager` respectively.
The rest of the component is designed around maintaining the state or providing the connection between these two interfaces.
See Figure 5 for all classes involved.

Registration When another part of Firefox has acquired a reference to the `CallMonitorManager` through the CallMonitor service, it can register a subscriber with the CallMonitor component; the callback and its associated config are then stored in the CallMonitorManager. It then returns a unique id back to the caller, which allows the caller to unregister the callback at a later time.
In the current implementation the CallMonitorManager does not forward any information to the rest of the component, however other approaches to configuration handling get discussed Section 3.3.2.

Event Capturing When an instrumented WebAPI function gets called, it acquires a reference to the `CallMonitorChild` by asking its manager (`PContentChild`) to either create it or return an existing reference.
It then calls the `submitEvent` method, which in turn forwards the event to the Main Process by calling `SendEventHappened`, a message defined in the `PCallMonitor` protocol. When the `CallMonitorParent` receives the message, it acquires a reference to the `CallMonitorManager` and forwards the message. The `CallMonitorManager` then checks the subscribers and invokes their callback, should they have registered for the specific WebAPI function.

3.3 Design Considerations

In this subsection I will discuss the design decisions I made while developing CallMonitor. For each decision point I will motivate its relevance, present multiple options, and judge them on their merit in a proof of concept and a production-grade implementation.

3.3.1 The Layers of a WebAPI Function Call

To capture WebAPI events there has to be code that gets run on each invocation of any WebAPI. But as each call passes through multiple layers of abstractions before returning a value to the user, it is unclear at which layer our instrumentation code should exist.
In this subsection I will present the different layers a WebAPI function call passes through and discuss whether the event capturing code should be at that layer.
To evaluate in which layer the instrumentation code should be written I will consider the following criteria:

Scalability of the Approach This criterion is meant to capture how easy it is to instrument the entire current WebAPI surface but also how much additional work is required to include future WebAPIs.

Facilities for Serialization To be able to send the captured event to the main process the instrumentation needs to be able to serialize to an IPC compatible format.

Access to Information about the Caller To achieve feature parity with OpenWPM's JavaScript instrument, CallMonitor needs to be able to capture information like the iFrame, the top-level URL and the URL of the caller.

The Website's JavaScript By injecting code into the websites scope, instrumentation code is able to observe arbitrary WebAPIs calls, use the JavaScript built-in serialization facilities and capture not only the arguments but the full callstack and other relevant context. This allows instrumentation at this layer to fully meet the three requirements; however, as discussed in Section 2.2 this approach also has significant drawbacks, that CallMonitor has to avoid in order to supersede OpenWPM's JavaScript instrument.

Interpreter Layer As JavaScript is an interpreted language the next layer below the website is the interpreter (see also Figure 6).
Firefox's interpreter, Spidermonkey, provides a JavaScript Debugger object, that a Firefox developer can use as a prototype for their own object and then define their specialized handlers for certain events such as `onFrameEnter` or `onError`. One of the events a user can specify a listener for is a native function call from the interpreter.

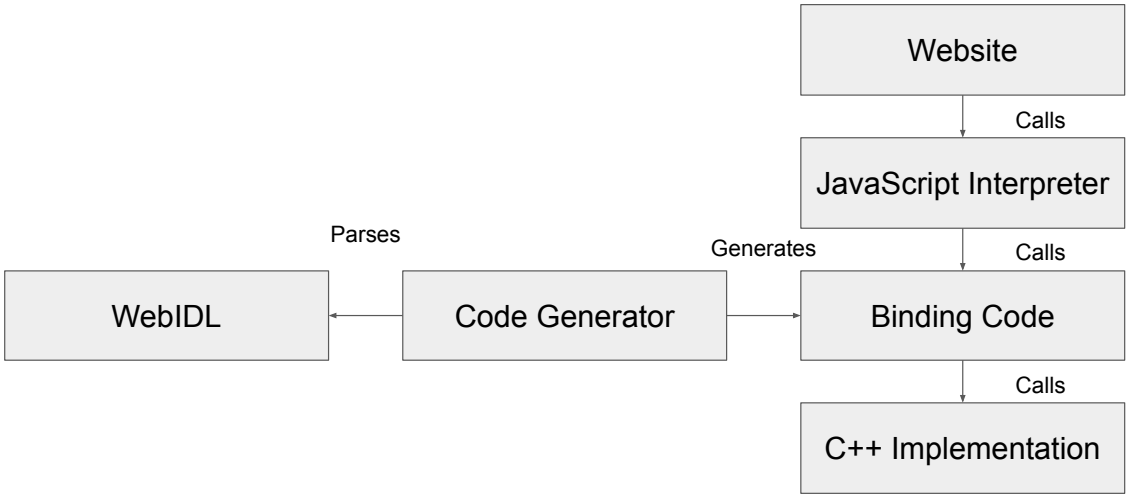

Figure 6: A schematic overview of the components involved in a single WebAPI function call. Each component of the call stack will be considered for instrumentation the Section 3.3.1

As all WebAPIs are currently implemented in C++, a `onNativeCall` listener would get invoked for every WebAPI function call. The listener receives a JSContext object when being invoked, which can be used to fully explore the interpreter's state, including the arguments to the native function call.

When evaluating this approach against the criteria set above, it seems like a good layer to inject the instrumentation, as the callback will be executed for every WebAPI function call, will have access to the JavaScript's serialization facilities through the JSContext and can also use it to capture any relevant information about the caller.

However, using the Debugger object has some challenges, that are not captured by the evaluation criteria. The first major challenge is that in order to capture native calls for a given document, the debugger object needs to be attached to it. This requires there is an appropriately configured debugger object in each web content process and that it gets attached before any JavaScript starts executing. The second challenge is, that SpiderMonkey has a just-in-time compiler (JIT), which compiles often used functions into native code. A compiled function's call to WebAPI function would no longer trigger the Debugger callback. To address the first challenge, I would need to modify the web content process creation code, which is a complex part of the codebase, so I avoided it for the proof of concept. The second challenge can not be addressed without deactivating the JIT, which is impossible for official builds of Firefox as it would result in significant performance loss. While this slowdown might be acceptable for research crawls, it still makes this option significantly less attractive.

Binding Layer One layer below the JavaScript interpreter is the generated binding code, which unwraps the arguments from JavaScript, calls the function and rewraps the result. This layer is generated by a python parser-generator from WebIDL files. To instrument at this layer, the parser-generator would need to generate code that checks if the call should be captured and if so, send all required information to the main process.

This approach is fully scalable as any WebAPI has to declare its interface in WebIDL. Since the parser-generator has access to the interface definition with full type information and argument names when generating the binding code, the generated serialization code could be written explicitly for the event that is being captured. Declaring the layout of the message at compile time might allow the compiler to emit high performance code. Information about the caller would be accessed via `SavedFrame` API, that is provided by SpiderMonkey to allow C++ code to access the current callstack. [24]

This approaches main drawback is also its implementation complexity, as the code generator is a single 23 thousand line file.

Implementation Layer Below the binding layer is the handwritten implementation of the WebAPIs. There are countless different files each implementing their distinct piece of functionality. Instrumentation at this layer requires reading the implementation, figuring out what arguments need to be part of the event and writing a serialization function by hand.

This approach does not scale, since every existing function and every future function would need to be manually adjusted. As the serialization code has to be handwritten, it is also possible to write helper functions to meaningfully serialize complex types. Information about the caller would also be accessed via `SavedFrame`.

In contrast to poor scalability, this approach minimizes the effort to instrument a given arbitrary API call, which makes it ideal for prototyping.

Conclusion I believe the ideal layer for inserting the instrumentation is the binding layer as the types and names present in the WebIDL definitions would allow generating serialization code that captures the maximum amount of information possible and achieve high performance by statically defining the serialization scheme; the complexity of modifying the code generator and the generated binding code make it inappropriate for a proof of concept implementation, so I chose the two functions and implemented them in the handwritten code.

3.3.2 Configuration Management

To achieve feature parity with the JavaScript instrument, CallMonitor needs to allow users to dynamically specify which WebAPIs they are interested in. Since polling is generally discouraged in Firefox, as it always introduces a trade off between resource overhead, when choosing a small polling interval, and latency, when choosing a big polling interval, the interface had to be callback based.

As callbacks are a common pattern in the codebase, there also exist conventions around how such a callback based interface should be designed. The general pattern is, that the consumer of the API should be able to register their callback and the associated metadata (in this case the configuration) and in return receive a unique identifier, that allows them to refer to this registration.

This design allows for multiple consumers to register themselves with the same API without interfering with one another, but it also complicates the internal state of the CallMonitor compared to the JavaScript instrument. Instead of baking in one configuration for a single consumer on startup, CallMonitor needs to maintain a mapping between identifiers and their associated callback-metadata pair, which can change at runtime, by consumers registering and unregistering themselves.

This left the following options:

Hard-coded Config As previously established, the WebAPIs get called in a different process than the one the registration is happening in. Specifying a fixed set of configurations at compile time and allowing the user to only register their callbacks for one of these configs, would sidestep the issue of synchronizing the internal state across multiple processes, especially since new web content processes get created for each new domain visited.

However, this would be a degradation in user experience compared to the JavaScript as compiling Firefox is both too resource intensive and too complex for privacy researchers, when they want to instrument a custom set of WebAPIs.

As there still would be no communication to the web content processes which configs currently have subscribers, they would still have to send all messages for all configs. A slight variation on this approach would be to require the callback code to also be written at compile time, to avoid the last issue mentioned.

While this approach would have been the easiest to implement, it breaks one of the core requirements, so it was discarded.

Centralized Config The approach I choose for the prototype was to keep the configs and callbacks exclusively in the `CallMonitorParent` and not share it out to web content processes.

This approach allows exposing an API that could achieve feature parity with the JavaScript instrument, while keeping the implementation complexity low.

As seen in Figure 7 the trade off for this approach is that all events get send to the main process, where the filtering happens. As PCallMonitor is currently a managed

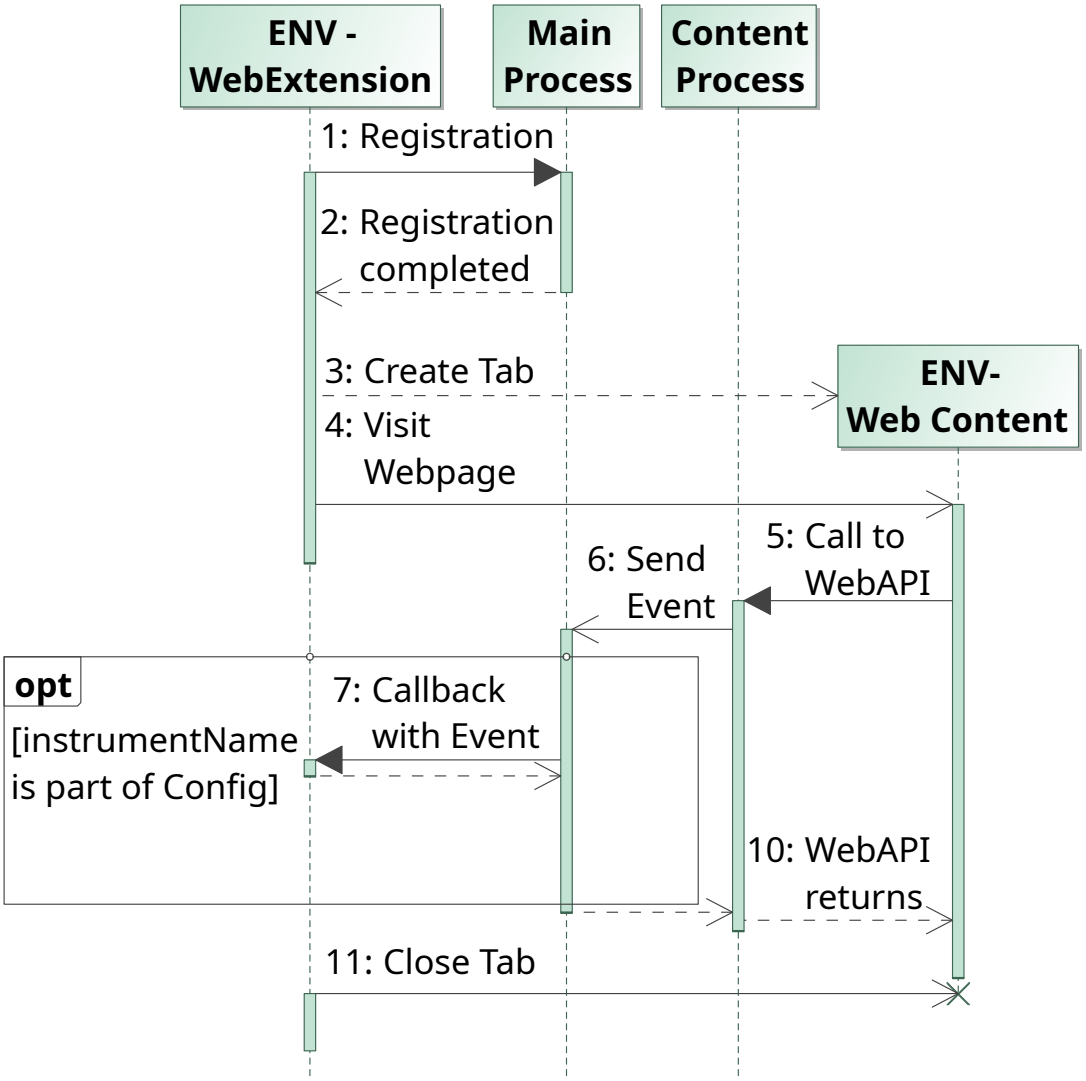

Figure 7: An UML sequence diagram showing a single consumer registering with CallMonitor and then visiting a web page; the page makes a single WebAPI request, which gets unconditionally propagated to the main process due to the centralized config approach

protocol, that might negatively impact the communication between the web content process and the main process, but as the proof of concept implementation only instrumented a rarely used functions, there was no such impact.

Distributed Config A production component should not make unnecessary IPC calls especially since a production-ready CallMonitor component would need to cover all WebAPIs, which means a single traversal of the DOM could generate hundreds if not thousands of events. As such web content processes must be able to decide locally whether an event needs to be sent to the main process.

To be able to make this decision, each content process would need to have at least the union of all active configs, but possibly better, a mapping from registration token to config, so it can send a pair of event and list of registration tokens to call via the `EventHappened` message.

Figure 8 shows the added round trip of propagating the config to an existing web content process when a new consumer registers itself as well as the added benefit of being able to make a local decision in the content process.

What the figure hides is the complexity of keeping all web content processes up to date. To achieve this, the main process would need to send the content process the current configuration when the process starts and then incrementally update the config every time it changes in the main process. As running any web content in a process without having the configuration initialize would result in missing events, CallMonitor would require synchronous IPC in the process creation. However, synchronous IPC is discouraged to the point that Firefox developers need to convince the IPC to allowlist their call in a special file.

Despite these implementation complexities, I still believe this is the right approach for a high throughput system.

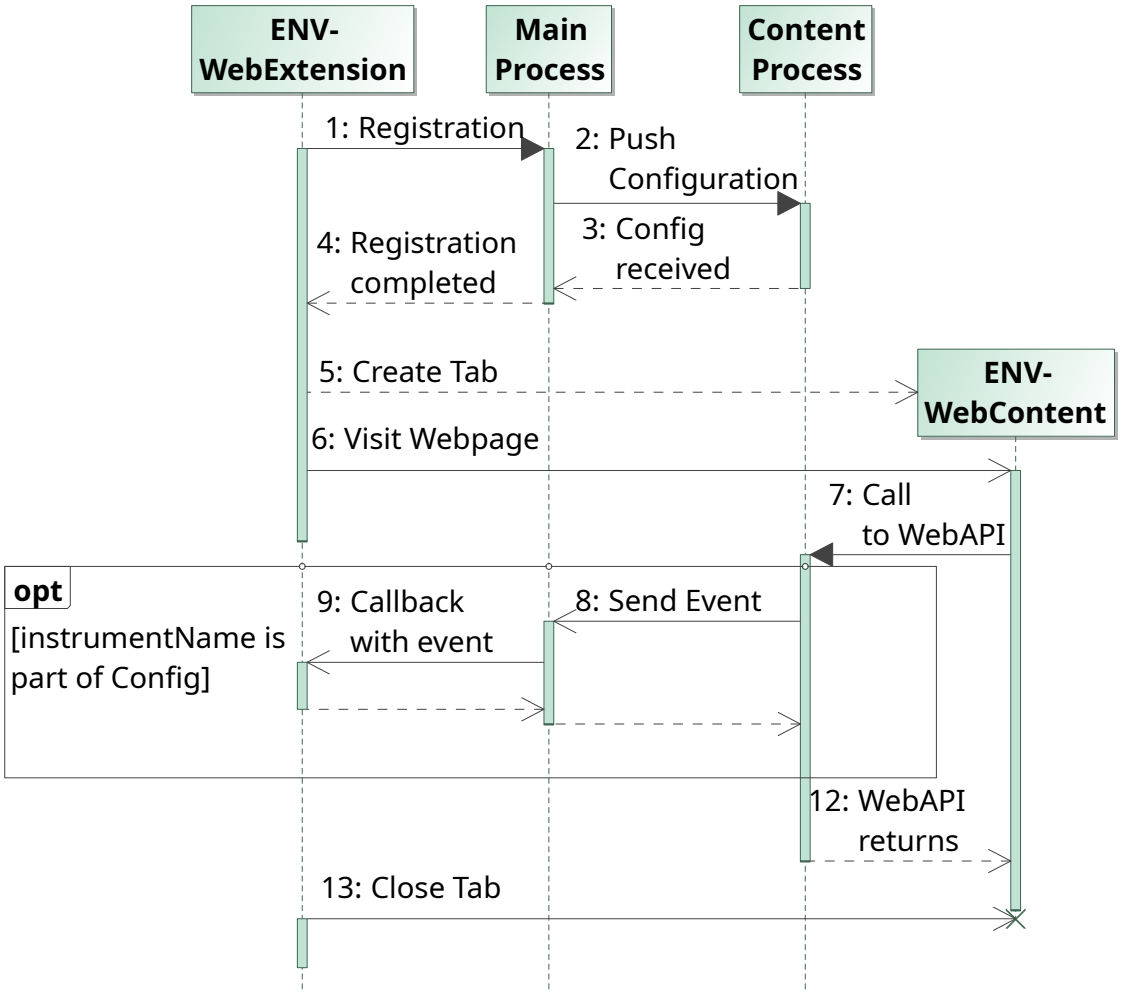

Figure 8: A UML sequence diagram showing a single consumer registering with CallMonitor and then visiting a web page; the page makes a single WebAPI request, which gets checked in the content process and only propagated if it needs to be due to the centralized config approach

3.3.3 Testing

While different parts of the Firefox code base use a variety of testing frameworks, such as GoogleTest [1], the native Rust framework [2] or xpcshell, a Mozilla internal framework to test XPCOM components [25], the majority of tests use mochitest, a JavaScript regression testing framework, that allows testing all scriptable components of the Firefox codebase [26].

Most mochitest, including mine, perform function testing as defined by Glenford J. Myers [27], as they do not test the module against the pure interface specification but rather test, that the component under test meets the external specification. This is necessary, as there are very few components which exist isolated in the Firefox code base. Instead, it is common, that to perform a single task a component has to interact with countless other systems.

All mochitest start a full browser which is then controlled via JavaScript running in different contexts. These different execution contexts are called flavors. There are four flavors of mochitest [28]:

- a11y — a11y or accessibility, is the practice of ensuring access to people with disabilities. All tests ensuring that the browser is usable for these people have the a11y flavor.
- plain — These tests run in content scope and only have access to a limited numbers of privileged APIs, through the `SpecialPowers` object.
- browser — Tests, that aim to test the browsers UI should use this flavor. They run in privileged scope.
- chrome — These tests are also run in a privileged mode, but aim to test the privileged APIs.

Originally, I assumed that the xpcshell framework would be sufficient for testing CallMonitor, however as xpcshell tests don't create a full browser but instead only initialize the XPCOM infrastructure, there was no way to submit events, as the necessary interface is only exposed in the content process, which does not get created as part of these tests.

Tests should only use as much enhanced access as required to make sure, the real use case and the test use the same code paths. As such, I evaluated the different flavors of mochitest against my testing requirements. The a11y flavor was immediately discarded as CallMonitor is not an accessibility feature. Plain mochitest run in the content process, which is insuffiecient to test the component as being able to receive events in the main process is it core purpose. Finally, as CallMonitor does not expose any UI, using the browser flavor was also out of question.

This left the chrome flavor, which is a fit, as CallMonitor exposes a privileged API in the main process.

So the test for the proof of concept works as following

[1] https://github.com/google/googletest

[2] https://doc.rust-lang.org/book/ch11-00-testing.html

1. It registers with the `CallMonitorManager` through the `CallMonitorSubscriptionManager` interface
2. It creates a new tab and points it to an HTML file that calls `atob`
3. It waits until it get notified, that the WebAPI function call has happened
4. It assert that the event contains the expected argument string

This setup tests the entirety of the current implementation. For a more complete implementation, that also provides the WebExtension API, there would have to be an additional test, which would need to register as an addon and use the API exposed there.

While this approach works very well for a proof of concept, it would have significant problems at scale, as creating a new test file for each WebAPI function wouldn't scale.

Testing every single instrumented API might not be required, once each API instrumentation is no longer handwritten, since a single error in the generated code would affect all paths.

However, to test a representative number of APIs to ensure, things are working as expected at scale, it could be reasonable to set up a channel between the web content and priviliged test context and pass a string to it, on which the web content could then call `eval`, which to the instrumentation would be indistinguishable from having the code be written in the file beforehand, except for perhaps the callstack. As there are only two functions instrumented in the current implementation, this approach seemed overly complex.

4 Evaluation

To evaluate the quality of the current implementation of CallMonitor, I visited the same website with CallMonitor and OpenWPM's JavaScript instrument and compared the collected data.

4.1 Setup

To compare the two instrumentation approaches I chose a configuration of the JavaScript instrument that included the WebAPIs currently instrumented by CallMonitor.

```
[
  {"window": {"propertiesToInstrument": ["atob", "btoa"]}},
   "screen"
]
```

This instructs the instrument to record all calls to `btoa` and `atob` and any access to properties of `screen`.
As for the CallMonitor configuration, I chose the following config to instrument all currently instrumented WebAPIs.

```
{items:[
{apiName: "Atob"},
{apiName: "Btoa"}
]}
```

4.2 Test case

To compare the two instrumentation approaches I created a website that calls the instrumented WebAPIs. The relevant code of this website I can be seen in Figure 9.
The code calls the instrumented `atob` and `btoa` APIs and the `screen` APIs which are currently not instrumented by CallMonitor. The latter are called to demonstrate the lack of generality in the current proof of concept.
When visiting the website with a Firefox that was compiled with the CallMonitor component, a subscriber that has the config shown above, collects the events shown in Table 1.
As I visited the website with OpenWPM and the presented configuration, the JavaScript instrument recorded its events into the SQLite database. A relevant subset of columns is shown for all events in Table 2. Data about the extensions, the tab, window and frame id, as well as the URLs of the script, the top level site and the current iFrame have been omitted for brevity. For a full event see Figure 3

```
1  function func(data) {
2    let encoded_data = window.btoa(data);
3    let decoded_data = window.atob(encoded_data);
4  }
5  function main() {
6    func("abcd");
7    let function_string = window.atob.toString();
8    if (function_string.includes("[native code]")) {
9      func("efgh");
10   }
11   document.getElementById(
12     "output"
13   ).innerText = `The screens coordinates are ${window.screen.width}
     x ${window.screen.height}`;
14 }
```

Figure 9: The JavaScript code of the test website. It calls en- and decodes a string, then checks if it can detect any tampering with the `atob` function and if it can detect no tampering performs a second encoding-decoding roundtrip with a different string. Finally, it prints the screen size in the output element on the site.

Records	0	1	2	3
apiName	Btoa	Atob	Btoa	Atob
arguments	abcd	YWJjZA==	efgh	ZWZnaA==

Table 1: The WebAPI name and the arguments captured by CallMonitor when visiting the test site

Records	0	1	2	3
script-line	6	7	20	20
script-col	33	33	52	77
func-name	func	func	main	main
symbol	window.btoa	window.atob	window.screen.width	window.screen.height
operation	call	call	get	get
value			1280	720
arguments	["abcd"]	["YWJjZA=="]		

Table 2: A subset of the columns recorded by the JavaScript instrument when visiting the test site. A notable omission are the transformations on "efgh" that were recorded by CallMonitor

4.3 Analysis

The collected data shows, that the website was able to evade the JavaScript instrument as Table 2 does not show the en- and decoding of "efgh". CallMonitor was able to record the calls, as the website didn't have a way to detect it. However, the JavaScript instrument is still more usable, as it is not limited in the APIs it can instrument and captures much richer events. While most of these data points could be recorded through the `SavedFrames` facilities as discussed in Section 3.3.1, this section compares the two instruments as they currently are and does not take future possibilities into account.

5 Outlook and Further Work

In this section I will evaluate the CallMonitor against the requirements I gave myself in Section 3.1 and against the JavaScript instrumentation as it exists in OpenWPM today.
To recap those requirements were:

- Capture the same information as the JavaScript instrument
- Being undetectable
- Follow the official Mozilla coding conventions
- Be easy to maintain, test and extend

As discussed in Section 4 the proof of concept implementation falls short on recording the same information as the JavaScript instrument, but a fully developed CallMonitor could capture the same or more information, as discussed in Section 3.3.1.
CallMonitor is undetectable by the methods used to identify the JavaScript instrument and should also be hard to detect in general as there is no change to the website's execution environment. This includes other common evasion techniques such as abusing the interaction of dynamically created iFrames and WebExtensions content script [29].
CallMonitor is following the coding conventions in the Firefox code base, it follows the official naming scheme for XPCOM interfaces, actor protocol definitions and method names. It uses the custom string and container types provided by the code base. The existing test uses the most common testing framework and respects established best practices regarding the level of testing and event handling in test code.
As discussed in Section 3.3.3 CallMonitor's test can be easily extended to cover any enhancements made upon the current proof of concept.
I also discussed in Section 3.3.2 and Section 3.3.1 what enhancements would be required to achieve a production-ready state. None of these enhancements would require a fundamental change of the architecture instead they could be progressively and independently applied. This shows that the current implementation is easy to extend.
Regarding the maintainability, I don't think there is a good metric other than that I kept the code as simple as the problem would allow, which should make it easy to maintain.

5.1 Further work

To summarize the findings from these subsections and the evaluation, the following work would need to be done to achieve a full replacement.

1. Collect data in binding layer using the WebIDL code generator
2. Push the CallMonitor configuration to the content processes
3. Use the `SavedFrame` API to capture information about the caller

While all of these issues are easily expressed succinctly, their respective implementations would probably each be another 3-5 month project. So to stay within the limits of a bachelor's thesis I constrained myself to implementing a proof of concept and only explore the full solution theoretically.

5.2 Outlook

Once these improvements have been made, CallMonitor is ready to replace the JavaScript instrument as part of OpenWPM. This would allow for more robust and less evadable privacy research and possibly allow to instrument a wider range of WebAPI function calls as is currently possible due to performance constraints. However, assuming it becomes part of the official Firefox codebase, the component could also open up a series of interesting new opportunities for browser developers. They would be able to instrument their browser dynamically, possibly even on users' machines through telemetry, to detect how WebAPIs are used in the wild. It would also be possible to integrate this tooling with the browser's developer tools, to allow web developers, to observe what WebAPI function calls their own website makes.

6 Sources

References

[1] P. Laperdrix, W. Rudametkin, and B. Baudry. “Beauty and the Beast: Diverting Modern Web Browsers to Build Unique Browser Fingerprints”. In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 878–894 (cit. on pp. 5, 7).

[2] WHATWG. *Introduction to "The DOM"*. Feb. 2022. URL: https://dom.spec.whatwg.org/#introduction-to-the-dom (cit. on p. 6).

[3] V. Beal. *Web Browser*. Aug. 2021. URL: https://www.webopedia.com/definitions/browser/ (cit. on p. 6).

[4] ECMA International. *ECMAScript® 2022 Language Specification*. Jan. 2022. URL: https://tc39.es/ecma262/ (cit. on p. 6).

[5] Mozilla and individual contributors. *Inheritance and the prototype chain*. Jan. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain (cit. on p. 6).

[6] W3C. *JavaScript Web APIs*. Jan. 2022. URL: https://www.w3.org/standards/webdesign/script.html (cit. on p. 6).

[7] Mozilla and individual contributors. *Document Object Model (DOM)*. Jan. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model (cit. on p. 7).

[8] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. “On the Incoherencies in Web Browser Access Control Policies”. In: *2010 IEEE Symposium on Security and Privacy*. 2010, pp. 463–478 (cit. on p. 8).

[9] S. Arshad, A. Kharraz, and W. Robertson. “Include me out: In-browser detection of malicious third-party content inclusions”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2016, pp. 441–459 (cit. on p. 8).

[10] DuckDuckGo. *DuckDuckGo Tracker Radar Collector*. Jan. 2022. URL: https://github.com/duckduckgo/tracker-radar-collector (cit. on p. 8).

[11] T. K. Panum, R. R. Hansen, and J. M. Pedersen. “Kraaler: A user-perspective web crawler”. In: *2019 Network Traffic Measurement and Analysis Conference (TMA)*. IEEE. 2019, pp. 153–160 (cit. on p. 8).

[12] S. Englehardt and A. Narayanan. “Online tracking: A 1-million-site measurement and analysis”. In: *Proceedings of ACM CCS 2016*. 2016 (cit. on pp. 8, 10).

[13] WebTAP - Princeton University. *Software: OpenWPM*. Feb. 2022. URL: https://webtap.princeton.edu/software/ (cit. on p. 10).

[14] Mozilla and individual contributors. *Browser Extensions*. Jan. 2022. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions (cit. on p. 10).

[15] Mozilla and individual contributors. *Anatomy of an extension*. Nov. 2021. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Anatomy_of_a_WebExtension (cit. on p. 11).

[16] Mozilla and individual contributors. *X Ray Vision*. Jan. 2022. URL: https://firefox-source-docs.mozilla.org/dom/scriptSecurity/xray_vision.html (cit. on p. 11).

[17] Mozilla and individual contributors. *Error.prototype.stack*. Jan. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error/Stack (cit. on p. 12).

[18] Mozilla and individual contributors. *<iframe>: The Inline Frame element*. Jan. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe (cit. on p. 14).

[19] Chromium Contributors. *Process Models*. Feb. 2022. URL: https://www.chromium.org/developers/design-documents/process-models/ (cit. on p. 16).

[20] WebKit Contributors. *WebKit2 - High Level Document*. June 15, 2016. URL: https://trac.webkit.org/wiki/WebKit2 (cit. on p. 16).

[21] Mozilla and individual contributors. *Process Model*. Jan. 2022. URL: https://firefox-source-docs.mozilla.org/dom/ipc/process_model.html (cit. on p. 17).

[22] WHATWG. *Web IDL*. Jan. 2022. URL: https://webidl.spec.whatwg.org/#introduction (cit. on p. 17).

[23] W. D. Clinger. "Foundations of actor semantics". In: *AITR-633* (1981) (cit. on p. 17).

[24] Mozilla. *SavedFrame*. Sept. 2021. URL: https://firefox-source-docs.mozilla.org/js/SavedFrame/index.html (cit. on p. 23).

[25] Mozilla and individual contributors. *XPCShell tests*. Jan. 2022. URL: https://firefox-source-docs.mozilla.org/testing/xpcshell/index.html (cit. on p. 28).

[26] Mozilla and individual contributors. *Mochitest*. Dec. 2020. URL: https://github.com/mdn/archived-content/blob/main/files/en-us/mozilla/projects/mochitest/index.html (cit. on p. 28).

[27] G. J. Myers. *The art of software testing / Glenford J. Myers, Corey Sandler, Tom Badgett*. eng. 3rd ed. Hoboken, N.J.: John Wiley & Sons, 2012, pp. 116–117 (cit. on p. 28).

[28] Mozilla. *Mochitest Flavors Explanation*. Sept. 2021. URL: https://github.com/mdn/archived-content/blob/0b3ab5eeab845edc099eaf0542629951cc061222/files/en-us/mozilla/projects/mochitest/index.html#L54-L61 (cit. on p. 28).

[29] gorhill. *Cosmetic filter for ad in dynamically created iframe*. June 2016. URL: https://github.com/gorhill/uBlock/issues/1740#issuecomment-227138235 (cit. on p. 33).

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den May 5, 2022 .....