Analysis of Methods for Background Execution in Modern Web Applications

Analyse von Verfahren für Hintergrundausführung in modernen Webanwendungen
Bachelor-Thesis von Yannick Reifschneider
Tag der Einreichung:

1. Gutachten: Nikolay Matyunin
2. Gutachten: Prof. Dr. Stefan Katzenbeisser

TECHNISCHE
UNIVERSITÄT
DARMSTADT

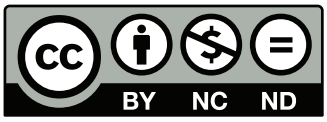
Fachbereich Informatik
Security Engineering

Analysis of Methods for Background Execution in Modern Web Applications
Analyse von Verfahren für Hintergrundausführung in modernen Webanwendungen

Vorgelegte Bachelor-Thesis von Yannick Reifschneider

1. Gutachten: Nikolay Matyunin
2. Gutachten: Prof. Dr. Stefan Katzenbeisser

Tag der Einreichung:



Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den August 25, 2019

_____

(Yannick Reifschneider)

Contents

1 Abstract

The application development industry shifts towards not developing native applications but instead use the browser as a universally available user interface platform. Web applications are indeed now capable of almost any task which would require a native application in the recent past. Popular word processing or spreadsheet applications for example are implemented with web technologies. A web site also has a completely different attack surface for a malicious actor then native applications. Browser engines do their best to protect their users of malicous web sites and also conserve energy.

In this paper we analyse major browsers regarding their behaviour of JavaScript code exection, when the execution page is in the background and not visible to the user. We show that the energy conserving methods of desktop browsers can easily be circumvented to do arbitrary calculations for unlimited time while the web site is not user visible. With these findings we trace popular websites to see if they use similar methods to execute uninterrupted in the background.

2 Introduction

Web browsers are a fundamental application on many computing devices. They also have to be trusted by the user of the device, because they run application code of any visited website.

Because browsers are so heavily used on many devices, they are encouraged to conserve battery and only use computing time, when they are really neccessary for the visited web page.

Eventually all tabs in the background should be halted completely and only woken up, when the page is moved to the foreground again. Due to the very different kind of applications which can be built using web technologies, not all tabs can be halted completely without breaking the application.

Many popular web sites earn money by showing ads on their page. These ads can in many cases also include JavaScript code, which then is executed on every visitor of this page.

There are also implementations of crypto currency miners implemented in JavaScript. These mining scripts are programmed to earn money for the website owner at the cost of the CPU and energy usage of the website visitors.

2.1 Motivation

- Possible side channel attack (i.e. via sensor readings)
- A XSS vulnerability in a popular website could use the visitors as a botnet for a DDOS attack

2.2 Related works

3 Background

JavaScript is the only programming language supported by all modern web browsers to enhance web sites or web applocications. If you compare JavaScript to other more traditional programming languages like C, C++ or Java you see some major differences between them.

- JavaScript has no functions for I/O like writing to a file or opening a socket. It depends on a runtime environment like a browser to provide I/O functionality.
- JavaScript has no concept of parallelism like operating system threads or processes.
- JavaScript is an event driven language and has a built in event loop.

All of these differences stem from the fact, that JavaScript was designed as an addition to HTML for enhancing web applications and the interface to the browser internal workings, such as the DOM. Many of these design decisions are choosen, because JavaScript is run in the browser and therefore can be run on any website you visit. This poses many security implications, because you may not want to give a website you visit the power to execute arbitrary programms on your computer.

Also explain why webassembly does nothing special in regards to background execution. That is, because it allows seamleas interop between webasm and Javascript functions. Therefore it has to run with the same runtime guarantees as JavaScript. WebASM Threads are based on Webworkers with a shared memory pool. WebASM therefore behaves exactly like javascript functions and have to yield to the event loop to not block the browser.

3.1 The JavaScript execution model

The JavaScript runtime uses an event loop to schedule tasks for execution[1]. The browser or other JavaScript code can put tasks on the task queue. During a single run of the event loop, the runtime removes the first item of the task queue and executes it. When the event queue is empty, the run loop waits until an event is placed into the queue.

When a JavaScript task is running, it is guaranteed to not be interrupted until it is completed. This insures, that a variable cannot change from outside. This is in contrast to many other programming languages like Java or C, where another thread can mutate variables at any time. To guarantee this behaviour, the JavaScript runtime in the web browser has to block and wait until the task finishes, because the JavaScript task has access to the browser internal state and DOM. Once the task is finished, the runtime can perform other work, such as layout calculations, which mutates its internal state. Due to these properties JavaScript tasks should be as small as possible to ensure responsiveness of the browser. If a JavaScript task takes a long time to complete (i.e. it is executing an infinite loop) most browser show a warning to the user that a script is slowing down the website. The user then has the option to kill the task or wait for the task to finish.

3.2 Web workers

Web workers are a mechanism to perform long running uninterruptable calculations. Due to the runtime guarantees which were described in the last section, Web workers have a completely seperate execution context. They don't share variables or resources with the invoking context.

The web worker and main thread context can communicate by passing messages to each other, which are handled by their respective event loops. In contrast to the main thread, which can manipulate the DOM of the browser, the Web worker has limited functionality. This limitation allows the web worker to run uninterrupted for longer times, because it does not does not block browser events.

4 Analysis of different background execution methods

4.1 Methods

4.1.1 Timers

Standard method for scheduling a recurring function in JavaScript. The setInterval function allows to specify a function and an interval in milliseconds after which a function is repeatedly called until the interval is cancelled.

4.1.2 Web workers

Using the worker-timers[1] library to run a scheduler on a web worker, which calls a callback on the main loop. This circumenvents the setInterval throttling, when a browser tab is in the background.

4.1.3 Timers with open WebSocket connection

In some browsers window timers in background tabs are not throttled, when either audible music is playing or a websocket connection is open. Playing a sound file or opening a websocket connection work equaliy in preventing the throttling, but they differ hugely in how visible both are the user of the web page. While playing audio is audible and also marks your tab with a speaker icon, so the user can quickly find out, which tab is producing the sounds, opening a websocket connection is invisible to the user. Also many browser prevent automatic playback of audio and videos, because it is considered bad behaviour of the website to autoplay audible sounds.

4.1.4 Service workers

Service workers have advantages. They run independent of the browser tab. They stay can stay aliver after the browser tab, which installed the service worker is closed.

- Multiple methods for background execution:
- Simple set interval after activation
- In response to network request (corresponding website has to be open to trigger a network call)
- Website push notifications (has to be allowed by user)
- Web Background Synchronization API[2]

[1] https://github.com/chrisguttandin/worker-timers

[2] https://wicg.github.io/BackgroundSync/spec/

4.2 Browser behaviour

Firefox budget-based throttling https://developer.mozilla.org/en-US/docs/Web/API/Page_Visibility_API#Policies_in_place_to_aid_background_page_performance

Chrome budget-based throttling https://developers.google.com/web/updates/2017/03/background_tabs#budget-based_background_timer_throttling

Safari throttling

Chrome Mobile

Mobile Firefox

Android Browser

Mobile Safari

Samsung Browser

4.3 Desktop web browsers

4.3.1 Google Chrome

Chrome on macOS does not allow sensor readings while in background.

AmbientLightSensor has to be enabled via flags.

4.3.2 Mozilla Firefox

Firefox does not support the Sensors API, but implements an older specification of the ambient light sensor API. This older API does not allow to specify a frequency in which the event handler is called. Also, this API is no longer enabled by default since Firefox 60 due to privacy concerns. Also Firefox does not call the event handlers, when the tab is in the background

4.3.3 Apple Safari

4.4 Mobile web browsers

On iOS we only analyse Mobile Safari, because Apple does not allow other browser engines in the Apple AppStore. Every other browser app has to use the system-provided webview to be in accordance with § 2.5.6 from Apple Review Guidelines[3].

On Android, we can differentiate between different browser engines.

4.4.1 iOS Mobile Safari

4.4.2 Chrome for Android

4.4.3 Firefox for Android

[3] https://developer.apple.com/app-store/review/guidelines/#software-requirements

5 Tracing of background execution on popular websites

In the first part of this thesis, we identified different methods to circumvent the default browser throttling mechanisms. With these findings in mind we can now trace popular websites[4] to analyse if these circumenvtion methods are used in the wild. Using the Alexa Top 1 million website list, we measure the average CPU usage of the websites. When we aggregate these tracing result, we can determine, if the browser throttling mechanisms are suiteable to limit CPU usage or if websites use these methods to actively or inadvertently bypass the background throttling mechanisms.

5.1 Automated measuring of background execution

To automate the tracing of popular websites we used the Puppeteer[2] library to control and automate Google Chrome. We choose Puppeteer because it allows us to access the Chrome internals like Web Profiler and also inject custom scripts into the loaded page to detect if circumvention methods which are described in part one of this thesis are being used. Puppeteer only allows to control Google Chrome, but with Google Chrome having the greatest market share among common Web browsers this is not a limiting factor.

The general idea for the tracing is, that we open a new Chrome instance and start the web profiling. Then we open the website to trace and wait for the initial load to complete. After that we move this website to the background by opening a new tab, so that the browser throttling mechanisms are activated. We profile the page for a 15 minutes time period. After the 15 minutes are completed, we close the browser instance and save the website trace for analysis.

The website trace allows us to understand if the website is running JavaScript code while it is in the background and which methods it used to initiate the execution. On a invidual site level the website trace profile allows us to see, which JavaScript functions were run and how long they took to complete.

The website trace profile allows us to dig deep into what a single website is executing while it is in the background. But to get a better picture of how popular websites in general behave in the background we have find a way to aggregate the analysis of the traces.

We propose to use the average CPU usage during the profiling to use as a score for a single website. The average CPU score can be calculated from the trace file by calculating summed duration, in which the website executed JavaScript code in the main thread and in spawned worker threads and dividing the sum by the time the trace was running. Wall clock time is a suitable replacement for CPU time in a scenario where the machine on which the measurements are taken is not under any other load from other processes. A website which uses no JavaScript at all should therefore receive a score of 0, whereas a website which does run uninterupted calculations on the main thread should receive a score of 1. If a website uses multiple worker threads and the machine on which the measurements are taken has more then one physical CPU core, then the website could receive a score which is greater then 1, because all workers could be active at the same time.

The background throttling mechanism put in place by Google Chrome does limit the timers of websites based on a CPU time budget. This budget is currently set so that websites on average only use 1% of CPU usage. That would equal a score of 0.01 in our proposed metric. This limit gives a good estimation, if websites try to overcome the background throttling mechanism of Chrome. Websites which score greater then 0.01 presumably use one or more methods to prevent the throttling.

[4] We used the first one thousand web sites of the Alexa Top 1m Website list

6 Evaluation of tracing results

7 Conclusion

8 References

[1] *Concurrency model and Event Loop*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop.

[2] *Puppeteer*. URL: https://pptr.dev/.