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

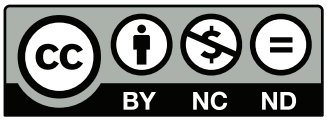
Fachbereich Informatik
Security Engineering

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 3, 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 information

3.1 The JavaScript execution model

The JavaScript language has no functions to perform I/O operations on it's own. It needs a runtime to perform the I/O task. All calls to the runtime are non-blocking.

JavaScripts concurrency model is based on events and callbacks.

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 web browser has to block and wait until the task finishes. Once the task is finished, the runtime can change and perform other work. Due to this properties JavaScript tasks should be as small as possible to ensure responsiveness of the browser. If a JavaScript task never completes (i.e. it is executing an inifite 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 longer.

Why a simple infinite loop is not feasable: Web page becomes unresponsive. If you don't yield back to the event loop, you can no longer react to changes in the environment, for example to detect, if the browser is now in the foreground again. This is at least true in the main loop, have to check for service worker or web worker context.

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.

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. But this limitation allows the web worker to run uninterrupted for long times. It does share resources with the browser renderer, and therefore does not block repaints or other browser events.

4 Analysis of different background execution methods

4.1 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.

4.2 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.

4.3 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.

4.4 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/

4.5 Desktop web browsers

4.5.1 Google Chrome

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

AmbientLightSensor has to be enabled via flags.

4.5.2 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

4.5.3 Apple Safari

4.6 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.

4.6.1 iOS Mobile Safari

4.6.2 Chrome for Android

4.6.3 Firefox for Android

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

5 Tracing of background execution on popular websites

Using the Alexa Top 100 Website list.

5.1 Method for measuring background execution

Maybe with puppeteer[4], web developer tools or with OpenWPM[5] or with simple hooking the JavaScript functions

[4] https://pptr.dev/

[5] https://github.com/mozilla/OpenWPM

6 Evaluation of tracing results

7 Conclusion