Analysis of Methods for Background Execution in Modern Web Applications

Analyse von Verfahren für Hintergrundausführung in modernen Webanwendungen
Bachelor thesis in Computer Science by Yannick Reifschneider
Date of submission: September 18, 2019

1. Review: Nikolay Matyunin
2. Review: Prof. Dr. Stefan Katzenbeisser
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science
Department
Security Engineering

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

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

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, den 18. September 2019

Y. Reifschneider

Contents

1 Abstract

In this thesis we analyse modern browsers regarding their behaviour of JavaScript code exection in background tabs. 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 technology received rapid advances in the last years. Many websites not only deliver static information in form of text and images, but are complete highly dynamic applications which compete with traditional software products. For many of the standard business applications like an E-mail client, a word processor or even a bitmap graphic manipulation software there are alterantives[1][2][3], which are developed using web technologies and used via a modern web browser. The web browser has grown from a renderer of static information to the operating system of web applications. Some reasons for the rise of these web applications are, that browsers are getting more capable every year and the JavaScript performance is increasing drastically so that these applications are possible in the first place. Developing a web application instead of traditional software has also many benefits for the developer of said software. Web apps don't need installation, just a modern web browser, which comes preinstalled with any common operating system and mobile device. Visiting a web site to try out a piece of software is a much smaller hurdle to overcome then downloading, installing and running a piece of software. These reasons make the web a popular platform for developing new applications.

At the same time, the revenue model for many websites is advertising. Websites include third party JavaScript from ad networks, which display ads and track impressions. The included code can be controlled by the advertiser. For a malicous entity, it is therefore possible to include JavaScript on reputable websites just by paying for the ad impression. This method of distribution for malicous script is known as malvertising[26].

The web browser is a fundamental application on most personal computing devices and is trusted by it's users. Because many modern applications are implemented as web apps, web browser usage is ubiquitous in the usage of computing devices. As such, many users leave the web browser open during the whole time they use their device, often times

[1] `https://www.google.com/gmail/about/`
[2] `https://products.office.com/en-us/free-office-online-for-the-web`
[3] `https://www.photopea.com/`

with many website tabs open. The browser vendors are encouraged to conserve as much energy as possible to enable users on battery powered devices, such as mobile phones or notebooks which are not tethered to a power outlet, a longer usage time. On the other side, the browser should not break web applications which require regular CPU time to function correctly. Web apps, which need regular CPU time are sites, which notify the user about new content such as a news site or a social network, a web application which plays audio or video, or a web video conference application. All modern browsers limit the amount of CPU time a tab in the background can use and how often web sites can schedule new work. For maximum energy efficiency, all tabs in the background should eventually be halted completely and only woken up, when the page is moved to the foreground again. This is also the goal of browser vendors[3], but the execution of this goal is easy without breaking existing web applications or providing a workaround for some applications.

In this thesis we find out, if the throttling mechanisms employed by browser vendors can be circumvented, so that web sites which are not visible to the browser can gain code execution for arbitrary time. We compare the different throttling mechanisms between desktop and mobile browsers and we trace popular websites to see if the found cirumvention methods are actively used in the wild.

2.1 Motivation

The JavaScript language was designed as an extension to HTML and CSS in the early days of the world wide web, with the goal to enable user interaction with web pages. Although JavaScript is nowadays used for much more then just simple event handlers for user interactions, the basic principle of the web did not change: When a user visits a web site, the browser downloads all resources belonging to this web site, like HTML, images and also scripts. These scripts are run, to build up the complete website. When users visit a web page and do not disable JavaScript completly, they implicitly allow this web site to execute arbitrary code on their machine. This implicit trust is also a major design factor for HTML5 APIs. The goal of the HTML5 API designers is, to allow for new kind of applications, without allowing a web application too much control over the machine it is running on. Nevertheless there are numerous activities, which malicous entities could perform with access to a web site with high amount of daily visits.

- The attacker could inject a browser crypto-currency miner to convert electricity of the website visitors into crypto currency. The existance of crypto currencies such as Monero, which uses proof of work algorithm, which is inefficient to calculate

on custom hardware, makes the browser a viable target for such mining. Existing implementations of Monero miners implemented in web assemlby are available for this attack.

- The browser could be used as a compute node in a botnet, which can be used for DDoS attacks or for cracking hashed passwords. Grossman et al.[12] show that these attacks can be used with standard web technologies and not exploiting any weaknesses.
- The computer of the visitor itself could be compromised by using exploits of the browser or the underlying system itself. Attacks which exploit hardware weaknesses such as Spectre[15], which could allow an attacker to read memory space of other browser tabs, could expose sensitive informations such as passwords or banking information of the uses. Rowhammer[14] attacks could even persistently infect the users system. These exploits usually need quite some time to be successful and could benefit from constant background execution in a background browser tab.

2.2 Related works

- Papadopoulos et al.[20] analyses the if Cryptocurrency miners are a viable alternative to traditional ad serving as a revenue possibility for web sites. They come to the conclusion, that cryptocurrency mining is more profitable when a user stays longer then 5.3 minutes on the website. When permanent execution of a crypto miners are possible in background tabs, then this time could easily be achieved.
- Papadopoulus et al.[21] show that browsers can be infected with malicous service workers to act as a puppet in a botnet. Their method for persistence requires an implementation of a draft proposel HTML5 API, which is not implemented in mainstream browsers yet. They focus in their research on service workers, which are independent of browser tabs, but poses other limitations. Our research instead focuses on web sites which are not user visible, i.e. in an inactive tab.
- Measurements of existing popular websites is done for privacy related as well as security releated research. Engelhardt et al.[11] for example developed the OpenWPM framework for doing privacy measurements on million of websites. This framework is also used for ...

Security related analysis of existing webpages are done for example: to measure the use of third party script inclusions[17], to assess the security claims provided by third party security seal providers[25] or to study malware distributed via ad networks[30].

- Pan et al. analyse the feasability to use the browser of website visitors for offloading large computing tasks[19]. They termed this usage of distributed data processing gray computing, because it can be done without the users explicit consent, for example while the users are watching video streams. Their research focus is the performance of web workers and the cost effectiveness in comparison to cloud computing offerings. Our research complements the findings of Pan, because they could allow grey computing even when the user is consuming other web sites.

3 Background

To understand, how JavaScript handles tasks and concurrency, whe first have to understand how a JavaScript execution context is structured and how the event loop handles long running tasks.

3.1 The JavaScript execution context and the event loop

A JavaScript execution context is organized into a task queue, a stack and a heap [7].

The task queue defines a list of tasks, which should be executed by the JavaScript runtime. A task consist of a callback function and optional arguments to this callback function. The JavaScript runtime implements an event loop, which processes the task queue. The event loop, waits for tasks to be added to the task queue. When the task queue is not empty, the event loop removes the first task from the queue and executes its corresponding callback function with the provided arguments. This is repeated until the task queue is empty again.

The stack is comparable to a call stack in other programming languages like Java or C. When a callback function from the task queue is executed, it creates a frame on the stack. Each function call inside the callback function also creates a stack frame. When a function returns, its frame is removed from the stack. A task from the task queue is finished, when the stack is empty.

The heap is a memory region for long living objects, which are not destroyed after the execution of a single task.

When a JavaScript task is running, it is guaranteed to not be interrupted until it is completed. This also insures, that every variable cannot be changed from outside during the execution of a task. 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 wait for task to finish before it can continue rendering the web page, because the JavaScript task has access to the browser internal state and DOM[1]. Due to these properties JavaScript tasks should be as small as possible to ensure responsiveness of the browser [16]. 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.

Tasks can be added to the task queue by registering event listeners for events, for example a onclick handler of a button. When this event occurs, the runtime then adds a new tasks to the task queue. Tasks can also be added by JavaScript code itself, for example by using a timer.

3.2 The JavaScript concurrency model

The JavaScript language does not support multi threading. Each browser tab has it's own JavaScript execution context. This execution context is tied to the rendering of the page, as explained in the section before. As long running JavaScripts tasks block the rendering of the web page, calls to potentially long running functions or other side effects are almost never blocking, but instead register a callback function to be called, when the result of function returns. The runtime then performs the side effect and creates a new task in the task queue with the registered callback function and the result of the side effect as a parameter for the callback. With this workaround it is possible to perform multiple long running tasks at the same time, by splitting the work into smaller tasks, which are suspended when they are waiting for other input and are resumed when the result becomes available.

3.3 Web workers

Web workers are a addition to the HTML5 API, to overcome some of the limitations, which were described before. Web workers create a new execution context, which is *not tied* to the browser rendering. They were introduced to perform long running uninterruptable

[1]The Document Object Model is the API for describing the representation of a web page. It describes the HTML element tree, styling and registered event handlers.

calculations, which would block the rendering process for too long, if they would be executed in the main browser context. They don't share variables or resources with the main JavaScript context. The worker context therefore does not have access to the DOM of the web page, because that would introduce shared memory between the worker and main context, which would invalidate the runtime guarantees of the JavaScript specification. Other resources, such as network requests, which are independent from the main context, are available to the worker context. The web worker and main thread context can communicate by passing messages to each other, which are handled by their respective event loops. This allows the result of a calculation which happened in the worker context, to be passed to the main context and then be made visible to the user by showing the result in the DOM .

4 Analysis of different background execution methods

As we have shown in the motivation section of this thesis, there numerous incentives to execute code in the background. Browsers vendors in contrast are motivated to limit the energy impact of JavaScript code in background tabs as much as possible to prolong the battery life of the users device. To assess the behaviour of the browser throttling mechanisms, we developed a framework to compare different methods for achieving background code execution. The framework handles the benchmarking and visualisation of each method. If new HTML5 APIs are released, that allow for a new method of scheduling tasks, this method can be plugged into the framework to compare it against existing methods. The framework also allows easy reproducability of our analysis, to test whether the throttling mechanisms of browsers changed in new versions.

Currently the browser market is dominated by only three different browser engines: Blink, the browser engine of Chrome which is also used in Opera and new versions of Microsoft Edge. Safari for Mac and Mobile Safari for iOS use the WebKit browser engine. WebKit is also used for every other browser in the iOS App Store, because Apple demands that every App, which shows web content, has to use the system-provided WebKit framework to be in accordance with the App Store Review Guidelines [2]. Apps that to not comply with these guidelines are not permitted in the iOS App Store. The third browser engine is Gecko used by Firefox for desktop and for Android. Of the global world wide web usage, these three browser engines cover 90 % of the market share according to [5]. Based on this data, we decided to focus our analysis on the following desktop browsers:

- Google Chrome 76
- Mozilla Firefox 69
- Apple Safari 12.1.2

For mobile browsers we evaluate these browsers:

- Google Chrome for Android 76
- Firefox for Android
- iOS 12.4.1 Mobile Safari

This selection covers the most used browser for each browser engine on desktop and mobile.

4.1 Methodology

In the following sections we look at different methods for scheduling JavaScript code, when the tab is in the background. We describe how each browser behaves when this method is used to execute JavaScript code in the background and we use our measureing framework to compare the browser behaviour for each method.

The measuring framework takes one parameter, to tune the measuring. With this parameter we can define the workload in milliseconds. This time defines, how long a simulated task should perform uninterrupted work. When this work is performed on the JavaScript main thread then the browser is unresponsive for this amount of time, until the simulated workload finished and yields back to the main thread. Google Chrome advises to keep all JavaScript tasks to under 50ms, to be percieved as immediate for the user [16]. For our analysis, we measure each method with two workload times for each browser, once for the recommended 50 ms workload and once for a long 1000 ms workload. The measurement is started automatically when the website page is moved to the background and stopped when it is visible again. The framework uses the Page Visibility API [18] for determining the current page visibility.

To test the browsers behaviour to background tasks, we developed a browser automation script using Selenium WebDriver [24]. WebDriver lets us automate different browsers using an API. In our automation script, we open our analysis page which embeds our measuring framework and starts the benchmark by opening a new empty browser tab. After a fixed amount of time, we close the empty browser tab to make the analysis page visible again and therefor stopping the measuring. This procedure is repeated for every browser, with every test method and the two defined workload times.

After the benchmarking for one scenario is complete, the measuring framework produces a CSV file for downloading the recorded invocations in the background. Each row in the CSV file corresponds to one invocation of the simulated work load. The recorded data

includes the time since the web page moved to the background, the time since the last invocation and a computed average CPU usage. This computed CPU usage is derived from the last invocation time and the simulated work load duration with the following formula:

$$CPU_{avg} = \frac{t_{workduration}}{\Delta t_{lastinvocation}}$$

With this data we can then plot the computed CPU usage over time and compare different browsers for each scenario.

4.2 Timer tasks

The default method to schedule new tasks for execution is to use the function `setTimeout()` or `setInterval()` which are available on window or worker global scope. These methods take at least two arguments. The first is the function which should be scheduled as a task. The second argument is the time in milliseconds to wait before the task should be executed. As this is the most straighforward way to schedule tasks, this method might be the one which gets throttled the most. The WHATWG specification for timers [13] even explicitly defines an optional waiting time which is user-agent defined to allow for optimization of power usage. Also, when `setTimeout()` calls are nested or `setInterval()` repeats the 5th time, the minimum waiting time is incresed to be 4ms.

Google Chrome throttles timers for a page, when it is in the background or not user visible [4]. Since Version 11 timers are batched at most once a second. This batching helps in reducing the battery impact of the background page. Since Version 57 Google Chrome also uses a budget based timer throttling. Budget based timer throttling works by introducing a timer budget for every page in the background. When the web site is in the background for longer then 10 seconds, the budget is considered. Every scheduled timer task is only executed when the budget for this page is greater then zero. The runtime of the task is subtracted from the budget. The budget regenerates for 0.01 seconds per second. This budget based throttling has multiple implications for background pages. First, it allows for sudden bursts in computing time, when these burst are very infrequent, because the budget regenereates continously, and can be depleted in a very short amount. Background web pages which want to use continous computation time, are limited to an average of 1% of CPU usage over time, because the budget regenerates at a rate of 1/100th of a second per second. The real CPU usage usually is a little higher then 1% on average though and

only approaches 1% the longer the site is in the background, because the budget can be overdrawn with a long lasting task at the end of the measurement.

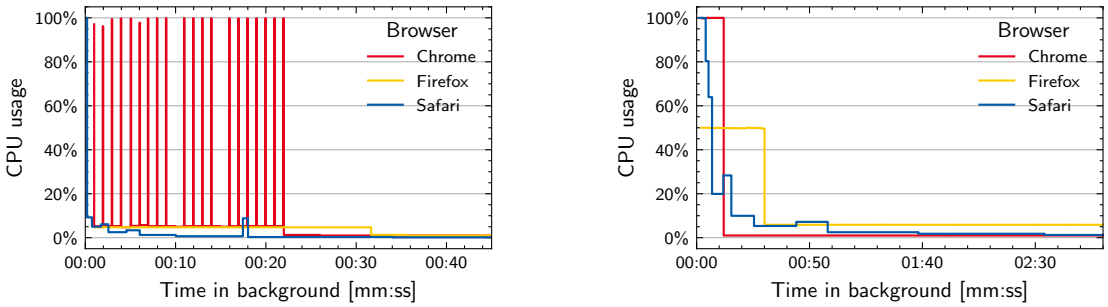
Firefox employs similiar throttling mechanisms to the ones from Google Chrome [18], though the details of the Firefox implementation differ slightly. When a page is in the background, Firefox invokes timers at least one second after the last timer finished. This is in contrast to Googles throttling to invocations once per second. The difference is most visible, when the task duration is 1 sec, because with Googles throttling implementation, the next task fires immediatly, whereas Firefox waits an additional second, before the next task is scheduled. Another differences is, that in Firefox, the budget based timer throttling is used after a page is in the background for 30 seconds, instead of 10 seconds for Google Chrome. Also Firefox caps the page budget at a minimum of -150 ms and a maximum of 50 ms. That means, that Firefox does not allow large burst of tasks, because the budget never increases above 50 ms, but longer lasting tasks, do not overdraw the budget more then -150 ms. This behaviour favors tasks, which are longer then 150ms. For example a repeated task with a duration of 1000 ms only needs to wait for 15 seconds, before it is executed again in Firefox, whereas the same task would have to wait for 100 seconds in Google Chrome, before the budget is posivite again. Firefox also employs additional throttling to tracking scripts [28], which limit repeated invocations to known tracker scripts to at most once every 10 seconds.

Safaris throttling behaviour is different then both Google Chrome's and Mozilla Firefox's behaviour. Scheduled tasks in background pages get invoked by Safari in increasing intervals. This ensures, that the CPU usage for background pages decreases much more rapidly then the CPU usage of the same page in Chrome or Firefox. Safari also coalesces timer calls, to prevent periodic wake ups of the CPU for the invocation of timers. Instead multiple timers are invoked one after the other to then let the CPU move to a energy saving state.

TODO: add iOS and Android

4.2.1 Timer tasks with WebSocket / AudioContext

Browser vendors try to strike a balance between throttling background pages as much as possible to conserve energy and keep the system more responsive and not breaking existing web applications. As backwards compatibility is an important factor, some browsers reduce their throttling, when they detect, that a web application might need more background processing time. The detection, if a web site needs more processing time is difficult though.



(a) 50 ms task duration

(b) 1000 ms task duration

Figure 4.1: CPU usage over time for continuously scheduled timer tasks

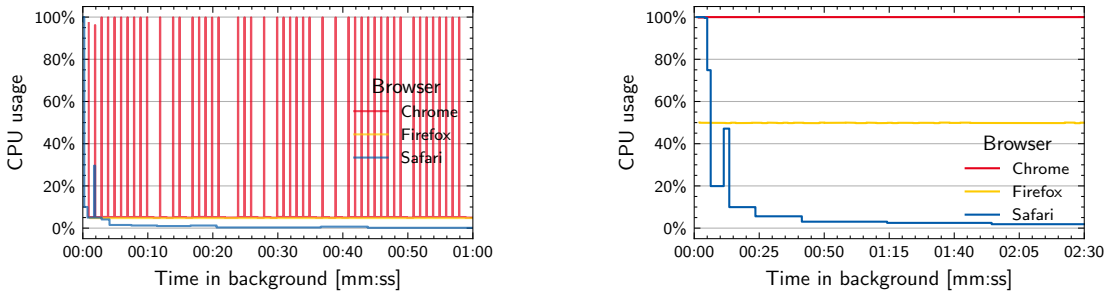
Due to the dynamic nature of JavaScript, static analysis of web scripts is very difficult. Therefore browsers use a simple heuristic, to determine if their default throttling should be reduced.

Chrome disables the budget-based throttling, when a web site has either an open WebSocket connection open or plays audible music. Playing silent music does not count as audible music playback though [4]. This behaviour could be misused to get more processing time for JavaScript tasks. Playing a audbile sound file or opening a WebSocket connection work equalliy in preventing the budget-based throttling, but they differ hugely in how visible they are to the user of the web page. While playing audio is obviously audible and marks your tab with a speaker icon in the tab bar[1], opening a websocket connection is invisible to the user and requires no user confirmation. Additionally Chrome prevents automatic playback of audio unless the user has interacted with the site before.

Firefox also disables the budget-based throttling when a open WebSocket connection is active. **TODO: Check for Audio Context.**

Safaris behaviours does not change, when a WebSocket connection is active.

[1] This is a convenience feature for users to quickly find out, which tab is producing the sounds. This is especially useful, if you have many tabs open at the same time.



(a) 50 ms task duration

(b) 1000 ms task duration

Figure 4.2: CPU usage over time for continuously scheduled timer tasks during an open WebSocket connection

4.3 `postMessage()` tasks

Besides `setTimeout()` or `setInterval()` there exists another API for putting tasks in the JavaScript task queue. The function `postMessage()` available on the window and worker scope is available to allow for communication between windows of different origins or between main thread and worker thread [27]. As each window or worker has its own JavaScript execution context, the communication between these context must happen explicitly via `postMessage()` calls. The receiving context has to register for the `message` event on the window or worker object. A call to `postMessage()` with the receiver set to a window or worker who registered for the event, adds a new task to the task queue for this window. The scheduling of tasks, which are created with the before mentioned method, are not subject to the throttling as explained in the timer tasks section. This is also true for when a `postmessage()` call is sent and received from the same context. Tasks created with this method are also not subject to the minimum delay of 4 ms as timer tasks are [8].

All desktop browsers do not throttle the scheduling of task created via `postMessage()`. This allows a web site to fully utilize the CPU even when it is on the background. Chrome and Firefox allow to use this method to run indefinitely, whereas Safari detects that a web page is using significant energy and reloads the page after around 8 minutes in the background.

With this API not being throttled when a web site is in the background, a malicous entity could use this to their advantage. This shows, that the throttling mechanisms put in place

for timer tasks are only considered for saving energy for well-behaving websites and not for protecting the users web sites which actively try to circumvent the throttling.

4.4 Web workers

Web workers allow us to spawn a seperate execution context, which behaves different then the main execution context [29]. Worker contexts are also treated differently by browsers with regards to timer throttling in the background. Timers created with `setInterval()` or `setTimeout()` inside the worker context are not subject to the throttling when the corresponding web page moves to the background. This fact is used by the library worker-timers [1], which implements a broker and scheduler for timer tasks which are not throttled when the page moves to the background. The scheduler component runs inside the web worker, whereas the broker is run from the main execution context and dispatches messages when a timer was created or cleared via `postMessage()` to the worker context. The scheduler then sets an appropriate interval inside the worker context. When the timer inside the worker context fires, the scheduler posts a message to the main thread, which then calls the scheduled function. Through this indirection, the main thread can schedule timers with the worker-timers library, as if it was using the native window timers, but without the throttling behaviour, when the page is in the background. We created a test case with our measuring framework, how each browser behaves with this workaround.

Chrome and Firefox do not throttle timers in web worker context for background tabs at all. This means you can max out the main thread context with the worker-timers library for indefinite time.

Safari does also not throttle timers in web worker context for background tabs, but it will detect significant energy usage and may reload the page after constant time with high CPU usage. There is also a feature in the macOS operating system called AppNap, which might suspend the browser at whole, when Safari itself is not visible to the user [10]. This feature kicks in, when the operating system decides, that the browser does not provide critical functionality at the moment and is also not visible. When the user is interacting with the Safari, but not the background tab itself then AppNap will not be invoked.

	Google Chrome	Mozilla Firefox	Apple Safari
Timer tasks	Coalesced invocations at most once per second. Budget-based throttling after 10 seconds in background.	Next invocation at least one second after last invocation. Capped budget-based throttling after 30 seconds in background.	Coalesced invocations. Time between invocations increases.
Timer tasks with open WebSocket / AudioContext	Coalesced invocations at most once per second. No budget-based throttling.	Next invocation at least one second after last invocation. No budget-based throttling.	Coalesced invocations. Time between invocations increases.
`postMessage()` tasks	No throttling	No throttling	No throttling, but page gets reloaded when it is using significant energy
Web worker timer tasks	No throttling	No throttling	No throttling, but page gets reloaded when it is using significant energy

Table 4.1: Summary of desktop browser background tab behaviour

4.5 Summary

In this section we analysed the behaviour of background tabs for desktop and mobile browsers. We saw that all browsers employ a timer throttling when the page is in the background. Chrome and Firefox use a budget-based timer throttling approach, whereas Safari increases the time between timer calls with every invocation. These timer throttlings can be circumvented by using the `postMessage()` API, by using web workers or by opening a websocket connection or an AudioContext. With all circumvention methods, we can saturate the main thread for indefinite amount of time in Chrome and Firefox. Only Safari detects continuous high CPU usage and reloads the page when it is using significant energy. Table 4.1 summarizes the findings for desktops browsers.

4.6 Future research

We have limited our detailed analysis of background execution methods to these methods listed above, but the measuring framework we developed can be used for further investigation of other methods to achieve arbitrary background execution.

4.6.1 Future browser updates and new APIs

Chrome and Firefox have adopted a rapid release cycle with new major releases of their product every couple weeks. When new versions are released with new energy conserving features, our research can be reliablty be verified with these new browser versions. Additionally with every major update, browser may ship new experimental APIs, which can be investigated for achieving background execution. These experimental and browser specific APIs may become a new HTML5 standard API. This is also a trend which emerged in the last years. New HTML5 standardization usually emerges by a browser implementing new features in a vendor specific API. When this API is proven to be useful and accepted by web site authors, it is converted to a API design specification to be implemented by other browsers.

4.6.2 Service workers

A notable omission to our detailed analysis are service workers. Although they have a similar name as web workers, they serve a complete different purpose. Service workers are complementary scripts, which belong to an URL for which they are registered. The service worker is invoked by the browser in response to certain events, which can be triggered by the website but also by the browser or the server which hosts the web site. Service workers are decoupled from the lifetime of the web page which registered the service worker. If two tabs of the same page are open, only one service worker is used by the browser to handle the events for these two tabs. The main use cases for service workers at the moment are a) providing a offline capable web site and b) allowing web site push notifications. To implement offline capable web sites, the service worker registers for intercepting network requests from the web site. If the user is offline and tries to do a network request, the service worker uses a cached response to handle the request or saves the payload for sending it later, when the user is online again. Service workers try to close the gap between web and native applications and may be extended to support more functionality.

We omitted the analysis of service workers, because the service workers specifications explicitly recommends, to terminate service workers which take longer then expected to perform their respective tasks. It could be possible, that browser implementations to not adhere to this recommendation, but this should be considered a browser bug and not intended behaviour. Our reseach instead focusses on the circumvention of battery conserving features of the browser implementations.

5 Tracing of background execution on popular websites

In the first part of our research we identified different methods to circumvent the browser timer throttling mechanisms for brackground tabs. With these findings in mind we can now trace popular websites to analyse if these circumenvtion methods are used in the wild. We used the first 1500 web sites included in the last free publication of the Alexa top 1 million site list[1]. We decided to just trace the home page of each web site and no sub pages to limit the scope of this research. This is a known compromise of this tracing research, because many web applications only show a largly static landing page on the home page, while the script-heavy application itself is only available after login. On the other hand, it is not feasable to provide login credentials for a mass analysis of web sites, but it could be the topic of further research in this area. After we visit the home page of the web sites we measure the CPU usage caused by scripting after it was moved to a background tab. With the aggregated result of our tracing, we can determine if the browser throttling mechanisms are suiteable to limit CPU usage or if large number of websites use these methods to actively or inadvertently bypass the background throttling mechanisms of the browser. For this part of our research we focus only on desktop browsers, because the potential for misuse of desktop browsers is much larger. As the browser usage on mobile is limited to smaller timespans, usage of browsers on non-mobile devices is more ubiquitous as more and more applications are replaced with web apps. Often times the browser is open as long as the system itself, with many tabs open in the background during the whole time. The most used desktop browser in the year 2019 is Google Chrome with over 70 % of market share according to [9]. With this leading margin over other browsers, we decided to only trace web sites with Chrome.

[1] The list is available under `http://s3.amazonaws.com/alexa-static/top-1m.csv.zip`

5.1 Automated measuring of background execution

To trace a web site in the background we perform the following steps: First we open a new Chrome instance and start the browsers internal 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.

To automate the tracing we used Puppeteer [23]. Puppeteer is a NodeJS library which uses the Chrome DevTools Protocol [6] to control a Chrome. The DevTools Protocol is also used by Chrome DevTools which are bundled with Chrome and it allows to access the browser interals. With Puppeteer we can therefore use the same tracing capabilities of Chrome as the web profiler included in the Chrome DevTools. This allows us to evaluate our traces with the existing profiling tools of Chrome. The website trace includes information if the website is executing code and the duration of each JavaScript task that was executed while the web site was in the background.

Another capability of puppeteer is to inject custom JavaScript code before a web site is rendered. Due to the dynamic nature of JavaScript, it is possible to override the implementation of every existing function. We use this feature in our injected script to hook the API for creating Web workers, opening a WebSocket and using `postMessage()` calls, to record their respective usage for our tracing. A simplified example how the hooking is performed looks like this:

```
1 const orig_postMessage = window.postMessage;
2 window.postMessage = function () {
3   console.log('postMessage() was called');
4   return orig_postMessage(...arguments);
5 }
```

To hook a function, we first save the orginal function in a new variable in line 1. We then redefine the function to record the usage in line 3 and pass the arguments the original function in line 4. The `arguments` object is available in all non-arrow function and refers to the parameters passed to the function. With this hooking technique we can record the calls for APIs, which we are interested in, without the need to change the code of the web sites we trace.

5.2 Analysis of web site traces

The website trace together with the recoding of the hooked functions allows us to dig deep into what a single website is executing while it is in the background and which methods it used for scheduling the tasks, but to get a better picture of how all traced websites in general behave in the background we need to find a way to compare and aggregate the results of the traces.

To compare the result of two traces, it is beneficial to have a single score for each web site. We propose to use the average CPU usage triggered by scripting during the profiling to use as this score. The average CPU usage score can be calculated from the trace file generated by the web profiler. To calculate the CPU usage time for scripting we use the sum of the duration of all JavaScript tasks executed in the main thread as well as in all potentially spawned worker threads. We use the wall clock time for this measurement, because it 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. Additionally this is the same method which Chrome uses for the budget-based throttling calculation [4].

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. 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.

5.3 Evaluation of tracing results

6 Conclusion

The timer throttling does not protect users from malicous web sites, which actively try to use CPU time when they are in the background. It can only be viewed as a battery conserving method for most web sites.

To protect users from unwanted CPU usage of background tabs, browsers should suspend tabs completely when they are not visible to the user. This goal is already acknowledged from Google Chrome developers as stated in [3], although the roadmap for implementation is not adhered to.

Suspending all background tabs will of course break legitimate use of background timers for web sites, where users expect the page to update in the background. Safari implemented a good interim solution by detecting high energy consumption of background tabs and reloading them, when they reach a certain threshold. This solution allows web site to run computation in the background but only if it does impact battery life too much, but it protects users from unwanted CPU usage and battery drain from web miners or other high CPU workloads. We propose two possible solutions for browser vendors to let user-chosen web apps not become suspended when in the background while still suspending all other web sites:

The first proposed solution is to make background execution a permission, which web sites have to explicitly ask for. Browser could then display a popup where users can allow or deny the use of background execution. This popup based permission system is already implemented for other privacy related activities, for example when a web site ask for the users location or access to camera and microphone or when a web site wants to send push notifications. Figure 6.1 shows how these dialog look like for access to the users microphone. One could argue, that even more permission popups could train users to blindly accept, whatever is shown to them. This behaviour is also reinforced by numerous cookie and policy permission dialogs as implemented to conform to GDPR requirements. On the other hand, cookie and tracking policy permissions are shown inside the web site frame, whereas the proposed background execution permission dialog should be shown

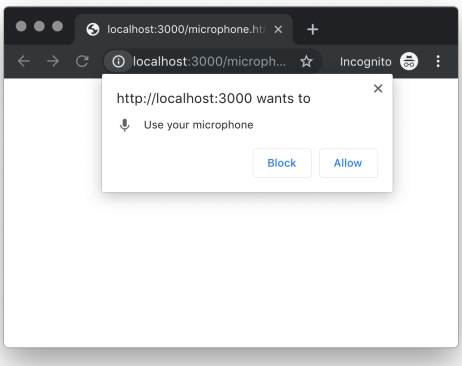


Figure 6.1: Chrome website permission dialog

from the browser UI and can therefore be recognized as a more important dialog. Web sites would then have to opt in to not be suspended when in the background via a new API. The question remains when this permission popup should be shown two the user. User experience research shows **TODO: search for proof**, that users are more likely to give permission for elevated access, when the web site first needs them and the usage of these permissions is obvous to the user. A good example is the microphone or camera permission dialog. Consider a video conferencing web sites, where these dialogs could be shown, when the users tries to join a call. For background execution permission the natural time, when the web site could ask for this permission is, when the user switches to another tab, but then the context of the web site is already lost and it could be confusing to the user, to then ask for the permission. Another obvious time would be, when the user first visits the web site, but this behaviour contradicts the experience, that users are less likely to give permissions as soon as the web site is loaded. It may not be clear to the user at this time, why the web site needs background execution to function properly. Another approach would be to show the dialog as soon, as the user reopens the tab a couple of times after it was in the background. At that time, it is confirmed, that the user frequently checks the content of this web site. Also the permission dialog could explain to the user, that the content of this page does not refresh in the background without the explicit permission for background execution.

The second proposed solution is to disable suspension of tabs, when they are pinned. Pinned tabs is a feature implemented by most desktop browsers, which allows easy access to heavily used web sites. Figure 6.2 displays a pinned tab besides a normal tab in Safari.

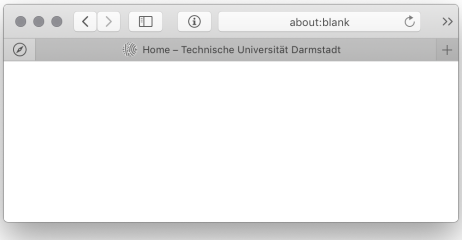


Figure 6.2: Pinned tab feature in Safari

Pinned tabs usually only show the favicon of the web site and hide the close button of the tab. [22] explains how pinned tabs work in Firefox. It also describes, why users should use pinned tabs: “The internet is now full of websites that we use more like programs than just static pages. Popular sites like Facebook and Gmail are like this – they’re used to accomplish tasks (or to avoid accomplishing tasks), they update themselves and notify you when they’ve changed.” Web sites, which users are likely to pin, are often times the same pages benefiting from background execution. It therefore makes sense, to combine the permission for background execution to pinned tabs. We claim, that the coupling of the background execution permission and pinned tabs is exactly what users expect for pinned tabs. With this solution, users do not have to be asked for another permission and it allows legitimate web sites to perform background tasks while still letting the user be in control of background CPU usage. It would also not require a new API for requesting the background execution permission and is therefore backwards compatible with existing web sites. Pinning a web site tab thereby converts the browser behaviour from a normal web site to a web application.

Mobile browsers do not have a concept of pinned tabs, but Android and iOS support adding web sites to the home screen. When added to the home screen, a web app can hide the browser chrome for navigation and the address bar to behave more like a fully native app. We propose to give background execution permission to these web sites, which were added to the home screen by the user. This would be the equivalent of a pinned tab on desktop browsers for background execution behaviour.

7 Bibliography

[1] *A replacement for setInterval() and setTimeout() which works in unfocused windows*. URL: https://github.com/chrisguttandin/worker-timers.

[2] *App Store Review Guidelines - Apple Developer*. URL: https://developer.apple.com/app-store/review/guidelines/#software-requirement.

[3] *Background tabs & offscreen frames: further plans*. URL: https://docs.google.com/document/d/18_sX-KGRaHcV3xe5Xk_l6NNwXoxm-23IOepgMx4OlE4/pub.

[4] *Background Tabs in Chrome 57*. URL: https://developers.google.com/web/updates/2017/03/background_tabs.

[5] *Browser Market Share Worldwide | StatCounter Global Stats*. URL: https://gs.statcounter.com/browser-market-share#monthly-201808-201908-bar.

[6] *Chrome DevTools Protocol Viewer*. URL: https://chromedevtools.github.io/devtools-protocol/.

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

[8] *David Baron's weblog: setTimeout with a shorter delay*. URL: https://dbaron.org/log/20100309-faster-timeouts.

[9] *Desktop Browser Market Share Worldwide | StatCounter Global Stats*. URL: https://gs.statcounter.com/browser-market-share/desktop/worldwide/2019.

[10] *Energy Efficiency Guide for Mac Apps: Extend App Nap — Apple Developer Documentation Archive*. URL: https://developer.apple.com/library/archive/documentation/Performance/Conceptual/power_efficiency_guidelines_osx/AppNap.html.

[11] Steven Englehardt and Arvind Narayanan. “Online Tracking: A 1-million-site Measurement and Analysis”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: ACM, 2016, pp. 1388–1401. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978313. URL: http://doi.acm.org/10.1145/2976749.2978313.

[12] Jeremiah Grossmann and Matt Johansen. “Million Browser Botnet, Conference talk at Black Hat USA 2013”. In: USA, 2013.

[13] *HTML Living Standard — Last Updated 3 September 2019, Timers*. URL: https://html.spec.whatwg.org/multipage/timers-and-user-prompts.html#timers.

[14] Yoongu Kim et al. “Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors”. In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA ’14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 361–372. ISBN: 978-1-4799-4394-4. URL: http://dl.acm.org/citation.cfm?id=2665671.2665726.

[15] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *40th IEEE Symposium on Security and Privacy (S&P’19)*. 2019.

[16] *Measure Performance with the RAIL Model*. URL: https://developers.google.com/web/fundamentals/performance/rail.

[17] Nick Nikiforakis et al. “You are what you include: large-scale evaluation of remote javascript inclusions”. In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012.

[18] *Page Visibility API*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Page_Visibility_API.

[19] Yao Pan et al. “Gray computing: an analysis of computing with background JavaScript tasks”. In: *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press. 2015, pp. 167–177.

[20] Panagiotis Papadopoulos, Panagiotis Ilia, and Evangelos P. Markatos. *Truth in Web Mining: Measuring the Profitability and Cost of Cryptominers as a Web Monetization Model*. 2018. arXiv: 1806.01994 [cs.CR].

[21] Panagiotis Papadopoulos et al. *Master of Web Puppets: Abusing Web Browsers for Persistent and Stealthy Computation*. 2018. arXiv: 1810.00464 [cs.CR].

[22] *Pinned Tabs - keep favorite websites open and just a click away | Firefox Help*. URL: https://support.mozilla.org/en-US/kb/pinned-tabs-keep-favorite-websites-open.

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

[24] *Selenium WebDriver*. URL: https://www.seleniumhq.org/projects/webdriver/.

[25] Tom Van Goethem et al. "Clubbing seals: Exploring the ecosystem of third-party security seals". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, pp. 918–929.

[26] Wikipedia contributors. *Malvertising — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Malvertising&oldid=911222206. [Online; accessed 31-August-2019]. 2019.

[27] *Window.postMessage() - Web APIs | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage.

[28] *WindowOrWorkerGlobalScope.setTimeout() — Reasons for delays longer than specified*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setTimeout#Reasons_for_delays_longer_than_specified.

[29] *Worker - Web APIs | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Worker.

[30] Apostolis Zarras et al. "The dark alleys of madison avenue: Understanding malicious advertisements". In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM. 2014, pp. 373–380.