MalloryWorker: Stealthy Computation and Covert Channels using Web Workers

Michael Rushanan, David Russell, Aviel D. Rubin

Department of Computer Science, Johns Hopkins University
Baltimore, MD, USA
{micharu1,drusse19,rubin}@cs.jhu.edu

Abstract. JavaScript execution and UI rendering are typically single-threaded; thus, the execution of some scripts can block the display of requested content to the browser screen. Web Workers is an API that enables web applications to spawn background workers in parallel to the main page. Despite the usefulness of concurrency, users are unaware of worker execution, intent, and impact on system resources. We show that workers can be used to abuse system resources by implementing a unique denial-of-service attack and resource depletion attack. We also show that workers can be used to perform stealthy computation and create covert channels. We discuss potential mitigations and implement a preliminary solution to increase user awareness of worker execution.

Keywords: web security, stealthy computation, covert channel

1 Introduction

Adobe Flash is an example third-party plugin that extends functionality like video streaming to web applications. HTML5 eliminates this necessity by providing new APIs that improve core functionality of the web browser (herein browser). Web Workers is one such API specified by the World Wide Web Consortium (W3C) [11] and Web Hypertext Application Technology Working Group (WHATWG) [10]. Web Workers enable web applications to spawn background *workers* (i.e., threads) in parallel to the main page. Workers are intended for long-lived and computationally intensive operations that would otherwise block the UI.

Encryption, motion detection, and simulated annealing are use cases for workers. Any application that has to have its execution broken up to avoid being prematurely terminated by the browser is a candidate for workers.

Despite the usefulness of concurrency in JavaScript, permissive execution of workers enables stealthy computation. Workers are instantiated unbeknownst to the user of a web application and can perform any number of computations. An attacker can cause a user to perform work for her by exploiting a cross-site scripting (XSS) vulnerability on a legitimate website or by placing an advertisement that hides the work in a worker.

We demonstrate the feasibility of stealthy computation using workers by implementing a distributed password cracker that uses the Web Workers API. We can compute 500,000 MD5 hashes per second using the Chrome 50.0.2661.94 browser on a Mid-2013 MacBook Air. We also implement a denial-of-service (DoS) attack that is unique to how OS X manages virtual memory. We define *wasteful* stealthy computation that exploits garbage collection mechanisms in Chrome, Firefox 46.0.1, and Safari 9.1. The execution of this computation results in high CPU and memory utilization that eventually fills the swap partition and causes a deadlock.

We target the Android browser and Android Chrome browser to perform wasteful stealthy computation on a mobile platform. We find exploiting garbage collection results in a resource depletion attack against the browsers. In fact, 55% of CPU load, 45% of memory usage, and an approximate 4°F increase in temperature was the direct result of five minutes of stealthy computation. We did not attempt this on the mobile Safari browser for iOS but believe that it is also susceptible because its operating system counterpart is. The mobile Safari browser is not susceptible to the DoS attack because it manages virtual memory differently than OS X.

A natural criticism to stealthy computation using workers is that a worker is *unnecessary* to perform attacker-controlled computation such as the DoS attack mentioned above. While the UI thread can carry out this type of computation, the thread becomes unresponsive and is later terminated by the browser. JavaScript Window Timers like `setTimeOut` avoid blocking the UI thread by executing code at specified time intervals. However, we find that our stealthy computation still results in unresponsiveness and later termination when using `setTimeOut`.

Lampson, when defining the confinement problem, first introduces covert channels as information leakage between processes that facilitate communication [14]. Covert channels are difficult to identify because other processes often obscure them. For example, CPU cycles can be used as a covert channel and it is affected by every single process on a system. Further, application firewalls and anti-virus software typically block non-whitelisted ports and anomalous behavior, not profile software system resource utilization.

We describe and implement a covert channel that is not unique to workers but is easily implemented using them. Our covert channel uses CPU and memory throttling to transmit bits to an unauthorized application. We find that CPU throttling is noisier than memory throttling because other processes can obscure our covertly transmitted bits (i.e., a random peak can corrupt bits or semantic structures such as a preamble). We throttle memory by exploiting garbage collection to create a peak and then terminating the web worker to force garbage collection.

This covert channel enables an attacker to transmit data from a website to an application on the user's system. This application may be untrusted or malicious. The attacker can send command-and-control instructions, binary updates, and sensitive data about the user's browsing without detection as browsers typically use a range of system resources depending on viewed content.

We scanned 7000 websites from Alexa's top sites to determine the prevalence of worker use. We found that 1.2% of them use workers to perform some computation. Websites such as yahoo.com, usbank.com, and mediafire.com use workers for various reasons. For example, usbank.com uses a worker defined in `foresee-worker.js` to compress session event logs.

In this paper, we are concerned with using the Web Workers API to create workers that enable stealthy computation and covert channels. We demonstrate the feasibility of these by implementing our own distributed password cracker using workers, a DoS attack against OS X, a resource depletion attack against Android, and a covert channel using memory throttling. We provide the necessary background for JavaScript code execution and Web Workers, discuss related work focused on HTML5 vulnerabilities, and we give the first mitigation strategy for the misuse of workers.

2 Background

Browsers typically have one thread that JavaScript and the UI share. Therefore, UI updates are blocked while the JavaScript interpreter executes code and vice versa. A shared task queue enables asynchronous execution of JavaScript and UI updates, allowing either to execute when the thread is available. Asynchronous execution does not solve the problem of an arbitrary script taking unusually long. The browser attempts to terminate any script that takes longer than some threshold regardless of its purpose or importance. The user is aware of this when the UI freezes. Not much later, the browser presents a status (i.e., terminate or continue) or crash message.

The browser's approach to ending long-running scripts is undesirable because it provides no context per the scripts execution. The user is unaware of what the script is meant to do and how long it has been running. Web application developers approach this issue by leveraging asynchronous execution and dividing their scripts into logical chunks that execute on some period. This method does not benefit from parallel execution where a computation is uninterrupted until it finishes.

HTML5 addresses these limitations with the Web Workers API. This API enables web applications to spawn background workers in parallel to the main page. Workers are unable to access the Dynamic Object Model (DOM) or the callers (i.e., parent object) variables and functions. Workers are instantiated as one of two types: shared or dedicated.

Shared workers can be accessed by multiple web applications but dedicated workers cannot. Web applications instantiate both shared and dedicated workers by providing a script object to the `Worker` constructor. The script object is either an externally loaded file or defined *inline* as a string description of the worker.

The string description is provided as input to the `blob` constructor, a file-like object, and is referenced by an output URL handle. This URL handle is provided to the `Worker` constructor. Listing 1.1 is an example inline instantiation. We note

that inline instantiation is important to our threat model because attackers that inject malicious scripts must be able to inject a worker.

```
<script id="mw" type="javascript/worker">
  self.onmessage = function(event) {
    self.postMessage({'msg': 'hello.',});
  }
</script>
<script language="javascript">
  var blob = new Blob([document.querySelector('#mw').
      textContext]);
  var m_worker = new Worker(window.URL.createObjectURL(blob))
      ;
</script>
```

Listing 1.1: Instantiate worker using blob.

Workers support communication with each other and their parent object via message passing. The `onMessage` method listens for messages and upon receiving one it will call the `postMessage` method to send a message. Workers continue to listen for messages until the user navigates away from the web application, or the parent object calls the `terminate` method on the worker. Terminating a web worker causes garbage collection on all allocated memory.

3 Threat Model

We use the definition of a *web attacker* and *gadget attacker* by Akhawe et al. [4] to define an attacker that maliciously misuses workers. A web attacker operates a malicious web application but has no visibility into the network beyond the requests directed to her application. A gadget attacker can inject content into otherwise legitimate web applications.

A web attacker that misuses workers hosts a web application with a mechanism for generating traffic (e.g., misleading domain name or social engineering). Every time a user visits the web application, stealthy computation is performed via a worker or workers. A gadget attacker that misuses workers exploits web vulnerabilities such as cross-site scripting to inject her workers. She may also purchase a web advertisement and bundle her workers in the ad. A user that visits a legitimate site will now perform some stealthy computation.

A web attacker is considered an insider threat; for example, a web application administrator. A gadget attacker is an outside threat. She is simply a web application user. We consider both attackers to be unsophisticated as neither has visibility or control of the network. Also, both attackers rely on generally accessible tools such as a laptop, internet access, and at most a web server.

The goals of both a web attacker and gadget attacker that misuse workers include: performing stealthy computation, mounting a DoS or resource depletion attack, and establishing a covert channel with an untrusted or malicious application. While we do not describe how to install such an application, we consider all typical malware delivery methods (e.g., flash drives, e-mail, etc.).

4 Web Worker Primitives

While creating stealthy computation is as simple as writing function x, a wasteful computation needs to exploit garbage collection mechanisms for multiple browsers. Covert channels also require a mechanism for throttling a system's CPU and Memory. We introduce three primitives to achieve wasteful stealthy computation: infinite loop sequences, CPU throttling, and memory throttling.

```
var cpu_work = function() {
  var scratch = [];

  // Fill the ArrayBuffer with random values.
  for(var j = 0; j < 1024; j++) {
      scratch.push(Math.random());
  }

  var firstArr = new Uint8Array(scratch);
  var secondArr = new Uint8Array(scratch);

  // ArrayBuffer concatenation.
  var concatBuf = new Uint8Array(firstArr.byteLength +
      secondArr.byteLength);
  concatBuf.set(new Uint8Array(firstArr), 0);
  concatBuf.set(new Uint8Array(secondArr), firstArr.length);
}
```

Listing 1.2: Browser CPU throttling.

Infinite Loop Sequences. An infinite loop is a sequence of instructions which loops endlessly because the boolean condition never changes (e.g., it always evaluates true). If an infinite loop is executed by the JavaScript interpreter, the browser UI will freeze due to blocking on the shared thread. However, blocking does not occur if this loop is executed in a worker.

We use an infinite loop such as `while(true){}` to perform a wasteful stealthy computation. This type of computation enables CPU and memory throttling. Again, the execution of this loop is undetected by the user because it does not block the UI thread.

CPU Throttling. Executing an empty infinite loop alone will throttle a modern CPU. We achieve throttling by looping on intensive operations such as recursive function calls and large data manipulation to quickly achieve maximum CPU utilization. Listing 1.2 implements a data manipulation loop that randomly fills two 1024-byte arrays and then concatenates them.

Memory Throttling. Throttling memory is browser specific as it exploits corner-cases not yet handled by the browser's garbage collection. We note that the browser does, in fact, do garbage collection correctly; however, the process is

approximate as deciding whether memory can be freed is *undecidable*. We use this knowledge to our advantage to discover browser-specific memory leaks and use them to throttle system memory.

In Listing 1.3 we use a technique outlined by Glasser [9] to demonstrate a memory leak in Firefox. This technique relies on JavaScript closures. Specifically, both `unused` and `bucket` are both defined inside of `RD_ATTACK_FIREFOX_SAFARI` scope, and if both functions access the variable `leak` it's imperative that both get the same object. So `leak` is never garbage collected.

In our experimentation with these primitives, we crashed Firefox and Chrome when throttling CPU and memory. We mitigate this by using the worker method `terminate()`. This method helps us avoid crashing the browser and completes our throttling primitives by exposing a mechanism for quickly freeing system resources.

```
var bucket = null;
var RD_ATTACK_FIREFOX_SAFARI = function () {
  var leak = bucket;
  var unused = function () {
    if (leak) {
      var hole_in_bucket = 1;
    }
  };
  bucket = {
    longStr: new Array(10000000).join(Math.random()),
    someMethod: function () {
      var hole_in_bucket = 2;
    }
  };
  // Placeholder for doing some repetitive operation.
  cpu_work();
};
```

Listing 1.3: Firefox memory throttling.

5 Stealthy Computation

We demonstrate the feasibility of stealthy computation using workers by implementing a distributed password cracker that uses the Web Workers API. We implement the main HTML page to define a target MD5 password hash, a worker instantiation, and an event listener to receive the result of password cracking (i.e., an MD5 collision was found).

The worker instantiation is on input `md5cracker.js`. This worker script defines the MD5 hashing algorithm, a dictionary download method, and the event listeners start and stop.

The start listener waits to receive the `onMessage` start string. When it receives the string, it downloads an array of passwords using the method `importScripts()`. This method synchronously imports a script into the worker's

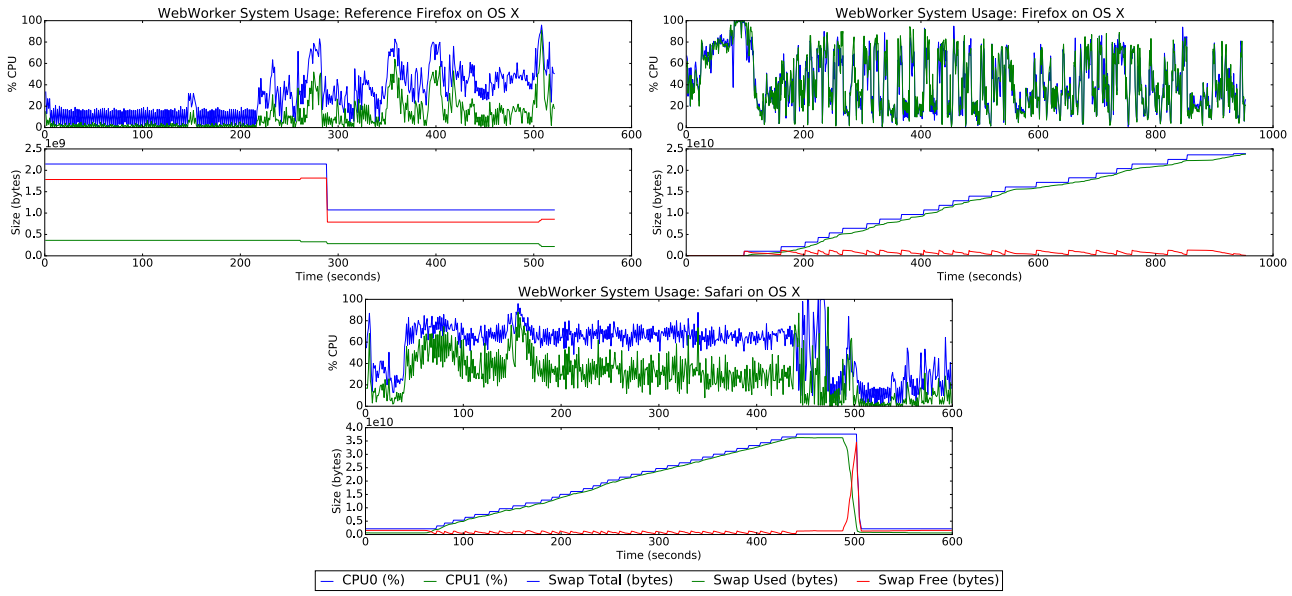

Fig. 1: OS X DoS attack.

scope. We use it to import an array of passwords because we want the worker to be self-contained. Specifically, if an attacker should inject a worker or upload an advertisement with a worker, she can not rely on the calling parent object to pass in any data such as an array of passwords.

After downloading the array, the worker selects a random index into the array and begins to hash each password and compares it to the target hash. If it finds a collision, it returns the result to the parent object, or it could use a web socket to send it elsewhere (e.g., the attacker's server).

The stop listener simply kills the worker once it is no longer useful.

We send 1 million passwords to the worker using `importScripts` which is approximately 13MB. This step adds approximately 50% latency on the dataset and takes 3 seconds to download. We can minimize this time by compressing the password array and partitioning it into multiple arrays. The password cracker performs 500K hashes per second on a Mid-2013 MacBook Air.

The average user visits a website for no longer than 15 seconds. Thus, one might think that workers that perform stealthy computation do not have much time to carry out any worthwhile computation. We find that media streaming sites such as Youtube or SoundCloud are ideal web applications for stealthy computation because users will remain on the page for a time much greater than 15 seconds. In addition, this technique has been proven via projects that unintentionally do stealthy computation using workers such as bitcoin mining [6]. We are the first, that we know, to point out the scope (i.e., all modern browsers) and potential of this type of computation.

5.1 Denial-of-Service

We use our loop and memory throttle primitives to mount a DoS attack against any 64-bit OS X device. This DoS is unique to OS X because of how virtual memory is handled. Specifically, OS X can grow its swap to the maximum available size of the backing store – the portion of hard disk responsible for storing virtual memory pages. On 32-bit systems the backing store is limited to 4 gigabytes, whereas 64-bit OS X systems can use up to 18 exabytes. Therefore, if we exploit garbage collection for an extended period, OS X will continually write out memory pages until deadlock. The period required for deadlock depends on the amount of memory leaked and the available space on disk.

In one test on a Mid-2013 MacBook Air with 50 GB free, 10 MB was leaked each loop iteration and deadlock occurred after 15 minutes. We can adjust the memory throttle primitive to allocate more memory each iteration and speed up deadlock.

This attack is successfully executed on Firefox and Safari only (specifically, versions Firefox 46.0.1 and Safari 9.1 (11601.5.17.1)). In Firefox, deadlock is always achieved since the upper-bound on paged memory is the full 18 exabytes. Deadlock is only achieved in Safari if the user has less than 32 GB available on disk. Otherwise, the browser kills the process.

We note that running the attack in a worker results in no UI indication; the user is unaware of the DoS attack. This is especially poignant in Firefox and Chrome, where running the attack *without* a worker results in an 'Unresponsive Script' notification. In Safari, no indication is given regardless of the payload's delivery (via UI thread or web worker). We therefore find that Safari and Firefox are susceptible to this attack, with Firefox's viability being dependent on workers.

The steadily growing swap in Figure 1 depicts our exploitation of garbage collection in Firefox and Safari. In the upper-left we have a normal profile of the browser's swap and CPU loads. The low-CPU section corresponds to no browser interaction on a static page. In the upper right and central figures, the scale for swap usage is now 10 GB. In the upper-right, swap is filled until deadlock occurs around 24 GB (the max available on the test system). In the central figure, swap is filled until it hits the 32 GB threshold, at which point Safari kills the process. In both cases, when deadlock occurs, OS X needs to be hard rebooted in order to recover. Fortunately, disk space is recovered and the swap returns to its original size.

5.2 Resource Depletion

The mobile Chrome browser also supports the Web Workers API. Figure 2 depicts user memory usage as it steadily increases from the stealthy computation. The over usage of memory results in I/O waiting toward the end of our experiment. Stealthy computation can exacerbate resource depletion as it uses system resources to perform wasteful work. Figure 3 illustrates resource depletion in terms of its effects on the battery. When the attack is initiated, battery temperature immediately spikes more than 8° F (in orange). Furthermore, projected battery lifetime drops significantly.

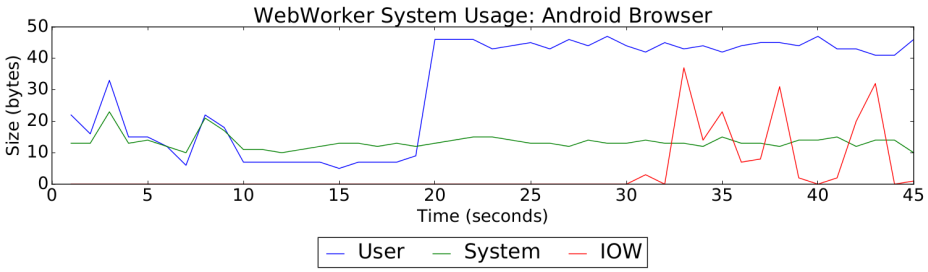

Fig. 2: Android Chrome resource depletion attack.

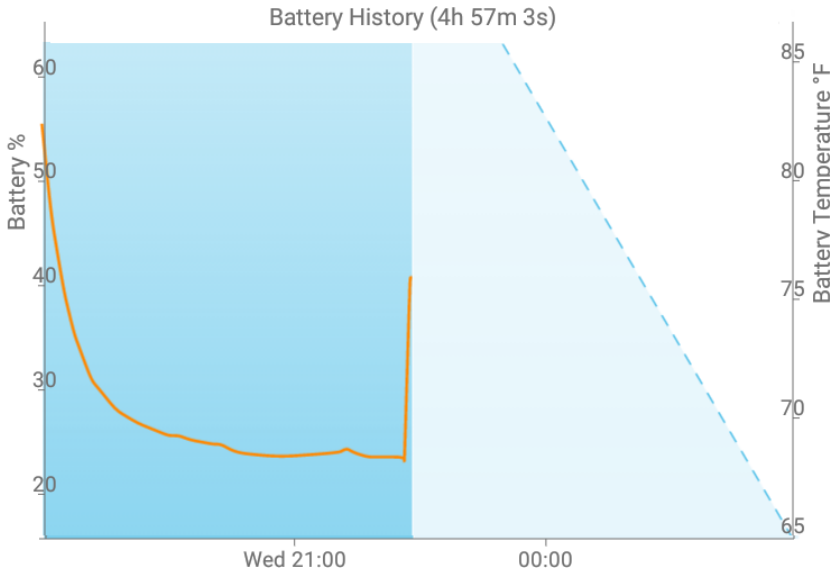

Fig. 3: Android Chrome resource depletion attack.

6 Covert Channel

A covert channel is a communication mechanism for two processes that are not supposed to be able or allowed to communicate. For example, Lampson first described a covert channel based on a program's effect on system resources [14]. The program attempting to transmit information can vary resources such as I/O or memory, and the receiver will observe the change. While this is a noisy channel, it can be corrected given a message encoding. Other covert channels include cachememory bus interactions [19], CPU scheduling [12], and network packets and protocols [7, 15]. These covert channels are timing channels because they transmit information by modulating system resources. Storage channels require access to storage locations whereby the transmission of information is written and read from the filesystem.

We use our CPU and memory throttling primitives to create a covert channel between a web application and some untrusted or malicious application on the user's system. The application is a desktop or mobile application that only requires access to monitor system resources, which is an unprivileged operation.

Application firewalls and anti-virus software can block TCP connections to non-whitelisted ports and anomalous behavior. Our covert channel circumvents these technologies by not using a standard channel like TCP. Further, if the covert channel could be identified, the result would be to block the entire browser. An attacker can use this channel to deliver command-and-control instructions, binary updates, and sensitive data about the user's browsing.

The use of both CPU and memory throttling primitives and the unauthorized application constitutes a timing channel. The web application, or injected script, transmits information about the user's browsing to the unauthorized application. Like other timing-based covert channels, this is difficult to detect. However, the covert channel cannot transmit information to other JavaScript scripts because the browser isolates execution and disallows access to system resources with sandboxing.

We first try CPU throttling to observe messages with a simple structure. Specifically, we do not define a pre or postamble; rather, we define a period in which to observe a bit based upon a CPU usage spike. We find that the CPU channel is noisy, as seen in Figure 4, and we can only achieve good accuracy by employing a high sampling rate. Unfortunately, we use PSUTIL to get current CPU usage and it imposes a sampling rate with a minimum bound of 100 milliseconds. Also due to JavaScript runtime limitations, anything less than one millisecond isn't feasible.

We attempt to minimize CPU noise by increasing the length between CPU spikes to 500 milliseconds and 1 second. We can obtain bits in the covert channel but under ideal conditions. For example, if any other work is done in the browser it significantly impacts our ability to discern relevant CPU spikes.

Next, we try our memory throttling primitive. Memory usage is a more deterministic channel and thus less noisy than CPU usage. This makes it more viable as a covert channel. We use our memory throttling primitive to fill a 40MB array and then clear the memory with a `terminate` worker method call. We can successfully send 1 bit per 5 seconds. We send the bits for "hello world" in Figure 4. Unlike the CPU covert channel, the memory covert channel is usable when the user browses the internet or streams videos. This finding is a consequence of the amount of memory used which far exceeds the memory needed to buffer a video in our tests.

We note that our covert channel does not require a web worker. However, when executing the covert channel in the UI thread, the browser is noticeably less responsive due to the looping execution of the memory primitive. Moreover, we find that, unlike workers, we do not have a mechanism to force garbage collection and thus create a clean signal (i.e., discernible peaks in memory). We also implement the covert channel with `setTimeout` and find it to be intractable. Web workers are unaffected by these limitations. Our ability to force garbage

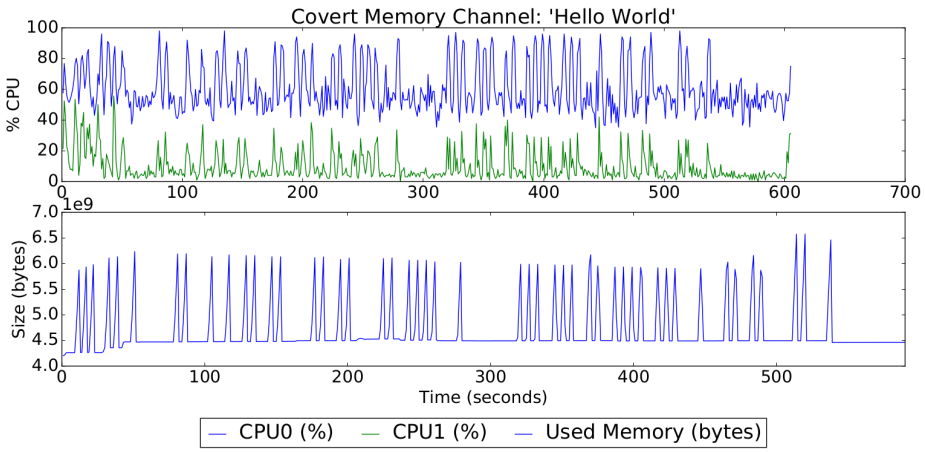

Fig. 4: Memory covert channel sending hello world.

collection by terminating the worker allows for distinct, deterministic memory peaks, as shown in Figure 4.

We implement and test this covert channel on OS X 10.11.4 using the Firefox 46.0.1 and Chrome 50.0.2661.94 browsers. The covert channel is inefficient regarding channel bandwidth; we can send approximately 1 bit per 5 seconds. We can speed this up by reducing the amount of memory throttled (e.g., less than 2GB peaks), by increasing the size of the leak to fill memory faster, and by using multiple workers to concurrently fill swap.

7 Potential Mitigations

The challenge for the Web Workers API is how to inform users a worker is executing, what the intent of the execution is, and how the execution is impacting system resources. We assert that the most effective solution is to provide fine-grained controls for workers similar to browser pop-up controls, and to restrict the Web Workers API in the ECMAScript specification. We envision a system administrator or user with an understanding of computer processes interacting with a dialog box that lists the options: Do not allow any site to execute computationally intensive scripts, Inform me when any site executes computationally intensive scripts (Recommended), Allow all sites to execute computationally intensive scripts.

In the interim, we implement a browser extension to mitigate worker stealthy computations partially. This mitigation is partial because the browser extension only informs the user of when a worker has executed. If the worker is named appropriately, the user is provided with some context of the workers intent, but name mangling and poor coding practices will undo this. We call our browser extension *wAudit*.

wAudit is a Google Chrome content script. Content scripts use the Document Object Model (DOM) to read and modify details of a visited web page. These scripts, however, cannot use or modify variables or functions defined by the visited web page. For wAudit to determine whether a worker exists it must be able to the later.

We programatically inject wAudit as a script into visited web pages using `document.createElement`. This function creates an HTML script element that we append to the document object's root element using the function `document.documentElement.appendChild`. The injected script recursively searches all DOM objects and identifies object types of `[object Worker]`.

The script alerts the user if it finds a worker or workers by drawing a banner at the bottom of the browser window. This banner includes the name of the worker and a UI button for terminating a selected worker. We implement the terminate function by crafting the string `"workers[i]+".terminate()`. This string contains the worker name and the method call to terminate. We call `eval` on the string input to execute.

8 Related work

Security researchers have found numerous vulnerabilities in the HTML5 APIs that enable traditional web application attacks such as CSRF and clickjacking, and HTML5-specific attacks such as cache poisoning and botnets.

Tian et al. [18] show that the HTML5 screen-sharing API can allow for cross-site request forgery (CSRF) attacks, even if the target website utilizes CSRF defenses such SSL and secure random tokens. The authors are also able to sniff user account, autocomplete, and browsing history data because it can be viewed directly on the user's screen.

The HTML5 FullScreen API displays web content that fills the user's entire screen. Aboukhadijeh [3] describes how a malicious website can trick users into clicking a link to a legitimate website (e.g., https://www.bankofamerica.com/), and then display a malicious website in fullscreen.

Kuppan [13] overviews multiple HTML5-specific attacks. For example, an attacker can use the HTML5 Drag and Drop API to trick users into setting target form fields with attacker controlled data, a clickjacking attack. An attacker can poison HTML5 caches designed to enable offline browsing with her own pages that recover user supplied data. Specific to our work, workers enable HTML5 botnets. These botnets can mount distributed denial-of-service (DDoS) attacks by sending cross-domain XMLHttpRequests.

Anibal Sacco et al. [16] use workers to optimize heap-spray attacks. By employing multiple workers, the authors show that they can populate the target systems' memory faster than conventional heap-spray attacks. They leverage HTML5 canvas objects to obtain both full control over consecutive heap pages and to provide byte-level access to pixel information. This gives four bytes per pixel for use in spray contents – typically a use-after-free exploit, heap-based

buffer overflow, or ROP chain. Also, due to the increasing prevalence of browser-based devices with HTML5 support (smartphones, TVs, consoles, etc.) the use of workers as an attack vector are largely platform and browser agnostic.

The Open Web Application Security Project (OWASP) blog [2] mentions the use of workers to perform DoS attacks. The post gives a cursory treatment of these vulnerabilities and does not provide any concrete details regarding implementation, measurement, or countermeasures.

In general, defenses for HTML5 API vulnerabilities include modifications to the APIs. Son and Shmatikov [17] find that many web applications perform origin checks incorrectly, if at all. The lack of stringent checking allows for cross-site scripting (XSS) attacks, as well as data injection into local storage. The authors propose accepting only messages from the origin of the page that loaded a frame and the parent of that frame.

Akhawe et al. [5] find that HTML5 web applications need better privilege separation. Rather than advocate for browser redesign or artificial limits on partitions, the authors propose a way for HTML5 applications to create an arbitrary number of unprivileged components. Each component executes with its own temporary origin, isolated from the rest of the components.

9 Conclusions

We described how the Web Workers API can be used to create workers that enable stealthy computation and covert channels. We demonstrated the feasibility of stealthy computation by implementing a distributed password cracker using workers, a DoS attack against OS X, and a resource depletion attack against Android. We evaluated the feasibility of a covert channel using CPU and memory throttling, and implemented the latter. Lastly, we gave the first mitigation strategy for the misuse of workers.

10 Acknowledgments

This research was funded by the National Science Foundation under award number CNS-1329737. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the sponsors.

References

1. Networked medical devices to exceed 14 million unit sales in 2018. `https://www.parksassociates.com/blog/article/dec2013-medical-devices` (Dec 2013)
2. Html5 security cheat sheet. `https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet#Web_Workers/` (apr 2014)
3. Aboukhadijeh, F.: Using the HTML5 fullscreen api for phishing attacks. `http://feross.org/html5-fullscreen-api-attack/` (Oct 2012), accessed May 27, 2014

4. Akhawe, D., Barth, A., Lam, P.E., Mitchell, J., Song, D.: Towards a formal foundation of web security. In: Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium. pp. 290–304. IEEE Computer Society (2010), http://dx.doi.org/10.1109/CSF.2010.27
5. Akhawe, D., Saxena, P., Song, D.: Privilege separation in html5 applications. In: Proc. 21st USENIX Conference on Security Symposium. pp. 23–23 (Aug 2012), http://dl.acm.org/citation.cfm?id=2362793.2362816
6. Biniok, J.: Hash me if you can - a bitcoin miner that supports pure javscript, webworker and webgl mining. https://github.com/derjanb/hamiyoca (2015)
7. Cabuk, S., Brodley, C.E., Shields, C.: Ip covert timing channels: Design and detection. In: Proceedings of the 11th ACM Conference on Computer and Communications Security. pp. 178–187. CCS '04, ACM, New York, NY, USA (2004), http://doi.acm.org/10.1145/1030083.1030108
8. Clark, S.S., Ransford, B., Rahmati, A., Guineau, S., Sorber, J., Xu, W., Fu, K.: Wattsupdoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices. In: Presented as part of the 2013 USENIX Workshop on Health Information Technologies. USENIX (2013)
9. Glasser, D.: An interesting kind of javascript memory leak. http://info.meteor.com/blog/an-interesting-kind-of-javascript-memory-leak (2014)
10. Group, W.H.A.T.W.: Web workers. http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html (Jul 2014)
11. Hickson, I.: Web workers editor's draft 19 may 2014. http://www.w3.org/TR/workers/ (May 2014)
12. Huskamp, J.C.: Covert communication channels in timesharing systems. Ph.D. thesis, California Univ., Berkeley (1978)
13. Kuppan, L.: Attacking with HTML5. In: Black Hat Abu Dhabi (Oct 2010), https://www.usenix.org/conference/healthsec12/workshop-program/presentation/Chang
14. Lampson, B.W.: A note on the confinement problem. Commun. ACM 16(10), 613–615 (Oct 1973), http://doi.acm.org/10.1145/362375.362389
15. Rowland, C.H.: Covert channels in the tcp/ip protocol suite. First Monday 2(5) (1997), http://firstmonday.org/ojs/index.php/fm/article/view/528
16. Sacco, A., Muttis, F.: Html5 heap sprays, pwn all the things (2012), https://eusecwest.com/speakers.html, eUSecWest
17. Son, S., Shmatikov, V.: The postman always rings twice: Attacking and defending postmessage in html5 websites. In: Proc. 20th Annual Network and Distributed System Security Symposium (NDSS). The Internet Society (2013), http://dblp.uni-trier.de/db/conf/ndss/ndss2013.html#SonS13
18. Tian, Y., Liu, Y.C., Bhosale, A., Huang, L.S., Tague, P., Jackson, C.: All your screens are belong to us: Attacks exploiting the HTML5 screen sharing api. In: Proc. 35th Annual IEEE Symposium on Security and Privacy (SP 2014) (May 2014)
19. Wu, Z., Xu, Z., Wang, H.: Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In: Proceedings of the 21st USENIX Conference on Security Symposium. pp. 9–9. Security'12, USENIX Association, Berkeley, CA, USA (2012), http://dl.acm.org/citation.cfm?id=2362793.2362802

Appendix: Health and Medical Systems

Health and medical systems are increasingly becoming networked. An industry report by Parks Associates predicts that networked medical systems will exceed

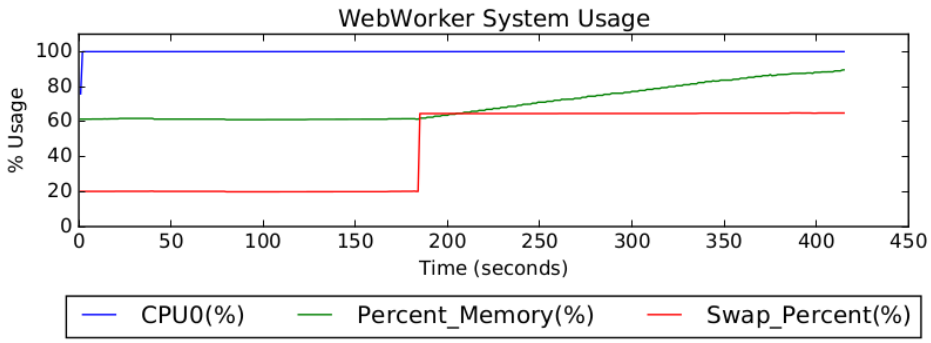

Fig. 5: Stealthy computation on Baxa ExactaMix.

14 million sales in 2018 [1]. These medical systems often employ commodity operating systems such as Windows Embedded and can access and be accessed over the internet.

We investigate the effects of running stealthy computation on Baxa ExactaMix. The Baxa ExactaMix is an embedded health and medical system that mixes total parenteral nutrition and other multi-ingredient solutions. The compounder runs Windows XP Embedded 2002 Service Pack 2 and has a 664 MHz VIA C5 x86 CPU with 496 MB of memory [8]. It also has Internet Explorer version 6.0, which does not support HTML5 APIs. However, since the Baxa ExactaMix can access the internet, we can install a modern browser. We installed Firefox 29 at the time of this experiment. We note that modern medical systems use more recent operating systems and thus support Web Workers without installing a third-party browser.

In our experiment, we first start the Baxa ExactaMix and wait for it to run its clinical software. We then begin measuring the CPU, memory, and swap usage of the device to establish a baseline of activity. Next, we launch Firefox and navigate to a website that we control. This website uses a worker to perform our stealthy computation, specifically, the DoS attack we describe earlier in Section 5. We continue our measurements for 3 minutes.

Results. We note a clear delineation between pre- and post-worker computation in Figure 5. Memory and swap usage are at 60% and 20%, respectively, when the Baxa ExactaMix first starts. As this is a single-core device, the CPU utilization remains high for the entire experiment because all processes are scheduled to execute on the same core. We note linearly increasing memory usage and a near-instantaneous spike in swap usage to 60% when we visit our website that performs the stealthy computation.

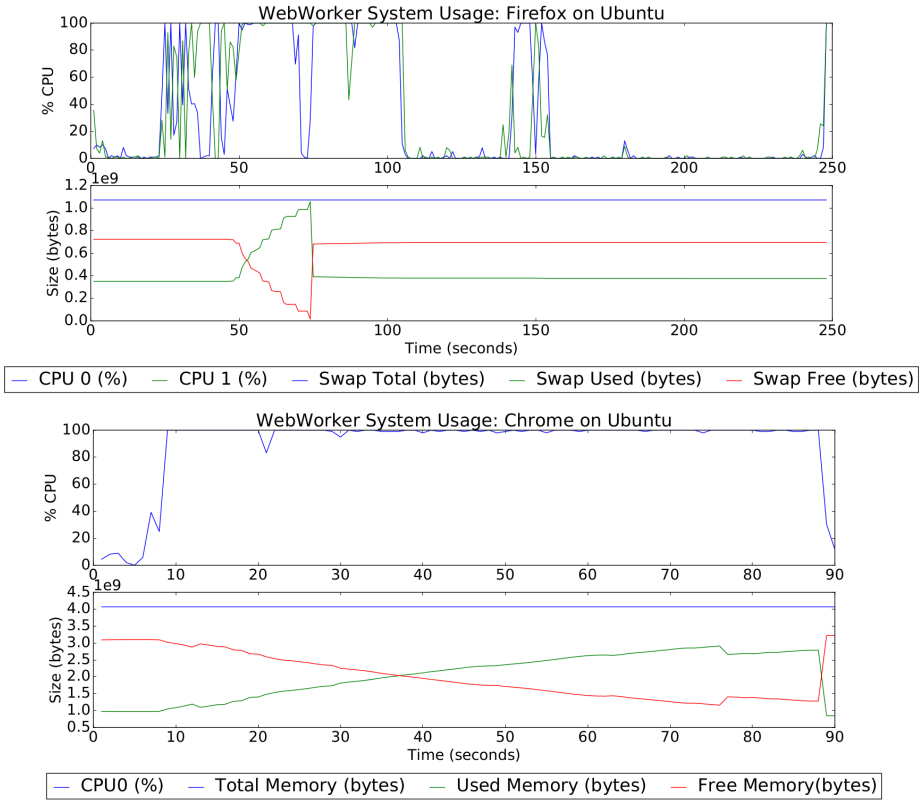

Fig. 6: Stealthy computation on Ubuntu 15.10 using Firefox and Chrome.

Appendix: Linux Stealthy Computation

We experiment with stealthy computation on other operating systems. We find that Chrome 48.0.2564.103 and Firefox 41.0.2 in Ubuntu 15.10 both allow stealthy computation using web workers. Figure 6 illustrates CPU and memory throttling in Chrome and Firefox. We can use these primitives to implement our covert channel as described in Section 6.

We also test our DoS attack described in Section 5.1. This attack does not work in Ubuntu, and Linux in general, because of how virtual memory and processes are managed. Specifically, virtual memory consists both of RAM and swap space. Swap space is managed as a file or partition on the hard disk, and holds inactive memory pages. We fill the swap to its maximum allowed space and note that the system becomes unresponsive. However, modern Linux distributions will terminate processes that consume resources, thus, we notice that free memory decreases and then rapidly increases when the process is killed in Figure 6.