Isolated and Exhausted: Attacking Operating Systems via Site Isolation in the Browser

Matthias Gierlings, Marcus Brinkmann, and Jörg Schwenk, *Ruhr University Bochum*

https://www.usenix.org/conference/usenixsecurity23/presentation/gierlings

This paper is included in the Proceedings of the
32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Isolated and Exhausted: Attacking Operating Systems via Site Isolation in the Browser

Matthias Gierlings, Marcus Brinkmann, Jörg Schwenk
Ruhr University Bochum

Abstract

Site Isolation [12, 40] is a security architecture for browsers to protect against side-channel and renderer exploits by separating content from different *sites* at the operating system (OS) process level. By aligning web and OS security boundaries, Site Isolation promises to defend against these attack classes systematically in a streamlined architecture. However, Site Isolation is a large-scale architectural change that also makes OS resources more accessible to web attackers, and thus exposes web users to new risks *at the OS level*.

In this paper, we present the first systematic study of OS resource exhaustion attacks based on Site Isolation, in the *web attacker model*, in three steps: (1) first-level resources directly accessible with Site Isolation; (2) second-level resources whose direct use is protected by the browser sandbox; (3) an advanced, real-world attack. For (1) we show how to create a *fork bomb*, highlighting conceptual gaps in the Site Isolation architecture. For (2) we show how to block all UDP sockets in an OS, using a variety of advanced browser features. For (3), we implement a fully working DNS Cache Poisoning attack based on Site Isolation, building on (2) and bypassing a major security feature of DNS. Our results show that the interplay between modern browser features and older OS features is increasingly problematic and needs further research.

1 Introduction

Site Isolation [40] is a security architecture for browsers that provides strong isolation for websites and thus mitigates risks from JavaScript, in particular, remote code execution by sandbox compromises [43] and microarchitectural side-channels like Spectre [26]. These benefits are achieved by performing all rendering and script execution from different *sites* in distinct processes, leveraging process security of the OS (Figure 1). Site Isolation is not without costs. The additional processes required to implement Site Isolation obviously consume system memory and CPU time. This overhead was minimized in Google Chrome through careful usage analysis and complex optimizations [40].

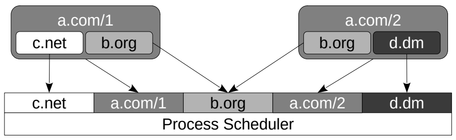


Figure 1: Site Isolation optimizes how sites referenced within web pages are mapped to OS processes (cf. Subsection 2.1).

Risks of Site Isolation. With Site Isolation, browsers now share responsibility with the OS for the allocation of computing and network resources. But in contrast to most local applications, a browser can be remotely controlled by a *web attacker* through the execution of malicious JavaScript code. Thus a remote, off-path web attacker may interfere with the local OS, using a browser with Site Isolation as intermediary.

Site Isolation was introduced to protect web applications from attacks *leveraging* the underlying OS, and we think that this goal has been achieved. Our work thus targets the opposite direction to answer the following research question:

Does Site Isolation make operating systems more vulnerable to web attacks?

Attacker Model. We use a weak attacker model, the *web attacker model* [2, Sec. II B]. According to [2], web attackers have no special network privileges – being off-path they can not observe, modify or block traffic between other parties on the Internet. However, web attackers can set up and control their own server infrastructure and have "root access" [2] to these servers. When choosing a provider that allows IP Spoofing, attackers can use utility programs such as `iptables` to configure spoofed IP addresses (cf. Subsection 6.1). According to [33], over a quarter of the ASes investigated allowed IP spoofing on egress, and two-thirds on ingress. To start the attack, the victim only needs to visit a web page hosted on one of these servers.

In contrast to the (stronger) *malware attacker*, who can directly access OS resources through native code, the web attacker is only allowed to use standard browser APIs. So more precisely, we ask the following research question:

> *Is a web attacker able to directly control OS resources when Site Isolation is enabled, thereby bypassing the current browser sandboxes?*

This research excludes (trivial) attacks on the browser itself. For example, we are interested in DoS attacks against the OS, but not in DoS attacks against the browser process alone.

Bypassing Browser Sandboxes. Controlling OS resources is fairly simple in the malware attacker model, but a web attacker is restricted by the intentional security boundaries of the browser sandbox. This sandbox limits access to resources by consumption quotas (e.g. HTML Web Storage), asserting implicit authorization (e.g. by a trusted event), or asking for explicit consent (e.g. through popups). We show how to overcome these obstacles for a web attacker in three steps:

1. First-Level Resources. We show how to use Site Isolation to implement a *fork bomb* DoS attack in JavaScript that circumvents browser watchdogs against simple DoS attacks on CPU and memory. This is a direct consequence of Site Isolation since extra processes are spawned for all *sites* included in the browser window. However, we show how to optimize this process by using IPv6 addresses as sites (Table 1, 1.).

2. Second-Level Resources. Second-level OS resources are resources that can not be allocated directly through JavaScript or web objects. In this work, we analyze UDP network sockets. Single network sockets can be opened easily, e.g. by accessing a QUIC-enabled webserver, but through techniques like multiplexing and timeouts, the browser sandbox prevents the opening of extra sockets. Moreover, many open sockets typically cause noisy network traffic and increase the CPU load until the browser becomes unresponsive. We use novel techniques exploiting WebRTC (Table 1, 2.–5.) to circumvent these sandbox and performance restrictions. Combined with the simplified method to spawn new processes, this allows us to block *all* available UDP source ports on a victim system.

3. Advanced Attack. We show how to use Site Isolation and the attacks on first- and second-level resources to implement DEMONS, a Cache Poisoning attack against the OS DNS resolver cache, in the web attacker model. We use the fact that we have blocked all UDP ports, in combination with the techniques from Table 1, 6.–8., to release a pair of UDP ports and learn their port numbers, which must then be used by the OS for DNS. Our off-path attacker is thus able to circumvent UDP port randomization, a major DNS security feature. We evaluate this attack twice: in a realistic Internet-based setup with poor network quality, achieving a success rate of 37%, and in a lab environment.

Known Attacks. Attacks on the OS in the web attacker model are rare, as JavaScript is an interpreted language that is strongly contained by the browser sandbox. The arguably most severe attack is Rowhammer.js, which uses malicious JavaScript to inject faults into neighboring system memory cells at the hardware level, bypassing all OS and sandbox memory access restrictions [17]. Another series of attacks uses side-channels to leak private data from other processes or the OS, such as memory deduplication [16], keystroke interrupts [31], and memory caches [15, 38, 46]. These techniques have in common that they do not attack the sandbox mechanism, but instead target the machine hardware directly.

More common are attacks against the browser sandbox itself, such as JIT spraying [14] or fuzzing [49]. After achieving native code execution within the sandbox process, the attacker may escalate the attack to the OS in the malware attacker model. One side-channel attack that leaks data from the sandbox is an implementation of Spectre in JavaScript [44].

In 2008, Dan Kaminsky [24] showed that in the web attacker model, DNS security can be broken by DNS Cache Poisoning. UDP port randomization was implemented as the only countermeasure against the Kaminski attack. Previous attacks defeating UDP port randomization relied on IP fragmentation and timing side-channels [35, 45, 50]. These attacks target intermediate devices like home routers or DNS servers directly. Client-side DNS Cache Poisoning attacks target end-user devices such as desktop computers and laptops. They have been analyzed by Alharbi et al. [3], but only in the stronger malware attacker model.

Main Insights. The main result of this paper is that with novel browser features, the boundary between the browser and the OS becomes weaker. Attacks that were previously only known in the malware attacker model may now become feasible in the web attacker model, allowing remote, off-path web attackers to compromise the OS. Our research only reveals the tip of the iceberg and future work may show that the mitigations introduced by browser vendors were insufficient.

1. Fork Bomb and UDP Port Blocking. Despite the optimization efforts by Google, the *site* concept is still too fine-grained and allows for effective DoS attacks, such as the fork bomb. With Chrome and Edge, it was possible to use Site Isolation and WebRTC to block all UDP ports in Windows with a single *visible* browser window. The root cause is that each IP (v4 or v6) address still counts as a separate site. The main difference between the tab process isolation introduced in 2009 and Site Isolation is that tab isolation limits OS resources per *visible* window. At the same time, opening additional windows through pop-ups was limited by introducing trusted events. Thus, users were able to control the resource consumption of web applications via the visible components of

	Objective	Malware Attacker	Web Attacker	Sec.
1.	**Create many processes**	Use standard OS API to fork processes.	Creates sites using IPv6, bypassing SI process consolidation.	new: 3.1
2.	**Allocate UDP Ports**	Use standard OS API to create sockets.	Indirectly via WebRTC connections.	new: 3.2
3.	**Keep Connections Alive**	Control socket lifetime over OS API.	Use pending connections and data streams.	new: 3.2
4.	**Avoid Network Traffic**	n/a	Use local WebRTC connections.	new: 3.2
5.	**Avoid DoS on CPU**	n/a	Use stream demultiplexing and munging.	new: 3.2
6.	**Find DNS Query Port**	Use standard OS API to observe ports.	SDP offer analysis, and exhaust & single release.	new: 5.1, A.1
7.	**Leak DNS Query Port**	Use standard OS API to leak over network.	Use standard browser API to leak over network.	well-known
8.	**Trigger DNS Request**	Use standard OS API to start DNS lookup.	Indirectly via XMLHttpRequest.	well-known

Table 1: DEMONS combines Site Isolation (SI) with eight additional browser techniques, six of these novel, to bypass the sandbox. Together these techniques, which are accessible in the web attacker model, replace the local malware attacker in [3].

the browser. With Site Isolation, this is no longer the case. We therefore propose, implement, and evaluate a concept to limit resource consumption rooted in the visible components.

2. DNS Cache Poisoning. We show that a web attacker can not only block all available UDP ports but also release a single pair of known ports, defeating UDP port randomization. In response to our findings, Chrome and Edge now limit the number of UDP ports that can be allocated globally by the browser (i.e. across all windows and tabs) to 6000, so that UDP port randomization remains effective at the OS level. While this mitigates the DEMONS attack this may not be sufficient in the future. When UDP port randomization was introduced, the designers considered adding 16 bits of randomness *for the DNS resolver alone* to reduce the success probability of a web attacker to one in 2^{32}. However, Windows only has 2^{14} freely available UDP ports, *shared among all processes*, resulting in a success probability of one in $2^{16+14} = 2^{30}$. Our web attacker could control all but two of these ports, with a success probability of one in $2^{16+1} = 2^{17}$. Even with the global limit, we still could control slightly less than 6000 ports, enhancing the success probability to one in $2^{16} \cdot (2^{14} - 6000) \approx 2^{29.34}$. This may still allow for future attacks. As a lasting countermeasure, we think that a critical re-evaluation of the OS socket API is necessary since the current API is not designed to be used as an entropy source.

Contributions. We make the following contributions:

1. We describe how Site Isolation in browsers can be exploited for novel resource exhaustion attacks against the client OS by a *web attacker* (Section 3). We provide an evaluation of these attacks and show that they can be used to implement DoS attacks against the operating system or the web browser (Section 4).
2. To show that possible attacks go beyond DoS, we implement DEMONS, a DNS Cache Poisoning attack that stealthily poisons the DNS cache of the Windows operating system, in the *web attacker model*. (Section 5). We evaluate DEMONS on the Internet and in a lab setting (Section 6). We show that under real-world conditions, DEMONS has a success rate of 37%. In the lab, we compare DEMONS to a malware-based attack, which has a slightly better success rate.
3. We identify conceptual weaknesses in the Site Isolation architecture and discuss countermeasures against resource exhaustion attacks based on Site Isolation, as well as mitigations to the DEMONS attack. Specifically, we develop, implement, and evaluate an efficient mitigation to resource exhaustion attacks (Section 7).

Responsible Disclosure. We reported our findings to Google, Microsoft, and Mozilla. Google assigned CVE-2020-6557 and now limits the number of allocated UDP sockets across all renderer processes. Microsoft also adopted this solution in the Chromium-based Edge browser. Google has also awarded a bug bounty to the authors for their findings.

Artifacts. All artifacts are available as Open Source.[1]

2 Background

2.1 Site Isolation

A decade ago, all major browsers abandoned the single process paradigm and used separate processes for the rendering of different browser windows and tabs. Content from different *sites* however was still rendered in the same process, e.g. when a cross-origin iframe was embedded in the webpage.

Site Isolation [40] improves content isolation based on process separation significantly because a new process is created for every *site*. For example, if a web page contains a cross-site iframe, at least two processes are used for rendering. The *site* concept is more coarse-grained than the better-known *web origin* concept: To extract the site from a web origin, only the protocol and the main domain are considered, subdomains and port numbers are omitted. For example, `https://a.com:4444` and `https://b.a.com` refer to different web origins but the same site. Sites referenced by IPv4 or IPv6 addresses instead of a domain name are considered distinct sites rendered in separate processes.

Since each process induces overhead in the OS, Site Isolation in Chrome has been optimized to reduce the total number of processes (Figure 1) with *process consolidation*. Suppose

[1] https://git.noc.rub.de/gierlmds/isolated-and-exhausted

two windows (or tabs) are open in the browser, where the document is loaded from the same site `a.com`. For each of these windows, a separate process is started. Additional processes are started for each iframe loaded from a different site; however, if the same site is loaded into iframes in two different windows (e.g., site `b.org` in Figure 1), only a single process is running which renders both iframes.

Site Isolation has been implemented by Google Chrome, which recently also became the base for Microsoft Edge. Mozilla rolled out their own Site Isolation implementation with Firefox 94 [13].

2.2 Exhaustible OS Resources

The OS manages the *resources* of a computing device, such as CPU time, main memory, and network sockets. Benign applications like browsers should cooperate with the OS to achieve a fair sharing of resources with other benign applications. Malware, on the other hand, may refuse cooperation and may try to use or block as many OS resources as possible.

Processes. OS processes are commonly identified by their globally unique *process ID (PID)*, which on many systems is a 32-bit integer. However, process creation is resource intensive, so RAM and CPU will be overloaded long before the system runs out of PIDs. A common attack on process-related resources is a *fork bomb* [5], which is a program that recursively spawns an exponentially growing number of clones. The attacker's goal is to overload the system to the point where it becomes unresponsive, e.g. due to memory page swapping or task scheduling latencies.

Network Sockets. A TCP or UDP *network socket* is abstractly defined as a 4-tuple $(\text{IP}_\text{dest}, \text{Port}_\text{dest}, \text{IP}_\text{src}, \text{Port}_\text{src})$ which identifies a network connection between two endpoints after a packet is received. However, in practice, the creation of *operating system sockets* through the Berkeley socket API (used in Windows, macOS, and Linux) is a multi-step process where often some parts of the 4-tuple are left undefined until a packet is fully transferred. As a consequence, the OS makes some simplifying assumptions. In particular, a source port number is reserved independently of the destination IP and port number. If one application allocates a socket for a specific local port number, no other application can allocate another socket for that port using the Berkeley socket interface. This can lead to port number exhaustion because only a small subset of possible 4-tuples is available to applications.

We note that TCP and UDP port numbers do not share the same namespace, nor do IPv4 and IPv6. In some cases, applications use dual-stack allocations to register IPv4 and IPv6 port numbers at the same time. We mainly consider UDP in this work, because it is an attractive target for packet injection, while TCP connections are already protected at the OS level by sequence numbers with a random start value.

System Ports, User Ports, and Ephemeral Ports. There are 65536 ports for each combination of TCP/UDP with IPv4/IPv6. Port numbers are grouped into three distinct use cases [7]. *System ports* (0–1023) are associated with well-known internet services (e.g., 53 for DNS). *User ports* (1024–49151) may be statically assigned for custom applications. *Ephemeral ports* (49152–65535) are used by clients for a single connection, such as a DNS query. Usually, the OS picks an arbitrary *unallocated* number from the ephemeral port range. Once a port number is bound to a socket, it uniquely identifies the socket over its lifetime. Actual port ranges can deviate from the above standards. For example, Linux typically uses 32768–60999 for ephemeral ports.

2.3 Domain Name System (DNS)

DNS is used for *name resolution*, a query-response protocol to translate domain names to IP addresses. We assume that the web browser uses the DNS resolver of the OS, which is configured with the IP address IP_NS of a default name server.

This is how a domain is resolved: 1. If the cache contains the IP address of the domain, it is returned. 2. Else, the resolver creates a UDP socket $S = (\text{IP}_\text{NS}, 53, \text{IP}_\text{src}, \text{Port}_\text{src})$, where 53 is the default port for DNS, IP_src is the resolver's external IP address, and Port_src is a random ephemeral port chosen by the OS for this connection. 3. The resolver sends a query to the name server over S, including a random 16-bit transaction ID (TXID). 4. The name server receives the query, and sends a response including the TXID. 5. The resolver receives the DNS response and verifies its content and the TXID. If the response is valid, it is cached up to its time-to-live (TTL), and the result is returned to the browser. 6. If the response is invalid, the resolver discards it. In Windows, the procedure is repeated from the first step up to five times, after which an error is returned. In Linux and macOS, the procedure is repeated from step 5 until a valid response is received or a timeout occurs.

The queried name server can either return the (authoritative or cached) result directly, recursively query another name server or indicate in the result that the client should iteratively query another name server.

DNS Cache Poisoning. In 2008, Dan Kaminsky [24] discovered a DNS Cache Poisoning off-path attack on name servers performing a recursive lookup to an authoritative name server by brute-forcing the 16-bit TXID of the request and sending a spoofed response with a malicious IP address. If the attacker can guess the correct TXID before the answer of the authoritative name server arrives, the victim name server caches the malicious entry, i.e., its cache is now poisoned. The primary mitigation for the Kaminsky attack is source port randomization (SPR) [6]. The goal is to increase the entropy of DNS queries, making it harder for an off-path attacker to successfully spoof a DNS response. Other countermeasures, such as 0x20 encoding [48], exclusive DNS over TCP [10], or DNS over HTTPS [20], are not as widespread due to compatibility concerns. Recently, different techniques to circumvent

SPR have been proposed: IP defragmentation [35, 45, 50] and blocking client OS source ports [3].

3 Resource Exhaustion Attacks Based on Site Isolation

Typically, benign applications only spawn a fixed number of processes to cooperate with the OS. Web browsers are an exception: they create a new process for each window or tab that is opened. In theory, this enables web browsers to allocate arbitrarily many resources, in practice, however, there is a limit on the number of windows and tabs that can be opened automatically by a malicious web application, and a trusted event (e.g. a mouse click) is needed to get the permission to open more. In contrast, if a user manually opens dozens of windows, the OS or browser should not prevent that, as the expressed intent of the user action implies authorization to allocate these resources. In a user study from 2009 [11], the maximum number of simultaneously open tabs was 42.

With Site Isolation, this simple relationship between windows or tabs on the one hand and OS processes on the other hand no longer holds. Instead, a web browser supporting Site Isolation may now open several processes per window or tab without user interaction. Major efforts have been made [40] to limit resource use even with Site Isolation, but in this paper, we show that these efforts can still be circumvented.

Furthermore, before Site Isolation, the browser maintained control over the allocation of secondary resources, such as network sockets allocated through WebRTC connections, by limiting their number per process. As the number of processes was bound, so was the number of secondary resource allocations. However, with the ability of the web attacker to allocate an arbitrary number of processes, limits on secondary resources can also be overcome by exploiting a combination of novel Site Isolation features, edge case configurations, and implementation bugs. In this section, we present the general ideas behind our evaluation. A detailed description can then be found in Section 4.

3.1 First Level Resource Exhaustion: Fork Bomb

Create Processes by Creating Sites. With Site Isolation, a web-attacker has the ability to create an arbitrary number of processes, despite the optimizations and sandbox restrictions in the browser. This can be used to perform a browser-based DoS attack that works similar to a fork bomb, but does not require shell access. The root cause for this issue is that an attacker can easily create many *sites* (Subsection 2.1) through the use of distinct domain names or IP addresses, and each site is rendered in a different process.

Attack Outline. An attacker hosts a malicious webpage which is assigned a large number N of IP addresses $IP_1, \ldots, IP_N$. The webpage (recursively) contains a total of N iframes with the source attribute set to `http://[IPi]`, for $i = 1, \ldots, N$. The webpage itself contains one iframe with its source attribute pointing to IP_1, and each loaded iframe contains two other iframes pointing to different IP addresses. With Site Isolation, loading this web page creates N processes on the victim system, leading to a fork bomb in the OS. In our implementation (Subsection 4.1), we use IPv6 addresses.

Novelty. While it is possible to manually assign domain names or IP addresses to a webserver, we implemented a much faster method using *non-local binds*. Non-local binds are an advanced feature of the Linux kernel IP stack that allows the server to listen to many IP addresses without assigning them to the network interface one by one.

3.2 Second Level Resource Exhaustion: UDP Port Exhaustion

Blocking UDP Ports via Browser APIs. We know about two browser APIs that can be used to block UDP ports from web pages: QUIC and WebRTC. Initial experiments with QUIC were inefficient due to the high computational cost associated with a large number of parallel QUIC handshakes. Thus, we focused on WebRTC.

WebRTC. WebRTC is an open web platform for real-time communication in telephony and video conferencing applications. Essentially, it gives websites access to audio and video peripherals (camera, microphone), and provides an API to stream the data from these devices to other endpoints supporting WebRTC using UDP or TCP. Metadata is exchanged using the Session Description Protocol (SDP [18], see Figure 8 for an example). The format is highly flexible and allows both ends to negotiate the number and type of media channels (audio, video, or data), possible communication endpoints (e.g., P2P, or use of a TURN server), and multiplexing options. In our attack implementation, we exclusively use data channels because video and audio channels require explicit permission from the user and consume more resources, increasing the footprint of the attack unnecessarily.

Local WebRTC in Offer State. Commonly, SDPs are exchanged between endpoints through a *signaling service*. Our attacks solely rely on local endpoints and thus do not involve a signaling service. Because we never complete any WebRTC handshake, we also do not need a peer object. Instead, we only create *local WebRTC objects*, put them into the offer state so that they allocate some UDP ports in preparation for the handshake, and then let the objects sit idle, keeping only a reference to prevent garbage collection. From our experiments, this is the most lightweight way to use WebRTC objects for port allocation, although other configurations might also work.

WebRTC Objects Allocate an Even Number of Ports. During the initial examination of individual WebRTC objects with a single data channel, denoted by WebRTC$^{[p]}$ in Table 2, we found that a single (non-multiplexed) data channel in a WebRTC object allocates *not one but two* UDP ports: one for interactive connectivity establishment using the ICE/STUN protocol and one for data transfer using SCTP-over-DTLS. ICE/STUN can not be disabled in the browser because it is also used to verify communication consent (see section 4.2 in [41]) and thus serves as a security mechanism. In consequence, WebRTC objects can not allocate ports individually but only in pairs.

WebRTC Data Channels and Multiplexing. Chrome allocates a thread for every WebRTC object, causing a high load with many WebRTC objects. Thus, we looked for ways to reduce the number of WebRTC object creations for the same number of allocated UDP ports.

Our measurement results in Table 2 document the effect of adding multiple data channels to the same WebRTC object, denoted by WebRTC$^{[u]}$. Simply adding data channels did not lead to more port allocations, because by default all data channels are multiplexed over the same connection.

However, multiplexing can be disabled for WebRTC. A feature of the WebRTC programming interface allows JavaScript to edit (or *munge*) the SDP generated by the browser locally before offering it to the receiving end. Based on this insight we made two modifications: First, we disabled multiplexing by removing the `BUNDLE=0` option [21] from the SDP. Second, we added copies of the data channel with their own unique identifiers `mid` (see Figure 8). We denote the resulting WebRTC objects with WebRTC$^{[m]}$ in Table 2.

Attack outline. The attacker allocates many first level WebRTC objects until an error message indicates that the per-process limit has been reached. Depending on the SDP, each WebRTC object causes the allocation of two or more UDP port numbers at the second level. Using Site Isolation, the attacker can then scale up the attack by repeating it in multiple processes, leading to resource exhaustion of the ephemeral UDP port table in the OS.

Novelty. We describe new methods to stealthily block many UDP ports via browser APIs. Our technique involves the (mis-)use of WebRTC, using data streams to avoid detection, pending connections to keep the ports blocked, and loopback connections to avoid network traffic. Disabling multiplexing through munging reduces load on the victim system while simultaneously increasing the attack speed.

4 Evaluation of Resource Exhaustion Attacks

We evaluated the effect of Site Isolation on first and second-level resource exhaustion attacks against Windows and Linux. For Windows 10 (1909 Build 18363.815), we used the production version of two popular web browsers, Google Chrome (83.0.4103.106) and Microsoft Edge (83.0.478.45, based on Chromium), as well as the development version of Firefox (Nightly 86.01a) that implements an experimental prototype for Site Isolation called Project Fission [12]. For Linux (Kubuntu 18.04.5 LTS), we used Chromium (83.0.4103.106), which is the Open Source version of Chrome, and Firefox (Nightly 86.01a). Edge is not available for Linux, so we had to exclude it from that platform. An overview of the results can be found in Table 2. Yellow cells indicate settings where intentional browser limits could be bypassed. Red cells with strong borders indicate successful attacks (either fork bomb or UDP port exhaustion).

4.1 Fork Bomb Evaluation

We measured the number of processes created while the browser attempts to render the iframe tree until the browser crashes, the OS becomes unresponsive, or no new processes are created. Both Windows and Linux can use disk space as virtual memory, which may change the number of processes that can be created in a system. To evaluate this, we repeated the measurement with "swap off" and "swap on". Windows dynamically calculates the swap size based on the disk size, so we included two different disk configurations. On the other hand, Kubuntu Linux uses a fixed 1 GB swap partition by default. For every combination of (browser, OS, swap configuration), the measurement was repeated five times, and Table 2 shows the median number of created processes.

Without Site Isolation, only a small number of processes were created, and we could not overload the browser or OS for any of the tested browsers on Windows or Linux.

With Site Isolation enabled, we could reliably crash the browser (ϟ) or even – in more than half of the cases – make the operating system unusable (💣) (see Appendix C).

4.2 UDP Port Exhaustion Evaluation

Chrome and Edge without Site Isolation. The measured results for Chrome were identical on Windows and Linux. Every renderer process allows the creation of at most 500 WebRTC objects at the same time. With un-munged WebRTC$^{[p]}$ or WebRTC$^{[u]}$ objects, we can allocate two UDP ports per WebRTC object, of at most 1000 UDP ports per renderer process (window or tab). Using a munged WebRTC$^{[m]}$ object, we bypass this limit and allocate up to 3000 UDP ports per renderer process.

Chrome and Edge with Site Isolation. Since we were able to allocate 3000 ports per process, we expected that this number can be multiplied by the number of site-based processes. On Windows, this strategy succeeds in exhausting the UDP ephemeral port range at the OS level completely (at most one open port due to allocation in pairs), using any of the WebRTC object variants (●). On Linux, we also could exceed the browser allocation limit for UDP ports using any of the WebRTC variants, and allocate about 8000 UDP ports (◐) instead of 3000 (◔). However, at that point the browser entered a failure state, where no more ports could be allocated until the browser was restarted.

OS		Browser	Chrome[3]/Chromium[4]			Edge[5]			Firefox[6]		
		Sites	Single	Multiple		Single	Multiple		Single	Multiple	
		Site Isolation	-	off	on	-	off	on	-	off	on
Windows[2]	Processes	swap large[a]	8	5	837 ϟ💣	7	5	822 💣	10	7	876 ϟ💣
		swap small[b]	8	5	522 ϟ💣	7	5	514 ϟ💣	10	7	457 ϟ
		swap off	8	5	275 ϟ💣	7	5	267 ϟ	10	7	200 ϟ
	Sockets	WebRTC[p]	○	○	●	○	○	●	○	○	○
		WebRTC[u]	○	○	●	○	○	●	○	○	○
		WebRTC[m]	◔	◔	●	◔	◔	●	-	-	-
Linux[1]	Processes	swap on[c]	12	10	435 ϟ💣	-	-	-	9	6	233 ϟ
		swap off	12	10	446 💣	-	-	-	9	6	271 ϟ
	Sockets	WebRTC[p]	○	○	◐	-	-	-	○	○	○
		WebRTC[u]	○	○	◐	-	-	-	○	○	○
		WebRTC[m]	◔	◔	◐	-	-	-	-	-	-

🟨 Allocation over intentional browser limits. 🟥 Exploitable in a fork bomb or DEMONS attack.

ϟ The browser crashes. 💣 The operating system becomes unusable.

Ports blocked: ○ $\leq$ 10% ◔ $\leq$ 25% ◐ $\leq$ 50% ● $\approx$ 100% (at most one open port due to allocation in pairs)

WebRTC objects: [p] with a single data channel, [u] with multiple data channels, [m] with munging.

OS versions: [1]Windows 10 (1909 Build 18363.815), [2]Kubuntu Linux 18.04.5 LTS (Kernel 5.4.0-62)

Swap configuration: [a]automatically managed (240 GB disk), [b]automatically managed (64 GB disk), [c] 1 GB swap partition.

Browser versions: [3]Chrome 83.0.4103.106, [4]Chromium 83.0.4103.0, [5]Edge 83.0.478.45, [6]Firefox Nightly 86.01a

Hardware configuration: Dell Latitude 5280, Intel Core i5 7200U, 8 GiB RAM, 240 GiB M.2 SATA SSD

Table 2: Site Isolation resource allocations in browsers and their adverse consequences. The columns show different browsers, attack variants (single vs. multiple sites), and Site Isolation configurations (off/on). The rows describe the OS, resource type, and variant. Table cells for processes show the maximum number of processes we could allocate, and a symbol indicating if crashes of the browser and/or OS were observed (cf. Appendix C for details). Table cells for sockets describe the percentage of the UDP sockets that could be allocated. For example, in Windows with a small swap space configuration, we observed that Firefox with Site Isolation visiting the multi-site attack allocated 457 processes, and then crashed. As another example, in Windows with Chrome, WebRTC[m] could be used to bypass browser limits for socket allocation even with a single-site attack, but Site Isolation and a multi-site attack are required to allocate enough sockets for a DEMONS attack.

Firefox. In contrast to Chrome and Edge, Firefox validates the munged SDP and rejects our two modifications with an error message. This means we had to exclude WebRTC[m] objects from our evaluation for Firefox.

As for the total number of WebRTC and UDP port allocations, Firefox globally limits the total number of allocated UDP ports to 1000 across all browser processes, regardless of Site Isolation. Thus with Firefox, UDP ports in the OS can not be exhausted (○).

5 Advanced Attack: DNS-Poisoning by Exhaustive Misappropriation of Network Sockets (DEMONS)

DEMONS is a novel Cache Poisoning attack against the DNS resolver of the client OS, in the web attacker model. DEMONS disables UDP port randomization by blocking all client-side UDP ports except two, and by informing the poisoner about these open ports (see Figure 2). Disabling UDP port randomization was introduced by Alharbi et al. [3] in an unprivileged malware attacker model. Table 1 summarizes the difference between their work and ours.

DEMONS consists of two phases. In the *setup phase*, source port randomization is disabled through second-level resource exhaustion (3.2, 4.2). During the *poisoning phase*, malicious entries are injected into the DNS resolver cache of the victim's client OS. Only the first phase is novel, the second phase is similar to other DNS Cache Poisoning attacks, such as [3, 24, 35]. We only evaluate Windows 10 as a client OS in this work, and refer to [3] for how to treat differences in Linux and macOS.

Architecture. The infrastructure required by the attacker consists of the following components (see Figure 3):

1. *Web Server:* The web server hosts the malicious web page that will be delivered to the victim's browser.
2. *Poisoner:* A system that, upon receiving a signal from the attacker's web application, sends a large number of spoofed DNS responses with randomly chosen TXIDs to the victim. Optionally, multiple poisoners can run simultaneously.
3. *Malicious Server:* The system whose IP address is inserted into the victim's DNS cache under the target domain. After a successful attack, the malicious server can impersonate the benign target server to the victim.

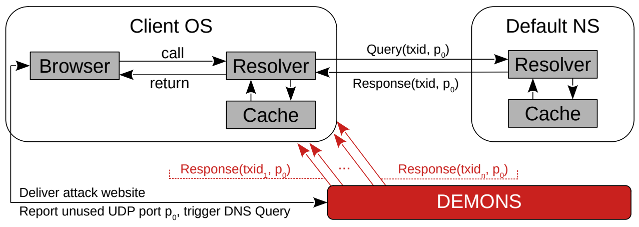


Figure 2: Resolution of `www.example.com` and the off-path DEMONS attacker sending responses to the victim.

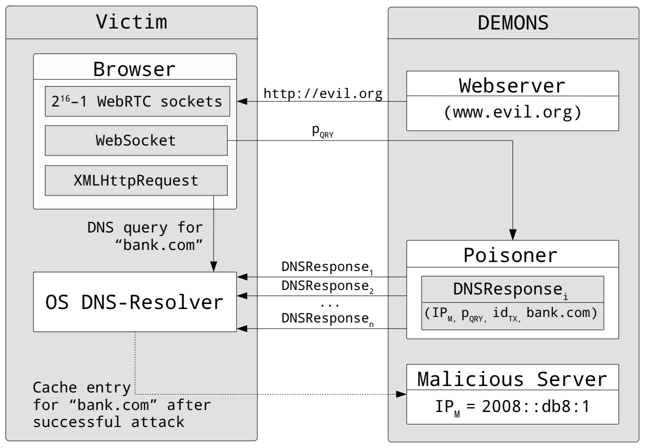


Figure 3: Architecture of the DEMONS attacker.

Limitations. DEMONS requires the attacker to impersonate a benign default name server by spoofing the source IP address in forged DNS responses. This is easy for an attacker to achieve because many providers do not filter IP spoofing [33]. However, private IP addresses are not internet-routable, therefore the position of the attacker's poisoner relative to the victim's default name server determines the feasibility of DEMONS. We distinguish three cases:

1. The attacker's poisoner and the victim's name server are located in the same local network.
2. The attacker's poisoner and the victim's name server are *not* located in the same local network.
   (a) The victim's name server has a *public* IP address.
   (b) The victim's name server has a *private* IP address.

IP spoofing is feasible in cases 1 and 2(a). Case 1 is a typical public network scenario; the attacker and victim both use the same public network (e.g. in cafes, airports, libraries, schools, etc.). Case 2(a) occurs in large business and cloud scenarios or in cases where home users use a public name server. Case 2(b) is the default for most home users connected to the internet via a home router. Home routers typically run a local name server which is advertised to all attached devices via DHCP. If users choose to change this default behavior, e.g., to defend against [50], or to bypass provider DNS-level filtering, they transition to case 2(a) and become vulnerable to DEMONS.

Impact. DEMONS allows for the attacker to gain control over the network communication of any process in the OS that relies on DNS security. While most web applications today rely on TLS for security, this is not true for all applications in general. For example, by rerouting the Network Time Protocol (NTP [36]), the attacker can get control over the system time, potentially influencing certificate validation or license management. Other examples are email protocols such as SMTP [25], IMAP [9], and POP3 [47], as well as the file transfer protocol FTP [39], used for anything from firmware updates to transferring sensitive business documents. Although these protocols can be protected by TLS, they are often used completely unsecured. Also, some software repositories use HTTP rather than HTTPS for automatic download [28]. In all these cases, client-side DNS Cache Poisoning can give the attacker access to a wide range of attacks on data privacy and system integrity.

5.1 Setup Phase

Source port randomization in DNS depends on free ephemeral UDP ports. With an increasing number of allocated UDP ports, this pool shrinks and eventually runs empty, effectively reducing the randomness in DNS queries back to the 16 bits provided by the TXID. However, at least one UDP port must remain unallocated, or no DNS query can be sent and poisoning is not possible. So, the goal of the web attacker during the setup phase is to force the browser to allocate all but one or a small number of known UDP ephemeral ports. This requires two steps:

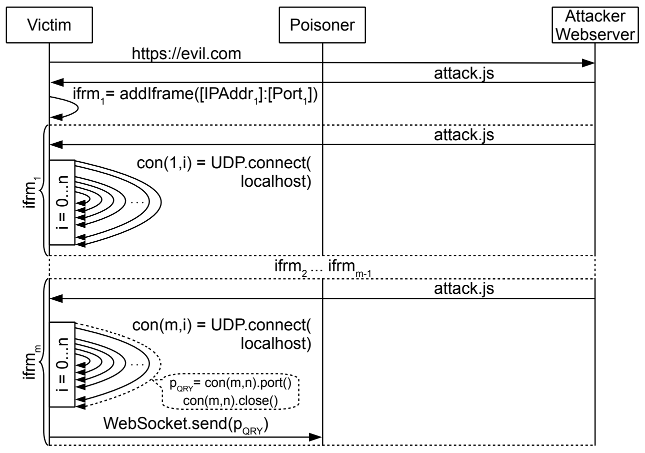
1. *Exhaustion:* The attacker allocates (almost) *all* available ports by creating a sufficient number of port-allocating browser objects, e.g. WebRTC connections. This process is finished when error messages indicate resource exhaustion, or when so many objects were created that they would surely consume at least the maximum number of ephemeral ports available in the OS, in the event that no error messages are seen (silent failure).
2. *Single release:* The attacker destroys a single object, thereby releasing one (or a small number) of ports back into the OS pool. The attacker must be able to determine the port numbers that were associated with the object, either directly with JavaScript, or, in the case of a remote connection, by observing the destruction at the remote end controlled by the attacker.

At this point, the OS has one or few free UDP source ports available, and the attacker knows their numbers. In case the port numbers were read out by the attacker script in the victim's browser, they can now be leaked to the poisoner in preparation of the poisoning phase, e.g. through a WebSocket or an HTTP request to the attacker's webserver. If the port numbers were observed at the remote end of a connection, the observing service has to leak them to the poisoner instead. See Subsection A.1 and Subsection A.2 for details about the setup phase in our implementation of the DEMONS attack.

We will now describe in detail how the OS resource exhaustion attacks based on Site Isolation can be used to implement an efficient setup phase for the DEMONS attack. The attack starts with a victim's web browser loading the attacker's website and executing the included malicious JavaScript code (see Figure 4). This script performs two tasks:

1. Establish a WebSocket for bi-directional communication with the poisoner. This is used to leak the possible DNS query ports at the end of the setup phase.
2. Allocate almost all ephemeral UDP ports by the "exhaustion" and "single release" technique, using a large number of WebRTC objects (cf. Subsection 3.2).

Reserving UDP Ports for Later Release. To follow the "exhaust" and "single release" approach of the setup phase, the attacker first creates a local WebRTC object RTC_0 with a single data channel. As explained in Subsection 3.2, this will allocate two UDP ports $(\mathrm{STUN}_0, \mathrm{DTLS}_0)$ and reserve them for later release. Note that in Windows almost certainly these port numbers are allocated consecutively, so we can assume that $\mathrm{STUN}_0 = \mathrm{DTLS}_0 - 1$.



Victim
Poisoner
Attacker Webserver
https://evil.com
attack.js
ifrm_1= addIframe([IPAddr_1]:[Port_1])
attack.js
con(1,i) = UDP.connect(localhost)
ifrm_1
i = 0...n
ifrm_2 ... ifrm_{m-1}
attack.js
con(m,i) = UDP.connect(localhost)
ifrm_m
i = 0...n
p_{QRY}= con(m,n).port()
con(m,n).close()
WebSocket.send(p_{QRY})

Figure 4: The Setup Phase blocks open ports on the victim's system.

Exhausting All Ephemeral UDP Source Ports. Combining munged WebRTC objects and Site Isolation, the attacker can allocate enough UDP ports to exhaust the entire UDP ephemeral port pool of the OS. Under Windows 10, six sites, each creating one WebRTC object with up to 1500 data channels, are sufficient to achieve port exhaustion. The setup phase takes a while to complete, which allows for a race condition where some UDP ports were allocated before and released during the setup phase by some unrelated process. To safeguard against this, we finish the exhaustion step by quickly allocating a small number of simple WebRTC objects RTC_n, RTC_{n-1}, RTC_{n-2}, ..., each consuming up to two ports.

Releasing Two UDP Source Ports. The attacker must now release at least one UDP port back into the OS pool. Otherwise, the DNS resolver embedded in the OS would not be able to send any more DNS queries, and the attack would result in DoS instead of a successful DNS Cache Poisoning. This is why the attacker created the RTC_0 object before starting the port exhaustion process. The attacker now determines DTLS_0 from RTC_0 (cf. Subsection A.1 in the appendix) and leaks the port number to the poisoner via the WebSocket created in the setup phase. Finally, the attacker closes RTC_0, releasing both STUN_0 and DTLS_0 back into the OS pool.

The Case of an Odd Number of Free Ports. If the OS has an odd number of free UDP ephemeral ports at the beginning of the setup phase, the exhaustion phase will be incomplete because only an even number of ports can be allocated by the attacker with WebRTC objects. Thus, one more port LAST will be left unallocated in addition to STUN_0 and DTLS_0. We found experimentally that most of the time this is the port just before STUN_0, likely an artifact of the mostly sequential UDP port allocation strategy in Windows. Usually, the attack starts at a point in time where the port just before STUN_0 is unallocated, which is $\mathrm{LAST} := \mathrm{STUN}_0 - 1$. Thus, the poisoner has to consider the three ports LAST, STUN_0, DTLS_0 for potential use by the DNS resolver.

Finding the DNS Query Port. Under ideal conditions, there would only be one free UDP ephemeral port usable by the OS resolver, known to the attacker. However, due to the use of WebRTC objects, we are left with either two or three possible source ports after the setup phase, depending on the number of free ports (even or odd) before the attack. To maximize our success rate, we send each spoofed response in the poisoning phase a total of three times, once to DTLS_0, $\mathrm{STUN}_0 := \mathrm{DTLS}_0 - 1$ and $\mathrm{LAST} := \mathrm{DTLS}_0 - 2$ each. Because packets sent to the wrong source port are silently discarded by the OS, the only impact of this change is that we need three times the bandwidth to perform the attack than in the case of a single free port.

5.2 Poisoning Phase

The poisoner receives the leaked DNS query port from the malicious JavaScript code over the WebSocket, and waits for the signal that the JavaScript code is about to trigger a DNS query for the target domain by the DNS resolver embedded into the victim's OS. The poisoner then proceeds to the poisoning phase (see Figure 5), which is similar to that of other DNS Cache Poisoning attacks [3,24,35]. For clarity and completeness, we include here a description of the poisoning phase as implemented and evaluated in our attack prototype.

1. Burst of Spoofed Responses: On activation, the Poisoner sends a burst of spoofed DNS responses. Every such response within a burst has a fresh, randomly chosen TXID and resolves the chosen target domain to the IP address of the malicious server. It is important to note that the first spoofed DNS responses arrive *early*, i.e., before the matching DNS query is generated. These will be considered unsolicited and dropped by the victim's DNS resolver. This maximizes the chance that the malicious response arrives first and the authentic response is never processed by the victim.

2. Triggering a DNS Request: Once a stream of spoofed responses is established, the attacker forces the victim to issue a query matching the target domain in the query section of the spoofed responses that are already in transit. The JavaScript that is still running in the victim's browser as part of the malicious website generates an XMLHttpRequest to a resource hosted on the chosen target domain, e.g., *bank.com*. The sole purpose of this request is to trigger a DNS query to the target domain, which the browser must resolve before the XMLHttpRequest can be sent.

From the moment this lookup is initiated, all spoofed responses that are in transit towards the victim become *potentially valid* because now there exists a query matching the target domain in the query section of the spoofed DNS responses. The only remaining property that can prevent the victim from accepting a potentially valid response is a mismatching TXID.

3. DNS Query Retransmissions: At this point, it is important to understand how the victim system deals with TXID mismatches because the attacker can not expect to guess the correct TXID right away. In accordance with [3], we observed during our experiment that incoming spoofed responses with mismatching TXIDs trigger an immediate *DNS query retransmission* (see Figure 5). A retransmission is a DNS query that is sent out repeatedly to a DNS server in an attempt to retry a previously failed DNS lookup. This retransmission is repeated up to four times for a single DNS query before the resolver aborts the name resolution with an error. Because the attacker maintains a steady stream of spoofed responses with the burst technique, every retransmission attempt is almost immediately answered with a spoofed response, long before the authentic name server even receives the retransmission.

4. Blocking the Correct DNS Response: After four retransmissions, the active query is invalidated and responses will no longer be accepted, even if their TXID would match the ID of the original query. This includes the answer of the benign server, which will also be rejected. Even though the retransmission-limit interferes with the attacker's ability to brute force a large amount of TXIDs in a short time, the net effect is advantageous because with a high likelihood it also prevents the authentic name server from placing the correct record in the client's cache.

5. Rinse and Repeat: To obtain more guesses, the attacker only needs to repeat the poisoning phase by sending another burst of spoofed DNS responses and triggering another DNS query shortly after. Once the TXID of one of the spoofed responses matches the ID used in the victim's DNS query, the attacker observes the incoming XMLHttpRequest on the malicious server and can end the poisoning phase.

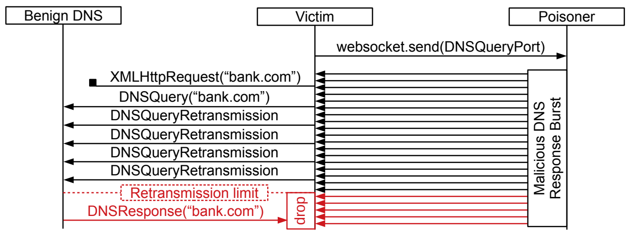


Figure 5: DNS retransmissions in Windows during the poisoning phase.

	Attack Duration			Spoofing Burst		Success Rate
	Min	Mean	Max	#Responses	⌀Duration	
DEMONS (Internet)	15 s	214 s	1162 s	2550	695 ms	37%
DEMONS (Lab)	32 s	243 s	517 s	525	63 ms	36%
Malware (Lab)	5 s	333 s	1586 s	525	50 ms	57%

Table 3: Performance of the web-based DEMONS attack in an Internet setting, in the lab, and in comparison to a malware attacker. The number of responses in a spoofing burst was set in advance.

6 Evaluation of DEMONS

We evaluated DEMONS twice: (1) In an internet setting, using a hosting service that allowed IP spoofing, but also provided an unstable network connection. The results exemplify the possible success rate of a real-world attacker, who may also have to cope with such unstable connections. (2) In a closed lab environment. Here we had optimal control over the network, and our results can be reproduced. For comparison, we also used the lab environment for an unprivileged malware attacker as described by [3], substituting the setup phase of the DEMONS attacker with that of a malware attacker, while preserving all other aspects of the experiment. Table 3 summarizes all three evaluations.

6.1 Setup of Internet Evaluation

Attacker Setup. We deployed the DEMONS infrastructure (Webserver, Poisoner, Malicious Server, see Figure 3) at an internet hosting provider in Moscow that allowed IP spoofing (April 2021). For these servers, we measured an upstream bandwidth that fluctuated between 1 and 200 MBit/s, average latencies of 70 ms (with outliers up to 200 ms), and intermittent episodes of packet loss of up to 20%. Although these conditions were far from ideal, we could implement the DEMONS attack in this setting with a significant success rate.

We adapted DEMONS to these network conditions as follows: To compensate for the overall latency, we inserted a 65 ms delay between the attack start signal sent to the Poisoners and the first XMLHttpRequest triggered by the malicious JavaScript. The latency jitter and packet loss were mostly compensated by distributing the poisoner across three different servers. In addition, we increased the burst size from 105 to 850 spoofed DNS responses per poisoner (for a total of 2550 DNS responses per burst). Longer bursts improve the success rate at the cost of a large drop in attack performance. To counteract this performance loss we increased the number of XMLHttpRequests per burst from 1 to 24 (for a total of 120 DNS queries), with a 3 ms delay in-between.

Victim Setup. The victim machine is a Windows 10 VM running on a desktop computer[2] in the home network of one of the authors. The victim host was connected to a home router via ethernet cable. The internet connection is an end-user DSL connection providing roughly 27 Mbit/s upstream and 80 Mbit/s downstream. The Google resolver (8.8.8.8) was configured as the default DNS server in the client OS.

Stealthiness. During the attack, we observed an average network traffic of roughly 3-4 MBit/s on the victim machine. The internet connection of our victim was utilized normally during the attack and did not show any reduction in service quality during typical home office tasks, browsing, telephony, and video streaming. A victim is unlikely to notice a running DEMONS attack unless the network traffic is actively monitored for suspicious activity. During the poisoning phase the CPU load stayed well below 20%, and only during the setup phase, which took 15 s, did CPU utilization spike up to 100% due to the overhead caused by creating WebRTC objects.

6.2 Results of Internet Evaluation

Table 3 summarizes the results for both DEMONS experiments. Over a course of 24 hours, we ran the DEMONS experiment a total of 351 times. We recorded 131 (37%) successful DNS cache poisonings. The experiment failed 219 (62%) times because the authentic DNS server managed to respond to a DNS query before it was invalidated by the Poisoner. The experiment was aborted one time because it did not produce a result before the limit of 2000 bursts was reached.

6.3 Setup of Lab Evaluation

For our lab setup, we used three Dell Optiplex 960[3] desktop computers connected via a GBit-Ethernet-Switch[4]. The first computer took the role of the victim, running Google Chrome on a stock installation of Windows 10[5]. The second system acted as benign DNS Server. The third system was configured as a router simulating infrastructure between the victim's and the attacker's ISP and the benign DNS server. A Thinkpad T480s ran the attacker's web server, poisoner, and a script to monitor and log the experiment results. To simulate realistic network conditions we used traffic control to set the latency to 1 ms and limit the attacker's bandwidth to a maximum of 20 Mbit/s. Since the lab setup provides a much more consistent connection than the internet setup, we used only one poisoner and a smaller burst size of 525 responses.

[2] Oracle Virtual Box 6 VM with 4 cores, 8 GiB RAM on Intel Core i7 3770k, 32 GiB RAM host.

[3] Intel Core2Quad Q9400, 4 GiB RAM, Intel 82567LM-3 Gigabit NIC

[4] D-Link DGS-108 Gigabit ethernet switch

[5] Chrome 83.0.4103.106 on Windows 10 (1909 Build 18363.815)

6.4 Results of Lab Evaluation

Out of a total of 133 experiments in the lab setup the DEMONS attack succeeded 48 times (36%) and failed 85 times (64%).

To compare DEMONS in the web attacker model with a malware attacker as described in [3], we implemented a collaborative, unprivileged malware attacker in Python. The setup phase of the malware simply allocates all UDP sockets in the system, except one, and leaks the remaining port to the attacker over the network (see Table 1). We used this setup phase *in situ* as a replacement for the victim browser in the DEMONS lab evaluation, keeping the network configuration and all other aspects of the attacker the same. Out of a total of 125 experiments in the lab setup the malware attack succeeded 71 times (57%) and failed 54 times (43%).

6.5 Discussion

Comparing DEMONS and the malware attacker, we see that the minimum attack duration is smaller for the malware attacker due to the faster setup phase. The DEMONS setup phase has more overhead caused by the relatively slow creation of WebRTC objects in the browser. Despite the faster setup phase, the mean and maximum attack duration is longer for the malware attacker compared to DEMONS because the malware attacker can sustain the poisoning phase for longer periods. This results in both higher total run times and a higher overall success rate for the malware attacker.

Overall, the DEMONS attacker has a 21% lower success rate than the malware attacker. We suspect that this is partly because of the jitter in the timing of the DNS queries triggered from JavaScript, compared to the malware attacker written in Python, and partly due to the additional DNS and other activity in the system from running the browser itself.

We note that in our experimental setup, we count a poisoning attempt as a failure when the benign DNS response is accepted by the victim once. In contrast, the evaluation of [3] is based on a DNS entry with a time-to-live (TTL) of 30 s and an attacker who retries the attack after that time, leading to almost perfect success rates overall. We make no such assumption about the TTL used by the benign DNS response.

7 Mitigations

7.1 Mitigating the Fork Bomb

The Google Chrome Team did not consider the fork bomb attack to be a security vulnerability and did not implement any countermeasures. This leaves all browsers based on Chrome susceptible to Site Isolation-based attacks like the fork bomb. In this section, we give some suggestions on how Site Isolation can be improved to prevent the fork bomb attack.

Applying OS Resource Limits. On Linux, cgroups can be used to limit the available resources for single applications – e.g. web browsers – in the OS (other systems offer similar functionality). So by e.g. limiting the number of processes available to Site Isolation-enabled browsers, we can prevent a DoS attack on the OS. At the same time, however, DoS attacks on the browser itself will become easier – the limit assigned by cgroups can be exhausted by multiple tabs, or by a single tab as in the Site Isolation-based fork bomb attack.

Browser Process Consolidation for IP Addresses. In Site Isolation, process consolidation [40, section 4.1.1] is used to reduce the overall number of processes: If two tabs include the same site in two different iframes, only one process is launched for this site. A browser could apply process consolidation also to IPv6 address blocks, such that all IP addresses from one block count as a single origin. However, if these blocks are too big, they can be used to circumvent Site Isolation – if an attacker manages to rent a single IP address in the same address block as an IP address used to access a target website, they may get access to the target website's process in the victim browser.

7.2 Our Solution: Limit Processes by Visible Windows and Tabs

We implemented and evaluated the following countermeasure to the fork bomb attack in the browser: Each window/tab is assigned a limit $\mathcal{L}$ of processes. Each time the user opens a new window/tab w_i, the browser initializes a local limit $\mathcal{L}_i := \mathcal{L}$ and tracks the use of resources for w_i in a variable $\mathcal{C}_i$. As long as $\mathcal{C}_i \leq \mathcal{L}_i$, page loading proceeds normally. If the maximum number $\mathcal{L}_i$ of processes is reached, the browser interrupts page processing with an alert message, offering the user to increase $\mathcal{L}_i$ by some fixed value $\Delta\mathcal{L}$. Our solution does not require special OS interfaces.

We evaluated the Tranco[6] [27] Top 1000 web pages and measured the number of processes created for each of these pages. We found that the maximum number of processes created by a single page from this set was 19. Including some additional headroom, we set $\mathcal{L} = 30$. With this limit, an attacker must open at least 7 tabs to trigger a browser crash, and at least 10 tabs to render the OS unusable (Table 2).

We implemented the described modification in Chromium 101.0.4951.64[7]. The resulting patch is available as part of our artifacts[1] and was submitted to the Chromium and Firefox developers.

For an additional False Positive evaluation, we sorted the Tranco Top 1000 according to the number of processes created for each website. Opening the top 50 web pages from the reordered list in 50 tabs lead to 171 processes running in the OS, which was well below the threshold for DoS attacks we identified in Table 2.

To verify that the changes introduced by our patch do not measurably impact the browser's performance we recorded the page load times of the Tranco Top 5 websites using the profiling tools integrated into Chromium. We found no significant difference in performance.

We also verified the effectiveness of our patch by opening the Tranco Top 50 pages together with our Site Isolation-fork-bomb attack page in Chromium Site Isolation-patched. The attack page was interrupted after creating 30 sites and both Chromium Site Isolation-patched and the OS (Kubuntu 18.04 LTS) stayed stable. Repeating the experiment with an unpatched Chrome led to an OS freeze. In contrast to our malicious attacker website, none of the benign websites triggered the Site Isolation-process-limit dialog. This indicates that our patch is unlikely to impact user experience with false positive warnings.

7.3 Mitigating UDP Port Exhaustion

Google Chrome. The Google Chrome Team considered DEMONS to be a security vulnerability. As DEMONS is a complex attack, it is easy to mitigate by removing any of its preconditions for success. In the case of Chrome, the developers implemented a configurable global limit of 6000 UDP sockets across the whole browser instance. We re-evaluated Chrome with this countermeasure and confirmed that this limit is now effectively enforced (see Table 2 in the appendix for detailed results). This aligns the behavior of Chromium-based browsers with that of Firefox, which already has a global limit of 1000 ports. We note that for both browsers the global limit is high enough to reduce the number of available ephemeral ports for DNS queries significantly, reducing the effectiveness of source port randomization as a countermeasure to the Kaminsky attack.

Redesigning Network Sockets in the OS. The root cause behind DNS Cache Poisoning attacks using UDP port exhaustion is that the pool of ephemeral ports must be shared among *all* IP addresses. However, in calculations regarding the effectiveness of source port randomization, it is commonly

[6] We use the Tranco list generated on 29 March 2022 (`https://tranco-list.eu/list/254Y9`)

[7] commit `7f19001c12cf2eee0b7ddd213a40c1652086c9da`

assumed that *for each* destination IP address the full range of available ephemeral UDP source ports would be available [24, 35, 45]. This mismatch between abstract network sockets and actual OS sockets created through the Berkeley socket API creates an attack surface where unrelated subsystems using UDP can interfere with each other. Unfortunately, changing the socket API would require a complete redesign of the network stack and its use in applications.

7.4 Mitigating DNS Cache Poisoning Attacks

DNS-over-HTTPS (DoH). DEMONS DNS Cache Poisoning can be mitigated by using DoH but only for those web applications running in a browser that uses DoH. Browsers with DoH can still be used as attack vectors to block UDP ports of the OS using the techniques described in this paper. Thus, source port randomization can still be disabled for applications relying on classical DNS. However, for a complete attack setup, the attacker now must control two applications: One for blocking UDP ports and releasing a single port (e.g., the browser), and one for sending a DNS request that shall be poisoned. A bigger obstacle is the limited support for DoH. Currently, only Mozilla supports DoH in the default configuration, and only in certain countries. None of our tested browsers used DoH in its default configuration. Moreover, a downgrade attack from DoH to classical DNS-over-UDP has been discovered recently [22].

Other Solutions. As a straightforward approach to mitigate DNS Cache Poisoning, the size of the DNS TXID could be extended, rendering source port randomization irrelevant. However, no standardization activities in this direction are known. This may be due to the now 24-year struggle to deploy DNSSEC [1, 4]. DNSSEC would solve the problem, yet a complete mitigation can only be achieved if *all* domains use DNSSEC, or if an application can determine which domains are secured and which are not. Additionally, OS resolvers would have to verify the DNSSEC signature chain.

8 Related Work

Site Isolation was developed by Google for the Chrome web browser, and is described by Reis, Moshchuk, and Oskov in [40]. Before, process isolation has been used to isolate different windows at the OS level to protect against remote code execution vulnerabilities in the renderer of the browser. As shown in [23, 43], the misalignment between web origin and browser boundaries could be exploited by a web attacker to target the local OS. The urgency for Site Isolation was increased by the publication of the Spectre [26] and Meltdown [32] side-channel attacks.

Just-in-time (JIT) compilation of JavaScript provided many examples of attacks on local processes [14, 29, 30, 34]. Before that, drive-by-downloads could be used to install malware on the local OS [8, 42]. Other attacks target the victim's machine hardware itself [15–17, 31, 38, 44, 46].

The concept of a fork bomb is comprehensively described by Berlot and Sang [5]. Fork bombs can be difficult to detect and mitigate. Nakagawa and Oikawa [37] suggest a quarantine procedure to reduce harm to honest processes in case of False Positives detection, but in practice, the best strategy is to limit and control the number of processes by careful application design, in particular in the case of sandbox environments running untrusted code [40].

The idea to allocate most of the UDP socket table to disable port randomization and thus re-enable past DNS Cache Poisoning attacks [24] was first described by Alharbi et al. [3], who carefully analyzed the performance of the attack on Windows, Linux, and macOS under realistic network latencies. The actual port exhaustion in their attack is achieved using a collaborative, unprivileged malware. Although [3] conjecture that the browser can be used instead of malware, to our knowledge no such port exhaustion in the web attacker model was known prior to Site Isolation and our work.

Other recent DNS Cache Poisoning attacks on various network devices have also bypassed UDP port randomization. Shulman and Waidner [45] use IP fragmentation to inject spoofed DNS responses. Man et al. [35] build a side-channel from a complex combination of ICMP error messages on UDP open port queries and ICMP limits, to detect open UDP ports at resolvers, which are then used in spoofed DNS responses. Zheng et al. [50] use oversized DNS resources at an attacker-controlled DNS server to enforce splitting of the DNS response into two UDP packets, where only the first packet contains the random TXID, and the second UDP packet is spoofed by the attacker who only has to guess the correct UDP port. As a limitation, the attacker must be in the same (W)LAN as the victim.

9 Conclusion and Future Work

Site Isolation is an important security architecture to protect against side channel and renderer exploits. Our work aims at improving Site Isolation, not at diminishing it.

While the sophisticated DEMONS attack could be successfully mitigated by introducing a global limit on the web browser's UDP ports, the fork bomb attack is still a threat. It may be surprising to some that even very old attacks such as fork bombs and other resource exhaustion attacks are still effective against current operating systems, and that browsers are fulfilling an important role in protecting users against such threats. In fact, for years now browsers have provided a safe and reliable environment for users to run untrusted, even malicious code, arguably a safer and more reliable environment than the operating system itself.

In this context, the flaws we found in Site Isolation are an unfortunate regression. We note with some concern that as browsers are evolving to meet the ever-increasing demands of web application developers, more and more resources of the OS will be available more or less directly to web attackers. This includes network sockets as well as hardware resources such as arbitrary USB devices. We hope that our work highlights the emergent risks of this trend.

Future Work. Our findings were limited to common web browsers but could be extended to other browsers, browser extensions, and native applications built with an embedded browser framework, such as Electron.[8] Another class of targets could be headless browsers running on servers for web crawlers or to create preview images of links for messenger apps. We did not evaluate all possible browser features for Site Isolation based resource exhaustion. In particular, UDP socket allocation may be possible using the QUIC protocol. Also, exhaustion of graphic card resources through WebGL and other rendering APIs could be considered.

Acknowledgments

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

References

[1] D. E. 3rd and C. Kaufman, "Domain Name System Security Extensions," RFC 2065 (Proposed Standard), Internet Engineering Task Force, Jan. 1997, obsoleted by RFC 2535. [Online]. Available: http://www.ietf.org/rfc/rfc2065.txt

[2] D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song, "Towards a formal foundation of web security," in *CSF 2010: IEEE 23st Computer Security Foundations Symposium*, A. Myers and M. Backes, Eds. IEEE Computer Society Press, 2010, pp. 290–304.

[3] F. Alharbi, J. Chang, Y. Zhou, F. Qian, Z. Qian, and N. Abu-Ghazaleh, "Collaborative Client-Side DNS Cache Poisoning Attack," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 1153–1161.

[4] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose, "DNS Security Introduction and Requirements," RFC 4033 (Proposed Standard), Internet Engineering Task Force, Mar. 2005, updated by RFCs 6014, 6840. [Online]. Available: http://www.ietf.org/rfc/rfc4033.txt

[5] M. Berlot and J. Sang, "Dealing with process overload attacks in unix," *Information Security Journal: A Global Perspective*, vol. 17, no. 1, pp. 33–44, mar 2008. [Online]. Available: https://doi.org/10.1080%2F19393550801929547

[6] C. C. Center, "Vulnerability Note VU#800113: Multiple DNS implementations vulnerable to cache poisoning," 2008. [Online]. Available: https://www.kb.cert.org/vuls/id/800113

[7] M. Cotton, L. Eggert, J. Touch, M. Westerlund, and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry," RFC 6335 (Best Current Practice), Internet Engineering Task Force, Aug. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6335.txt

[8] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 281–290.

[9] M. Crispin, "INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1," RFC 3501 (Proposed Standard), Internet Engineering Task Force, Mar. 2003, updated by RFCs 4466, 4469, 4551, 5032, 5182, 5738, 6186, 6858. [Online]. Available: http://www.ietf.org/rfc/rfc3501.txt

[10] J. Dickinson, S. Dickinson, R. Bellis, A. Mankin, and D. Wessels, "Dns transport over tcp - implementation requirements," Internet Requests for Comments, RFC Editor, RFC 7766, March 2016.

[11] P. Dubroy and R. Balakrishnan, "A study of tabbed browsing among mozilla firefox users," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 673–682. [Online]. Available: https://doi.org/10.1145/1753326.1753426

[12] M. Foundation, "Mozilla firefox - projekt fission," 2020. [Online]. Available: https://wiki.mozilla.org/Project_Fission

[13] ——, "Firefox 94.0, see all new features, updates and fixes," 2021. [Online]. Available: https://www.mozilla.org/en-US/firefox/94.0/releasenotes/

[14] R. Gawlik and T. Holz, "Sok: Make jit-spray great again," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. Baltimore, MD: USENIX Association, Aug. 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/gawlik

[8] https://www.electronjs.org/

[15] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by key-extraction cache attacks from portable code," in *Applied Cryptography and Network Security*, B. Preneel and F. Vercauteren, Eds. Cham: Springer International Publishing, 2018, pp. 83–102.

[16] D. Gruss, D. Bidner, and S. Mangard, "Practical memory deduplication attacks in sandboxed javascript," in *Computer Security – ESORICS 2015*, G. Pernul, P. Y A Ryan, and E. Weippl, Eds. Cham: Springer International Publishing, 2015, pp. 108–122.

[17] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, ser. DIMVA 2016. Berlin, Heidelberg: Springer-Verlag, 2016, p. 300–321. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_15

[18] M. Handley, V. Jacobson, and C. Perkins, "SDP: Session Description Protocol," RFC 4566 (Proposed Standard), Internet Engineering Task Force, Jul. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4566.txt

[19] I. Hickson *et al.* (2019) WebRTC 1.0: Real-time Communication Between Browsers. W3C and Google Inc. and Apple Computer Inc. and Mozilla Foundation and Opera Software ASA. [Online]. Available: https://www.w3.org/TR/webrtc/#dom-peerconnection-localdescription

[20] P. Hoffman and P. McManus, "DNS Queries over HTTPS (DoH)," Internet Requests for Comments, RFC Editor, RFC 8484, October 2018.

[21] C. Holmberg, H. Alvestrand, and C. Jennings, "Negotiating Media Multiplexing Using the Session Description Protocol (SDP)," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-mmusic-sdp-bundle-negotiation-54, December 2018. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-mmusic-sdp-bundle-negotiation-54.txt

[22] Q. Huang, D. Chang, and Z. Li, "A comprehensive study of DNS-over-HTTPS downgrade attack," in *10th USENIX Workshop on Free and Open Communications on the Internet (FOCI 20)*, 2020.

[23] Y. Jia, Z. L. Chua, H. Hu, S. Chen, P. Saxena, and Z. Liang, ""The Web/Local" Boundary Is Fuzzy: A Security Study of Chrome's Process-based Sandboxing," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 791–804.

[24] D. Kaminsky, "Black ops 2008: It's the end of the cache as we know it," *Black Hat USA*, vol. 2, 2008.

[25] J. Klensin, "Simple Mail Transfer Protocol," RFC 5321 (Draft Standard), Internet Engineering Task Force, Oct. 2008, updated by RFC 7504. [Online]. Available: http://www.ietf.org/rfc/rfc5321.txt

[26] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[27] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, "Tranco: A research-oriented top sites ranking hardened against manipulation," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, ser. NDSS 2019, Feb. 2019.

[28] J. Leitschuh, "Want to take over the Java ecosystem? All you need is a MITM!" 2019. [Online]. Available: https://medium.com/bugbountywriteup/want-to-take-over-the-java-ecosystem-all-you-need-is-a-mitm-1fc329d898fb

[29] W. Lian, H. Shacham, and S. Savage, "Too lejit to quit: Extending jit spraying to arm." in *NDSS*. Citeseer, 2015.

[30] ——, "A call to arms: Understanding the costs and benefits of jit spraying mitigations." in *NDSS*, 2017.

[31] M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C. Maurice, and S. Mangard, "Practical keystroke timing attacks in sandboxed javascript," in *Computer Security – ESORICS 2017*, S. N. Foley, D. Gollmann, and E. Snekkenes, Eds. Cham: Springer International Publishing, 2017, pp. 191–209.

[32] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[33] M. J. Luckie, R. Beverly, R. Koga, K. Keys, J. A. Kroll, and k claffy, "Network hygiene, incentives, and regulation: Deployment of source address validation in the internet," in *ACM CCS 2019: 26th Conference on Computer and Communications Security*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM Press, Nov. 2019, pp. 465–480.

[34] G. Maisuradze, M. Backes, and C. Rossow, "Dachshund: digging for and securing against (non-) blinded constants in jit code," in *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[35] K. Man, Z. Qian, Z. Wang, X. Zheng, Y. Huang, and H. Duan, “Dns cache poisoning attack reloaded: Revolutions with side channels,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1337–1350. [Online]. Available: https://doi.org/10.1145/3372297.3417280

[36] D. Mills, J. Martin, J. Burbank, and W. Kasch, “Network Time Protocol Version 4: Protocol and Algorithms Specification,” RFC 5905 (Proposed Standard), Internet Engineering Task Force, Jun. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5905.txt

[37] G. Nakagawa and S. Oikawa, “Fork bomb attack mitigation by process resource quarantine,” in *2016 Fourth International Symposium on Computing and Networking (CANDAR)*. IEEE, nov 2016. [Online]. Available: https://doi.org/10.1109%2Fcandar.2016.0124

[38] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, “The spy in the sandbox: Practical cache attacks in javascript and their implications,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1406–1418. [Online]. Available: https://doi.org/10.1145/2810103.2813708

[39] J. Postel and J. Reynolds, “File Transfer Protocol,” RFC 959 (INTERNET STANDARD), Internet Engineering Task Force, Oct. 1985, updated by RFCs 2228, 2640, 2773, 3659, 5797, 7151. [Online]. Available: http://www.ietf.org/rfc/rfc959.txt

[40] C. Reis, A. Moshchuk, and N. Oskov, “Site Isolation: Process Separation for Web Sites within the Browser,” in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1661–1678. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/reis

[41] E. Rescorla, “Security considerations for webrtc,” Working Draft, IETF Secretariat, Internet-Draft draft-ietf-rtcweb-security-12, July 2019. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-security-12.txt

[42] K. Rieck, T. Krueger, and A. Dewald, “Cujo: efficient detection and prevention of drive-by-download attacks,” in *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010, pp. 31–39.

[43] R. Rogowski, M. Morton, F. Li, F. Monrose, K. Z. Snow, and M. Polychronakis, “Revisiting browser security in the modern era: New data-only attacks and defenses,” in *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, 2017, pp. 366–381.

[44] S. Röttger and A. Janc, “A spectre proof-of-concept for a spectre-proof web,” 2021. [Online]. Available: https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html

[45] H. Shulman and M. Waidner, “Fragmentation considered leaking: Port inference for dns poisoning,” in *Applied Cryptography and Network Security*, I. Boureanu, P. Owesarski, and S. Vaudenay, Eds. Cham: Springer International Publishing, 2014, pp. 531–548.

[46] A. Shusterman, A. Agarwal, S. O’Connell, D. Genkin, Y. Oren, and Y. Yarom, “Prime+probe 1, javascript 0: Overcoming browser-based side-channel defenses,” in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2863–2880. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/shusterman

[47] R. Siemborski and A. Menon-Sen, “The Post Office Protocol (POP3) Simple Authentication and Security Layer (SASL) Authentication Mechanism,” RFC 5034 (Proposed Standard), Internet Engineering Task Force, Jul. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc5034.txt

[48] P. Vixie and D. Dagon, “Use of bit 0x20 in dns labels to improve transaction identity,” Working Draft, IETF Secretariat, Internet-Draft draft-vixie-dnsext-dns0x20-00, March 2008. [Online]. Available: https://www.ietf.org/archive/id/draft-vixie-dnsext-dns0x20-00.txt

[49] W. Xu, S. Park, and T. Kim, “Freedom: Engineering a state-of-the-art dom fuzzer,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 971–986. [Online]. Available: https://doi.org/10.1145/3372297.3423340

[50] X. Zheng, C. Lu, J. Peng, Q. Yang, D. Zhou, B. Liu, K. Man, S. Hao, H. Duan, and Z. Qian, “Poison Over Troubled Forwarders: A Cache Poisoning Attack Targeting DNS Forwarding Devices,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

A DEMONS Implementation Details

A.1 Tracking the port numbers used by WebRTC Objects

The `RTCPeerConnection` (RTCPC), which is part of the JavaScript WebRTC API, has an `onconnectionstatechange` property. Any custom event handler function assigned to this property is called upon state change of the RTCPC. Furthermore, the RTCPC has a `localDescription` property that describes the session for connections local endpoint [19]. Among other information this session description contains the port number proposed during the connection negotiation. The malicious JavaScript attaches the custom event handler shown in Figure 6 to every RTCPC it establishes. Once the event fires for any connection the handler passes the local session description to the `getPort` function shown in Figure 7. The function `getPort` extracts the UDP port from the string representation of the session description associated with the connection that triggered the event. At the end of the Setup Phase the malicious JavaScript closes one of the established RTCPCs and sends the port stored for this connection to the Poisoner using a WebSocket.

```
function onConnectionStateChange(ev, cnContainer, cnIndex, cnType, eventHandler) {
  // store local description ports
  if(cnType == "LCON") {
    // Get handler for the n-th RTCPC
    cn = cnContainer.connections.local[cnIndex];
    // Extract and store local port for the n-th
    // RTCPC
    cnContainer.ports.local[cnIndex] = getPort(cn);
    if(eventHandler != null) {
      eventHandler(cn, cnContainer.ports.local[cnIndex]);
    }
  }
  // store remote description ports
  ...
}
```

Figure 6: Intercepting WebRTC connection state changes

```
function getPort(rtcpc) {
    sdp = rtcpc.localDescription.sdp.split("\n");
    cand = sdp.filter(i => i.startsWith("a=candidate"));
    return cand[0].split(/\s+/)[5];
}
```

Figure 7: Retrieving the UDP port from a session description

A.2 Modifying the SDP

Adding multiple copies of the same data channel with different unique identifiers (`mid` in Figure 8) allows for the reservation of multiple UDP ports with a single RTCPeerConnection. This significantly reduces the CPU overhead compared to using multiple RTCPeerConnections with only a single data channel. We achieve this by using SDP munging, where the SDP offer is manipulated outside of the WebRTC implementation (see Figure 9).

```
1   v=0
2   o=- 6271792437987180154 2 IN IP4 127.0.0.1
3   s=-
4   t=0 0
5   a=group:BUNDLE 0
6   m=application 9 UDP/DTLS/SCTP webrtc-datachannel
7   c=IN IP4 0.0.0.0
8   a=mid:0
9  +m=application 9 UDP/DTLS/SCTP webrtc-datachannel
10 +c=IN IP4 0.0.0.0
11 +a=mid:1
12 +m=application 9 UDP/DTLS/SCTP webrtc-datachannel
13 +c=IN IP4 0.0.0.0
14 +a=mid:2
15  ...
```

Figure 8: SDP offer manipulation (excerpt). The attacker inserts a copy of lines 6–8 once per extra media channel to be allocated, consuming two more UDP ports each time.

```
function mungeChannels(offer, mungeChannelCount, offs ) {
  const midx = offer.sdp.indexOf("m=");
  const mdef = offer.sdp.substr(midx);
  let sdp = offer.sdp;
  for(let i=0; i < mungeChannelCount; i++) {
    sdp += mdef.replace(/mid:\d+/, "mid:" + (
      10 + mungeChannelCount * offs + i));
  }
  offer.sdp = sdp;
  return offer;
}
```

Figure 9: SDP offer manipulation program code.

B Evaluation Results of the DEMONS Mitigations deployed in Chrome/Chromium and Edge

OS			Browser	Chrome[1]/Chromium[2]			Edge[3]			Firefox[4]		
			Sites	Single-Site	Multi-Site		Single-Site	Multi-Site		Single-Site	Multi-Site	
			Site Isolation	-	off	on	-	off	on	-	off	on
Windows	**Processes**			8	6	311ϟ💣	9	7	306ϟ💣	10	7	215ϟ
	Sockets	**UDP v4/v6**	**WebRTC[p]**	1000/500	1000/500	5996/2998	1000/500	1000/500	5996/2998	0/999	0/998	0/999
			WebRTC[u]	1000/500	1000/500	5996/2998	1000/500	1000/500	5996/2998	0/998	0/999	1/999
			WebRTC[m]	3001/1500	3001/1500	5997/2996	3001/1500	3000/500	5996/2997	-	-	-
Linux	**Processes**			11	9	460💣	-	-	-	6	6	224ϟ
	Sockets	**UDP v4/v6**	**WebRTC[p]**	1000/500	1000/500	6000/3000	-	-	-	1000/998	999/998	999/998
			WebRTC[u]	1000/500	1000/500	6000/3000	-	-	-	999/998	999/998	999/998
			WebRTC[m]	3000/1500	3001/1500	6000/2999	-	-	-	-	-	-

[p] WebRTC objects with a single data channel. [u] WebRTC objects with multiple data channels. [m] WebRTC objects with munging.
[1]Chrome v89.0.4389.114 on Windows 10; [2]Chromium v89.0.4389.90 on Kubuntu Linux 18.04 LTS;
[3]Edge (Chromium based) v89.0.774.75; [4]Firefox Nightly 89.0a1 2021-04-11 *experimental Site Isolation enabled*
ϟ Browser Crash; 💣 Operating system unusable
Note: Chrome/Chromium allocate one UDP port in each WebRTC channel in dual-stack mode, causing an extra allocation in IPv6. Firefox allocates all UDP ports IPv6 only, but in Linux, they are mapped to IPv4 by the OS. Therefore entries like 1000/500 refer to the number of IPv4/IPv6 ports blocked, resp.

Table 4: Re-evaluation of Resource Exhaustion Attacks based on Site Isolation with DEMONS mitigations deployed in Chrome/Chromium and Edge.

C Site Isolation-Based Fork Bomb Details

	Observed behavior	🐧[1]		⊞[2]		
		◎[3]	🦊[4]	◎[5]	🦊[4]	℮[6]
ϟ	The Browser became unresponsive to user interactions.	●	●	●	●	
	The browser window closed without notification.			●	●	●
	The browser showed a crash report dialog.		●		●	
	Moving the browser window caused artifacts, the desktop was not redrawn properly.		●			
	The browser window became transparent and could not be dragged, shortly after clicking the window the browser automatically closed and restarted without notification.			●		
	The screen turned black for a short time, then the browser crashed with a dialog titled „chrome.exe – Application Error“, message text: „The application was“. After closing the message another error message with the same title appeared, message text: „The exception unknown software exception (0xe0000008) occurred in the application at location 0x00007FFC8111A799.“			●		
	The browser became unresponsive. The OS displayed a dialog titled „WerFault.exe – Application Error“, message text: „The application was unable to start correctly (0xc000012d). Click OK to close the application. After confirming the dialog by pressing the "OK" button the browser window remained open and unresponsive.			●		
	The screen turned black for a short time, then the browser crashed with a dialog titled „msedge.exe – Application Error“, message text: „The exception unknown software exception (0xe0000008) occurred in the application at location 0x00007FF9E242A799.“. After confirming the dialog a white unresponsive browser window stayed open.					●
	The browser crashed with a dialog titled „firefox.exe – Application Error“, message text: „The exception Breakpoint A breakpoint has been reached. (0x80000003) occurred in the application at location 0x00007FF9DB4C0955. Click on OK to terminate the program“.				●	
💣	The OS froze.	●	●		●	
	The screen turned black.			●		●
	The screen was no longer redrawn properly.			●		●

OS versions: [1]Kubuntu Linux 18.04.5 LTS (Kernel 5.4.0-62), [2]Windows 10 (1909 Build 18363.815)
Browser versions: [3]Chromium 83.0.4103.0, [4]Firefox Nightly 86.01a, [5]Chrome 83.0.4103.106, [6]Edge 83.0.478.45
Hardware configuration: Dell Latitude 5280, Intel Core i5 7200U, 8 GiB RAM, 240 GiB M.2 SATA

Table 5: During our evaluation of the fork bomb we observed effects that affected browsers and operating systems. This table provides a detailed description of behavior we classified as "browser crash (ϟ)" and "OS unusable (💣)".