Server-Side Browsers: Exploring the Web's Hidden Attack Surface

Marius Musch
TU Braunschweig
Germany

Robin Kirchner
TU Braunschweig
Germany

Max Boll
TU Braunschweig
Germany

Martin Johns
TU Braunschweig
Germany

ABSTRACT

As websites grow ever more dynamic and load more of their content on the fly, automatically interacting with them via simple tools like *curl* is getting less of an option. Instead, headless browsers with JavaScript support, such as *PhantomJS* and *Puppeteer*, have gained traction on the Web over the last few years. For various use cases like messengers and social networks that display link previews, these browsers visit arbitrary, user-controlled URLs. To avoid compromise through known vulnerabilities, these browsers need to be diligently kept up-to-date. In this paper, we investigate the phenomenon of what we coin *server-side browsers* at scale and find that many websites are running severely outdated browsers on the server-side. Remarkably, the majority of them had not been updated for *more than 6 months* and over 60% of the discovered implementations were found to be vulnerable to publicly available proof-of-concept exploits.

CCS CONCEPTS

• **Security and privacy** → **Web application security**.

KEYWORDS

Browser; Fingerprinting; Instrumentation; Server-Side Requests; Web Security

ACM Reference Format:
Marius Musch, Robin Kirchner, Max Boll, and Martin Johns. 2022. Server-Side Browsers: Exploring the Web's Hidden Attack Surface. In *Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security (ASIA CCS '22), May 30-June 3, 2022, Nagasaki, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3488932.3517414

1 INTRODUCTION

Nowadays, many applications that were originally intended as ordinary desktop software, such as messengers or word processors, are moving to the Web. As a result, new technologies are implemented to support this transition. One of these are *server-side requests* (SSRs), i.e., when the web server also acts as a client to fetch additional information from other locations on the Web. Such SSR implementations can often be triggered by unprivileged users of a website, e.g., social networks showing a preview for all user-submitted links.

Traditional SSR implementations resemble tools like *wget* or *curl* and only fetch the main HTML document in a single request without executing active content. However, the rise of JavaScript-heavy frameworks and single page applications (SPAs) made this approach largely incompatible with modern web development, as the relevant content is often dynamically inserted afterward. Consequently, this led to an increased usage of what we call *server-side browsers* (SSBs), i.e., real browser like *Headless Chrome* running on the server-side. By using such an SSB instead of a simple SSR, the back end can not only send HTTP requests and receive the responses but render entire web pages built with modern JavaScript frameworks.

However, running a real browser that *anyone* can summon to an *arbitrary* URL naturally poses a much higher risk than downloading a page without executing its code. Despite browser vendors' best efforts to harden their code [50], new vulnerabilities with high or even critical security impact are found on a regular basis [e.g. 18, 19, 60]. Therefore, it is crucial to apply updates on time, to avoid being vulnerable to publicly disclosed exploits. Google, for example, just recently announced that they will shorten their stable release cycle from six weeks down to four [20], in another effort to tighten the *patch gap*, i.e., the time between a security bug fix being merged into their open-source repository and the release of a new, stable version that includes this bug fix for all users [38]. While modern browsers apply updates automatically for end-users [32, 61], popular SSB implementations do not, e.g., because part of the automation API could have changed and an unattended update could silently break an important service [34]. Consequently, this opens a unique and dangerous attack surface on the server-side. As the amount of features and supported file formats in browsers is only increasing, so is the attack surface, e.g., through the addition of the new *WebAssembly* format [39, 63].

In this paper, we systematically explore the phenomenon of these outdated server-side browsers in the wild. We first develop a methodology to automatically discover SSBs at scale that entices servers we visit to request a website under our control. We find that about 6% of all domains that we crawl do visit our monitoring server afterward and 16% of those even did so with a real browser that had JavaScript enabled. To determine which of these SSBs are

ASIA CCS '22, May 30-June 3, 2022, Nagasaki, Japan
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9140-5/22/05…$15.00
https://doi.org/10.1145/3488932.3517414

vulnerable to known exploits, we use a fingerprinting technique to extract their browser version, without needing to rely on potentially spoofed user agent strings. Thereby, we find that over 60% of all SSB implementations in the wild are vulnerable to publicly available exploits and *more than half* of them even had not been updated for *over 6 months.*

To summarize, we make the following contributions:

- We show how to automatically discover server-side browsers in the wild (Section 3).
- We present a methodology to discern human visitors from bots based on a combination of timing information and bot indicators. Moreover, we also accurately determine which browser versions these bots use, even if they send spoofed user agent information (Section 4).
- We conduct the first large-scale study of SSBs and find that over 60% of the 254 sites we discovered to make use of an SSB are vulnerable to publicly available exploits (Section 5).

2 SERVER-SIDE BROWSERS

In this section, we first introduce our terminology and then describe multiple use cases for server-side browsers that can not be solved with simpler alternatives, showing their usage is often intended and unavoidable. After that, we introduce the most popular SSB implementations and discuss the potential security pitfalls posed by this technology.

Terminology. Basically, SSRs are conducted each time one server requests data from another, e.g., from a service offering an HTTP API. These SSR implementations are characterized by the fact that they only handle the connection on the HTTP level, but treat the returned content as a string or simple data format like JSON. In particular, they do not parse and render HTML, do not load embedded resources, and do not execute active content like JavaScript code. Popular SSR implementations include command-line tools like *curl* and *wget*, as well as the *http* package for Node.js, and the *file_get_contents* function for PHP. SSBs, on the other hand, are the equivalent of running a normal desktop browser on the server-side. The obvious difference here is that there is no human interacting with a visible graphical user interface involved. Instead, these SSBs are fully automated tools running in the background. Under the hood, they support exactly the same features as their desktop equivalent, in particular, they parse and execute all active JavaScript content unless this is explicitly disabled. Our terminology implies that each SSB also conducts one or more SSRs while loading a webpage, but not each SSR necessarily comes from an SSB.

Use cases. There are various use cases that require running a fully-featured browser on the server-side and can not be solved by resorting to simpler, SSR-like solutions. For example, major search engines such as *Google* and *Bing* rely on SSBs for their web crawlers [2, 4]. In principle, statically crawling a web page without rendering it would be more convenient as it uses a lot less resources. Yet, as many of today's web applications are built dynamically with JavaScript, web crawlers need to be capable of executing this scripting language to properly index the page's content. Therefore, using an SSB with a native rendering engine increases the accuracy of the indexed content, as SPAs might only show a blank page if JavaScript is disabled. Another use case are web security services like *Symantec Sitereview* [5] and *VirusTotal* [6] offering ratings and categorizations for URLs, allowing users to check whether a website is potentially malicious. As previous work finds, a significant percentage of search results and ads employ cloaking techniques [52, 83, 84] which involve blocklisting of IPs and user agents affiliated with search engines or detecting the absence of JavaScript execution. This way, they try to hide from these services by serving harmless content to web crawlers while serving malicious sites to potential victims [44]. These techniques make it especially important for providers of web security scanners to use real, automated browsers to mimic a human user in order to scan websites from a human's and not from a bot's perspective.

Automated Browsers. Generally, there are several different options available to implement an SSB, of which we introduce two in more detail in the following. PhantomJS [41] is a headless browser, i.e., a browser that can be executed in the background on a server without a graphical user interface. It is based on the rendering engine *Webkit* and can be controlled via a JavaScript API. After its launch in 2011, PhantomJS enjoyed great popularity for a few years. A total of about 50 million downloads from npm highlight the success of the framework. However, at the timing of writing, the last official release was about four years ago and its creator announced in early 2018 that the development will be discontinued [40]. Surprisingly, there are nearly 14 million downloads registered for the package since it was officially discontinued [81]. As an alternative, *Headless Chrome* allows interacting with a website using all of Chrome's features without having a visible user interface. Internally, it utilizes the DevTools [36] protocol to communicate with and control a browser instance. With Puppeteer [34], the Chrome team also provides a high-level API in form of a Node library to control a Chromium instance since the beginning of 2018 [11]. In contrast to PhantomJS, download counts for the `puppeteer` package are continuously increasing with a total of 163 million downloads since its release [82].

2.1 Attack Scenario

As long as the URL in an SSR/SSB implementation is hard-coded to visit only one trustworthy server, running them is generally unproblematic. Problems arise, however, if this URL depends on user input, e.g., in all the previously outlined use cases. An apparent threat in this scenario are *server-side request forgery* (SSRF) attacks. In the most common SSRF scenario, an attacker tries to bypass firewall rules to gain access to privileged parts of the network. In the simplest case, submitting an internal IP address such as http://10.0.0.1 to a vulnerable SSR service would result in that server forwarding the internal content to the external attacker, as shown in part (a) of Figure 1. Other, similar attacks against SSRs are abusing them as an attack proxy, e.g., in a denial of service (DoS) attack, or abusing them to confuse client-side filters integrated into browsers in a so-called *origin laundering attack.* The high prevalence of SSRs as a convenience feature, as well as its increasing severity due to complex architectures and higher adoption rates of cloud and web services has earned SSRF a place in this year's OWASP Top 10 [67]. These scenarios and their potential consequences were studied in detail in a paper on SSRF attacks published in 2016 by Pellegrino et al. [69].

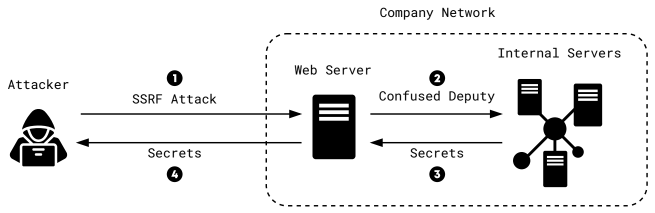
On the other hand, one so far over-looked scenario is directly attacking the implementation of the requesting mechanism, i.e., the headless browser itself. Since these SSBs are based on a real browser and visit arbitrary, attacker-supplied URLs, they too are vulnerable to JavaScript exploits like every other desktop browser. Successful exploitation means the attacker could gain control of the server the SSB is running on. As part (b) of Figure 1 shows, in this scenario, there is no request forged and no deputy confused, instead the attacker uses the request service *as intended* and prompts it to visits an *external* website under the attacker's control. From there, they can launch their JavaScript payload and probe the visiting user agent for exploitable vulnerabilities. If successful, they might be able to completely compromise the server and begin lateral movement through the network from there.

A major threat to all browsers is the large number of publicly disclosed vulnerabilities. Google Chrome, for example, enjoys a rapid update cycle to keep up with security and new features of the constantly evolving web ecosystem. A major update of the stable channel is pushed to the public every six weeks [33] while minor releases come every two to three weeks according to Chrome's update strategy [35]. Google even announced plans to further increase the frequency of their updates and plan to release a major update every four weeks by the end of 2021 [20]. They aim to further reduce the *patch gap*, i.e., the time between a security bug fix being merged into their open-source repository and the release of a new, stable version that includes this bug fix for all users [38].

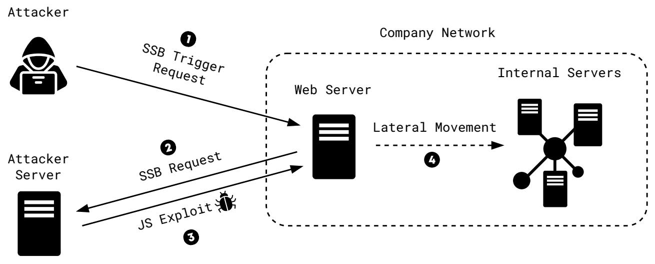
Yet one main difference between headless browsers and their desktop equivalent is the way updates are handled: While desktop browsers usually update automatically nowadays, headless browsers require manual intervention to keep up with security patches. Not updating these SSBs automatically has good reason, since the automation API or other important features might have changed and thus could silently break tools that rely on them. For example, Google writes in the official Puppeteer repository: "We see Puppeteer as an *indivisible entity* with Chromium. Each version of Puppeteer bundles a specific version of Chromium – the *only version* it is guaranteed to work with. [...] This is not an artificial constraint." [34] This means there is a significant risk of outdated versions of SSBs still running in the wild if they are not constantly monitored and maintained. In this paper, we thus focus on *automated but outdated browsers* that we can lure to visit a site under our control to (theoretically) deliver a publicly available JavaScript exploit from there. To summarize, we consider the following to be in and out of scope for this work respectively:

In scope. All automated, server-side browsers that run a real rendering engine and execute JavaScript, regardless of which automation framework or headless browser implementation they use.

Out of scope. All SSR implementations that do not use a real browser (e.g., curl) or use one, but have disabled JavaScript execution. Also, all SSRF attacks like accessing the internal network, bypassing URL filters, or origin laundering.



(a) Abusing an SSR service as a confused deputy in an SSRF attack



(b) Compromising a server-side browser in an SSB attack

Figure 1: Comparison of SSRF and SSB attacks

2.2 Research Questions

So far, we have seen that running a fully-featured browser on the server-side is a relatively recent phenomenon, probably caused by the rise in complexity in modern websites. The popularity of the now-defunct *PhantomJS* and its de-facto successor *Puppeteer* highlight the relevance of their use cases. However, the extent of usage and the security implications of running such an SSB as a service, i.e., where the attacker fully controls the URL destination, have—so far—not been studied. To better understand this phenomenon, we investigate and answer the following research questions:

- *RQ1:* How many websites trigger SSRs, i.e., visit other websites based on user-provided URLs?
- *RQ2:* How many of these SSRs are SSBs, i.e., use a real browser in their back end?
- *RQ3:* Which browsers are used in these SSB implementations and at which major version?
- *RQ4:* How many of these SSBs are running outdated browsers vulnerable to publicly available exploits?

3 DISCOVERING SSRS

As SSRs are implemented in the back end of a web application, their presence is not immediately obvious by accessing the front end, i.e., the loaded website with HTML, CSS, and JS content. Therefore, we require an empirical method to interact with these websites and reveal any potential SSRs. This means we need to automatically visit websites and provide them with unique URLs pointing to a server under our control. If we afterward observe a request to the URL we provided, there is a good chance that we successfully triggered an SSR. In general, there are three ways to entice the targeted server to visit our monitoring server: First, we can supply *additional headers* that contain our URLs in each of our requests, e.g., in the *Referer* header. Second, we can modify existing *URL parameters* in GET requests that contain encoded URLs pointing to other domains and instead make them point at our domain. Third, we can submit our URLs in the *HTTP body* of a POST requests, e.g., by filling out HTML forms on the pages we visit. In the following, we will describe each approach in more detail.

3.1 HTTP Headers

HTTP headers offer a convenient option for transferring metadata to a web server. Usually, they serve various purposes such as transmitting user agent strings, language preferences, or cookies with session information. However, there are also some lesser-known headers like *True-Client-IP* or *Forwarded* that usually contain hostnames or IPs and can cause certain systems like reverse proxies or load balancers to reveal themselves and send requests to these remote hosts [72]. Moreover, the Referer header is of particular interest to analytics software, which often visits the referring site to analyze its content with the aim of identifying what drives visitors to its own website [71].

In a preliminary study, we investigated the impact of 25 different headers taken from the *Collaborator Everywhere* project [73], which is a plugin for the *Burp Suite* proxy [3] that injects specific headers of this type into every intercepted HTTP request. During our crawls with these additional headers, we found that with 98%, almost all SSRs triggered by headers were caused by supplying a URL in the Referer header. On the other hand, some of these other 24 headers decreased the amount of successfully visited websites by up to 28%. Due to these results, we only use the Referer header and do not send other headers to discover SSR implementations.

3.2 URL Parameters

To discover parameters that could trigger an SSR, we scan the URL query string of all HTTP requests to all the included resources that were requested while loading a page. If the query string of one of these requests contains one or more (potentially encoded) URLs, we mark its position as a potential SSR candidate. Moreover, we also search existing query keys for interesting names like *url*, *host*, or *domain* and mark their value as an SSR candidate, regardless of if their value contained a URL or not. We then replace each SSR candidate separately with a unique subdomain pointing to our monitoring server and request the resulting URL again. Figure 2 contains our complete list of 22 SSR candidate names and is based on previous research on the detection of SSRF vulnerabilities and bug bounty reports [15, 27, 45, 73].

```
action, addr, address, domain, from, host, href, http_host, load, page, preview,
↪  proxy, ref, referer, referrer, rref, site, src, target, url, uri, web
```

Figure 2: The full list of 22 additional SSR URL parameter candidate names

It is important to note that only one parameter is modified at a time while the other remains unchanged, to increase the likelihood of the back end successfully processing our request. If our monitoring server then receives an incoming HTTP request on the respective unique subdomain, we can trace it back to the initiating parameter. Compared to the previously presented approach of inserting additional HTTP headers, this procedure thus requires multiple requests to the same resource—one for each SSR candidate.

```
http://example.com/some/service?param=foo&id=42
↪  &url=http%3A%2F%2Fsome-other-service.example.org&exec=yes

http://example.com/some/service?param=foo&id=42
↪  &url=http%3A%2F%2Funique-subdomain.our.monitoring&exec=yes
```

Figure 3: One example of an URL before and after our replacement of SSR candidates

3.3 HTML Forms

HTML forms often accept URLs in one or more fields and thus present another opportunity to discover SSRs. However, unlike URL parameters that can be collected by simply visiting websites, these POST requests usually have to be triggered by user interaction. Moreover, many forms implement both client- and server-side validation mechanisms designed to reject any input entered in the wrong format. Hence, to submit a form we need to fill each field with the correct input of the expected data type while inserting as many SSR–URLs as possible. For this, we extended our crawler with a form-filling algorithm, which tries to satisfy the constraints of the form while at the same inserts as many URLs pointing to our monitoring server as possible. A more detailed description of this approach as well as the pseudo code for our form filling algorithm can be found in Appendix A. Compared to the approach for URL parameters, this procedure does not allow us to simply collect submission requests and resend them later with arbitrary modifications as often as we want. Due to security measures such as *CSRF tokens*, submitting the same form multiple times or after a long delay would cause errors on the server-side and the transmitted data would be discarded. Therefore—and also for ethical reasons as outlined next—we only submit each form exactly once, potentially inserting a URL pointing to our monitoring server into multiple fields of the same form

3.4 Ethical Considerations

Since our work requires submitting *POST* requests to websites, we have to make sure that our method is as non-intrusive as possible. First of all, we do not provide an actual attack payload and instead only provide them with a URL that leads to an innocuous monitoring server. Moreover, we try to reduce the number of outgoing requests to these websites providers as much as possible. Since some forms like search bars or newsletter subscription forms may be present on multiple subpages of the same domain, we use a form deduplication technique to significantly reduce the number of form submissions. Finally, at both the experiment's subdomains and the root URL of our monitoring server, we deploy a disclaimer page explaining our research and presenting our organization's imprint and data protection regulation, intended for human visitors. We also offer anybody to opt-out of further crawls and respond to these requests in the time frame of about one business day. During our final data collection, we received and obliged to 5 opt-out requests on top of the 27 domains already blocked from previous crawls.

4 IDENTIFYING SSBS

In this section, we discuss the other end of our setup, i.e., the monitoring that collects data for all incoming requests. In particular, we need a way to accurately determine which of these incoming requests are from vulnerable browsers. The most conclusive way to test if the connecting browsers are vulnerable would be to use real JavaScript exploits and test which of them successfully compromises the visitors. Clearly, this would be neither legal nor ethical and is not an option. Instead, we resort to safe fingerprinting techniques that can determine the most likely user agent of our visitors and then map this information to a list of known vulnerabilities.

4.1 JavaScript Metadata

First of all, our server records some basic information like the IP address, *User-Agent* header, and timestamp for each incoming request. To distinguish simple SSRs from real browsers, we then reply with an HTML page that contains one small inline script. If this script executes, it sends a notification to our monitoring server. This way, we can not only discern visitors that execute JavaScript from those who do not but also collect additional metadata and send it to our back end. We specifically designed this website to also work with very old browsers, by not using any HTML5 features and writing our inline script according to the ECMAScript 5 standard, which was released in 2009 [1]. This website also does not include any external resources to load with a single request and the inline script executes in about 5 ms, to make sure that the notification is sent to our server before the page is closed again.

4.2 Bots vs. Human Visitors

While discerning SSR tools with and without JavaScript is straightforward with our monitoring website, discerning bots from humans, on the other hand, is a much more difficult task and often solved by either behavioral observation [47] or with CAPTCHAs [9]. Observing visitors over a series of requests allows finding indicators for bot-like behavior. We, however, do not run a real website with content that we want to protect from scraping, and instead need to discern humans from bots within *a single request* to our monitoring back end, making behavioral analysis not an option. Using a CAPTCHA is similarly not applicable, because every human visitor to our website that is asked to solve one, but instead closes the tab, would be misclassified as a bot. Previous work has shown that no programmatic bot indicator holds on its own and that even commercial fingerprinting companies lack robustness against concealed crawlers [80]. However, we have the unique advantage of knowing *when* to expect incoming bot traffic, because we first supply unique URLs to other websites in our attempts to trigger visits to our monitoring server. For this reason, our bot detection is based on the time period that passed between our initial request and the received SSR. We choose a very narrow time frame of 3 minutes starting from our initial trigger, because chances are extremely high that requests in that period come from a completely automated system. Even the most zealous administrator is unlikely to respond *that fast* to a new URL submitted to their website. Moreover, we extend this timing-based approach with additional bot indicators as described in the following subsection.

4.3 Additional Bot Indicators

To better incorporate slower bots, we add several additional bot indicators and increase our timing threshold accordingly. For this, we use the following insight: even though bots might try to conceal themselves as human visitors, hardly any human visitor will try to mimic an automated browser. Therefore, while the absence of bot properties is rather meaningless, their presence presents—to some extent—a useful indicator. In the following, we briefly present four bot indicators and then describe how we combine them with our timing-based methodology.

Known crawler user agents. Lists of known crawler user agents or patterns to detect those are widely available [e.g. 10, 56]. While crawlers regularly spoof the user agent information, we can assume that human visitor mimicking a known crawler are unlikely. Therefore, if the requested user agent is known as a bot or crawler, we use this as one supplemental indicator for our evaluation.

Inconsistent user agents. When bot operators try to conceal their automation tools, e.g., by altering the user agent string, inconsistencies can emerge when comparing the user agent as reported by the HTTP headers with the values reported by `navigator.userAgent` and `navigator.platform`. For example, a HTTP user agent of *Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:79.0)* is clearly spoofed, if `navigator.platform` reports to be running on *Linux x86_64*. While the values reported by the `navigator` object are also not trustworthy, inconsistencies in the reported user agent values from different sources are hard to explain with normal user behavior.

Inconsistent screen dimensions. Typically, crawlers operate in headless mode, i.e., the visited page is not rendered to decrease performance overhead, which can lead to mismatches in the reported screen resolution of the browser. Therefore, if the available inner screen dimensions exceed the reported outer dimensions, we consider the data as inconsistent and use it as a bot indicator. This reportedly does not necessarily hold true for mobile operating systems [8, 68] so we only apply this indicator for visitors who do not claim to be mobile devices.

Missing plugin information. Another consequence of specifically running Chromium-based browsers in headless mode, is that they do not support browser extensions when running without a GUI [24]. Under normal circumstances, Chrome always reports several plugins when reading the value of `navigator.plugins`, as the *Chrome PDF Viewer* and the *Native Client* are pre-installed. However, for Headless Chrome these plugins are missing, resulting in another useful bot indicator that is unlikely to apply for human visitors.

For each of these four indicators that is present during the visit, we double the initial threshold of 3 minutes during which we consider requests to be originating from bots. Therefore, if three of the four additional indicators would be present, the request would be labeled as bot traffic if it arrives within the first 24 minutes after our visit to their page. To summarize, we consider every visitor to be a human, until we find some hard indicators that they have to be a bot, e.g., because they are too fast and provide inconstant information about themselves. While this approach still overlooks slower bots, as well as particularly stealthy bots, we get a very reliable data set with little to no false positives, as it is very unlikely that human visitors have these indicators present and even if they do, it is even more unlikely that they are also fast enough for us to flag them as bots.

4.4 Feature Fingerprinting

Now that we know which visitors are bots running a server-side browser, we need to determine their browser version in order to identify those that are based on outdated browsers. However, as already described, the user agent string is trivial to spoof and sometimes even the user agent provided in the HTTP header and the user agent according to `navigator.platform` do not match—a clear indication of intentional manipulations. Therefore, the inline script on our monitoring website conducts a JavaScript *feature fingerprinting* that tests which features are supported by the visiting browser, to estimate the most likely *actual* user agent in a more reliable fashion. For this fingerprinting, we first probe for a few specific features to distinguish between the different browser *products*. For example, the presence of `InstallTrigger` is a unique indicator for the Firefox browser [77]. Now, to also distinguish between the different browser *versions*, we need a more sophisticated approach that works as follows: First, we compile a list of all JavaScript objects and properties of the global `window` object once, using the latest alpha release of Google Chrome. It is important to note, that we compiled this list while visiting an *HTTP* origin, as some newer features such as sensors are only available on a *HTTPS* origin. While our monitoring website is served on both protocols, for this fingerprinting we only rely on features that are always available, regardless of the protocol. At the time of writing, this list was compiled using Chrome 89 and contains 590 entries, such as `AggregateError`, `SVGAngle`, `Uint8Array`, and `WebGLQuery`. We embed this whole list into our inline script so that for each visitor with JavaScript enabled the presence of each of the 590 features is tested. We then encode the presence or absence of each feature into a long binary string and send this *feature vector* to our monitoring back end, along with the previously described other metadata.

4.5 Resulting User Agent

In our back end, we now need to decode this collected feature vector and map it back to the most likely browser versions. For this, we make use of the raw compatibility data provided by Mozilla's MDN [59], which is available on GitHub in their *browser-compat-data* repository [58]. As an example, their data shows that the `AggregateError` object is available in Chrome and Edge since v85, in Firefox since v79, in Safari since v14, and not supported by Opera. With this information for each of the 590 different features, we can narrow down the most likely browser version for each individual feature string. As the examples in Table 1 show, even with only a handful of features the range of possible versions can be quite small. For example, if a visitor supports `WeakRef` but not `AggregateError` then they are using Chrome, as Firefox introduced both features in the same update and all other browsers do not support `WeakRef` yet. Moreover, as `WeakRef` was introduced in Chrome 84 and `AggregateError` in Chrome 85, we now even know their exact major version for certain. However, it should be noted that not all fingerprint vectors are as distinct as this example and sometimes two successive releases might be indistinguishable.

Table 1: Five selected features and since which release the different browsers support them. The combination of presence (✓) and absence (✗) of the different features concludes that example 1 must be exactly Chrome 84 and example 2 must be Firefox 79 or newer.

Feature	Supported since				Example	
	Chrome	Firefox	Opera	Safari	1	2
AggregateError	85	79	–	14	✗	✓
MutationObserver	26	14	15	7	✓	✓
RTCCertificate	49	42	36	12	✓	✓
TrustedScript	83	–	69	–	✓	✗
WeakRef	84	79	–	–	✓	✓

This means we now have three potentially different user agents: from the HTTP headers, from the JavaScript navigator object, and from our feature fingerprinting. If these three agree with each other, then there is a high chance that the provided user agent information is indeed correct. When they do not match with each other, we could in doubt rely on the fingerprinting results as it is the most difficult to spoof. Yet, there is a chance that some administrators intentionally disabled a few features as an additional security measure. Therefore, we instead use the user agent with the highest browser version of the three as a conservative estimate, which in doubt defaults to the most secure of the three.

5 SSBS IN THE WILD

To answer the research questions outlined in Section 2.2, we conduct a large-scale study on SSBs in the wild on the 100,000 most popular websites according to the Tranco list [51] generated on March 2, 2021. On each website, we visit same-site links up to a depth of 10 or until we visited 50 pages, whichever comes first. If there are more than 50 same-site links on the landing page already, we randomly select 50 from among them. On each page, our crawler waits up to 30 seconds for the load event to trigger, otherwise, we flag the site as failed and move on. After the load event, we wait up to 3 more seconds for pending network requests to resolve to better handle pages that dynamically load additional content.

We started 60 parallel crawlers using Chromium 89.0.4389.72 on March 3 and finished the crawl about one week later on March 11. Of all the sites of the initial 100,000, we could only successfully visit about 79%. Of those that failed, about 8% were due to network errors, in particular, the DNS lookup often failed to resolve. In another 4%, the server returned an HTTP error code on the initial front page already. Additionally, 5.5% of these sites redirected to another domain which we subsequently discarded, since they are either duplicates like `blogger.com` and `blogspot.com`, or redirect to a location that is not part of the top 100k and thus out of scope. The remaining 3.5% failed due to various other issues, like failing to load before our 30 seconds timeout hit. In total, we successfully visited around 2.6M pages on about 79k sites. On these, we discovered 22.2M forms and submitted about 2.5M of them, the rest were considered duplicates. Additionally, we sent a total of 18M modified GET requests to about 5.6M different URLs in an attempt to discover SSRs that are triggered by URL parameters.

5.1 Postprocessing

First of all, we only recorded requests to subdomains with a unique ID generated for each possible SSR candidate on each website that we visited. Thus, generic requests from scanners and crawlers are not part of this data. For a meaningful analysis, we also have to prevent that our data is dominated by a few big companies, as they sometimes offer third-party scripts that trigger SSRs, e.g., DoubleClick and WordPress. These would then cause an incoming request any time we visit a domain that includes one of their affected scripts. Therefore, we use the *target domain* for attribution, e.g., if we submit a form included on *a.com* with an action URL to *b.com* that triggers an SSR, we consider *b.com* as the target domain and thus responsible for the incoming request, as it does matter less where we found this form and more where we submitted its data to. The same applies to SSRs triggered by modified URL parameters of third-party scripts.

Moreover, since these third parties are by nature present on a lot of websites, they would in total also send the most requests by far. As these companies also usually have access to vast IP ranges and heavily distribute their workload, we apply further deduplication based on the autonomous system number (ASN) — and not based on the IP address. Therefore, we define *unique requests* as those requests where the tuple *<target domain, asn, user agent>* is unique. If there are multiple, non-unique requests we use the fastest and ignore the rest. Additionally, we specifically exclude all requests from the *Googlebot* user agent, since their analytics products are widely used and result in many SSB visits from Google servers that would skew the analysis.

Finally, we only consider incoming requests that we received within one week of visiting the page. Otherwise, popular websites with a high rank, i.e., sites that we visited at the very beginning, would have had almost twice as much time to send an SSR than websites which we visited towards the end of our one-week-long crawl. Obviously, we continued the data collection for one week after we had visited the last page with our crawler.

5.2 All Incoming Requests

RQ1: How many websites trigger SSRs, i.e., visit other websites based on user-provided URLs?

In total, we recorded over 168,000 incoming requests, as Table 2 shows. Of these requests, we only consider 11,367 requests as unique according to our definition in Section 5.1. The JavaScript usage is quite low when looking at all recorded requests with 4.5%, however this is to be expected since by far not all automated requests need the capabilities of a full browser. Yet these 7,500 requests with JavaScript enabled already cover around 17% of the unique requests, as the total number of requests is heavily skewed towards a few third-party services that still often use simple SSR implementations without a real browser. Overall, we triggered requests on 4,850 different domains of the initial 100,000. Therefore, our experiments enticed about 6% of the successfully crawled 79,000 sites to visit us back at a URL we presented. And about 16% of these domains even did so with a real browser that has JavaScript execution enabled. Additional analyses about which of the triggers, i.e., HTTP headers, URL parameters, and HTML forms, were the most effective can be found in Appendix B.

Table 2: SSRs recorded during our large-scale study.

	# Total	# with JavaScript
All requests	168055	7503 (4.5%)
Unique requests	11367	1973 (17.4%)
Unique domains	4850	760 (15.7%)
Unique IPs	8636	1571 (18.2%)
Unique AS	917	610 (66.5%)

Looking into the temporal dimension, we found that around 35% of all requests arrived within 1 minute after we had visited their page, as Figure 4 shows. Yet, these 35% of requests in the first minute already cover about 50% of all domains due to repeated visits. Thus, on about 50% of sites where we could trigger *any* SSR, we had at least one visit *within the first minute*. On the other hand, the same is only true for 22% of the sites that visited us with JavaScript enabled. There are multiple reasons for this, e.g., SSBs could operate a bit slower than plain SSR implementations, and there are also likely humans still distorting the data, who manually visit our website much later after discovering our URL in their logs.

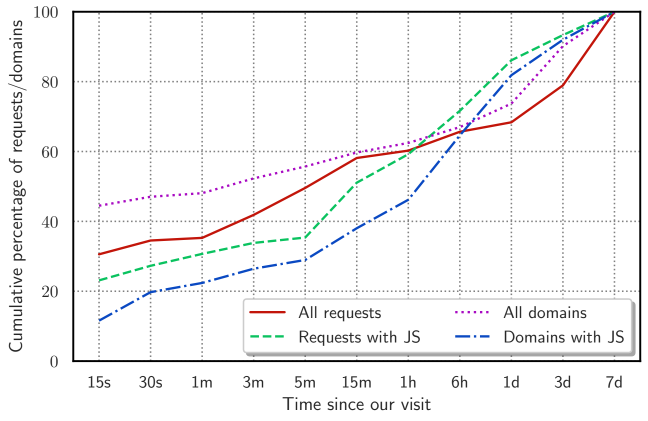


Figure 4: Incoming requests accumulated over time. Note the non-linear time axis.

5.3 Prevalence of SSBs

RQ2: How many of these SSRs are SSBs, i.e., use a real browser in their back end?

So far, we analyzed all incoming requests that we recorded. This also contains simple SSRs without a real browser, or from real browsers controlled by humans instead of automated systems. For all following analyses, we will now focus exclusively on SSBs, i.e., *real, automated browsers* with JavaScript enabled. As described in Section 4, discerning SSR tools without JS from real browsers with JS is straight-forward. On the other hand, discerning bots from humans with only a single request and without interaction is much more difficult. In order to do so nevertheless, we use the time difference between our visit to their website and their visit to our monitoring server and only consider cases where this time difference was less than 3 minutes, as such a fast reaction is a strong indicator for automated behavior. Moreover, as previously described in Section 4.3, we also extend this time frame if additional bot indicators are present.

As already shown in Figure 4, focusing on only those requests that have JavaScript enabled and happened within in the first few minutes after our visit to their server reduces the total number of requests significantly. As Table 3 shows, 532 unique requests on 254 different domains were caused by bots running an SSB. Of these, 433 unique requests on 192 domains were labeled as bots because they arrived within 3 minutes. The remaining 99 requests on an additional 62 domains were labeled as bots due to a combination of a time threshold and our additional bot indicators. Moreover, Table 3 also shows the *type of triggers*, i.e., whether the visit was caused by the referer header we sent, a modified URL parameter, or a form submission. As the table shows, forms were most successful in attracting SSBs, being responsible for about 65% of all cases.

Table 3: Prevalence of server-side browsers on the left and their causes on the right-hand side. This is a summary of Tables 1 and 2, but only with the requests made by SSBs.

	# Requests
All requests	3264
Unique requests	532
Unique IPs	440
Unique AS	206

	# Domains
All domains	254 (100.0%)
Header	58 (22.8%)
Param	34 (13.4%)
Form	167 (65.7%)

Next, we analyze the characteristics of the affected websites themselves and found that SSB implementations were significantly more common on the 10,000 most popular websites with 51 SSBs compared to the average of 23 SSBs per 10,000 sites in the remaining 90,000 sites. We found 30 of them within the top 5,000 and 14 of them even within the top 1,000 of the most popular websites according to the Tranco ranking. Moreover, we also investigated what types of categories these 254 sites belong to, using the *WebPulse Site Review* service [5] operated by the security company Symantec. As Table 4 shows, these sites are mostly related to technology and business.

Table 4: Categories of the sites with an SSB.

Category	# Sites	Category	# Sites
Analytics	7	News	14
Business	57	Other	39
Entertainment	3	Shopping	25
Education	21	Technology	69
Government	8	Uncategorized	11

Besides the websites themselves, we also investigated the origin of these SSB connections. We found that despite our deduplication, for some websites we received multiple, unique requests from real browsers even within the first few minutes. On 63 out of the 254 domains, we received requests from IP addresses belonging to two or more different ASNs. 20 of these even connected from 4 or more different ASNs and in one extreme case, one website even caused incoming connections from 22 different ASNs. On the other hand, we also found some networks that are responsible for a greater number of incoming connections for *different domains*. The three most popular ones were *AS8075* with connections triggered on 35 different domains, *AS15169* with 34 different domains, and *AS14618* and *AS16509* each with 18 domains. Closer investigation reveals that all these ASNs are related to cloud hosting, the first is owned by Microsoft and used for Azure, the second by Google for their cloud platform, while the other two are part of Amazon's AWS. Therefore, we can not assume a direct relation between these incoming requests, as many different companies likely just rely on the same, third-party hosting provider.

5.4 Investigation of Browser Versions

RQ3: Which browsers are used in these SSB implementations and at which major version?

The 532 unique requests by bots (with JavaScript enabled and arriving within our time threshold) were conducted by 157 different HTTP user agents (UAs). The supposedly most popular used browser versions were *Chrome 84* in 122 requests, *Chrome 85* in 48 requests, and *Chrome 88* in 34 requests. However, only 64 requests had an HTTP UA that additionally clearly indicated that a bot is visiting us. For example, by either containing a reference to their service like *SpeedCurve* [76] in 3 cases or containing a general reference to an automation framework like *Headless Chrome* in 36 cases, *PhantomJS* in 7 cases, and *Lighthouse* [37] in 6 cases as the most popular tools in our data set.

As previously outlined, this UA information in the HTTP header is easily spoofed and should not be trusted blindly. Thus, we next compare these supposed values to the UA and platform information provided by JavaScript's `navigator` object. Looking into our collected data, we find that 13 unique requests definitely lied to us based on a mismatch between the HTTP UA and the JavaScript UA. Moreover, we also find that another striking 124 cases where the two UAs actually *do match* with each other but do not match with the platform information reported in the JavaScript environment. For example, these have a UA starting with *Mozilla/5.0 (Windows NT 10.0; Win64; x64)*, but `navigator.platform` reports *Linux x86_64* as their operating system. Table 5 list the most common examples of UA and platform mismatches.

Table 5: Most frequently faked user agent strings (abbreviated) and the reported but mismatching platform

# Req.	HTTP Header	Platform
17	CPU iPhone OS 13_7 [...] Version/13.1.2	Linux x86_64
9	Windows NT 6.1 [...] Chrome/83.0.4103.106	Linux x86_64
9	Windows NT 6.1 [...] Firefox/77.0	Linux x86_64
7	Windows NT 10.0 [...] Chrome/79.0.3945.79	Linux x86_64
4	iPad; CPU OS 11_4 [...] Version/11.0	Linux x86_64

These findings not only confirm that these are indeed bots, but also that they take some efforts to stay undetected by spoofing both UA string values. As expected, these bots with UAs claiming to be running on a Windows PC, an iPhone, or iPad, are actually all running on a Linux server, the preferred distribution for servers running automated tasks. With 137 requests with obviously spoofed information in total, about *one quarter* of the unique bot requests lied about their user agents, confirming that this voluntarily provided information should indeed not be relied upon. Moreover, these findings should only be seen as a lower bound, as the value of `navigator.platform` could obviously be fake, as well.

To analyze their browsers in greater detail, we instead conduct a *feature fingerprinting* of all visitors, as described in Section 4.4. This way, we can determine their UA in an objective manner, without relying on potentially spoofed user agent strings. For 282 of the 532 unique requests, the version determined by the fingerprinting indeed did match the version of their HTTP and JavaScript UA value. In this case, we define *match* as within one major release as the fingerprint might not always be distinguishable for browsers with a fast release cycle. Of the remaining 250 requests where the user agents did not match our feature fingerprint, 80% claimed to be older than our fingerprint determined them to be with 201 requests, while 41 requests came from browsers claiming to be newer than our fingerprinting determined them to be. The remaining 8 requests did not send a browser version in their user agent information at all. Additionally, of those 124 requests that were previously found to be lying about their OS, in only 12 cases did their provided UA browser version match the results of our fingerprinting. This means, that in the remaining 112 cases, their browser version was apparently manipulated, too.

As described in Section 4.5, we then use the newest version derived from the three UAs (HTTP, JavaScript, fingerprint) as a conservative estimate for the actually used browser version. Table 6 shows the results, in which Chrome 84 is the most popular browser version with 150 unique requests. Combined with the other outdated, but popular versions Chrome 85 and Chrome 86, these three already make up for over 60% of all unique requests that we received. Chrome 88, which was the latest stable version of Chrome at the time of our crawl, only was responsible for 100 (19%) of SSB requests. In the next section, we will discuss the consequences to the security of all these servers running outdated SSBs in detail.

Table 6: Number of requests by the five most popular user agents in SSBs as reported by our three different indicators. When the indicated versions differ, we use the newest of the three as the resulting user agent.

Browser	Indicator			Resulting UA
	HTTP Header	JavaScript	Fingerprint	
Chrome 88	34	29	112	100
Chrome 86	2	2	84	84
Chrome 85	48	48	48	95
Edge 85	12	12	0	12
Chrome 84	122	127	204	150

5.5 Vulnerable SSBs in the Wild

RQ4: How many of these SSBs are running outdated browsers vulnerable to publicly available exploits?

While there is a certain risk that even a fully up-to-date browser is exploited [74], in our scenario, we instead focus on outdated browsers in the wild. In the following, we describe our methodology to determine which browsers were vulnerable at the time of our crawl and how realistic it is to obtain a working exploit for them. Since 93% of the 532 requests were using Google Chrome or a Chromium-based browser like Edge, we will only discuss the security of different versions of this browser in detail.

Google released the latest stable Chrome version 89 for all platforms on March 2, i.e., one day before we started our crawl. This update contains 8 security fixes with a high severity, however they only started to roll out the update "over the coming days/weeks" [19]. The previous version 88 had been available for several weeks already and also contains many important security fixes, some of which earned a bug bounty of 10,000 dollars or more [18]. However, at time of writing, the details of these recently patched vulnerabilities in Chrome 88 and 89 had not yet been revealed in Google's bug tracker [e.g., 25], therefore we do not know if the bug can be abused without user interaction. Nevertheless, reverse engineering patched software to create a working 1-day exploit is, in general, easier than searching for 0-day vulnerabilities from scratch [64]. As major browsers are nowadays open-source software, attackers do not even need to employ *binary diffing* techniques [e.g., 12, 31, 55] but can directly look a human-readable diffs including detailed comments about the changes. As previous research has shown, sometimes it might even be possible to automatically discover the relevant security patches among the huge list of changes [86], as well as automatically create exploits from the identified patches [14].

However, for vulnerabilities patched with the release of Chrome 87 and earlier, even the full details of the bugs including proof of concept (PoC) exploits and a discussion by Chrome engineers were already publicly available at the time of our crawl [e.g., 23]. Thus, while Chrome 87 and 88 could likely be exploited by a skilled attacker reverse-engineering the patches, we nevertheless use a very conservative estimate here and only consider browser versions to be vulnerable if publicly disclosed, detailed information about their vulnerabilities exists. Therefore, we consider version 86 and all older versions of Chrome to be vulnerable at the time of our crawl. This also applies to Microsoft Edge, which is based on Chromium and shares their version naming scheme, and Puppeteer, which uses Headless Chrome internally.

Table 7: The five most popular SSBs with the number of requests and affected domains in our study. For reference, we also include their release date and a high/critical CVE with a PoC exploit for that specific version.

Browser	Requests	Domains	Release	CVE	PoC
Chrome 88	100	83	01/21	—	—
Chrome 86	84	44	10/20	2020-16015	[23]
Chrome 85	95	39	08/20	2020-6575	[21]
Edge 85	12	10	08/20	2020-6575	[21]
Chrome 84	150	68	07/20	2020-6559	[22]

With this information about the security of these different releases in mind, we now come back to the 532 unique requests. Of these, 405 (76%) were conducted with a browser version vulnerable to publicly available exploits, resulting in 168 out of the 254 domains with SSBs to be vulnerable. This means, that *two out of three*

SSB implementations that visit and subsequently execute arbitrary, attacker-controlled JavaScript code, are running a severely outdated and vulnerable browser with publicly disclosed PoC exploits. As shown in Table 7, the most popular browser versions were already quite dated during our experiments in March 2021 and many of them were released more than *half a year ago*. For reference, the table also includes three CVEs for the most popular browsers versions we encountered. All of these three CVEs work on all major platforms, including Linux, are exploitable without user interaction, and have their details including a PoC exploit publicly available. When looking at Figure 5, we see that even in the top 10,000 domains, over half of the websites with an SSB are vulnerable. All in all, this demonstrates that server-side browsers pose a considerable risk to any organization that makes use of them and is a, so far, widely overlooked problem.

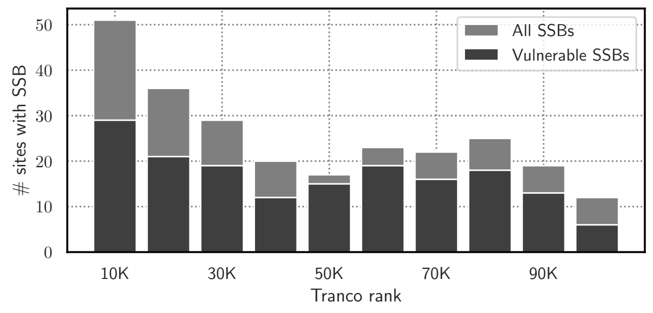


Figure 5: Distribution of vulnerable SSB implementations over Tranco ranks. The most popular sites are on the left.

6 DISCUSSION

In this section, we discuss how various aspects of our methodology and analysis might have influenced our findings. False positives in the sense that we misclassify a simple SSR as an SSB are impossible, as the former would never be able to generate the exact request required to register as a client with JavaScript execution. False positives in the sense that we misclassify a human visitor as a bot are also quite unlikely since we generate a unique URL that must be visited within 3 minutes if no additional indicators for an automated system are present in their request. Therefore, we are confident that all our findings are indeed valid and represent actual, vulnerable server-side browsers. However, it is not clear how many of them are secured by additional means, e.g., make exploitation difficult because they are running in an isolated network or inside a restricted virtual environment such as a sandbox. Unfortunately, without actually abusing the vulnerabilities to explore the underlying system in more detail, we can not report on the prevalence of these countermeasures. On the other hand, there are also three reasons why we might have missed SSB implementations:

Crawling depth and user interaction. One obvious limitation of our large-scale study is that we can only visit a limited amount of sub-pages on each domain and might have missed opportunities to trigger SSBs simply by not crawling deep enough. Moreover, SSBs might be hidden behind advanced user interaction, e.g., only accessible for logged-in users. While recently some efforts have been made towards *authenticated crawling* [48], this still represents a largely unsolved research challenge and is out of scope for this work.

Server-side filters. An inherent problem of *blind* attacks is that we only get notified about successfully triggered implementations. Fortunately, this is usually only then a problem when trying to launch an SSRF attack, e.g., when trying to access internal resources. The important difference here is that we are not trying to circumvent server-side filters and other countermeasures against attacks, but rather use their SSB service *as intended* by simply requesting it to visit an external website.

Bot detection. As outlined in Section 4, discerning human visitors from bots *within a single request* is challenging. To not accidentally introduce any false positives into our data set, we use a very conservative cutoff and ignoring all requests that did not arrive within 3 minutes of our visit to their page. Naturally, this means we likely missed many slower bots that did visit us too late, e.g., because they still had a queue of other URLs to visit first.

In summary, this means there are multiple reasons why we probably reported an *underapproximation* of the actual extent of the problem and vulnerable server-side browsers might be actually even more prevalent than our findings suggest. However, the main goal of our study was not to precisely measure the exact amount of SSBs at a specific point in time, as the Internet is a very dynamic system and these numbers fluctuate daily anyways. Rather, this work aims to show that server-side browsers are an underestimated attack surface for the servers running them, as the majority of implementations that we discovered were vulnerable to publicly available exploits.

7 RELATED WORK

In this section, we first discuss the process of automated vulnerability scanning and the concept of blind vulnerabilities in general. Then, we describe previous work on the topic of server-side requests in detail and discuss how it relates to this work.

7.1 Vulnerability Scanners

Generally speaking, our approach to discover SSR services is related to *black-box vulnerability scanners* [e.g., 17, 28, 43, 49]. More specifically, server-side requests are one instance of a vulnerability class called *blind vulnerabilities.* One classic example of this class are *BlindSQL* [65] injections, where the attacker might not get a direct response of the query's results in text form, but can still infer their contents based on the application's response time to different queries. Another example of this are *Blind XML External Entities (XXE)* [75] attacks, in which links in XML sheets are abused to leak data. To aid the discovery of blind vulnerabilities, PortSwigger added the *Collaborator Everywhere* plugin [73] to their popular attack proxy *Burp Suite* [70].

7.2 Browser Fingerprinting and Bot Detection

Web fingerprinting as a means to recognize repeated visitors, i.e tracking users without storing a client-side state such as cookies, has been studied for over two decades [e.g., 7, 29, 30, 53, 54]. However, in contrast to these previous works, detecting *reoccurring visits* by the same browser is not in our scope and we also can not rely on longer-term behavioral analysis [26, 47] of visitors in order to differentiate bots from humans. Furthermore, we can not resort to crawler-specific bot indicators, such as crawler traps [13, 80], or access log and traffic analysis [78, 85] on a series of requests, since the SSBs we target do not necessarily behave like crawlers. Instead, we are interested in identifying automated visitors with a real browser (RQ2) and their underlying browser version (RQ3) within a single request.

Thus, more closely related to our focus is research on detecting browser versions through JavaScript runtime and performance information [42, 57]. Unfortunately, these techniques suffer from long time requirements, making it impractical to analyze bots which, unlike humans, do not tend to leave browser tabs open for a long time. Moreover, approaches like Red Pills to detect virtualization are susceptible to processing and network bottlenecks which might introduce noise [16, 42]. Therefore, to infer the browser version, we instead leverage a JavaScript engine fingerprinting approach similar to Mulazzani et al. [62]. However, as previous work has shown, differentiating between bots and humans based on such fingerprints is problematic [80]. Yet in contrast to previous works we have additional knowledge about *when* to expect a visit and can identify bots based on this timing information.

7.3 Server-Side Requests

Research specifically on the topic of server-side requests is rare. Stivala and Pellegrino [79] studied how link previews on social media platforms can be manipulated to create benign-looking previews for malicious links. While the underlying link preview implementation makes use of server-side requests to fetch this information, the security of these implementations was not studied as part of their paper. In 2017, Orange Tsai [66] presented their findings on how differences in URL parsers cause filter bypasses leading to SSRF vulnerabilities in seemingly secure implementations. In 2021, Jabiyev et al. [46] performed a manual analysis of 61 HackerOne SSRF vulnerabilities and found that developer awareness for this vulnerability was still low. The authors propose a generic defense mechanism that proxies all SSRs through a helper server with no access to the internal network of that company, preventing the exfiltration of internal information. However, as all requests are forwarded through the proxy and still rendered and executed on the original, internal server, their defense would not protect against our attacker model.

Most closely related to our work is the 2016 publication titled "Uses and Abuses of Server-Side Requests" by Pellegrino et al. [69]. For their research, they developed *Günther*, a scanning tool that probes server-side backends for SSRF vulnerabilities. While they discuss the threat of the SSR implementation itself getting exploited, their attacker model in this case only considers DoS attacks, such as keeping the SSR provider busy with decompression tasks eventually leading to memory exhaustion. On the other hand, our work is, to the best of our knowledge, the first to study the consequences of running full *server-side browsers* with JavaScript execution in the wild. Unlike previous publications, we do not investigate traditional SSRF vulnerabilities and do also not try to circumvent filters or to confuse parsers. Instead, we use the SSR service *as intended* and instead directly attack those requesting clients that run a fully-featured, but outdated browser engine. Moreover, we are the first to systematically investigate this phenomenon on a large-scale and report on server-side requests conducted by the 100,000 most popular websites as compared to the previous studies on less than 100 websites.

8 CONCLUSION

In this paper, we studied the phenomenon of running an automated browser on the server-side. These SSBs represent a unique attack surface as they execute untrusted code on the server in a full browser, which is one of the most complex pieces of software of our times. Consequently, critical security bugs are constantly found and at a much higher rate than in other server software typically exposed to the public such as *Nginx* or *Apache*. Even worse, if an library such as *Puppteer* is included as a dependency in a custom tool, the underlying browser will not update automatically when security patches are installed on the server, thus further increasing the risk.

During our large-scale crawl of the top 100,000 domains, we discovered that we could trigger server-side requests on a significant fraction of around 6% of the visited domains. Moreover, 16% of these even conducted their requests with an SSB, i.e., with a real browser that also executes JavaScript code. We analyzed these browsers in detail and used a technique based on feature fingerprinting to determine the version number of these browsers, without relying on potentially spoofed user agent information. We found that of the 254 domains where we could confirm that an SSB is used, with 168 of them the majority used severely outdated browsers. Not only are the details of their vulnerabilities publicly disclosed, but they are even accompanied by a full proof-of-concept exploit. This emphasizes how underestimated the risks of running a full browser on the server-side currently are.

ACKNOWLEDGMENTS

We would like to thank all anonymous reviewers for their valuable comments and suggestions. Moreover, we gratefully acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

REFERENCES

[1] 2009. ECMAScript 2009 Language Specification 5th Edition. Online https://www.ecma-international.org/wp-content/uploads/ECMA-262_5th_edition_december_2009.pdf.
[2] 2020. Google Search Central - Googlebot evergreen rendering in our testing tools. Online https://developers.google.com/search/blog/2019/08/evergreen-googlebot-in-testing-tools.
[3] 2021. Burp Suite. Online https://portswigger.net/burp.
[4] 2021. Microsoft Bing Blogs - bingbot Series: JavaScript, Dynamic Rendering, and Cloaking. Oh My! Online https://blogs.bing.com/webmaster/october-2018/bingbot-Series-JavaScript,-Dynamic-Rendering,-and-Cloaking-Oh-My.
[5] 2021. Symantec Sitereview: WebPulse Site Review Request. Online https://sitereview.bluecoat.com/.
[6] 2021. VirusTotal. Online https://www.virustotal.com/gui/home/url.
[7] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. 2013. FPDetective: Dusting the Web for Fingerprinters. In *Proc. of ACM Conference on Computer and Communications Security (CCS).*

[8] Apple. 2021. iOS SDK Release Notes for iOS 8.0 GM. Online https://developer.apple.com/library/archive/releasenotes/General/RN-iOSSDK-8.0/.

[9] Babak Amin Azad, Oleksii Starov, Pierre Laperdrix, and Nick Nikiforakis. 2020. Web runner 2049: Evaluating third-party anti-bot services. In *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*.

[10] Mark Beech. 2021. GitHub: Crawler-Detect. Online https://github.com/JayBizzle/Crawler-Detect.

[11] Eric Bidelman. 2021. Headless Chrome: an answer to server-side rendering JS sites. Online https://developers.google.com/web/tools/puppeteer/articles/ssr.

[12] Martial Bourquin, Andy King, and Edward Robbins. 2013. BinSlayer: accurate comparison of binary executables. In *Proc. of the ACM SIGPLAN Program Protection and Reverse Engineering Workshop (SSPREW)*.

[13] Douglas Brewer, Kang Li, Laksmish Ramaswamy, and Calton Pu. 2010. A Link Obfuscation Service to Detect Webbots. In *Proc. of IEEE International Conference on Services Computing*.

[14] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proc. of IEEE Symposium on Security and Privacy*.

[15] Bugcrowd. 2018. GitHub: HUNT - SSRF Python script. Online https://github.com/bugcrowd/HUNT/blob/master/ZAP/scripts/passive/SSRF.py.

[16] Elie Bursztein, Artem Malyshev, Tadek Pietraszek, and Kurt Thomas. 2016. Picasso: Lightweight Device Class Fingerprinting for Web Clients. In *Proc. of ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*.

[17] Jan-Min Chen and Chia-Lun Wu. 2010. An automated vulnerability scanner for injection attack based on injection point. In *Proc. of International Computer Symposium (ICS)*.

[18] Chrome Releases. 2021. Chrome 88 Stable Channel Update for Desktop. Online https://chromereleases.googleblog.com/2021/02/stable-channel-update-for-desktop.html.

[19] Chrome Releases. 2021. Chrome 89 Stable Channel Update for Desktop. Online https://chromereleases.googleblog.com/2021/03/stable-channel-update-for-desktop.html.

[20] Chromium Blog. 2021. Speeding up Chrome's release cycle. Online https://blog.chromium.org/2021/03/speeding-up-release-cycle.html.

[21] Chromium Bug Tracker. 2020. Issue 1081874: Double free on NodeChannel. Online https://crbug.com/1081874.

[22] Chromium Bug Tracker. 2020. Issue 1137630: PDFium heap-use-after-free. Online https://crbug.com/1137630.

[23] Chromium Bug Tracker. 2020. Issue 1146670: TFC chrome full chain. Online https://crbug.com/1146670.

[24] Chromium Bug Tracker. 2020. Issue 706008: Extensions support in headless Chrome. Online https://bugs.chromium.org/p/chromium/issues/detail?id=706008.

[25] Chromium Bug Tracker. 2021. Issue 1138143: segmentation fault in mojom. Online https://crbug.com/1138143.

[26] Zi Chu, Steven Gianvecchio, Aaron Koehl, Haining Wang, and Sushil Jajodia. 2013. Blog or block: Detecting blog bots through behavioral biometrics. *Computer Networks* (2013).

[27] d0nut. 2018. HackerOne: SSRF on duckduckgo.com/iu/. Online https://hackerone.com/reports/398641.

[28] Adam Doupé, Marco Cova, and Giovanni Vigna. 2010. Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*.

[29] Peter Eckersley. 2010. How unique is your web browser?. In *Proc. of Privacy Enhancing Technologies Symposium (PETS)*.

[30] Steven Englehardt and Arvind Narayanan. 2016. Online Tracking: A 1-million-site Measurement and Analysis. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*.

[31] Halvar Flake. 2004. Structural comparison of executable objects. In *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*.

[32] Google. 2021. Chrome keeps you up to date. Online https://www.google.com/chrome/update/.

[33] Google. 2021. Chrome Release Cycle. Online https://chromium.googlesource.com/chromium/src/+/master/docs/process/release_cycle.md.

[34] Google. 2021. Puppeteer. Online https://pptr.dev/.

[35] Google. 2021. Understanding your Chrome Browser update options. Online https://services.google.com/fh/files/misc/chromeenterprisebrowser_updatestrategies_mktgwp_5.1.19.pdf.

[36] Google Developers. 2020. Chrome DevTools. Online https://developers.google.com/web/tools/chrome-devtools/.

[37] Google Developers. 2021. Lighthouse. Online https://developers.google.com/web/tools/lighthouse.

[38] Google Groups. 2021. Q4 2019 Summary from Chrome Security. Online https://groups.google.com/a/chromium.org/g/security-dev/c/fbiuFbW07vI.

[39] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[40] Ariya Hidayat. 2018. GitHub: PhantomJS – Archiving the project: suspending the development. Online https://github.com/ariya/phantomjs/issues/15344.

[41] Ariya Hidayat. 2020. PhantomJS: Scriptable Headless Browser. Online https://phantomjs.org/.

[42] Grant Ho, Dan Boneh, Lucas Ballard, and Niels Provos. 2014. Tick Tock: Building Browser Red Pills from Timing Side Channels. In *Proc. of USENIX Workshop on Offensive Technologies (WOOT)*.

[43] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. 2003. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *Proc. of the International World Wide Web Conference (WWW)*.

[44] Luca Invernizzi, Kurt Thomas, Alexandros Kapravelos, Oxana Comanescu, Jean-Michel Picod, and Elie Bursztein. 2016. Cloak of Visibility: Detecting when Machines Browse a Different Web. In *Proc. of IEEE Symposium on Security and Privacy*.

[45] Noriaki Iwasaki. 2019. HackerOne: SSRF in Search.gov via ?url= parameter. Online https://hackerone.com/reports/514224.

[46] Bahruz Jabiyev, Omid Mirzaei, Amin Kharraz, and Engin Kirda. 2021. Preventing Server-Side Request Forgery Attacks. In *Proc. of ACM Symposium on Applied Computing (SAC)*.

[47] Gregoire Jacob, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2012. PUBCRAWL: Protecting Users and Businesses from CRAWLers. In *Proc. of USENIX Security Symposium*.

[48] Hugo Jonker, Stefan Karsch, Benjamin Krumnow, and Marc Sleegers. 2020. Shepherd: a generic approach toautomating website login?. In *Proc. of Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*.

[49] Stefan Kals, Engin Kirda, Christopher Krügel, and Nenad Jovanovic. 2006. SecuBat: a Web vulnerability scanner. In *Proc. of the International World Wide Web Conference (WWW)*.

[50] Christoph Kerschbaumer, Tom Ritter, and Frederik Braun. 2020. Hardening Firefox against Injection Attacks. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*.

[51] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proc. of Network and Distributed System Security Symposium (NDSS)*.

[52] Nektarios Leontiadis, Tyler Moore, and Nicolas Christin. 2011. Measuring and Analyzing Search-Redirection Attacks in the Illicit Online Prescription Drug Trade.. In *Proc. of USENIX Security Symposium*.

[53] Jonathan R Mayer. 2009. Any person... a pamphleteer": Internet Anonymity in the Age of Web 2.0.

[54] Jonathan R. Mayer and John C. Mitchell. 2012. Third-Party Web Tracking: Policy and Technology. In *Proc. of IEEE Symposium on Security and Privacy*.

[55] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *Proc. of USENIX Security Symposium*.

[56] Martin Monperrus. 2021. GitHub: crawler-user-agents. Online https://github.com/monperrus/crawler-user-agents.

[57] Keaton Mowery, Dillon Bogenreif, Scott Yilek, and Hovav Shacham. 2011. Fingerprinting Information in JavaScript Implementations. In *Proc. of IEEE S&P Web 2.0 Security & Privacy Workshop (W2SP)*.

[58] Mozilla. 2021. GitHub: mdn/browser-compat-data. Online https://github.com/mdn/browser-compat-data.

[59] Mozilla. 2021. MDN Web Docs. Online https://developer.mozilla.org/.

[60] Mozilla. 2021. Security Advisories for Firefox. Online https://www.mozilla.org/en-US/security/known-vulnerabilities/firefox/.

[61] Mozilla. 2021. Update Firefox to the latest release. Online https://support.mozilla.org/en-US/kb/update-firefox-latest-release.

[62] Martin Mulazzani, Philipp Reschl, Markus Huber, Manuel Leithner, Sebastian Schrittwieser, and Edgar Weippl. 2013. Fast and Reliable Browser Identification with JavaScript Engine Fingerprinting. In *Proc. of IEEE S&P Web 2.0 Security & Privacy Workshop (W2SP)*.

[63] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*.

[64] Jeongwook Oh. 2009. Fight against 1-day exploits: Diffing Binaries vs Anti-diffing Binaries. Online http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.694.8684&rep=rep1&type=pdf.

[65] Open Web Application Security Project. 2021. Blind SQL Injection. Online https://owasp.org/www-community/attacks/Blind_SQL_Injection.

[66] Orange Tsai. 2017. A New Era of SSRF – Exploiting URL Parser in Trending Programming Languages. Online https://www.blackhat.com/docs/us-17/thursday/us-17-Tsai-A-New-Era-Of-SSRF-Exploiting-URL-Parser-In-Trending-Programming-Languages.pdf.

[67] OWASP. 2021. OWASP Top 10. Online https://owasp.org/Top10/.

[68] James Pearce. 2021. First, Understand Your Screen. Online https://tripleodeon.com/2011/12/first-understand-your-screen/.
[69] Giancarlo Pellegrino, Onur Catakoglu, Davide Balzarotti, and Christian Rossow. 2016. Uses and Abuses of Server-Side Requests. In *Proc. of International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.
[70] PortSwigger. 2021. Burp Suite. Online https://portswigger.net/burp.
[71] PortSwigger. 2021. SSRF via the Referer header. Online https://portswigger.net/web-security/ssrf.
[72] PortSwigger Ltd. 2021. Cracking the lens: Targeting HTTP's Hidden Attack-Surface. Online https://portswigger.net/research/cracking-the-lens-targeting-https-hidden-attack-surface.
[73] PortSwigger Ltd. 2021. GitHub: collaborator-everywhere. Online https://github.com/PortSwigger/collaborator-everywhere.
[74] Project Zero. 2021. In-the-Wild Series: October 2020 0-day discovery. Online https://googleprojectzero.blogspot.com/2021/03/in-wild-series-october-2020-0-day.html.
[75] Christopher Späth, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. 2016. SoK: {XML} parser vulnerabilities. In *Proc. of USENIX Workshop on Offensive Technologies (WOOT)*.
[76] SpeedCurve Ltd. 2021. SpeedCurve: Monitor front-end performance. Online https://speedcurve.com/.
[77] StackOverflow Community. 2020. How to detect Safari, Chrome, IE, Firefox and Opera browser? Online https://stackoverflow.com/a/9851769.
[78] Athena Stassopoulou and Marios D. Dikaiakos. 2009. Web robot detection: A probabilistic reasoning approach. *Computer Networks*.
[79] Giada Stivala and Giancarlo Pellegrino. 2020. Deceptive previews: A study of the link preview trustworthiness in social platforms. In *Proc. of Network and Distributed System Security Symposium (NDSS)*.
[80] Antoine Vastel, Walter Rudametkin, Romain Rouvoy, Xavier Blanc, and Antoine Vastel Datadome. 2020. FP-Crawlers: Studying the Resilience of Browser Fingerprinting to Block Crawlers. In *Proc. of Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*.
[81] Paul Vorbach. 2021. npm-stat: phantomjs. Online https://npm-stat.com/charts.html?package=phantomjs&from=2015-01-25&to=2021-02-24.
[82] Paul Vorbach. 2021. npm-stat: puppeteer. Online https://npm-stat.com/charts.html?package=puppeteer&from=2017-06-01&to=2021-02-24.
[83] David Y Wang, Matthew Der, Mohammad Karami, Lawrence Saul, Damon McCoy, Stefan Savage, and Geoffrey M Voelker. 2014. Search+ seizure: The Effectiveness of Interventions on SEO Campaigns. In *Proc. of Internet Measurement Conference (IMC)*.
[84] David Y Wang, Stefan Savage, and Geoffrey M Voelker. 2011. Cloak and Dagger: Dynamics of Web Search Cloaking. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*.
[85] Haitao Xu, Zhao Li, Chen Chu, Yuanmi Chen, Yifan Yang, Haifeng Lu, Haining Wang, and Angelos Stavrou. 2018. Detecting and Characterizing Web Bot Traffic in a Large E-commerce Marketplace. In *Computer Security*.
[86] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. SPAIN: Security Patch Analysis for Binaries towards Understanding the Pain and Pills. In *Proc. of International Conference on Software Engineering (ICSE)*.

A FORM FILLING AND DEDUPLICATION

Here, we describe in more detail how our form filling algorithm works. The algorithm depicted in Figure 6 has two primary goals: First, we want to reduce the number of outgoing requests to website providers as much as possible. Secondly, to discover SSRs, we want to insert as many URLs pointing to our monitoring server as possible, while satisfying potential input validation.

```
1  for form in page.findAllForms():
2      formHash = calculateFormHash(form)
3      receiverUrl = getReceiverUrl(form)
4
5      // Skip duplicates
6      if (receiverUrl, formHash) not in submissionCache:
7          // Fill and submit form
8          for field in form.getInputFields():
9              if allowsUrl(field):
10                 field.setValue(generateMonitoringUrl())
11             else:
12                 field.setValue(getDefaultValue(input))
13
14         form.submit()
15
16         // Prevent duplicates
17         submissionCache.add((receiverUrl, formHash))
```

Figure 6: Pseudo code showing our form filling and deduplication mechanism

For our first goal, we employ a deduplication mechanism that detects forms with high similarity on multiple subpages of the same or different websites. To recognize already submitted forms, we create a hash representation of each form's `outerHTML` excluding already filled values like hidden tokens and collect the URL the form is posted to (line 2-3). A form is only submitted, if no form with an identical hash representation has been submitted to the same receiver URL (line 6). For websites that, for example, employ a newsletter subscription form in their sidebar, or a comment functionality on every subpage of a paginated blog, this technique drastically reduces the number of submissions, since each form is only filled once.

If the form is new, we decide for each individual input field whether we can insert a URL (line 8-9). For that purpose, the algorithm determines the semantic data type of each HTML input element by parsing their `type` attribute, which directly specifies the expected content type, e.g. color picker, password or plain text. Since many websites use text fields for all purposes and conduct the actual content validation via JavaScript, we also parse the `name` attribute of input tags or if missing their closest HTML label. Names like city, age, or username give information to deduce the required input type. Naturally, we want to insert as many URLs pointing to our monitoring server as possible (line 10) to identify as many SSBs as possible. However, if length restrictions of input fields prohibit this or when the input type is not a string, we instead try to satisfy potential input validation as much as possible by inserting fitting inputs for the deduced type, e.g. by inserting a random age, date, or a city depending on the type (line 12). After submitting a new form, a tuple of the receiver URL and the form's hash representation is added to the submission cache, to prevent identical forms from being submitted again (line 14 and 17).

B TYPES OF TRIGGERS

Here, we present additional information about the three types of triggers for SSRs, i.e., HTTP headers, URL parameters, and HTML forms. We find that simply supplying our URL in the `Referer` HTTP request header resulted in the most incoming requests as Table 8 shows. Of these, about 4.4% were conducted from a browser with JavaScript enabled. The table also shows that each of the three types of triggers has its specific advantage over the others: The `Referer` header triggered both most requests in total and the highest number of different domains. Modifying URL parameters, on the other hand, resulted in the highest amount of requests with JavaScript enabled. Submitting forms led to comparatively few requests, yet triggered a relatively high amount of requests with JavaScript enabled and a higher percentage of requests from different domains.

Table 8: The different triggers and how each affected the number of requests with and without JavaScript, as well as the number of domains.

Trigger	# Requests	# Requests with JS	# Domains
All	168055	7503 (4.5%)	4850 (100.0%)
Header	114833	2373 (2.1%)	3799 (78.3%)
Param	46135	2627 (5.7%)	353 (7.3%)
Form	7087	2503 (35.3%)	794 (16.4%)

Looking at the influence of time on the incoming requests, Figure 7 shows that changing URL parameters leads to the fastest responses. Both forms and parameters are quite effective to provoke immediate—and thus most likely automated—responses within the first few minutes. The `Referer` header, on the other hand, does trigger more requests over a longer time frame. However, these late requests are then more likely to contain requests from humans, e.g., people who look at their analytics dashboard and investigate where their visitors come from.

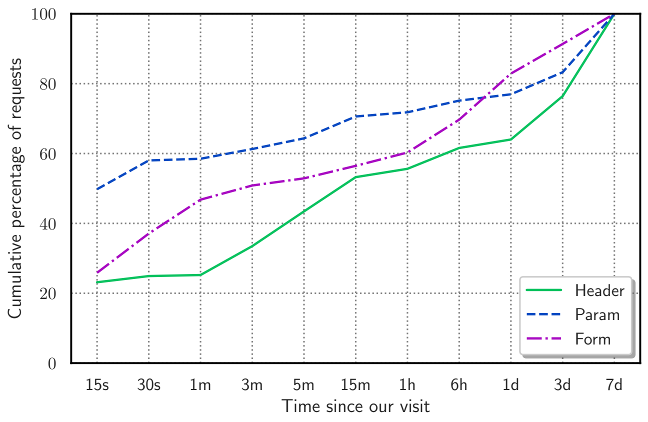


Figure 7: Incoming requests by type of trigger. Note the non-linear time axis.