Web-based Attacks to Discover and Control Local IoT Devices

Gunes Acar*, Danny Yuxing Huang*, Frank Li†, Arvind Narayanan*, Nick Feamster*

*Princeton University, †UC Berkeley

ABSTRACT

In this paper, we present two web-based attacks against local IoT devices that any malicious web page or third-party script can perform, even when the devices are behind NATs. In our attack scenario, a victim visits the attacker's website, which contains a malicious script that communicates with IoT devices on the local network that have open HTTP servers. We show how the malicious script can circumvent the same-origin policy by exploiting error messages on an HTML5 interface or by carrying out DNS rebinding attacks. We demonstrate that the attacker can gather sensitive information from the devices (e.g., unique device identifiers and precise geolocation), track and profile the owners to serve ads, or control the devices by playing arbitrary videos and rebooting. We propose potential countermeasures to our attacks that users, browsers, DNS providers, and IoT vendors can implement.

CCS CONCEPTS

• **Human-centered computing** → **Ubiquitous and mobile computing**; • **Security and privacy** → *Web protocol security*; Browser security;

KEYWORDS

Internet of Things, DNS rebinding, JavaScript, privacy

ACM Reference Format:

Gunes Acar*, Danny Yuxing Huang*, Frank Li†, Arvind Narayanan*, Nick Feamster*. 2018. Web-based Attacks to Discover and Control Local IoT Devices. In *IoT S&P'18: ACM SIGCOMM 2018 Workshop on IoT Security and Privacy , August 20, 2018, Budapest, Hungary*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3229565.3229568

1 INTRODUCTION

The rapid proliferation of IoT devices, from smart toys to smart appliances, enables a more connected and automated home. However, we have recently witnessed a number of IoT privacy and security fiascoes that have led to high-profile investigations, charges, and settlements [1–4]. Beyond the security and privacy concerns of end users, including children [5, 6], vulnerable IoT devices have been leveraged against the Internet ecosystem at large. For example, the 2016 Mirai botnet compromised IoT devices to launch one of the most devastating DDoS attacks the Internet had ever witnessed [7].

Attacks like that of the Mirai botnet are possible because many IoT devices are publicly exposed on the Internet (e.g., due to port forwarding). However, devices that are not Internet accessible (e.g., those behind NATs) are not safe either. In this paper, we present two web-based attacks against IoT devices with HTTP servers on the local area network (LAN). In our attack scenario, a victim on the LAN visits a web page hosting malicious JavaScript (either directly or loaded through a third-party). This script communicates with the HTTP endpoints of the IoT devices on the LAN by carrying out one or both of the following attacks[1]:

Attack ①: Discovering local devices. We show that an attack script can identify the presence of certain IoT devices on the LAN by exploiting an HTML5 element[2] interface error messages. In Chrome and Firefox, the resulting error message reveals if the resource exists. With this technique, attackers can detect the presence of popular devices, potentially profiling and tracking their owners.

Attack ②: Accessing local devices. We demonstrate that an attack script can fully access the HTTP endpoints of certain IoT devices on the LAN via an age-old attack that circumvents the same-origin policy, DNS rebinding [8]. We are the first to apply this attack to the IoT realm, showing that attackers can obtain sensitive information (e.g., MAC addresses and geolocations) from devices and send commands to control the devices (e.g., rebooting or playing arbitrary videos).

Any web publisher, advertiser, or third-party script embedded on a visited page can deploy either or both of these attacks at scale. We note these two attacks are independent of each other. While Attack ① aids Attack ② by identifying devices of interest, Attack ② is still feasible, albeit at a slower rate, without Attack ①. Neither attacks require a powerful adversary such as an ISP with man-in-the-middle capabilities; nor do they involve compromising routers or the lateral movement of malware within the network.

We observe that it may be difficult to curb these two attacks, as the browser features that we exploit have legitimate use cases [9, 10]. Still, we propose mitigations that users, browsers, DNS providers, and IoT vendors can implement.

In summary, this paper provides three primary contributions:

- ❖ We show that a web-based attacker can identify certain IoT devices on the LAN by exploiting timing side channels and HTML5 error messages, thereby sidestepping the same-origin policy (SOP).
- ❖ We demonstrate how a web-based attacker can access and control these devices. Additionally, we show what information attackers can extract, via DNS rebinding attacks, which also circumvent the SOP.
- ❖ We evaluate both attacks across seven IoT devices, four browsers, and three operating systems. We identify the limitations of these attacks and propose attack countermeasures.

IoT S&P'18, August 20, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5905-4/18/08…$15.00
https://doi.org/10.1145/3229565.3229568

[1] We demo the attacks at: https://iot-inspector.princeton.edu/iot-sigcomm-18/

[2] We redact the attack details to respect the disclosure process. We will release the redacted information and device models and manufacturers in the next months.

2 RELATED WORK

Network mapping and device discovery: As early as 2006, researchers presented web-based attacks that target local network devices. Lam et al. showed how an attacker can use malicious web pages to propagate worms and scan internal devices for vulnerabilities [11]. Grossman presented a web-based attack to scan local networks using *onerror* handlers of image resources [12]. Stamm et al. used this attack to hijack the DNS servers of home routers [13]. This very attack was recently observed in the wild, used by the DNSChanger exploit kit [14]. Several variants of the web-based network or port scanners have been developed since then [15–21]. Compared to this prior work, our device discovery attack: (1) works on HTTPS websites (i.e., circumvents mixed content protections); (2) can be used to detect HTTP endpoints that do not contain images (e.g., REST API endpoints); (3) works in parallel.

Gallagher proposed an improvement to web-based private network mapping attacks by using WebSockets and WebWorkers to circumvent network throttling by the browsers [22, 23]. Following this work, we too use WebWorkers to avoid throttling.

Lee et al. [24] improved web-based LAN scanning attacks by using AppCache mechanisms to identify the status of cross-origin requests. Lee et al.'s attack takes about 76 seconds on OS X and 21 seconds on Windows, as it requires waiting for the HTTP connection to terminate. Compared to theirs, our attack is much faster, uses a novel way to detect cross-origin request statuses, and does not depend on a deprecated technology.

DNS rebinding attacks: DNS rebinding attacks were discovered more than two decades ago [8]. Researchers explored DNS rebinding to circumvent firewalls, exfiltrate sensitive material from corporate intranets, and steal cryptocurrencies [25, 26]. Several freely available libraries and services implement DNS rebinding [27, 28].

Browser vendors [29], firewalls [30], and DNS providers [31] responded to DNS rebinding by attempting to implement defenses that include DNS pinning and blocking of private IPs in DNS responses. Jackson et al. presented one of the notable studies on defenses, conducting an extensive analysis of DNS rebinding and its mitigations [25]. The authors also presented a software called `dnswall` [32], which blocks DNS responses with private addresses [25]. However, research has shown that those defenses are not commonly deployed and a determined attacker can circumvent them. For instance, FireDrill by Dai and Resig floods the browser's DNS cache to circumvent DNS pinning defenses [33]. Recently, Young presented Jaqen, which combines the state-of-the-art DNS rebinding attacks into a JavaScript library [34].

In this work, we explore the feasibility of launching DNS rebinding attacks on today's browsers and operating system, and the damage these attacks can inflict against IoT devices.

3 PREPARING THE ATTACKS

The goal of the adversary in our model is to use JavaScript to discover, access, and control local IoT devices running HTTP servers without authentication mechanisms. This requires the adversary to issue `HTTP` requests (either `GET` or `POST`) to specific endpoints on the web servers. Therefore, the attacker must acquire prior knowledge of these endpoints before launching any attacks. The adversary may take the following two steps to identify these endpoints.

IoT Device	*Attack*
Camera 1	①
Camera 2	① ②
Google Home	① ②
Google Chromecast	① ②
Camera 3	① ②
Smart TV	① ②
Smart Switch	① ②

Table 1: IoT devices with open HTTP servers, and to which attacks (① and/or ②) they are vulnerable.

Step 1. Capturing packets from devices: An attacker can first interact with devices of interest in a test environment, capture the packets as a result of the interactions, and analyze the HTTP endpoints. To simulate this step, we connected 15 IoT devices to a Raspberry Pi wireless access point that also captured packets.[3] In addition, we connected an Android phone to the same wireless network, downloaded apps that corresponded to the IoT devices, and used the apps to interact with the devices. We then analyzed the captured packets offline to identify all local HTTP requests, i.e., requests between the phone and a device, and requests between the devices. Of our 15 devices, we identified seven that had local HTTP servers, listed in Table 1 (we did not detect HTTP endpoints on the Amazon Echo, Geeni and LIFX light bulbs, Halo smoke alarm, Hello Barbie toy, Samsung SmartThings Hub, and TP-Link and Orvibo smart plugs). The servers listened on a range of ports, all accepting unencrypted requests. We may have missed certain HTTP endpoints, for instance, because our interactions failed to cover all features of the device.

Step 2. Searching for other endpoints: To expand our coverage, we sampled two of the discovered HTTP endpoints for each of the seven devices, queried the paths on Google, and looked for documentation of other endpoints that we had not yet discovered. For example, we identified new endpoints in blog posts by other security researchers who were analyzing similar devices [36]. We verified these new endpoints by sending the same `HTTP` requests to our own devices and ensuring no errors were flagged in the responses.

Summary: In total, we collected 35 endpoints that accepted `GET` requests and eight endpoints that accepted `POST` requests across the seven devices. For the rest of the paper, we assume that an adversary has knowledge of these endpoints before discovering or accessing local IoT devices from the web.

4 ATTACK ①: DISCOVER LOCAL DEVICES

Using the 35 `HTTP` endpoints from Section 3, an adversary can launch Attack ①. At a high level, the adversary hosts a malicious JavaScript on a TLS-enabled web server and embeds the script on a website or an ad. Using TLS here allows the JavaScript to be loaded on both HTTP and HTTPS sites without triggering mixed-content errors. While using her home network, a victim visits the page with the malicious JavaScript, which automatically executes and discovers certain IoT devices on the victim's LAN. Discovering a victim's devices may potentially allow an adversary to launch

[3] CableLabs [35], a non-profit organization with ties to major cable network operators, selected and donated these devices; their interest in these devices presumably reflects the relative popularity among consumers.

further, device-specific attacks (e.g., Attack ②), or to infer the victim's lifestyle if she owns certain specialty devices.

4.1 How the attack works

The malicious JavaScript takes the following steps to identify local IoT devices:

(1) The script obtains the victim's local IP address via the WebRTC Session Description Protocol [37].
(2) The script sends a `GET` request via the `Fetch` API [38] to port 81 of every IP address in the local /24 subnet (e.g., https://192.168.1.123:81/), while measuring the timing of each response. As port 81 is rarely used, active devices are likely to immediately respond with a TCP RST packet, while the TCP connection times out for non-active IP addresses [17]. Using this timing side channel, the script infers which IP addresses correspond to active devices.
(3) To every active IP address, the script sends a request using the HTML5 element for the 35 device-specific endpoints that accept `GET` requests. Based on the resulting error messages, the script infers if the responding IP address is associated with one of the seven known devices (Section 3), and if so, which device.

Circumventing throttling with WebWorkers: For Step 2, if the attacker's script sends the `GET` requests (via `Fetch`) sequentially, scanning the entire /24 subnet could take several minutes. Instead, the attacker can take advantage of the asynchronous nature of `Fetch` and issue the `GET` requests in parallel. However, the browser may throttle these requests, such that requests to active devices may be queued and delayed. These delays may pollute the timing measurements as the attacker may incorrectly attribute the delays to inactivity.

To mitigate this problem, the attacker's script can use multiple WebWorkers — effectively separate threads — for the `Fetch` requests. As the browser throttles requests on a per-thread basis, this technique allows the script to send more requests in parallel [22]. As a further optimization, the script can cancel a `Fetch` request whose execution time has exceeded a certain threshold (we discuss choosing the threshold in Section 4.2), thus minimizing the waiting time for requests with long timeouts.

Circumventing SOP: We plan to publish the details of this attack once the disclosure period ends.

Reducing false positives with random paths: Some devices, such as the Smart Switch, respond to any HTTP `GET` requests with the `200 OK` code. If the attacker requests a non-existent endpoint on the Smart Switch, the HTML element will generate an error message that suggests that the endpoint exists, thus resulting in a false positive. To mitigate this problem, for every active IP address, the attack can request a randomly generated path between Steps 2 and 3. Thereafter, the attacker can exclude a device for which the error message indicates the existence of the random path.

4.2 Evaluating the severity of the attack

Presumably, one of the adversary's objectives is to maximize the number of devices identified, while minimizing the duration of the attack (as the victim may navigate away from the attack page, thus terminating the attacker's script). As such, we evaluated the attack with the following two metrics.

OS	*Chrome*	*Firefox*
Ubuntu 16.04	6.0	7.0
macOS 10.13	4.0	7.0
Windows 10	5.4	6.0

Table 2: To evaluate the first attack, we show the number of devices (out of 7 devices) discovered using HTML5 error messages, averaged over 5 attack attempts per OS/browser pair. The attack does not work on Safari or Edge.

Number of devices identified: To determine how many of the seven devices (Table 1) the attacker could identify, we loaded the attack script on different operating systems (i.e., Ubuntu 16.04, macOS 10.13, and Windows 10) and browsers (i.e., Chrome 65.0.3325, Firefox 59.0.1, Safari 11.0.3, and Microsoft Edge 41.16299.15). For each OS/browser pair, we executed the script in the browser sequentially for five iterations. For each page load, we counted the number of devices correctly identified. Between each page load, we disabled and re-enabled the wireless interface; otherwise, on macOS any subsequent `Fetch` requests to non-active IP addresses may time out quickly and become conflated with active IP addresses. Across the five page loads, we computed the average number of devices correctly identified, as shown in Table 2.

The attack script correctly identified all seven devices if a victim loaded it in Firefox on Ubuntu and macOS. In Firefox on Windows 10, the script consistently failed to identify the Camera 3 in each of the five page loads, as the port 81 `Fetch` requests timed out and thus the camera's IP address was identified as inactive. Manually sending an HTTP `GET` request via cURL showed that, in fact, the device was active.

In Chrome, the attack script failed to identify the Camera 2 in all 15 page loads across the three operating systems. We observed that the camera responded with `200 OK` to `GET` requests from the HTML5 element to random paths[4]; thus, the attack script considered the device as a false positive and removed it. On Ubuntu, the script identified the remaining six devices. However, it could not reliably identify the Camera 3 and the Smart Switch on macOS and Windows, as the `Fetch` requests timed out on these two devices.

On Safari, all the `Fetch` requests timed out, so the attack script considered all IP addresses as inactive. In contrast, the script could use `Fetch` to correctly identify the active IP addresses on Edge, but the Edge browser did not expose detailed HTML5 error messages. Thus, the attack script was unable to identify any devices on Edge.

Based on these findings, Firefox and Chrome were the only two browsers that the current version of the attack worked on. However, an attacker could potentially fine-tune the attack parameters for other browsers and OSes and fall back to alternative attacks (`<img>` and `onerror` based) where possible.

Duration of attack: For each `Fetch` request that the attack script sent, we measured the time it took to fail with an error (as `Fetch` was requesting a non-existent resource). We found that requests to IP addresses with active devices took 141 ms on average to fail with an error (i.e., due to TCP RSTs). In comparison, requests to the inactive IP addresses took approximately 3s, 21s and 76s to fail (i.e., due to timeouts) on Ubuntu, Windows and macOS respectively. These timeout values are consistent with Lee et al.'s earlier study [24].

[4]The Camera 2 did not respond with `200 OK` to Firefox; we speculate that this variation is due to minor differences in HTTP headers sent by Chrome and Firefox.

Such variations in timeouts are likely due to how different operating systems handle TCP timeouts.

An attacker with knowledge of these timings can optimize `Fetch` requests by cancelling those that take more than 141 ms (e.g., setting the threshold at 2 seconds).

4.3 Limitations of the attack

One key limitation to this attack is that the adversary can only target IoT devices with local HTTP servers. Eight of the 15 devices we studied did not satisfy this criteria, and thus were resistant to attack. Furthermore, the attacker needs to have prior knowledge of device-specific HTTP endpoints – for instance, by first buying these devices and interacting with them (similar to Section 3).

This attack has two sources of errors. First, even with this prior knowledge, an attacker may not always correctly identify a certain device. Devices produced by the same manufacturer but branded differently may not be distinguishable. For example, the Google Home and the Chromecast have a matching HTTP endpoint. Even if the HTML5 error message suggests that this endpoint exists, an attacker can only infer that the device is likely a Google product, not if it is a Google Home or Chromecast (although other endpoints can help with distinguishing). Another source of error is false negatives. For example, we use WebWorkers in Step 2 of the attack to mitigate browser throttling, but it cannot completely eliminate throttling effects. As such, `Fetch` requests to IP addresses with active devices may time out, causing the attacker to fail to identify those devices.

4.4 Countermeasures for the attack

What home users can do: In Step 1 of the attack, the malicious script obtains the local IP address via WebRTC SDP. A home user can prevent this by toggling a preference in their browser (e.g., `media.peerconnection.enabled` on Firefox), although an adversary can try to iterate over the *.1 addresses of the private IP ranges and discover an active device (which is typically the local router).

In Step 2 of the attack, the malicious script assumes that the IoT devices are on the same /24 subnet as the victim's computer. A home user can take advantage of this assumption and reconfigure the DHCP on their home routers to give out IP addresses in the /16 subnet, for instance.

What browsers can do: Privacy-focused browsers (or browser extensions) can limit the access to private IP ranges from web pages with public domain names. This restriction can prevent scanning of the LAN from the web, while still allowing access to web pages served from the LAN (e.g., home router interfaces).

What IoT vendors can do: IoT vendors could configure devices with HTTP servers to respond to any HTTP `GET` request with the `200 OK` code. This technique would make fingerprinting the device by its HTTP endpoints infeasible, although potentially at the cost of more difficult debugging and development.

5 ATTACK ②: ACCESS LOCAL DEVICES

While Attack ① can identify the presence or absence of specific HTTP endpoints, it cannot read any returned data due to the SOP. However, an attacker can circumvent the SOP with DNS rebinding [8], as we will discuss in this section. We show that this attack allows an adversary to extract private information (e.g., serial numbers, user names, or geolocations) from local devices or even control these devices (e.g., rebooting or playing arbitrary videos).

5.1 How the attack works

Traditionally, attackers have used DNS rebinding to target local routers, printers, firewalls, and corporate intranet servers [25]. IoT exacerbates the impact of this attack, as it significantly increases the number of potentially targeted devices, many of which connect to physical processes or manage sensitive user data.

At a high level, an attacker executes DNS rebinding by operating a web server at a domain (call it `domain.tld`) with a remote IP address X and serving a malicious JavaScript from that domain. The attacker also controls the domain's authoritative name server, such that resolving `domain.tld` initially will return the IP address X, while subsequent resolutions will return a local IP address — i.e., "rebinding" the domain to the new local IP address. The rebinding to a local destination allows the attack script (loaded from the attack domain) to access local HTTP endpoints without triggering browser cross-origin errors. Specifically, the attack proceeds as follows:

(1) A victim visits http://domain.tld/ for the first time. During the DNS lookup for the domain, the authoritative name server resolves the domain to X with a short TTL (e.g., one second). The victim loads and executes the malicious JavaScript hosted at the domain.
(2) The JavaScript requests another existing resource at the attacker-controlled domain, e.g., http://domain.tld/evil-test.
(3) The local DNS caches at the victim may or may not have evicted `domain.tld`. If not, `domain.tld` still resolves to X, and the request for http://domain.tld/evil-test would still return `200 OK`. In this case, the JavaScript waits a few seconds before re-attempting Step (2).
(4) If `domain.tld` has expired in the local DNS caches, the victim will query the attacker's authoritative name server and resolve `domain.tld` again.
(5) This time, the name server returns a local IP address, Y. (From Attack ①, the attacker could have learned that Y is associated with a known IoT device for this victim.[5]) As the domain has been rebound to a new IP address, any request for http://domain.tld/evil-test now returns a `404` error. At this point, DNS rebinding is complete. The attack script can now send HTTP requests directly to HTTP endpoints on the device at IP address Y, and read responses, allowing the attacker to extract information from or send commands to the device.

To perform DNS rebinding, an attacker can use Jaqen [34], an open-source DNS rebinding attack server that automatically manages and optimizes the name server and web server responses to perform DNS rebinding attacks.

5.2 Which devices were vulnerable

We implemented and executed an attack script that used DNS rebinding to access the devices from Table 1.[6] On Ubuntu and Windows, this attack completed within 2 to 4 seconds across all

[5] Without Attack ①, an attacker would have to try all the 35 known `GET` requests until one of them returns `200 OK`. While Attack ① is not absolutely needed, it does speed up Attack ②.
[6] Omitted from this evaluation was the Camera 1, which had only one known `HTTP` endpoint that returned an icon image. We chose not to attack this device because the endpoint did not return any unique identifiers or sensitive information.

Capabilities	*A*	*B*	*C*	*D*	*E*	*F*
Get Software Version or Model	✓	✓	✓	✓	✓	✓
Get Current SSID	✓		✓	✓	✓	✓
Get Nearby SSIDs	✓		✓	✓		✓
Get Device Unique Identifier	✓	✓	✓	✓	✓	✓
Get Owner's Username				✓		
Change State	✓		✓		✓	✓

Table 3: What Attack ② could do to IoT devices: [A]: Google Chromecast, [B]: Camera 2, [C]: Google Home, [D]: Camera 3, [E]: Smart TV, and [F]: Smart Switch.

browsers, while the completion time was between 9 and 13 seconds across the different browsers on macOS, possibly due to longer DNS caching on macOS. In the following, we enumerate the salient data that the attack script was able to extract from the devices via GET requests, along with the control commands that it could issue using POST requests. We summarize our findings in Table 3.

Google Home: An attacker could extract device configurations and other information using GET requests, including: software or firmware build versions, which attackers could use to identify outdated devices with known vulnerabilities; the SSID and BSSID of the wireless network the device was using, which could also be used for fingerprinting and geolocation; the night mode settings, which could reveal user sleep schedules; alarms and timers set on the device, which could also leak user schedules; and the device's unique identifier.

Using HTTP POST requests, the attacker could initiate a wireless network scan by the Google Home and retrieve the SSID, BSSID, and signal strength for each detected network. This scan could provide a unique fingerprint for the local network, and potentially could reveal the precise physical location of the device [39]. The attacker could also trigger a reboot of the Google Home with an HTTP POST request. Our device took over 30 seconds to reboot, so this capability could be used to launch a denial-of-service attack on Google Homes or damage the reputation of Google products.

Google Chromecast: Attackers could exploit the Google Chromecast similarly to the Google Home. The same information could be extracted from using HTTP GET requests except for night mode parameters and alarms/timers, which were not supported by the Chromecast.

Like the Google Home, miscreants could use HTTP POST requests to extract local WiFi network information from the Chromecast, and reboot the device. In addition, attackers could send a POST request to trigger the Chromecast to play any YouTube video, simply specifying the YouTube video ID. This capability could be leveraged to conduct view fraud, gaining a large number of views for attacker videos and generating ad revenue.

Smart TV: The HTTP GET interfaces exposed information about the device, such as: device model details, useful for identifying outdated or vulnerable devices; a unique device ID, making the TV fingerprintable; and the BSSID of the wireless network the device was using, also helpful for fingerprinting and geolocation. Furthermore, the device also exposed UPnP endpoints that accepted HTTP POST requests. We found that an attacker could trigger the TV to play any audio or video file at a given URL.

Smart Switch: The Smart Switch provided an HTTP endpoint that revealed the device model, its MAC address, a unique device identifier, the firmware version, and its current on/off state. This information could be useful for fingerprinting the device, tracking switch usage, and identifying outdated vulnerable devices.

Like the Smart TV, the Smart Switch also supported UPnP via HTTP POST requests that allowed for further data extraction and full control of the device. An attacker could extract the schedules set for the device and identify all local wireless network SSIDs, data useful for user tracking or profiling. With UPnP commands, attackers could also turn on and off the switch, change its schedule, change the WiFi settings, and trigger a firmware update.

OS	*Request*	*Chrome*	*Firefox*	*Safari*
Ubuntu	GET	A B C D E F	A B C D E F	N/A
	POST	A C E F	A C E F	N/A
macOS	GET	A B C D E F	*A B C D E* F	A B C D E F
	POST	A C E F	A C E F	A C E F
Windows	GET	A B C D E F	A B C D E F	N/A
	POST	A C E F	A C E F	N/A

Table 4: Which operating systems and browsers were vulnerable to Attack ② against the following devices: [A]: Google Chromecast, [B]: Camera 2, [C]: Google Home, [D]: Camera 3, [E]: Smart TV, and [F]: Smart Switch. An unformatted letter indicates that the attack was successful on all known HTTP endpoints on a given device; an underline indicates unsuccessful attacks on all of the HTTP endpoints; and italics indicates that some of the endpoints were vulnerable to our attack. We omit Microsoft Edge as DNS rebinding always failed on it.

Camera 3: We identified 5 HTTP endpoints that expected GET requests, but none for HTTP POST requests. These GET requests returned the camera model, the serial number (for which Camera 3's vendor provided an API that returned the username of the device), the current network's SSID, and the SSIDs of networks nearby.

Camera 2: The Camera 2 provided 6 HTTP endpoints for GET requests, but none for POST requests. One endpoint returned the device model, the build version, the MAC address, and the local router's IP address — all useful information for fingerprinting devices and identifying vulnerable models.

The other endpoints for GET requests required authentication. If an attacker could guess the password (as the username remained "admin"), they could directly access the live stream video and sound from the camera. However, the camera setup process prompted the user to change the device password, and browsers displayed login prompts, which would alert the user to suspicious behavior.

5.3 Which OSes and browsers were vulnerable

The feasibility of the attack differed depending on the operating system (Windows 10, Ubuntu 16.04, macOS 10.13) and the browser (Chrome 65.0.3325, Firefox 59.0.1, Safari 11.0.3, and Microsoft Edge 41.16299.15). We observed that an attacker could successfully issue GET requests on all six devices in Chrome across all three OSes, while in Firefox an attacker could potentially achieve the highest success rate for POST requests. We summarize the findings in Table 4.

Chrome: On all OSes, DNS rebinding succeeded for all targeted HTTP GET and POST endpoints. All HTTP GET requests succeeded in retrieving device data. Of our four devices with HTTP POST

interfaces, an attacker could control the Smart Switch and the Smart TV. The Google Home and Chromecast disallowed our attack HTTP `POST` requests as it appeared these devices denied requests that contained "Mozilla" in the user-agent header, and Chrome did not permit JavaScript to modify the user-agent header value. We suspect this user-agent filtering was a measure against web browsers accessing the HTTP interface.[7] In Section 5.4, we propose that the `Origin` header and other forbidden header names [41] are more suitable for filtering browser-originated requests.

Firefox: Firefox allowed scripts to overwrite user-agent headers. Thus we could bypass user-agent based filtering by the Google Home and Chromecast to send `POST` requests.

On Windows 10, DNS rebinding and all subsequent HTTP `GET` and `POST` requests were successfully executed. Thus, an attacker could conduct the full range of attacks discussed in Section 5.2.

On Ubuntu and macOS, the DNS rebinding failed for some HTTP endpoints. In these cases, we observed that the attack domain was successfully rebound to a local IP address, but only for a single network connection. We have not determined the cause of this odd behavior, and we speculate it is due to a limitation in Jaqen.

On Ubuntu, this DNS rebinding issue prevented attackers from accessing any HTTP endpoints on the Smart TV, the Camera 3, the Camera 2, and the Smart Switch. For all HTTP endpoints on other devices, adversaries could successfully make HTTP `GET` and `POST` requests to conduct the attacks.

On macOS, the rebinding issue for Firefox was less extensive, affecting 1 of 9 Smart TV endpoints, 2 of 5 Camera 2 endpoints, and the single Smart Switch endpoint. The HTTP `GET` requests were successful on all other endpoints, and all of the HTTP `POST` requests were correctly executed (except on the Smart Switch). The HTTP `POST` endpoints on the Smart TV were not affected by the DNS rebinding failure.

Safari: For Safari on macOS, all HTTP `GET` attacks worked correctly, but attackers could not trigger actions on the Google Home and Chromecast, as Safari also disallowed user-agent modifications. Like on Chrome, an attacker could control the Smart Switch and the Smart TV through HTTP `POST` requests.

Microsoft Edge: For Edge on Windows 10, DNS rebinding failed in all cases, so an adversary could not execute any of the attacks. It appeared that Edge had built-in DNS rebinding protection, although we did not find confirmed documentation of this defense.

5.4 Countermeasures for the attack

What home users can do: Users can install `dnsmasq`, a local DNS forwarder that protects against DNS rebinding by dropping RFC 1918 addresses from DNS replies, similar to `dnswall` [42]. Alternatively, users can use OpenWRT routers, which use `dnsmasq` under the hood [43] to drop private IPs in DNS replies.

What browser vendors can do: Prior attempts to mitigate DNS rebinding in browsers not only broke some web services [29], but led to new security vulnerabilities [44]. Some browser vendors seem to have adopted the view that it is infeasible to completely mitigate this attack in the browser [45, 46]. We believe a browser-based defense remains as an open research problem.

[7] All modern browsers include "Mozilla" in their user-agent string for historical compatibility reasons [40].

What IoT manufacturers can do: Vendors can validate the `Host` headers of incoming requests (which none of our six devices did) and only allow requests that contain the device's IP address or mDNS name in the `Host` header. Moreover, we propose that *forbidden header names* such as the `Origin` header, which cannot be overwritten by web scripts, can be used to filter out requests that originate from arbitrary web pages and browsers [41, 47].

What DNS providers can do: DNS providers can use `dnswall` [25] or similar software to filter out private IPs from DNS replies. We checked eight popular DNS providers and found that none of the following ISPs and DNS providers filtered out private IPs from their replies: ATT, Comcast, Verizon Fios, Google DNS, Cox, Time Warner, Orange (Spain), and VyprVPN.

6 RESPONSIBLE DISCLOSURE

We reported vulnerabilities discovered throughout our research to respective browser (Mozilla [48] and Chromium [49]) and IoT vendors (Google and three other vendors). The Chromium project awarded a bug bounty for our disclosure.

7 CONCLUSION

In this paper, we have shown that a web script can detect the presence of IoT devices that have local HTTP interfaces, and that it can access the devices using DNS rebinding. Malicious web pages or third-party ads can perform these two attacks, possibly without user awareness. As we did not identify well-known defenses from major browsers, DNS providers, and IoT vendors, the attacks are likely to present major security and privacy concerns to IoT users.

ACKNOWLEDGMENTS

This work was partially supported by NSF awards CNS-1526353, CNS-1539902, CNS-1535796, and CNS-1237265, the William and Flora Hewlett Foundation, a Google Faculty Research Award, and the Princeton University Center for Information Technology Policy Internet of Things Consortium.

REFERENCES

[1] Federal Trade Commission. FTC Charges D-Link Put Consumers' Privacy at Risk Due to the Inadequate Security of Its Computer Routers and Cameras. https://www.ftc.gov/news-events/press-releases/2017/01/ftc-charges-d-link-put-consumers-privacy-risk-due-inadequate, 2017.
[2] Federal Trade Commission. ASUS Settles FTC Charges That Insecure Home Routers and "Cloud" Services Put Consumers' Privacy At Risk. https://www.ftc.gov/news-events/press-releases/2016/02/asus-settles-ftc-charges-insecure-home-routers-cloud-services-put, 2016.
[3] Federal Trade Commission. Electronic Toy Maker VTech Settles FTC Allegations That it Violated Children's Privacy Law and the FTC Act. https://www.ftc.gov/news-events/press-releases/2018/01/electronic-toy-maker-vtech-settles-ftc-allegations-it-violated, 2018.
[4] VIZIO to Pay $2.2 Million to FTC, State of New Jersey to Settle Charges It Collected Viewing Histories on 11 Million Smart Televisions without Users' Consent. https://www.ftc.gov/news-events/press-releases/2017/02/vizio-pay-22-million-ftc-state-new-jersey-settle-charges-it, 2017.
[5] Emily McReynolds, Sarah Hubbard, Timothy Lau, Aditya Saraf, Maya Cakmak, and Franziska Roesner. Toys that Listen: A Study of Parents, Children, and Internet-Connected Toys. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2017.
[6] German Parents Told to Destroy Cayla Dolls over Hacking Fears. http://www.bbc.com/news/world-europe-39002142, 2017.
[7] Wikipedia. 2016 Dyn cyberattack. https://en.wikipedia.org/wiki/2016_Dyn_cyberattack, 2016.
[8] Drew Dean, Edward W Felten, and Dan S Wallach. Java Security: From HotJava to Netscape and Beyond. In *IEEE Symposium on Security and Privacy (S&P)*, 1996.
[9] Mozilla. High rate of MEDIA_ERR_SRC_NOT_SUPPORTED error being thrown. https://bugzilla.mozilla.org/show_bug.cgi?id=1417869#c3, 2017.

[10] Chromium. Better logging for cause of media playback failure. https://bugs.chromium.org/p/chromium/issues/detail?id=492845, 2015.
[11] VT Lam, Spyros Antonatos, Periklis Akritidis, and Kostas G Anagnostakis. Puppetnets: Misusing Web Browsers as a Distributed Attack Infrastructure. In *ACM Conference on Computer and Communications Security (CCS)*, 2006.
[12] Jeremiah Grossman and T Niedzialkowski. Hacking Intranet Websites from the Outside: JavaScript Malware Just Got a Lot More Dangerous. *Blackhat USA*, 2006.
[13] Sid Stamm, Zulfikar Ramzan, and Markus Jakobsson. Drive-By Pharming. In *International Conference on Information and Communications Security (ICICS)*, 2007.
[14] Proofpoint. Home Routers Under Attack via Malvertising on Windows, Android Devices. https://www.proofpoint.com/us/threat-insight/post/home-routers-under-attack-malvertising-windows-android-devices, 2015.
[15] beefproject. XSS Rays. https://github.com/beefproject/beef/wiki/Xss-Rays, 2012.
[16] JS-Recon. Port Scanning with HTML5 and JS-Recon . http://blog.andlabs.org/2010/12/port-scanning-with-html5-and-js-recon.html, 2010.
[17] Taylor Hornby. Port Scanning Local Network From a Web Browser. https://defuse.ca/in-browser-port-scanning.htm, 2015.
[18] Peppersoft. Local Network Scanner with JavaScript. http://peppersoft.net/local-network-scanner-javascript/, 2017.
[19] JavaScript LAN scanner. https://www.myria.de/lan-scan/index.php, 2007.
[20] Joe Vennix. lan-js: Probe LAN devices from a web browser. https://github.com/joevennix/lan-js, 2015.
[21] Matthew Bryant. sonar.js: A Framework for Identifying and Launching Exploits against Internal Network Hosts. https://github.com/mandatoryprogrammer/sonar.js, 2015.
[22] Tom Gallagher. Port Scanning and WebSockets. https://datatracker.ietf.org/meeting/96/materials/slides-96-saag-1/, 2016.
[23] Tom Gallagher. Security Enhancement for WebSockets to Prevent Private Network Mapping. https://tools.ietf.org/html/draft-gallagher-hybiwebsocketenhancement-00, 2016.
[24] Sangho Lee, Hyungsub Kim, and Jong Kim. Identifying Cross-origin Resource Status Using Application Cache. In *Network and Distributed System Security Symposium (NDSS)*, 2015.
[25] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting Browsers from DNS Rebinding Attacks. *ACM Transactions on the Web (TWEB)*, 3(1):2, 2009.
[26] Jazzy. How Your Ethereum Can Be Stolen Through DNS Rebinding. https://ret2got.wordpress.com/2018/01/19/how-your-ethereum-can-be-stolen-using-dns-rebinding, 2018.
[27] Rebind. https://tools.kali.org/sniffingspoofing/rebind, 2014.
[28] Taviso Ormandy. rbndr. https://github.com/taviso/rbndr, 2017.
[29] Bug 149943 - Use "DNS pinning" to Prevent Princeton-like Exploits. https://bugzilla.mozilla.org/show_bug.cgi?id=149943, 2002.
[30] pfSense. DNS Rebinding Protections. https://doc.pfsense.org/index.php/DNS_Rebinding_Protections, 2011.
[31] Cisco. Finally, a Real Solution to DNS Rebinding Attacks. https://umbrella.cisco.com/blog/2008/04/14/finally-a-real-solution-to-dns-rebinding-attacks/, 2008.
[32] Andrew Bortz. google-dnswall. https://github.com/abortz/google-dnswall/, 2015.
[33] Yunxing Dai and Ryan Resig. FireDrill: Interactive DNS Rebinding. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2013.
[34] Luke Young. Open Sourcing Jaqen, A Tool For Developing DNS Rebinding PoCs | LinkedIn Engineering. https://engineering.linkedin.com/blog/2017/07/open-sourcing-jaqen--a-tool-for-developing-dns-rebinding-pocs, 2017.
[35] CableLabs. https://www.cablelabs.com/, 2018.
[36] Google Home Local API. https://rithvikvibhu.github.io/GHLocalApi/, 2018.
[37] Suhas Nandakumar and Cullen Jennings. SDP for the WebRTC. https://tools.ietf.org/html/draft-nandakumar-rtcweb-sdp-00, 2012.
[38] Mozilla. Fetch API. https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API, 2018.
[39] WiGLE.net. WiGLE: Wireless Network Mapping. https://wigle.net/, 2018.
[40] Pawel Piejko. List of User Agent strings. https://deviceatlas.com/blog/list-of-user-agent-strings, 2018.
[41] Mozilla. Forbidden Header Name. https://developer.mozilla.org/en-US/docs/Glossary/Forbidden_header_name, 2017.
[42] Simon Kelley. Man page of dnsmasq. http://www.thekelleys.org.uk/dnsmasq/docs/dnsmasq-man.html, 2018.
[43] OpenWrt. DNS and DHCP Configuration. https://wiki.openwrt.org/doc/uci/dhcp#common_options, 2018.
[44] Bug 174590 - DNS "Pinning" behavior leads to security hole. https://bugzilla.mozilla.org/show_bug.cgi?id=174590, 2002.
[45] Bug 689835 - DNS rebinding attack using cached resources. https://bugzilla.mozilla.org/show_bug.cgi?id=689835#c9, 2011.
[46] Bug 274464 - Security: DNS cache can be flooded, which leads to DNS rebinding, circumvents same origin policy. https://bugs.chromium.org/p/chromium/issues/detail?id=274464#c10, 2013.
[47] Issue #37: 'user-agent' header control - whatwg/fetch. https://github.com/whatwg/fetch/issues/37, 2018.
[48] Bug 1450853 - MediaError Message Property Leaks Cross-Origin Response Status. https://bugzilla.mozilla.org/show_bug.cgi?id=1450853, 2018.
[49] Bug 828265 - MediaError Message Property Leaks Cross-Origin Response Status. https://bugs.chromium.org/p/chromium/issues/detail?id=828265, 2018.