

Leaky Images: Targeted Privacy Attacks in the Web

Cristian-Alexandru Staicu and Michael Pradel, *TU Darmstadt*

https://www.usenix.org/conference/usenixsecurity19/presentation/staicu

This paper is included in the Proceedings of the 28th USENIX Security Symposium.

August 14–16, 2019 • Santa Clara, CA, USA

978-1-939133-06-9

Open access to the Proceedings of the 28th USENIX Security Symposium is sponsored by USENIX.

Leaky Images: Targeted Privacy Attacks in the Web

Cristian-Alexandru Staicu
Department of Computer Science
TU Darmstadt

Michael Pradel
Department of Computer Science
TU Darmstadt

Abstract

Sharing files with specific users is a popular service provided by various widely used websites, e.g., Facebook, Twitter, Google, and Dropbox. A common way to ensure that a shared file can only be accessed by a specific user is to authenticate the user upon a request for the file. This paper shows a novel way of abusing shared image files for targeted privacy attacks. In our attack, called *leaky images*, an image shared with a particular user reveals whether the user is visiting a specific website. The basic idea is simple yet effective: an attacker-controlled website requests a privately shared image, which will succeed only for the targeted user whose browser is logged into the website through which the image was shared. In addition to targeted privacy attacks aimed at single users, we discuss variants of the attack that allow an attacker to track a group of users and to link user identities across different sites. Leaky images require neither JavaScript nor CSS, exposing even privacy-aware users, who disable scripts in their browser, to the leak. Studying the most popular websites shows that the privacy leak affects at least eight of the 30 most popular websites that allow sharing of images between users, including the three most popular of all sites. We disclosed the problem to the affected sites, and most of them have been fixing the privacy leak in reaction to our reports. In particular, the two most popular affected sites, Facebook and Twitter, have already fixed the leaky images problem. To avoid leaky images, we discuss potential mitigation techniques that address the problem at the level of the browser and of the image sharing website.

1 Introduction

Many popular websites allow users to privately share images with each other. For example, email services allow attachments to emails, most social networks support photo sharing, and instant messaging systems allow files to be sent as part of a conversation. We call websites that allow users to share images with each other *image sharing services*.

This paper presents a targeted privacy attack that abuses a vulnerability we find to be common in popular image sharing services. The basic idea is simple yet effective: An attacker can determine whether a specific person is visiting an attacker-controlled website by checking whether the browser can access an image shared with this person. We call this attack *leaky images*, because a shared image leaks the private information about the victim's identity, which otherwise would not be available to the attacker. To launch a leaky images attack, the attacker privately shares an image with the victim through an image sharing service where both the attacker and the victim are registered as users. Then, the attacker includes a request for the image into the website for which the attacker wants to determine whether the victim is visiting it. Since only the victim, but no other user, is allowed to successfully request the image, the attacker knows with 100% certainty whether the victim has visited the site.

Beyond the basic idea of leaky images, we describe three further attacks. First, we describe a targeted attack against groups of users, which addresses the scalability issues of the single-victim attack. Second, we show a pseudonym linking attack that exploits leaky images shared via different image sharing services to determine which user accounts across these services belong to the same individual. Third, we present a scriptless version of the attack, which uses only HTML, and hence, works even for users who disable JavaScript in their browsers.

Leaky images can be (ab)used for targeted attacks in various privacy-sensitive scenarios. For example, law enforcement could use the attack to gather evidence that a suspect is visiting particular websites. Similarly but perhaps less noble, a governmental agency might use the attack to deanonymize a political dissident. As an example of an attack against a group, consider deanonymizing reviewers of a conference. In this scenario, the attacker would gather the email addresses of all committee members and then share leaky images with each reviewer through some of the various websites providing that service. Next, the attacker would embed a link to an external website into a paper under review, e.g.,

Table 1: Leaky images vs. related web attacks. All techniques assume that the victim visits an attacker-controlled website.

Threat	Who can attack?	What does the attacker achieve?	Usage scenario
Tracking pixels	Widely used ad providers and web tracking services	Learn that user visiting site A is the same as user visiting site B	Large-scale creation of low-entropy user profiles
Social media fingerprinting	Arbitrary website provider	Learn into which sites the victim is logged in	Large-scale creation of low-entropy user profiles
Cross-site request forgery	Arbitrary website provider	Perform side effects on a target site into which the victim is logged in	Abuse the victim's authorization by acting on her behalf
Leaky images	Arbitrary website provider	Precisely identify the victim	Targeted, fine-grained deanonymization

a link to a website with additional material. If and when a reviewer visits that page, while being logged into one of the image sharing services, the leaky image will reveal to the attacker who is reviewing the paper. The prerequisite for all these attacks is that the victim has an account at a vulnerable image sharing service and that the attacker is allowed to share an image with the victim. We found at least three highly popular services (Google, Microsoft Live, and Dropbox) that allow sharing images with any registered user, making it straightforward to implement the above scenarios.

The leak is possible because images are exempted from the same-origin policy, and because image sharing services authenticate users through cookies. When the browser makes a third-party image request, it attaches the user's cookie of the image sharing website to it. If the decision of whether to authorize the image request is cookie-dependent, then the attacker can infer the user's identity by observing the success of the image request. Related work discusses the dangers of exempting JavaScript from the same-origin policy [24], but to the best of our knowledge, there is no work discussing the privacy implications of observing the result of cross-origin requests to privately shared images.

Leaky images differ from previously known threats by enabling arbitrary website providers to precisely identify a victim (Table 1). One related technique are tracking pixels, which enable tracking services to determine whether two visitors of different sites are the same user. Most third-party tracking is done by a few major players [13], allowing for regulating the way these trackers handle sensitive data. In contrast, our attack enables arbitrary attackers and small websites to perform targeted privacy attacks. Another related technique is social media fingerprinting, where the attacker learns whether a user is currently logged into a specific website.[1] In contrast, leaky images reveal not only whether a user is logged in, but precisely which user is logged in. Leaky images resemble cross-site request forgery (CSRF) [33], where a malicious website performs a request to a target site on behalf of the user. CSRF attacks typically cause side effects on the server, whereas our attack simply retrieves an image. We discuss in Section 5 under what conditions defenses proposed against CSRF, as well as other mitigation techniques, can reduce the risk of privacy leaks due to leaky images.

To understand how widespread the leaky images problem is, we study 30 out of the 250 most popular websites. We create multiple user accounts on these websites and check whether one user can share a leaky image with another user. The attack is possible if the shared image can be accessed through a link known to all users sharing the image, and if access to the image is granted only to certain users. We find that at least eight of the 30 studied sites are affected by the leaky images privacy leak, including some of the most popular sites, such as Facebook, Google, Twitter, and Dropbox. We carefully documented the steps for creating leaky images and reported them as privacy violations to the security teams of the vulnerable websites. In total, we informed eight websites about the problem, and so far, six of the reports have been confirmed, and for three of them we have been awarded bug bounties. Most of the affected websites are in the process of fixing the leaky images problem, and some of them, e.g., Facebook and Twitter, have already deployed a fix.

In summary, this paper makes the following contributions:

- We present leaky images, a novel targeted privacy attack that abuses image sharing services to determine whether a victim visits an attacker-controlled website.
- We discuss variants of the attack that aim at individual users, groups of users, that allow an attacker to link user identities across image sharing services, and that do not require any JavaScript.
- We show that eight popular websites, including Facebook, Twitter, Google, and Microsoft Live are affected by leaky images, exposing their users to be identified on third-party websites.
- We propose several ways to mitigate the problem and discuss their benefits and weaknesses.

2 Image Sharing in the Web

Many popular websites, including Dropbox, Google Drive, Twitter, and Facebook, enable users to upload images and to

[1] See https://robinlinus.github.io/socialmedia-leak/ or https://browserleaks.com/social.

share these images with a well-defined set of other users of the same site. Let i be an image, U be the set of users of an image sharing service, and let $u^i_{owner} \in U$ be the owner of i. By default, i is not accessible to any other users than u^i_{owner}. However, an owner of an image can share the image with a selected subset of other users $U^i_{shared} \subseteq U$, which we define to include the owner itself. As a result, all users $u \in U^i_{shared}$, but no other users of the service and no other web users, have read access to i, i.e., can download the image via a browser.

Secret URLs To control which users can access an image, there are several implementation strategies. One strategy is to create a *secret URL* for each shared image, and to provide this URL only to users allowed to download the image. In this scenario, there is a set of URLs L^i (L stands for "links") that point to a shared image i. Any user who knows a URL $l^i \in L^i$ can download i through it. To share an image i with multiple users, i.e., $|U^i_{shared}| > 1$, there are two variants of implementing secret URLs. On the one hand, each user u may obtain a personal secret URL l^i_u for the shared image, which is known only to u and not supposed to be shared with anyone. On the other hand, all users may share the same secret URL, i.e., $L^i = \{l^i_{shared}\}$. A variant of secret URLs are URLs that expire after a given amount of time or after a given number of uses. We call these URLs session URLs.

Authentication Another strategy to control who accesses an image is to authenticate users. In this scenario, the image sharing service checks for each request to i whether the request comes from a user in U^i_{shared}. Authentication may be used in combination with secret URLs. In this case, a user u may access an image i only if she knows a secret URL l^i and if she is authenticated as $u \in U^i_{shared}$. The most common way to implement authentication in image sharing services are cookies. Once a user logs into the website of an image sharing service, the website stores a cookie in the user's browser. When the browser requests an image, the cookie is sent along with the request to the image sharing service, enabling the server-side of the website to identify the user.

Image Sharing in Practice Different real-world image sharing services implement different strategies for controlling who may access which image. For example, Facebook mostly uses secret URLs, which initially created confusion among users due to the apparent lack of access control[2]. Gmail relies on a combination of secret URLs and authentication to access images attached to emails. Deciding how to implement image sharing is a tradeoff between several design goals, including security, usability, and performance. The main advantage of using secret URLs only is that third-party content delivery networks may deliver images, without any cross-domain access control checks. A drawback of secret URLs is that they should not be used over non-secret channels, such as HTTP, since these channels are unable to protect the secrecy of requested URLs. The main advantage of authentication is to not require links to be secret, enabling them to be sent over insecure channels. On the downside, authentication-based access control makes using third-party content delivery networks harder, because cookie-based authentication does not work across domains.

[2] `https://news.ycombinator.com/item?id=13204283`

Same-Origin Policy The same-origin policy regulates to what extent client-side scripts of a website can access the document object model (DOM) of the website. As a default policy, any script loaded from one origin is not allowed to access parts of the DOM loaded from another origin. Origin here means the URI scheme (e.g., *http*), the host name (e.g., *facebook.com*), and the port number (e.g., 80). For example, the default policy implies that a website *evil.com* that embeds an `iframe` from *facebook.com* cannot access those parts of the DOM that have been loaded from *facebook.com*. There are some exceptions to the default policy described above. One of them, which is crucial for the leaky images attack, are images loaded from third parties. In contrast to other DOM elements, a script loaded from one origin can access images loaded from another origin, including whether the image has been loaded at all. For the above example, *evil.com* is allowed to check whether an image requested from *facebook.com* has been successfully downloaded.

3 Privacy Attacks via Leaky Images

This section presents a series of attacks that can be mounted using leaky images. At first, we describe the conditions under which the attack is possible (Section 3.1). Then, we present a basic attack that targets individual users (Section 3.2), a variant of the attack that targets groups of users (Section 3.3), and an attack that links identities of an individual registered at different websites (Section 3.4). Next, we show that the attack relies neither on JavaScript nor CSS, but can be performed by a purely HTML-based website (Section 3.5). Finally, we discuss how leaky images compare to previous privacy-related issues, such as web tracking (Section 3.6).

3.1 Attack Surface

Our attack model is that an attacker wants to determine whether a specific victim is visiting an attacker-controlled website. This information is important from a privacy point of view and usually not available to operators of a website. An operator of a website may be able to obtain some information about clients visiting the website, e.g., the IP and the browser version of the client. However, this information is limited, e.g., due to multiple clients sharing

Table 2: Conditions that enable leaky image attacks.

Authentication (e.g., cookies)	URL of image: Publicly known	Secret URL shared among users	Per-user secret URL
Yes	(1) Leaky image	(2) Leaky image	(3) Secure
No	(4) Irrelevant	(5) Secure	(6) Secure

the same IP or the same browser version, and often insufficient to identify a particular user with high confidence. Moreover, privacy-aware clients may further obfuscate their traces, e.g., by using the Tor browser, which hides the IP and other details about the client. Popular tracking services, such as Google Analytics, also obtain partial knowledge about which users are visiting which websites. However, the use of this information is legally regulated, available to only a few tracking services, and shared with website operators only in anonymized form. In contrast, the attack considered here enables an arbitrary operator of a website to determine whether a specific person is visiting the website.

Leaky image attacks are possible whenever all of the following four conditions hold. First, we assume that the attacker and the victim are both users of the same image sharing service. Since many image sharing services provide popular services beyond image sharing, such as email or a social network, their user bases often cover a significant portion of all web users. For example, Facebook announced that it has more than 2 billion registered users[3], while Google reported to have more than 1 billion active Gmail users each month[4]. Moreover, an attacker targeting a specific victim can simply register at an image sharing service where the victim is registered. Second, we assume that the attacker can share an image with the victim. For many image sharing services, this step involves nothing more than knowing the email address or user name of the victim, as we discuss in more detail in Section 4. Third, we assume that the victim visits the attacker-controlled website while the victim's browser is logged into the image sharing service. Given the popularity of some image sharing services and the convenience of being logged in at all times, we believe that many users fulfill this condition for at least one image sharing service. In particular, in Google Chrome and the Android operating system, users are encouraged immediately after installation to login with their Google account and to remain logged in at all times.

The fourth and final condition for leaky images concerns the way an image sharing service determines whether a request for an image is from a user supposed to view that image. Table 2 shows a two-dimensional matrix of possible implementation strategies, based on the description of secret URLs and authentication-based access control in Section 2. In one dimension, a website can either rely on authentication or not. In the other dimension, the site can make an image available through a publicly known URL, a secret URL shared among the users allowed to access the image, or a per-user secret URL. Out of the six cases created by these two dimensions, five are relevant in practice. The sixth case, sharing an image via a publicly known URL without any authentication, would make the image available to all web users, and therefore is out of the scope of this work. The leaky image attack works in two of the five possible cases in Table 2, cases 1 and 2. Specifically, leaky images are enabled by sites that protect shared images through authentication and that either do not use secret URLs at all or that use a single secret URL per shared image. Section 4 shows that these cases occur in practice, and that they affect some of today's most popular websites.

[3] https://techcrunch.com/2017/06/27/facebook-2-billion-users/

[4] https://www.businessinsider.de/gmail-has-1-billion-monthly-active-users-2016-2

3.2 Targeting a Single User

After introducing the prerequisites for leaky images, we now describe several privacy attacks based on them. We start with a basic version of the attack, which targets a single victim and determines whether the victim is visiting an attacker-controlled website. To this end, the attacker uploads an image i to the image sharing service and therefore becomes the owner of the image, i.e., $u_{attacker} = u^i_{owner}$. Next, the attacker configures the image sharing service to share i with the victim user u_{victim}. As a result, the set of users allowed to access the image is $U^i_{shared} = \{u_{attacker}, u_{victim}\}$. Then, the attacker embeds a request for i into the website s for which the attacker wants to determine whether the victim is visiting the site. Because images are exempted from the same-origin policy (Section 2), the attacker-controlled parts of s can determine whether the image gets loaded successfully and report this information back to the attacker. Once the victim visits s, the image request will succeed and the attacker knows that the victim has visited s. If any other client visits s, though, the image request fails because s cannot authenticate the client as a user in U^i_{shared}. We assume that the attacker does not visit s, as this might mislead the attacker to believe that the victim is visiting s.

Because the authentication mechanism of the image sharing service ensures that only the attacker and the victim can access the image, a leaky image attack can determine with 100% accuracy whether the targeted victim has visited the site. At the same time, the victim may not notice that she was tracked, because the image can be loaded in the background.

For example, Figure 1 shows a simple piece of HTML code with embedded JavaScript. The code requests a leaky image, checks whether the image is successfully loaded, and sends this information back to the attacker-controlled web

```
<script>
window.onload = function() {
 var img = document.getElementById("myPic");
 img.src = "https://imgsharing.com/leakyImg.png";
 img.onload = function() {
   httpReq("evil.com", "is the target");
 }
 img.onerror = function() {
   httpReq("evil.com", "not the target");
 }
}
</script>
<img id="myPic">
```

Figure 1: Tracking code included in the attacker's website.

server via another HTTP request. We assume `httpReq` is a method that performs such a request using standard browser features such as `XMLHttpRequest` or `innerHTML` to send the value of the second argument to the domain passed as first argument. Alternatively to using `onload` to detect whether the image has been loaded, there are several variations, which, e.g., checking the width or height of the loaded image. As we show below (Section 3.5), the attack is also possible within a purely HTML-based website, i.e., without JavaScript.

The described attack works because the same-origin policy does not apply to images. That is, the attacker can include a leaky image through a cross-origin request into a website and observe whether the image is accessible or not. In contrast, requesting an HTML document does not cause a similar privacy leak, since browsers implement a strict separation of HTML coming from different origins. A second culprit for the attack's success is that today's browsers automatically include the victim's cookie in third-party image requests. As a result, the request passes the authentication of the image sharing service, leaking the fact that the request comes from the victim's browser.

3.3 Targeting a Group of Users

The following describes a variant of the leaky images attack that targets a group of users instead of a single user. In this scenario, the attacker considers a group of n victims and wants to determine which of these victims is visiting a particular website.

As an example, consider a medium-scale spear phishing campaign against the employees of a company. After preparing the actual phishing payload, e.g., personalized emails or cloned websites, the attacker may include a set of leaky images to better understand which victims interact with the payload and in which way. In this scenario, leaky images provide a user experience analysis tool for the attacker.

A naive approach would be to share one image i_k ($1 \leq k \leq n$) with each of the n victims. However, this naive approach does not scale well to larger sets of users: To track a group of 10,000 users, the attacker needs 10,000 shared images and 10,000 image requests per visit of the website. In other words, this naive attack has $\mathcal{O}(n)$ complexity, both in the number of leaky images and in the number of requests. For the above example, this naive way of performing the attack might raise suspicion due to the degraded performance of the phishing site and the increase in the number of network requests.

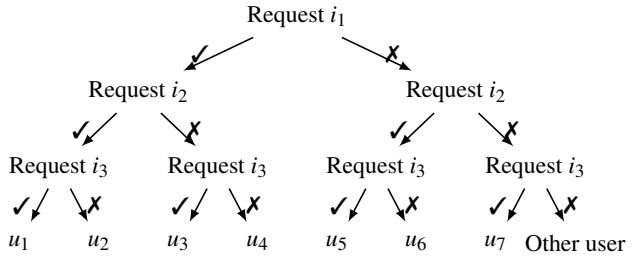

Figure 2: Binary search to identify individuals in a group of users u_1 to u_7 through requests to leaky images i_1 to i_3.

To efficiently attack a group of users, an attacker can use the fact that image sharing services allow sharing a single image with multiple users. The basic idea is to encode each victim with a bit vector and to associate each bit with one shared image. By requesting the images associated with each bit, the website can compute the bit vector of a user and determine if the user is among the victims, and if yes, which victim it is. This approach enables a binary search on the group of users, as illustrated in Figure 2 for a group of seven users. The website includes code that requests images i_1, i_2, and i_3, and then determines based on the availability of the images which user among the targeted victims has visited the website. If none of the images is available, then the user is not among the targeted victims. In contrast to the naive approach, the attack requires only $\mathcal{O}(log(n))$ shared images and only $\mathcal{O}(log(n))$ image requests, enabling the attack on larger groups of users.

In practice, launching a leaky image attack against a group of users requires sharing a set of images with different subsets of the targeted users. This process can be automated, either through APIs provided by image sharing services or through UI-level web automation scripts. However, this process will most likely be website-specific which makes it expensive for attacking multiple websites at once.

3.4 Linking User Identities

The third attack based on leaky images aims at linking multiple identities that a single individual has at different image sharing services. Let *siteA* and *siteB* be two image sharing services, and let u_{siteA} and u_{siteB} be two user accounts, registered at the two image sharing services, respectively. The

```
1  <!-- Three users (u1, u2, u3) have access to two
2  images (i1, i2) as follows: u1 to (i1);
3  u2 to (i2); u3 to (i1, i2) -->
4  <object data="leaky-domain.com/i1.png">
5   <object data="evil.com?info=not_i1?sid=2342"/>
6  </object>
7  <object data="leaky-domain.com/i2.png">
8   <object data="evil.com?info=not_i2?sid=2342"/>
9  </object>
10
11 <object data="leaky-domain.com/invalidImg.png">
12  <object data="leaky-domain.com/invalidImg2.png">
13   <object data="leaky-domain.com/invalidImg3.png">
14    <object data="evil.com?info=loaded?sid=2342"/>
15   </object>
16  </object>
17 </object>
```

Figure 3: HTML-only variant of the leaky image group attack. All the `object` tags should have the `type` property set to `image/png`.

attacker wants to determine whether u_{siteA} and u_{siteB} belong to the same individual. For example, this attack might be performed by law enforcement entities to check whether a user account that is involved in criminal activities matches another user account that is known to belong to a suspect.

To link two user identities, the attacker essentially performs two leaky image attacks in parallel, one for each image sharing service. Specifically, the attacker shares an image i_{siteA} with u_{siteA} through one image sharing service and an image i_{siteB} with u_{siteB} through the other image sharing service. The attacker-controlled website requests both i_{siteA} and i_{siteB}. Once the targeted individual visits this site, both requests will succeed and establish the fact that the users u_{siteA} and u_{siteB} correspond to the same individual. For any other visitors of the site, at least one request will fail because the two requests only succeed if the browser is logged into both user accounts u_{siteA} and u_{siteB}.

The basic idea of linking user accounts generalizes to more than two image sharing services and to user accounts of more than a single individual. For example, by performing two attacks on groups of users, as described in Section 3.3, in parallel, an attacker can establish pairwise relationships between the two groups of users.

3.5 HTML-only Attack

The leaky image attack is based on the ability of a client-side website to request an image and to report back to the attacker-controlled server-side whether the request was successful or not. One way to implement it is using client-side JavaScript code, as shown in Figure 1. However, privacy-aware users may disable JavaScript completely or use a security mechanism that prevents JavaScript code from reading details about images loaded from different domains.

We present a variant of the leaky image attack implemented using only HTML code, i.e., without any JavaScript or CSS. The idea is to use the `object` HTML tag, which allows a website to specify fallback content to be loaded if there is an error in loading some previously specified content.[5] When nesting such `object` elements, the browser first requests the resource specified in the outer element, and in case it fails, it performs a request to the inner element instead. Essentially, this behavior corresponds to a logical *if-not* instruction in pure HTML which an attacker may use to implement the leaky image attack.

Figure 3 shows an example of this attack variant. We assume that there are three users u_1, u_2, and u_3 in the target group and that the attacker can share leaky images from leaky-domain.com with each of them. The comment at the beginning of Figure 3 specifies the exact sharing configuration. We again need $log(n)$ images to track n users, as for the JavaScript-based attack against a group of users (Section 3.3). We assume that the server-side generates the attack code upon receiving the request, and that the generated code contains a session ID as part of the reporting links pointing to evil.com. In the example, the session ID is 2342. Its purpose is to enable the server-side code to link multiple requests coming from the same client.

The main insight of this attack variant is to place a request to the attacker's domain as fallbacks for leaky image requests. For example, if the request to the leaky image i_1 at line 4 fails, a request is made to evil.com for an alternative resource in line 5. This request leaks the information that the current user cannot access i_1, i.e., `info=not_i1`. By performing similar requests for all the leaky images, the attacker leaks enough information for precisely identifying individual users. For example, if in a given session, evil.com receives `not_i1`, but not `not_i1`, the attacker can conclude that the user is u_2. Because the server-side infers the user from the absence of requests, it is important to ensure that the current tracking session is successfully completed before drawing any conclusions. Specifically, we must ensure that the user or the browser did not stop the page load before all the nested `object` tags were evaluated. One way to ensure this property is to add a sufficiently high number of nested requests to non-existent images in lines 11 to 13 followed by a request that informs the attacker that the tracking is completed, in line 14. The server discards every session that does not contain this last message.

As a proof of concept, we tested the example attack and several variants of it in the newest Firefox and Chrome browsers and find the HTML-only attack to work as expected.

[5] `https://html.spec.whatwg.org/multipage/iframe-embed-object.html#the-object-element`

3.6 Discussion

Tracking pixels Leaky images are related to the widely used tracking pixels, also called web beacons [14, 8, 47], but both differ regarding who learns about a user's identity. A tracking pixel is a small image that a website s loads from a tracker website s_{track}. The image request contains the user's cookie for s_{track}, enabling the tracker to recognize users across different page visits. As a result, the tracking service can analyze which pages of s users visit and show this information in aggregated form to the provider of s. If the tracker also operates services where users register, it can learn which user visits which site. In contrast, leaky images enable the operator of a site s to learn that a target user is visiting s, without relying on a tracker to share this information, but by abusing an image sharing service. As for tracking pixels, an attacker can deploy leaky image attacks with images of 1x1 pixel size to reduce its impact on page loading time.

Fingerprinting Web fingerprinting techniques [12, 29, 10, 22, 1, 2, 30] use high-entropy properties of web browsers, such as the set of installed fonts or the size of the browser window, to heuristically recognize users. Like fingerprinting, leaky images aim at undermining the privacy of users. Unlike fingerprinting, the attacks presented here enable an attacker to determine specific user accounts, instead of recognizing that one visitor is likely to be the same as another visitor. Furthermore, leaky images can determine a visitor's identity with 100% certainty, whereas fingerprinting heuristically relies on the entropy of browser properties.

Targeted attacks versus large-scale tracking Leaky images are well suited for targeted attacks [37, 6, 26, 16], but not for large-scale tracking of millions of users. One reason is that leaky images require the attacker to share an image with each victim, which is unlikely to scale beyond several hundreds users. Another reason is that the number of image requests that a website needs to perform increases logarithmically with the number of targeted users, as discussed in Section 3.3. Hence, instead of aiming at large-scale tracking in the spirit of tracking pixels or fingerprinting, leaky images are better suited to target (sets of) individuals. However, this type of targeted attacks is reported to be increasingly popular, especially when dealing with high-value victims [37].

4 Leaky Images in Popular Websites

The attacks presented in the previous section make several assumptions. In particular, leaky images depend on how real-world image sharing services implement access control for shared images. To understand to what extent popular websites are affected by the privacy problem discussed in this paper, we systematically study the prevalence of leaky images. The following presents our methodology (Section 4.1), our main findings (Section 4.2), and discusses our ongoing efforts toward disclosing the detected problems in a responsible way (Section 4.3).

4.1 Methodology

Selection of websites To select popular image sharing services to study, we examined the top 500 most popular websites, according to the "Top Moz 500" list[6]. We focus on websites that enable users to share data with each other. We exclude sites that do not offer an English language interface and websites that do not offer the possibility to create user accounts. This selection yields a list of 30 websites, which we study in more detail. Table 3 shows the studied websites, along with their popularity rank. The list contains all of the six most popular websites, and nine of the ten most popular websites. Many of the analyzed sites are social media platforms, services for sharing some kind of data, and communication platforms.

Image sharing One condition for our attacks is that an attacker can share an image with a victim. We carefully analyze the 30 sites in Table 3 to check whether a site provides an image sharing service. To this end, we create multiple accounts on each site and attempt to share images between these accounts using different channels, e.g., chat windows or social media shares. Once an image is shared between two accounts, we check if the two accounts indeed have access to the image. If this requirement is met, we check that a third account cannot access the image.

Access control mechanism For websites that act as image sharing services, we check whether the access control of a shared image is implemented in a way that causes leaky images, as presented in Table 2. Specifically, we check whether the access to a shared image is protected by authentication and whether both users access the image through a common link, i.e., a link known to the attacker. A site that fulfills also this condition exposes its users to leaky image attacks.

4.2 Prevalence of Leaky Images in the Wild

Among the 30 studied websites, we identify a total of eight websites that suffer from leaky images. As shown in Table 3 (column "Leaky images"), the affected sites include the three most popular sites, Facebook, Twitter, and Google, and represent over 25% of all sites that we study. The following discusses each of the vulnerable sites in detail and explains how an attacker can establish a leaky image with a target user. Table 4 summarizes the discussion in a concise way.

[6] `https://moz.com/top500`

Table 3: List of analyzed websites, whether they suffer from leaky images, and how the respective security teams have reacted to our notifications about the privacy leak.

Rank	Domain	Leaky images	Confirmed	Fix	Bug bounty
1	**facebook.com**	**yes**	**yes**	**yes**	**yes**
2	**twitter.com**	**yes**	**yes**	**yes**	**yes**
3	**google.com**	**yes**	**yes**	**planned**	**no**
4	youtube.com	no			
5	instagram.com	no			
6	linkedin.com	no			
8	pinterest.com	no			
9	wikipedia.org	no			
10	**wordpress.com**	**yes**	**no**	**no**	**no**
15	tumblr.com	no			
18	vimeo.com	no			
19	flickr.com	no			
25	vk.com	no			
26	reddit.com	no			
33	blogger.com	no			
35	**github.com**	**yes**	**no**	**no**	**no**
39	myspace.com	no			
54	stumbleupon.com	no			
65	**dropbox.com**	**yes**	**yes**	**planned**	**yes**
71	msn.com	no			
72	slideshare.net	no			
91	typepad.com	no			
126	**live.com**	**yes**	**yes**	**planned**	**no**
152	spotify.com	no			
160	goodreads.com	no			
161	scribd.com	no			
163	imgur.com	no			
166	photobucket.com	no			
170	deviantart.com	no			
217	**skype.com**	**yes**	**yes**	**planned**	**no**

Facebook Images hosted on Facebook are in general delivered by content delivery networks not hosted at the facebook.com domain, but, e.g., at fbcdn.net. Hence, the fact that facebook.com cookie is not sent along with requests to shared images disables the leaky image attacks. However, we identified an exception to this rule, where a leaky image can be placed at `https://m.facebook.com/photo/view\_full\_size/?fbid=xxx`. The `fbid` is a unique identifier that is associated with each picture on Facebook, and it is easy to retrieve this identifier from the address bar of an image page. The attacker must gather this identifier and concatenate it with the leaky image URL given above. By tweaking the picture's privacy settings, the attacker can control the subset of friends that are authorized to access the image, opening the door for individual and group attacks. A prerequisite of creating a leaky image on Facebook is that the victim is a "friend" of the attacker.

Twitter Every image sent in a private chat on Twitter is a leaky image. The victim and the attacker can exchange messages on private chats, and hence send images, if one of them checked "Receive direct messages from anyone" in their settings or if one is a follower of the other. An image sent on a private chat can only be accessed by the two participants, based on their login state, i.e., these images are leaky images. The attacker can easily retrieve the leaky image URL from the conversation and include it in another page. A limitation of the attack via Twitter is that we are currently not aware of a way of sharing an image with multiple users at once.

Google We identified two leaky image channels on Google's domains: one in the thumbnails of Google Drive documents and one in Google Hangouts conversations. To share documents with the victim, an attacker only needs the email address of the victim, while in order to send Hangouts messages, the victim needs to accept the chat invitation from the attacker. The thumbnail-based attack is more powerful since it allows to easily add and remove users to the group of users that have access to an image. Moreover, by unselecting the "Notify people" option when sharing, the victim users are not even aware of this operation. An advantage of the Hangouts channel, though, is that the victim has no way to revoke its rights to the leaky image, once the image has been received in a chat, as opposed to Drive, where the victim can remove a shared document from her cloud.

Wordpress To create a leaky image via Wordpress, the attacker needs to convince the victim to become a reader of his blog, or the other way around. Once this connection is established, every image posted on the shared private blog is a leaky image between the two users. Fulfilling this strong prerequisite may require non-trivial social engineering.

GitHub Private repositories on GitHub enable leaky images. Once the victim and the attacker share such a repository, every committed image can be accessed through a link in the web interface, e.g., `https://github.com/johndoe/my-awesome-project/raw/master/car.jpg`. Only users logged into GitHub who were granted access to the repository *my-awesome-project* can access the image. To control the number of users that have access to the image, the attacker can remove or add contributors to the project.

Dropbox Every image uploaded on Dropbox can be accessed through a leaky image endpoint by appending the HTTP parameter `dl=1` to a shared image URL. Dropbox allows the attacker to share such images with arbitrary email addresses and to fine-tune the permissions to access the image by including and excluding users at any time. Once the image is shared, our attack can be successfully deployed,

Table 4: Leaky images in popular websites, the attack's preconditions, the image sharing channel and the implemented authentication mechanism as introduced in Table 2

Domain	Prerequisites	Image sharing channel	Authentication mechanism
facebook.com	Victim and attacker are "friends"	Image sharing	(5), (2)
twitter.com	Victim and attacker can exchange messages	Private message	(2)
google.com	*None*	Google Drive document Private message	(3), (2)
wordpress.com	Victim is a viewer of the attacker's private blog	Posts on private blogs	(2)
github.com	Victim and attacker share a private repository	Private repository	(3), (2)
dropbox.com	*None*	Image sharing	(3), (6), (2)
live.com	*None*	Shared folder on OneDrive	(3), (2)
skype.com	Victim and attacker can exchange messages	Private message	(2)

without requiring the victim to accept the shared image. However, the victim can revoke its rights to access an image by removing it from the "Sharing" section of her account.

Live.com Setting up a leaky image on One Drive, a cloud storage platform available on a live.com subdomain, is very similar to the other two file sharing services that we study, Google Drive and Dropbox. The attacker can share images with arbitrary email addresses and the victim does not need to acknowledge the sharing. Moreover, the attacker can easily deploy a group attack due to the ease in changing the group of users that have access to a particular image.

Skype In the Skype web interface, every image sent in a chat is a leaky image. Note that most of the users probably access the service through a desktop or mobile standalone client, hence the impact of this attack is limited to the web users. Moreover, Skype automatically logs out the user from time to time, limiting the time window for the attack.

Our study of leaky images in real-world sites enables several observations.

Leaky images are prevalent The first and perhaps most important observation is that many of the most popular websites allow an attacker to create leaky images. From an attacker's point of view, a single leaky image is sufficient to track a user. If a victim is registered as a user with at least one of the affected image sharing services, then the attacker can create a user account at that service and share a leaky image with the victim.

Victims may not notice sharing a leaky image Several of the affected image sharing services enable an attacker to share an image with a specific user without any notice given to the user. For example, if the attacker posts an image on her Facebook profile and tweaks the privacy settings so that only the victim can access it, then the victim is not informed in any way. Another example is Google Drive, which allows sharing files with arbitrary email addresses while instructing the website to not send an email that informs the other user.

Victims cannot "unshare" a leaky image For some services, the victim gets informed in some way that a connection to the attacker has been established. For example, to set up a leaky image on Twitter, the attacker needs to send a private message to the victim, which may make the victim suspicious. However, even if the victim knows about the shared image, for most websites, there is no way for a user to revoke its right to access the image. Specifically, let's assume the victim receives a cute cat picture from a Google Hangouts contact. Let us now assume that the victim is aware of the leaky image attack and that she suspects the sender of the image tracking her. We are not aware of any way in which the victim can revoke the right to access the received image.

Image sharing services use a diverse mix of implementation strategies Secret URLs and per-user authenticated URLs are widely implemented techniques that protects against our attack. However, many websites use multiple such strategies and hence, it is enough if one of the API endpoints uses leaky images. Identifying this endpoint is often a hard task: for example, in the case of Facebook, most of the website rigorously implements secret URLs, but one API endpoint belonging to a mobile subdomain exposes leaky images. After identifying this endpoint we realized that it can be accessed without any problem from a desktop browser as well, enabling all the attacks we describe in Section 3.

The attack surface varies from site to site Some but not all image sharing services require a trust relation between the attacker and the victim before a leaky image can be shared. For example, an attacker must first befriend a victim on Facebook before sharing an image with the victim, whereas no such requirement exists on Dropbox or Google Drive. However, considering that most users have hundreds of friends on social networks, there is a good chance that a trust channel is established before the attack starts. In the case of Wordpress the prerequisite that the "victim is a viewer of the attacker's private blog" appears harder to meet and may require advanced social engineering. Nonetheless, we believe that such leaky images may still be relevant in certain targeted attacks.

Moreover, three of the eight vulnerable sites allow attackers to share images with arbitrary users, without any prerequisites (Table 4).

Since our study of the prevalence of leaky images is mostly manual, we cannot guarantee that the 22 sites for which we could not create a leaky image are not affected by the problem. For some sites, though, we are confident that they are not affected, as these sites do not allow users to upload images. A more detailed analysis would require in-depth knowledge of the implementation of the studied sites, and ideally also access to the server-side source code. We hope that our results will spur future work on more automated analyses that identify leaky images.

4.3 Responsible Disclosure and Feedback from Image Sharing Services

After identifying image sharing services that suffer from leaky images, we contacted their security teams to disclose the problem in a responsible way. Between March 26 and March 29, 2018, we sent a detailed description of the general problem, how the specific website can be abused to create leaky images, and how it may affect the privacy of users of the site. Most security teams we contacted were very responsive and eager to collaborate upon fixing the issue.

Confirmed reports The last three columns of Table 3 summarize how the security teams of the contacted companies reacted to our reports. For most of the websites, the security teams confirmed that the reported vulnerability is worth fixing, and at least six of the sites have already fixed the problem or have decided to fix it. In particular, the top three websites all confirmed the reported issue and all have been working on fixing it. Given the huge user bases of these sites and the privacy implications of leaky images for their users, this reaction is perhaps unsurprising. As another sign of appreciation of our reports, the authors have received bug bounties from (so far) three of the eight affected sites.

Dismissed reports Two of our reports were flagged as false positives. The security teams of the corresponding websites replied by saying that leaky images are a "desired behavior" or that the impact on privacy of their user is limited. Comparing Table 3 with Table 4 shows that the sites that dismiss our report are those where the prerequisites for creating a leaky image are harder to fulfill than for the other sites: Creating a leaky image on GitHub requires the attacker and the victim to share a private repository, and Wordpress requires that the victim is a viewer of the attacker's private blog. While we agree that the attack surface is relatively small for these two sites, leaky images may nevertheless cause surprising privacy leaks. For example, an employee might track her colleagues or even her boss if their company uses private GitHub repositories.

Case study: Fix by Facebook To illustrate how image sharing services may fix a leaky images problem, we describe how Facebook addressed the problem in reaction to our report. As mentioned earlier, Facebook employs mostly secret URLs and uses content delivery networks to serve images. However, we were able to identify a mobile API endpoint that uses leaky images and redirects the user to the corresponding content delivery network link. This endpoint is used in the mobile user interface for enabling users to download the full resolution version of an image. The redirection was performed at HTTP level, hence it resulted in a successful image request when inserted in a third-party website using the `<a>` HTML tag. The fix deployed by Facebook was to perform a redirection at JavaScript level, i.e. load an intermediate HTML that contains a JavaScript snippet that rewrites `document.location.href`. This fix enables a benign user to still successfully download the full resolution image through a browser request, but disables third-party image inclusions. However, we believe that such a fix does not generalize and cannot be deployed to the other identified vulnerabilities. Hence, we describe alternative ways to protect against a leaky image attacks in Section 5.

Case study: Fix by Twitter A second case study of how websites can move away from leaky images comes from Twitter that changed its API[7] in response to our report[8]. First, they disabled cookie-based authentication for images. Second, they changed the API in a way that image URLs are only delivered on secure channels, i.e., only authenticated HTTPS requests. Last, Twitter also changed the user interface to only render images from strangers when explicit consent is given. Essentially, Twitter moved from implementation strategy (2) to (5) in Table 2 in response to our report.

Overall, we conclude from our experience of disclosing leaky images that popular websites consider it to be a serious privacy problem, and that they are interested in detecting and avoiding leaky images.

5 Mitigation Techniques

In this section, we describe several techniques to defend against leaky image attacks. The mitigations range from server-side fixes that websites can deploy, over improved privacy settings that empower users to control what is shared with them, to browser-based mitigations.

[7] `https://twitter.com/TwitterAPI/status/1039631353141112832`

[8] `https://hackerone.com/reports/329957`

5.1 Server-Side Mitigations

The perhaps easiest way to defend against the attack presented in this paper is to modify the server-side implementation of an image sharing service, so that it is not possible anymore to create leaky images. There are multiple courses of actions to approach this issue.

First, a controversial fix to the problem is to disable authenticated image requests altogether. Instead of relying on, e.g., cookies to control who can access an image, an image sharing service could deliver secret links only to those users that should access an image. Once a user knows the link she can freely get the image through the link, independent of whether she is logged into the image sharing service or not. This strategy corresponds to case 5 in Table 2. Multiple websites we report about in Table 3 implement such an image sharing strategy. The most notable examples are Facebook, which employs this technique in most parts of their website, and Dropbox, which implements this technique as part of their link sharing functionality. The drawback of this fix is that the link's secrecy might be compromised in several ways outside of the control of the image sharing service: by using insecure channels, such as HTTP, through side-channel attacks in the browser, such as cache attacks [20], or simply by having the users handle the links in an insecure way because they are not aware of the secrecy requirements.

Second, an alternative fix is to enforce an even stricter cookie-based access control on the server-side. In this case, the image sharing service enforces that each user accesses a shared image through a secret, user-specific link that is not shared between users. As a result, the attacker does not know which link the victim could use to access a shared image, and therefore the attacker cannot embed such a link in any website. This implementation strategy corresponds to case 3 in Table 2. On the downside, implementing this defense may prove challenging due to the additional requirement of guaranteeing the mapping between users and URLs, especially when content delivery networks are involved. Additionally, it may cause a slowdown for each image request due to the added access control mechanism.

Third, one may deploy mitigations against CSRF.[9] One of them is to use the `origin` HTTP header to ensure that the given image can only be embedded on specific websites. The `origin` HTTP header is sent automatically by the browser with every request, and it precisely identifies the page that requests a resource. The server-side can check the request's `origin` and refuse to respond with an authenticated image to unknown third-party request. For example, facebook.com could refuse to respond with a valid image to an HTTP request with the `origin` set to evil.com. However, this mitigation cannot defend against tracking code injected into a trusted domain. For example, until recently Facebook allowed users to post custom HTML code on their profile page. If a user decides to insert leaky image-based tracking code on the profile page, to be notified when a target user visits the profile page, then the CSRF-style mitigation does not prevent the attack. The reason for this is that the request's origin would be set to facebook.com, and hence the server-side code will trust the page and serve the image.

Similarly, the server-side can set the `Cross-Origin-Resource-Policy` response header on authenticated image requests and thus limit which websites can include a specific image. Browsers will only render images for which the origin of the request matches the origin of the embedding website or if they correspond to the same site. This solution is more coarse-grained than the previously discussed `origin` checking since it does not allow for cross-origin integration of authenticated images, but it is easier to deploy since it only requires a header set instead of a header check. The `From-Origin` header was proposed for allowing a more fine-grained integration policy, but to this date there is no interest from browser vendors side to implement such a feature.

Another applicable CSRF mitigation is the `SameSite` cookie attribute. When set to "strict" for a cookie, the attribute prevents the browser from sending the cookie along with cross-site requests, which effectively prevents leaky images. However, the "strict" setting may be too strict for most image sharing services, because it affects all links to the service's website. For example, a link in a corporate email to a private GitHub project or to a private Google Doc would not work anymore, because when clicking the link, the session cookie is not sent along with the request. The less restrictive "lax" setting of the `SameSite` attribute does not suffer from these problems, but it also does not prevent leaky images attacks, as it does not affect GET requests.

A challenge with all the described server-side defenses is that they require the developers to be aware of the vulnerability in the first place. From our experience, a complex website may allow sharing images in several ways, possibly spanning different UI-level user interactions and different API endpoints supported by the image sharing service. Since rigorously inspecting all possible ways to share an image is non-trivial, we see a need for future work on automatically identifying leaky images. At least parts of the methodology we propose could be automated with limited effort. To check whether an image request requires authentication, one can perform the request in one browser where the user is logged in, and then try the same request in another instance of the browser in "private" or "incognito" mode, i.e., without being logged in. Comparing the success of the two requests reveals whether the image request relies in any form of authentication, such as cookies. Automating the rest of our methodology requires some support by image sharing services. In particular, automatically checking that a leaky image is accessible only by a subset of a website's users, requires APIs

[9] `https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet`

to handle user accounts and to share images between users.

Despite the challenges in identifying leaky images, we believe that server-side mitigations are the most straightforward solution, at least in the short term. In the long term, a more complete solution would be desirable, such as those described in the following.

5.2 Browser Mitigations

The current HTTP standard does not specify a policy for third-party cookies[10], but it encourages browser vendors to experiment with different such policies. More precisely, the current standard lets the browser decide whether to automatically attach the user's cookie to third-party requests. Most browsers decide to attach third-party cookies, but there are certain counter-examples, such as the Tor browser. In Tor, cookies are sent only to the domain typed by the user in the address bar.

Considering the possible privacy implications of leaky images and other previously reported tracking techniques [8], one possible mitigation would be that browsers specify as default behavior to not send cookies with third-party (image) requests. If this behavior is overwritten, possibly using a special HTTP header or tag, the user should be informed through a transparent mechanism. Moreover, the user should be offered the possibility to prevent the website from overwriting the default behavior. We believe this measure would be in the spirit of the newly adopted European Union's General Data Protection Regulation which requires *data protection by design and by default*. However, such an extreme move may impact certain players in the web ecosystem, such as the advertisement ecosystem. To address this issue, advertisers may decide to move towards safer distribution mechanisms, such as the one popularized by the Brave browser.

An alternative to the previously discussed policy is to allow authenticated image requests, but only render them if the browser is confident that there are no observable differences between an authenticated request and a non-authenticated one. To this end, the browser could perform two image requests instead of one: one request with third-party cookies and one request without. If the browser receives two equivalent responses, it can safely render the content, since no sensitive information is leaked about the authenticated user. This solution would still allow most of the usages of third-party cookies, e.g. tracking pixels, but prevent the leaky image attack described here. A possible downside might be the false positives due to strategy (3) in Table 2, but we hypothesize that requests to such images rarely appear in benign third-party image requests. A second possible drawback of this solution may be the increase in web traffic and the potential performance penalties. Future work should test the benefits of this defense and the cost imposed by the additional image request.

[10] `https://tools.ietf.org/html/rfc6265#page-28`

To reduce the cost imposed by an additional image request, a hybrid mechanism could disable authenticated image requests by default, and allow them only for the resources specified by a CSP directive. For the allowed authenticated images, the browser deploys the double image requests mechanism described earlier. We advocate this as our preferred browser-level defense since it can also defend against other privacy attacks, e.g. reading third-party image pixels through a side channel [21], while still permitting benign uses.

Similarly to ShareMeNot [32], one can also implement a browser mechanism in which all third-party image requests are blocked unless the user authorizes them by providing explicit consent. To release the burden from the user, a hybrid mechanism can be deployed in which the website requires authenticated requests only for a subset of images for which the user needs to provide consent.

Another solution for when third-party cookies are allowed is for browsers to implement some form of information flow control to ensure that the fact whether a third-party request was successfully loaded or not, cannot be sent outside of the browser. A similar approach is deployed in *tainted canvas*[11], which disallows pixel reads after a third-party image is painted on the canvas. Implementing such an information flow control for third-party images may, however, be challenging in practice, since the fact whether an image has successfully loaded or not can be retrieved through multiple side channels, such as the `object` tag or by reading the size of the contained div.

The mechanisms described in this section vary both in terms of implementation effort required for deploying them and in terms of their possible impact on the existing state of the web, i.e., incompatibility with existing websites. Therefore, to aid the browser vendors to take an informed decision, future work should perform an in-depth analysis of all these defenses in terms of usability, compatibility and deployment cost, in the style of Calzavara et al. [9], and possibly propose additional solutions.

5.3 Better Privacy Control for Users

A worrisome finding of our prevalence study is that a user has little control over the image sharing process. For example, for some image sharing services, the user does not have any option to restrict which other users can privately share an image with her. In others, there is no way for a user to revoke her right to access a specific image. Moreover, in most of the websites we analyzed, it is difficult to even obtain a complete list of images privately shared with the current account. For example, a motivated user who wants to obtain this list must check all the conversations in a messaging platform, or all the images of all friends on a social network.

[11] `https://html.spec.whatwg.org/multipage/canvas.html#security-with-canvas-elements`

We believe that image sharing services should provide users more control over image sharing policies, to enable privacy-aware users to protect their privacy. Specifically, a user should be allowed to decide who has the right to share an image with her and she should be granted the right to revoke her access to a given image. Ideally, websites would also offer the user a list of all the images shared with her and a transparent notification mechanism that announces the user when certain changes are made to this list. Empowering the users with these tools may help mitigate some of the leaky image attacks by attracting user's attention to suspicious image sharing, allowing users to revoke access to leaky images.

The privacy controls for web users presented in this section will be useful mostly for advanced users, while the majority of the users are unlikely to take advantage of such fine-grained controls. Therefore, we believe that the most effective mitigations against leaky images are at the server side or browser level.

6 Related Work

Previous work shows risks associated with images on the web, such as malicious JavaScript code embedded in SVGs [17], image-based fingerprinting of browser extensions [35], and leaking sensitive information, such as the gender or the location of a user uploading an image [11]. This work introduces a new risk: privacy leaks due to shared images. Lekies et al. [24] describe privacy leaks resulting from dynamically generated JavaScript. The source of this problem is the same as for leaky images: both JavaScript code and images are excepted from the same-origin policy. While privacy leaks in dynamic JavaScript reveal confidential information about the user, such as credentials, leaky images allow for tracking specific users on third-party websites. Heiderich et al. [18] introduce a scriptless, CSS-based web attacks. The HTML-only variant of leaky images does not rely on CSS and also differ in the kinds of leaked information: While the attack by Heiderich et al. leaks content of the current website, our attacks leak the identity of the user.

Wondracek et al. [46] present a privacy leak in social networks related to our group attack. In their work, the attacker neither has control over the group structure nor can she easily track individuals. A more recent attack [41] deanonymizes social media users by correlating links on their profiles with browsing histories. In contrast, our attack does not require such histories. Another recent attack [44] retrieves sensitive information of social media accounts using the advertisement API provided by a social network. However, their attack cannot be used to track users on third-party websites.

Cross-Site Request Forgery (CSRF) is similar in spirit to leaky image attacks: both rely on the fact that browsers send cookies with third-party requests. For CSRF, this behavior results in an unauthorized action on a third-party website, whereas for leaky images, it results in deanonymizing the user. Existing techniques for defending [5] and detecting [31] CSRF partially address but do not fully solve the problem of leaky images (Section 5).

Browser fingerprinting is a widely deployed [1, 2, 30] persistent tracking mechanism. Various APIs have been proposed for fingerprinting: user agent and fonts [12], canvas [29, 10], ad blocker usage, and WebGL Renderer [22]. Empirical studies [12, 22] suggest that these technique have enough entropy to identify most of the users, or at least, to place a user in a small set of possible users, sometimes even across browsers [10]. The leaky image attack is complementary to fingerprinting, as discussed in detail in Section 3.6.

Another web tracking mechanism is through third-party requests, such as tracking pixels. Mayer and Mitchell [27] describe the tracking ecosystem and the privacy costs associated with these practices. Lerner et al. [25] show how tracking in popular websites evolves over time. Several other studies [42, 47, 13, 32, 8, 14] present a snapshot of the third-party tracking on the web at various moments in time. One of the recurring conclusion of these studies was that few big players can track most of the traffic on the Internet. We present the first image-based attack that allows a less powerful attacker to deanonymize visitors of a website.

Targeted attacks or advanced persistent threats are an increasingly popular class of cybersecurity incidents [37, 26]. Known attacks include spear phishing attacks [6] and targeted malware campaigns [26, 16]. Leaky images adds a privacy-related attack to the set of existing targeted attacks.

Several empirical studies analyze different security and privacy aspects of websites in production: postMessages [36], cookie stealing [4, 34], credentials theft [3], cross-site scripting [28, 23], browser fingerprinting [30, 1], deployment of CSP policies [45], and ReDoS vulnerabilities [38]. User privacy can also be impacted by security issues in browsers, such as JavaScript bindings bugs [7], micro-architectural bugs [20], and insufficient isolation of web content [19]. Neither of these studies explores privacy leaks caused by authenticated cross-origin image requests.

Van Goethem et al. [43] propose the use of timing channels for estimating the file size of a cross-origin resource. One could combine leaky images with such a channel to check if a privately shared image is accessible for a particular user, enabling the use of leaky images even if the browser would block cross-origin image requests. One difference between our attack and theirs is that leaky images provide 100% certainty that a victim has visited a website, which a a probabilistic timing channel cannot provide.

Several researchers document the difficulty of notifying the maintainers of websites or open-source projects about security bugs in software [24, 40, 39]. We experienced quick and helpful responses by all websites we contacted, with an initial response within less than a week. One reason for this difference may be that we used the bug bounty channels provided by the websites to report the problems [15, 48].

7 Conclusions

This paper presents leaky images, a targeted deanonymization attack that leverages specific access control practices employed in popular websites. The main insight of the attack is a simple yet effective observation: Privately shared resources that are exempted from the same origin policy can be exploited to reveal whether a specific user is visiting an attacker-controlled website. We describe several flavors of this attack: targeted tracking of single users, group tracking, pseudonym linking, and an HTML-only attack.

We show that some of the most popular websites suffer from leaky images, and that the problem often affects any registered users of these websites. We reported all the identified vulnerabilities to the security teams of the affected websites. Most of them acknowledge the problem and some already proceeded to fixing it. This feedback shows that the problem we identified is important to practitioners. Our paper helps raising awareness among developers and researchers to avoid this privacy issue in the future.

Acknowledgments
Thanks to Stefano Calzavara and the anonymous reviewers for their feedback on this paper. This work was supported by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, by the German Research Foundation within the ConcSys and Perf4JS projects, and by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

References

[1] G. Acar, C. Eubank, S. Englehardt, M. Juárez, A. Narayanan, and C. Díaz, "The web never forgets: Persistent tracking mechanisms in the wild," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, 2014, pp. 674–689. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660347

[2] G. Acar, M. Juárez, N. Nikiforakis, C. Díaz, S. F. Gürses, F. Piessens, and B. Preneel, "Fpdetective: dusting the web for fingerprinters," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, 2013, pp. 1129–1140. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516674

[3] S. V. Acker, D. Hausknecht, and A. Sabelfeld, "Measuring login webpage security," in *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, 2017, pp. 1753–1760. [Online]. Available: http://doi.acm.org/10.1145/3019612.3019798

[4] D. J. andZhaoGL15 Ranjit Jhala, S. Lerner, and H. Shacham, "An empirical study of privacy-violating information flows in javascript web applications," in *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, 2010, pp. 270–283. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866339

[5] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, 2008, pp. 75–88. [Online]. Available: http://doi.acm.org/10.1145/1455770.1455782

[6] S. L. Blond, A. Uritesc, C. Gilbert, Z. L. Chua, P. Saxena, and E. Kirda, "A look at targeted attacks through the lense of an NGO," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, 2014, pp. 543–558. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/le-blond

[7] F. Brown, S. Narayan, R. S. Wahby, D. R. Engler, R. Jhala, and D. Stefan, "Finding and preventing bugs in javascript bindings," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, 2017, pp. 559–578. [Online]. Available: https://doi.org/10.1109/SP.2017.68

[8] A. Cahn, S. Alfeld, P. Barford, and S. Muthukrishnan, "An empirical study of web cookies," in *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, 2016, pp. 891–901. [Online]. Available: http://doi.acm.org/10.1145/2872427.2882991

[9] S. Calzavara, R. Focardi, M. Squarcina, and M. Tempesta, "Surviving the web: A journey into web session security," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 13:1–13:34, 2017. [Online]. Available: https://doi.org/10.1145/3038923

[10] Y. Cao, S. Li, and E. Wijmans, "(cross-)browser fingerprinting via OS and hardware level features," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/cross-browser-fingerprinting-os-and-hardware-level-features/

[11] M. Cheung and J. She, "Evaluating the privacy risk of user-shared images," *ACM Transactions on Multi-*

media Computing, Communications, and Applications (TOMM), vol. 12, no. 4s, p. 58, 2016.

[12] P. Eckersley, "How unique is your web browser?" in *Privacy Enhancing Technologies, 10th International Symposium, PETS 2010, Berlin, Germany, July 21-23, 2010. Proceedings*, 2010, pp. 1–18. [Online]. Available: https://doi.org/10.1007/978-3-642-14527-8_1

[13] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016, pp. 1388–1401. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978313

[14] S. Englehardt, D. Reisman, C. Eubank, P. Zimmerman, J. Mayer, A. Narayanan, and E. W. Felten, "Cookies that give you away: The surveillance implications of web tracking," in *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, 2015, pp. 289–299. [Online]. Available: http://doi.acm.org/10.1145/2736277.2741679

[15] M. Finifter, D. Akhawe, and D. A. Wagner, "An empirical study of vulnerability rewards programs," in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, 2013, pp. 273–288. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/finifter

[16] S. Hardy, M. Crete-Nishihata, K. Kleemola, A. Senft, B. Sonne, G. Wiseman, P. Gill, and R. J. Deibert, "Targeted threat index: Characterizing and quantifying politically-motivated targeted malware," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, 2014, pp. 527–541. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/hardy

[17] M. Heiderich, T. Frosch, M. Jensen, and T. Holz, "Crouching tiger - hidden payload: security risks of scalable vectors graphics," in *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, 2011, pp. 239–250. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046735

[18] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk, "Scriptless attacks: Stealing more pie without touching the sill," *Journal of Computer Security*, vol. 22, no. 4, pp. 567–599, 2014. [Online]. Available: https://doi.org/10.3233/JCS-130494

[19] Y. Jia, Z. L. Chua, H. Hu, S. Chen, P. Saxena, and Z. Liang, ""the web/local" boundary is fuzzy: A security study of chrome's process-based sandboxing," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016, pp. 791–804. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978414

[20] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[21] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson, "Cross-origin pixel stealing: timing attacks using CSS filters," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, A. Sadeghi, V. D. Gligor, and M. Yung, Eds. ACM, 2013, pp. 1055–1062. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516712

[22] P. Laperdrix, W. Rudametkin, and B. Baudry, "Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, 2016, pp. 878–894. [Online]. Available: https://doi.org/10.1109/SP.2016.57

[23] S. Lekies, B. Stock, and M. Johns, "25 million flows later: large-scale detection of dom-based XSS," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, 2013, pp. 1193–1204. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516703

[24] S. Lekies, B. Stock, M. Wentzel, and M. Johns, "The unexpected dangers of dynamic javascript," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, 2015, pp. 723–735. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lekies

[25] A. Lerner, A. K. Simpson, T. Kohno, and F. Roesner, "Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lerner

[26] W. R. Marczak, J. Scott-Railton, M. Marquis-Boire, and V. Paxson, "When governments hack opponents: A look at actors and technology,"

in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, 2014, pp. 511–525. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/marczak

[27] J. R. Mayer and J. C. Mitchell, "Third-party web tracking: Policy and technology," in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, 2012, pp. 413–427. [Online]. Available: https://doi.org/10.1109/SP.2012.47

[28] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, "Riding out domsday: Toward detecting and preventing dom cross-site scripting," 2018.

[29] K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in html5," *Proceedings of W2SP*, pp. 1–12, 2012.

[30] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless monster: Exploring the ecosystem of web-based device fingerprinting," in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, 2013, pp. 541–555. [Online]. Available: https://doi.org/10.1109/SP.2013.43

[31] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow, "Deemon: Detecting CSRF with dynamic analysis and property graphs," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 1757–1771. [Online]. Available: http://doi.acm.org/10.1145/3133956.3133959

[32] F. Roesner, T. Kohno, and D. Wetherall, "Detecting and defending against third-party tracking on the web," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, 2012, pp. 155–168. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/roesner

[33] C. Shiflett, "Cross-site request forgeries," http://shiflett.org/articles/cross-site-request-forgeries.

[34] S. Sivakorn, I. Polakis, and A. D. Keromytis, "The cracked cookie jar: HTTP cookie hijacking and the exposure of private information," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, 2016, pp. 724–742. [Online]. Available: https://doi.org/10.1109/SP.2016.49

[35] A. Sjösten, S. V. Acker, and A. Sabelfeld, "Discovering browser extensions via web accessible resources," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017*, 2017, pp. 329–336. [Online]. Available: http://doi.acm.org/10.1145/3029806.3029820

[36] S. Son and V. Shmatikov, "The postman always rings twice: Attacking and defending postmessage in HTML5 websites," in *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013. [Online]. Available: https://www.cs.utexas.edu/~shmat/shmat_ndss13postman.pdf

[37] A. K. Sood and R. J. Enbody, "Targeted cyberattacks: A superset of advanced persistent threats," *IEEE Security & Privacy*, vol. 11, no. 1, pp. 54–61, 2013. [Online]. Available: https://doi.org/10.1109/MSP.2012.90

[38] C. Staicu and M. Pradel, "Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers," in *USENIX Security Symposium*, 2018, pp. 361–376.

[39] C.-A. Staicu, M. Pradel, and B. Livshits, "Understanding and automatically preventing injection attacks on Node.js," in *25th Annual Network and Distributed System Security Symposium, NDSS*, 2018.

[40] B. Stock, G. Pellegrino, C. Rossow, M. Johns, and M. Backes, "Hey, you have a problem: On the feasibility of large-scale web vulnerability notification," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, 2016, pp. 1015–1032. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/stock

[41] J. Su, A. Shukla, S. Goel, and A. Narayanan, "De-anonymizing web browsing data with social networks," in *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, 2017, pp. 1261–1269. [Online]. Available: http://doi.acm.org/10.1145/3038912.3052714

[42] M. Tran, X. Dong, Z. Liang, and X. Jiang, "Tracking the trackers: Fast and scalable dynamic analysis of web content for privacy violations," in *Applied Cryptography and Network Security - 10th International Conference, ACNS 2012, Singapore, June 26-29, 2012. Proceedings*, 2012, pp. 418–435. [Online]. Available: https://doi.org/10.1007/978-3-642-31284-7_25

[43] T. van Goethem, W. Joosen, and N. Nikiforakis, "The clock is still ticking: Timing attacks in the modern web," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, 2015, pp. 1382–1393. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813632

[44] G. Venkatadri, A. Andreou, Y. Liu, A. Mislove, K. P. Gummadi, P. Loiseau, and O. Goga, "Privacy risks with Facebook's pii-based targeting: Auditing a data Broker's advertising interface," 2018.

[45] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "CSP is dead, long live csp! on the insecurity of whitelists and the future of content security policy," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016, pp. 1376–1387. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978363

[46] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel, "A practical attack to de-anonymize social network users," in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, 2010, pp. 223–238. [Online]. Available: https://doi.org/10.1109/SP.2010.21

[47] Z. Yu, S. Macbeth, K. Modi, and J. M. Pujol, "Tracking the trackers," in *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, 2016, pp. 121–132. [Online]. Available: http://doi.acm.org/10.1145/2872427.2883028

[48] M. Zhao, J. Grossklags, and P. Liu, "An empirical study of web vulnerability discovery ecosystems," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, 2015, pp. 1105–1117. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813704