Paranoid: Privacy through Granular Separation of User Identities

Brandon Tan (tjiansin)
Brown University

Irvin Lim (ilim5)
Brown University

Abstract

Paranoid aims to allow users to regain control of their personal data. We explore a new paradigm of designing web services to present personal data on HTML webpages without having access to the data itself, with the assumption that the web service is untrusted. Paranoid achieves this by storing all personal data separately outside of the service, only substituting the private data on the client side via DOM manipulation using a Chromium extension, as well as a secure platform to share private data files with other users built on top of Keybase. We also show that such an approach is secure against various attack vectors, in the case of both malicious web services and users.

1 Introduction

1.1 Goals of the project

In the recent years, there have been an increasing number of high profile instances of users' personal information being compromised via web services. This includes, but is not limited to:

1. Data breaches arising from inadequate access control, e.g. Equifax announced a data breach in 2017 that exposed the personal information of over 140 million consumers worldwide [8];
2. Governments and law enforcement agencies that could potentially demand private corporations to hand over its users' personal information;
3. Lack of consent for services to pass on users' personal information to external vendors, e.g. the Facebook/Cambridge Analytica scandal which resulted in 87 million users' Facebook profiles being acquired by Cambridge Analytica for voter targeting in the 2016 US election;
4. Lack of guarantee for users' personal information being stored and processed securely by the service;
5. Personal data acquired from multiple sources can be correlated to paint a comprehensive overview of a users' online identity.

There have been various approaches in solving the above mentioned challenges, including information flow control (IFC) frameworks such as DStar [12], Resin [11] and Jacqueline [10], which require implicit trust in the service itself to adequately implement these guarantees. Additionally, while services like Facebook are starting to provide fine-grained control over who should be allowed access to what data, it is complex to implement and potentially buggy, as exemplified with a recent Facebook bug that affected 14 million users [9].

A web service may commonly ingest private data during registration, through the means of single sign-on (SSO) mechanisms using widely used accounts like Facebook and Google. The wide proliferation of SSO means that a user's identity could potentially be tracked and pieced together over multiple web services, by correlating user accounts with the same email address.

As such, we aimed to design a system which allows a user to control precisely who should be granted access to their personal data, such that without having been granted access, the user's identity on the service should remain anonymous from their point of view. This means that private data should not be divulged to the web service, even during registration where personal information like email addresses are commonly used to create a user account. At the same time, we want the web service to still be able to work as though it had the data in the first place.

1.2 Targeted services

We classify web services into two distinct categories: *data platforms* and *data consumers*.

A data consumer is a web service that manipulates private user data, and requires the actual private data to be divulged to the service, which otherwise would fail to perform a critical function and be of no value to the user. One such example would be Equifax, which requires the user's actual name

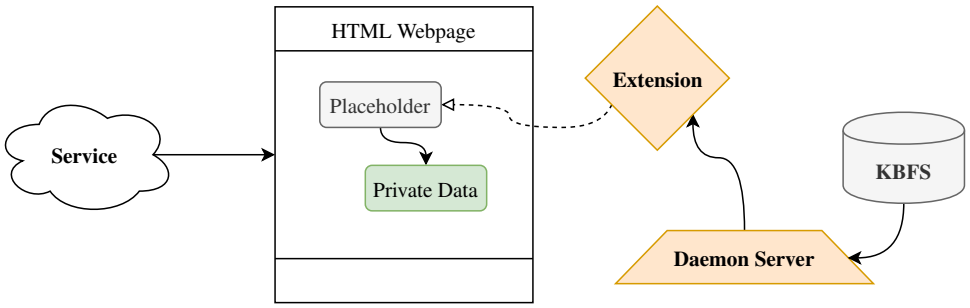

Figure 1: Main architecture of Paranoid

and social security number (SSN) in order to perform credit checks with credit bureaus and other financial institutions.

On the other hand, a data platform is a web service that serves user data to other users on the platform, and does not fail even if the data is hidden from the service. One such example would be social media sites like Facebook and Twitter, where the contents of a tweet can be completely transparent from the point of view of the service, but is only important to other users who view that tweet.

In this case, we will only be focusing on designing a system that can work on top of data platforms, since it would be pointless to come up with such a system for data consumers which require and expect input of true and accurate personal data.

Paranoid's key idea is thus to separate the presentation of the private user data from the contents of the data itself on these web services. Specifically, the HTML webpage returned by the web service should define the structure and appearance of the private data, while the private data is only substituted in the browser on the client side for users who have access rights to the data.

2 Overview of Paranoid

There are three main parts to our solution: 1) providing a public key-based authentication scheme that should be supported by a Paranoid-compliant service, 2) overlaying private data on top of a web page securely in a browser to users who have access to said private data, and 3) providing a means of defining which users have access to what pieces of data, as well as a storage and transport mechanism that will support it.

The main components of Paranoid is implemented as a Chromium browser extension, which would handle both authentication and overlaying of private data, as well as a daemon server, which the browser extension would communicate with in order to fetch and store data within the storage mechanism, as shown in Figure 1.

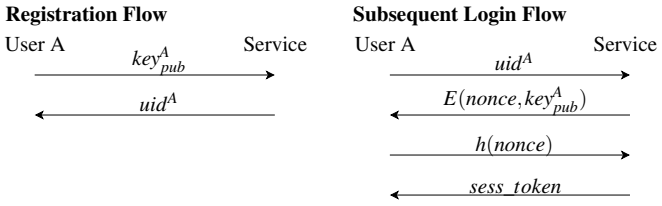

Figure 2: Registration and authentication flow in Paranoid using public keys.

2.1 Public key authentication

Paranoid provides a single sign-on (SSO) mechanism, similar to how users can log in to third-party services using their Google or Facebook accounts via OAuth 2.0. However, instead of sharing private data such as email address, name, etc. during registration, a new *service identity* is generated, which includes a private/public key pair. The public key portion is transmitted to the service, which stores it in order to authenticate the user for future logins. The web service returns a unique *UID* for the user, which is equivalent to the user's anonymous identifier for the service.

User authentication for the service is done using a challenge-response authentication scheme, where the service provides a randomly generated nonce challenge encrypted using the provided public key, and the user must prove they can retrieve the original nonce value by decrypting it with the corresponding private key, as shown in Figure 2. Once verified, the service can then provide a persistent session token, which is typically a cookie that allows a user to use it for subsequent authenticated requests.

We envision that Paranoid-compliant web services should support both existing login flows like email/password combinations or third-party SSO solutions, as well as to support the Paranoid authentication scheme using public key infrastructure (PKI). At the same time, web services should support a mix of both divulged data (i.e. data that was divulged to the service) as well as hidden data (i.e. data that should only be visible to other users through Paranoid).

2.2 Private data overlay in the DOM

A Paranoid user does not have any private data sent to the web service, and as such the service cannot display the private data values in the returned HTML webpage. We would require a Paranoid service to substitute the contents in the HTML elements with a custom `<paranoid>` HTML tag as a placeholder. By default, a custom HTML element is styled as an inline element, much like a `<span>` element, and we foresee that using this would not negatively affect the service's ability to style the placeholder element with normal CSS.

```
<paranoid uid="15" attribute="first_name">
  Hidden Content
</paranoid>
```

Figure 3: Example `<paranoid>` HTML tag.

The `<paranoid>` tag can contain several attributes, but the most important ones are `uid`, which specifies the UID of the user that this placeholder tag is associated with, and `attribute`, which is a canonical name for the private data field that the placeholder is masking. Examples of field names include `first_name` or `email`, or could even be custom defined on a per-service level. The body of the tag itself would typically be a placeholder value displayed if the user does not have the appropriate permissions to view the data value.

The browser extension can simply inspect the webpage for all such `<paranoid>` HTML tags, and replace the inner text of the tag with the data values by matching against the origin of the web page, and the UID and field name of the HTML tag. In other words, we associate the *field tuple*, as defined by $\langle origin, uid, field\_name \rangle$, with a specific private data value, and the browser extension simply enumerates all available data values to find if its field tuple matches that on the `<paranoid>` HTML tag.

2.3 Encapsulation using Shadow DOM

If we naively replace the HTML tags as a simple string replacement, the untrusted web service is able to extract the already-replaced HTML tags to retrieve the private information using client-side JavaScript. A notable example would be British Airways' data breach in 2018 [7], where malicious client-side JavaScript that was injected to the British Airways website was able to siphon credit card information from users when forms were submitted.

Instead, we make use of the the Shadow DOM specification [6], which is supported by most major browsers today. The Shadow DOM specification was developed for the purpose of the Web Component standard, which was designed as a tool to develop component-based (i.e. composition-based) web applications. For example, CSS defined within the shadow DOM would be locally scoped to the shadow DOM, preventing global selectors from affecting elements outside

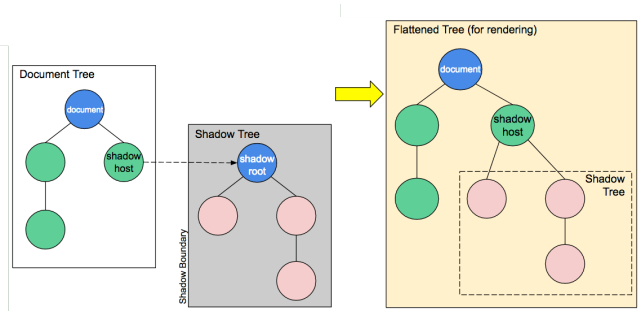

Figure 4: Shadow DOM allows DOM trees (rooted at a *shadow root*) to be attached to the main document tree, providing isolation from external JavaScript.

of the shadow DOM, and JavaScript element queries using `document.querySelector` (for example) would not return elements inside of shadow DOMs. This means that web developers can design components as standalone DOM trees which can be composed inside of other DOM trees without fear of polluting the global CSS and JS namespace. Figure 4 shows how shadow DOM trees can be attached to an existing DOM tree.

We exploited this useful API to encapsulate and hide private data within a shadow DOM. Specifically, we create a closed shadow root, as opposed to an open shadow root which is the default creation mode. This allows us to create a document fragment, or a subtree of the DOM, which cannot be introspected by other JavaScript code running on the same page. This is achieved in client-side JavaScript, by calling `Element.prototype.attachShadow` method on an existing `HTMLElement` on the page, as shown in Figure 5.

```
const span = document.createElement("span");
const shadowRoot = span.attachShadow({
    mode: closed,
});
shadowRoot.innerHTML = "Private Data";
```

Figure 5: Example JavaScript code which attaches a shadow root to a HTML element.

One may wonder if this method is truly secure, as it is also asserted on the Shadow DOM guide [6] that closed shadow roots may be seen by some as an "artificial security feature", since an attacker could trivially overload the `Element.prototype.attachShadow` method with a malicious implementation using JavaScript code executed on the same page.

To get around this, we execute the JavaScript code that calls `Element.prototype.attachShadow` within a content script [1], a feature of Chrome extensions. Content scripts are injected into the context of the current webpage, but executed in isolated JavaScript environments (known as *Isolated*

Worlds). The purpose of Chrome's Isolated Worlds feature allows Chrome extensions to have privileged access to Chrome APIs (such as `chrome.storage`) that should not be permissible by JavaScript running on the web page.

As such, any changes to the JavaScript environment in the web page would not affect the JavaScript environment within the content script of our browser extension, hence making it immune from `attachShadow` overloading attacks. Additionally, because of this the browser extension must be written for a Chromium browser, in order to support the execution of content scripts in isolation from the untrusted web page's JavaScript code.

2.4 Keybase-backed private data storage

We chose to use Keybase [2] as our storage and transport mechanism as it already provides both authenticity and confidentiality guarantees out of the box, as well as other neat features like filesystem abstractions (KBFS) [4] and an end-to-end encrypted message passing mechanism (Keybase Chat) [5].

KBFS is a network-backed filesystem, which can be mounted on local machines using the Filesystem in Userspace (FUSE) API or accessed via a command-line interface. In particular, it provides the `/keybase/private` top-level folder (TLF), where all files are both signed (ensuring integrity) and encrypted (ensuring confidentiality), while files in the `/keybase/public` TLF are only signed by default. This provides the property that all private data, stored in *data files*, will be encrypted at rest, and we can be sure that they have not been tampered with.

Keybase also uses the saltpack [3] message format for encrypting data for multiple users. The message is encrypted using each recipient's private key separately, such that each user would be able to decrypt the message independently using their own key. Furthermore, saltpack allows the public keys of recipients to be hidden in the encrypted message format. We found that this encrypted message format is amenable to storing data files due to the above two properties mentioned.

In the case of Paranoid, we store most metadata in the `/keybase/private` TLF, including information about services and identities, as well as the generated public/private key pair for an identity. On the other hand, we store data files, which can be shared with other users, in the `/keybase/public` TLF, encrypted using the keys of all users who have access to them, which does not expose the private values to anyone who does not have access, and also does not expose the set of users which have access to a given data file.

There is actually no real need to store the Paranoid metadata within the private TLF, since we only need to depend on the public TLF for the purposes of sharing encrypted data files. We could instead store the Paranoid metadata within an encrypted database stored on the user's local computer. However, by colocating all Paranoid data within KBFS, it gives us the nice side-effect of allowing Paranoid data to be synchronized over multiple devices and thus allowing a user to seamlessly use Paranoid on multiple devices, as long as each device is authorized to access the same Keybase account.

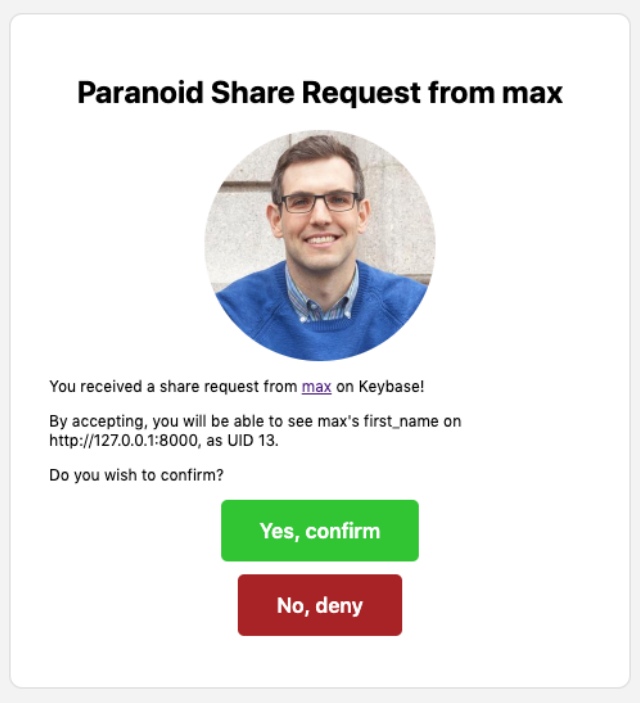

Figure 6: Example share request dialog that requests the user to confirm whether if a field tuple sent by `max` on Keybase should be added to the user's foreign map, to allow placeholder replacements for the specified field tuple. In this case, the field tuple is `(origin="http://127.0.0.1:8000", uid="13", field_name="first_name")`.

In order to not expose the list of services and the corresponding UIDs that a Keybase user has registered for, we use a deterministic hash function (specifically, SHA-256) to hash the field tuple as the filename for the data file. This still allows recipients to access the data file in the owner's `/keybase/public` TLF as long as they keep track of the list of data files that they have access to.

2.5 Share requests

KBFS also does not provide a push-like synchronization mechanism (a la Dropbox), but instead clients must pull files on demand. While this still allows a recipient to be aware of any changes to a data file that they had previously been granted access to by pulling the data file again from the owner's public TLF, we need to construct a way to notify the recipient that they have been granted access to a new data file.

As such, we introduce the notion of a share request. A

screenshot of an example share request in the browser can be seen in Figure 6. The owner *A* chooses to grant access of a data file to a recipient *B*, and as such re-encrypts the data file using *B*'s public key in addition to all previous users' public keys, which immediately allows *B* to decrypt the data file.

To notify *B* of a data file that is now made accessible to him, we send a Keybase Chat message originating from *A* to *B*, of a share request that *A* wants to send to *B*. *B* can then accept the share request, which would keep track of the field tuple in its private storage's *foreign map*, and thus allowing the browser extension to now replace tags corresponding to *A* on the service's web page.

3 Security overview

3.1 Threat model

3.2 Encryption

We make use of Keybase to provide encryption at rest of all private data files and metadata. One other alternative we considered was to simply store the data within HTML5 `localStorage` within the browser extension, but we found that it was implemented as an unencrypted database on the filesystem as an IndexedDB file. This database is accessible to all processes on the machine, even after the machine is shut off. On the other hand, all files in KBFS are stored encrypted, and require the user key to decrypt them.

As such, no keys or data files are stored in the browser such that it can be accessible by JavaScript, and instead all secrets must be decrypted and transmitted from the daemon server to the browser extension over HTTPS.

3.3 Potential attacks

3.3.1 JavaScript Prototype Poisoning

As described in §2.3, malicious JavaScript code might poison the prototype function for `Element.prototype.attachShadow`, overriding it with an implementation that still allows the web page's JavaScript to access the DOM elements despite creating a shadow root. The malicious implementation could simply ignore the `mode` option in the `attachShadow` arguments, which makes the resultant shadow root default to an open mode.

However, the use of content scripts in Chrome's *Isolated Worlds* feature [1] makes the browser extension immune to such prototype poisoning vulnerabilities.

3.3.2 Cross-Site Request Forgery (CSRF)

The daemon server sends all data files unencrypted to the browser extension, and thus it is crucial to lockdown access to the daemon server as much as possible. Furthermore, the browser extension has to be able to access the daemon server from the context of any origin, since it operates on the service's web page itself, and thus we have to enable Cross-Origin Resource Sharing (CORS) for all origins on the daemon server (i.e. the server sends the header `Access-Control-Allow-Origin: *`). This opens up the possibility for the web service to execute malicious JavaScript on the client side, to make unauthorized requests to the daemon server to exfiltrate secrets, such as private keys or private data files, using Cross-Site Request Forgery (CSRF).

While the typical CSRF protection usually involves browser enforcement of CORS or using CSRF tokens, this does not apply in our case, since requests have to be made automatically when a page is loaded for an untrusted origin. We realised the key idea is to perform authentication such that only the browser extension is able to make the request, and not arbitrary JavaScript on the page.

We thus make use of a session token to authenticate all HTTPS clients to the daemon server. This session token has to be made known to the browser extension before Paranoid is used. The browser extension sends a HTTP 'Authorization' header, which the daemon server uses to authenticate all incoming HTTP requests, and reject the request if the session token does not match.

This session token is stored within the extension's `localStorage` IndexedDB location, instead of a per-origin `localStorage`. Although unencrypted, we can ensure that untrusted JavaScript on third-party web pages do not have access to the session token to open up the system to CSRF.

3.3.3 Man-in-the-middle (MITM)

To prevent MITM attacks, the communication between the browser extension and the daemon server should be encrypted over TLS. This prevents network adversaries from snooping on plaintext private data, or to intercept and modify data that is sent to the browser extension.

Furthermore, we also want to minimize other possible attacks like DNS hijacking, and thus the daemon server's hostname should be resolved locally, preferably within the `/etc/hosts` file.

3.3.4 Impersonation attacks

Using our implementation, it is also possible for a user to impersonate another user's service identity. Suppose Alice is associated with UID 13 on the service `origin.com`. However, Harry sends Bob a share request claiming to be associated with UID 13, and if Bob accepts the share request, Bob would mistakenly think that user 13 on `origin.com` is Harry, and thus making Bob think that all content on the service's site associated with UID 13 belongs to Harry.

One may think we could simply use Keybase's mechanism of proofs to resolve this. However, we must remember that identities hidden via Paranoid are *intentionally* made anony-

mous to the public, which puts it at odds with the goals of Keybase identity proofs. As such, Alice's Keybase user account cannot be publishing a signed identity proof claiming to be UID 13 on `origin.com`, as this would give away her identity that she wanted to keep secret from the world.

One initially plausible solution would be for the service itself to publish Alice's public key, allowing Bob to prove the Alice's identity by requesting the user to solve a challenge signed with the public key. However, it is also entirely possible for the service to lie about a user's public key, since we do not trust the service.

As such, we see that the only viable solution would be to defer the responsibility of proof to a trusted platform. For example, we could modify the SSO flow such that when Alice registers on a service with her public key and receives a UID of 13, the service must publish this $\langle public\_key, uid \rangle$ association onto a distributed ledger that we all trust. If a user wants to send a share request to Bob claiming to be user 13, Bob can then look up the ledger to retrieve the associated public key with the UID 13, and send the user a challenge encrypted with the associated public key, in order for the user to prove ownership of the user account with UID 13, i.e. whether the user is indeed Alice.

Note that we must use a ledger so that adversaries cannot rewrite previously published associations. We did not implement this in our implementation, but we see how we could easily extend the current implementation to make use of a ledger for proving claims for a UID.

4 Implementation

4.1 Components

4.1.1 Browser extension

4.1.2 Daemon server

4.1.3 Sample Paranoid-compliant web service

4.2 Challenges faced

Since we are storing all data on KBFS, we quickly found that requests made to the daemon server were constantly taking over 2 seconds to complete. This is because a request may require many `read(2)` and `getdents(2)` calls of several different files and directories within KBFS. Considering that the private TLF is fully encrypted, and data files in the public TLF are also encrypted, much of the overhead may stem from the fact that we have to repeatedly perform decryption on each request.

We managed to improve the response time of the daemon server by implementing an in-memory cache that stores all decrypted metadata and data files, and updating the cache only when an update is made. This drastically improved the average response time of the server. In one of our tests, the response for a particular endpoint to return a single service identity's metadata and data files saw a 130x improvement (3703ms uncached vs 27ms cached).

5 Evaluation

References

[1] Content Scripts. https://developer.chrome.com/extensions/content_scripts.

[2] Keybase. https://keybase.io/.

[3] saltpack - a modern crypto messaging format. https://saltpack.org/encryption-format-v2.

[4] Understanding the Keybase filesystem. https://keybase.io/docs/kbfs/understanding_kbfs.

[5] Introducing Keybase Chat. https://keybase.io/blog/keybase-chat, 2017.

[6] BIDELMAN, E. Shadow DOM v1: Self-Contained Web Components. https://developers.google.com/web/fundamentals/web-components/shadowdom.

[7] NG, A. That British Airways breach shows hackers fine-tuning e-commerce attacks. https://www.cnet.com/news/heres-how-british-airways-was-hacked-according-to-researchers/, September 2018.

[8] O'BRIEN, S. A. Equifax data breach: 143 million people could be affected. https://money.cnn.com/2017/09/07/technology/business/equifax-data-breach/index.html, 2017.

[9] SHEERA, F. Facebook Bug Changed Privacy Settings of Up to 14 Million Users. https://www.nytimes.com/2018/06/07/technology/facebook-privacy-bug.html, 2018.

[10] YANG, J., HANCE, T., AUSTIN, T. H., SOLAR-LEZAMA, A., FLANAGAN, C., AND CHONG, S. Precise, Dynamic Information Flow for Database-backed Applications. *SIGPLAN Not. 51*, 6 (June 2016), 631–647.

[11] YIP, A., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Improving Application Security with Data Flow Assertions. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 291–304.

[12] ZELDOVICH, N., BOYD-WICKIZER, S., AND MAZIÈRES, D. Securing Distributed Systems with Information Flow Control. In *Proceedings of the 5th*

USENIX Symposium on Networked Systems Design and Implementation (Berkeley, CA, USA, 2008), NSDI'08, USENIX Association, pp. 293–308.