redact

Alexandre Pauwels, Hugh Whelan

Oct 27, 2021

INTRODUCTION

CHAPTER

ONE

GETTING STARTED

Broadly speaking, the steps to get started with Redact are as follows:

1. Get access to a redact-store instance
2. Install the redact-client locally on a device
3. Point your browser to a Redact-enabled website

1.1 Setup Redact-store

The storage service can be either self-hosted or provided by a third-party. Since it only stores encrypted data, the provider of the storage service does not need to be trusted, but should provide a reasonable level of protection against unauthorized requests.

A self-hosted storage is fairly easy to setup, and primarily involves procuring a database (currently only MongoDB is supported) and standing up the redact-store server to connect to it.

A third-party storage will simply provide a URL for the *Client* to connect to.

1.1.1 Self-hosted Storage

1. Get access to a MongoDB instance
   - Sign up for a free, fully-managed instance at mongodb.com (easy, quick)
   - Set up an instance on your local device or host your own instance (harder, time-consuming, more customizable)
   - Minimum supported MongoDB version is 3.6+
   - If running MongoDB locally and the storage cannot connect, try using 127.0.0.1 in the connection string instead of a hostname
2. Install Rust: https://www.rust-lang.org/tools/install
3. `git clone https://github.com/pauwels-labs/redact-store.git`
4. `echo "export REDACT_DB_URL=\"<mongo connection string>\"" >> config/config.env`
5. `echo "export REDACT_DB_NAME=\"<db name>\"" >> config/config.env`
6. `source config/config.env`
7. `cargo r`

The port and address listened on by the storage server will be provided to the client.

Support multimedia (e.g. images, video)

Redact leverages object storage services in order to store large chunks of data that are unlikely or difficult to fit in a traditional database.

Currently, the only supported object storage is a Google Cloud Storage bucket.

In order to add this functionality to your storer, do the following:

1. Sign up for Google Cloud and provision a Google Cloud Storage bucket
2. Go to the Permissions tab of the bucket details page and add a new principal with Storage Legacy Bucket Writer permissions
3. Go to the IAM & Admin section of Google Cloud and click on Service Accounts
4. Click on Keys > Create new key, and create a new JSON key
5. Download the key and save it to a safe place on your computer
6. `echo "export SERVICE_ACCOUNT=\"<path to file downloaded>\"" >> config/config.env`
7. `echo "export REDACT_GOOGLE_STORAGE_BUCKET_NAME=\"<bucket name>\"" >> config/config.env`
8. `source config/config.env`
9. `cargo r`

1.2 Install redact-client

1. `git clone https://github.com/pauwels-labs/redact-client.git`
2. Provide the *Storage* URL in `config/config.yaml#storage.url`
   - If you set-up your own storage server using the steps above, the URL will likely be https://localhost:8081
   - If you set-up your own storage server, notice that the config allows for specifying a custom server CA certificate at `storage.tls.server.ca.filepath`. You MUST copy the CA certificate generated by the storage server to this location. Within the redact-store directory, it should be located at `tls/server/cert/ca.pem`, copy this file to the client at `certs/storer-ca.pem`.

4. `cargo r`

1.3 Visit Redact-enabled website

We have an example website that allows you to demo Redact's current feature-set called Redact Feed, which allows you to post text and multimedia and see those posts displayed. Redact Feed will soon support data sharing and other social features.

Once the *Client* is setup locally and points to a working storage instance, Redact-enabled websites will "just work" (TM). The *Client* handles generation and coordination of cryptographic material with no further input.

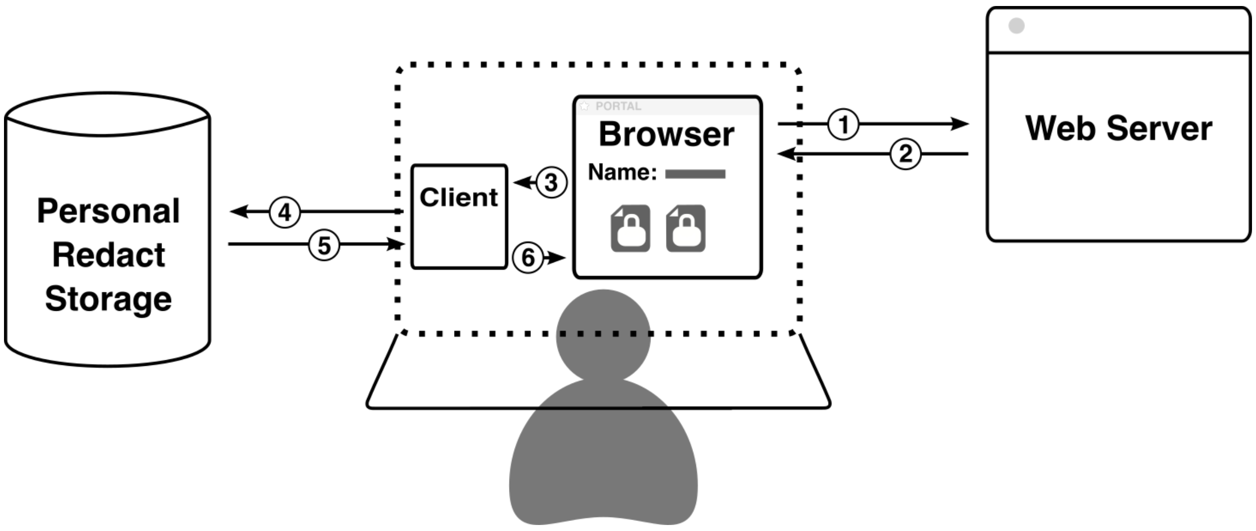
Warning: Redact currently only supports storing keys unencrypted on the file system. Support for hardware and software key vaults is upcoming.

CHAPTER

TWO

OVERVIEW

The best way to begin understanding how Redact works is to understand, at a high-level, what the general flow of information is when a user visits a Redact-enabled website. Below is a diagram outlining this flow.



1. Alice opens her browser and visits a Redact-enabled website (e.g. www.foo.com).
2. The website's server responds with the contents of the page, which include `<iframe>` elements for instances where the website displays "Redacted" data.
3. Alice's browser issues a request to the local *Client* for each iframe, each of which represents a different piece of secured data.
4. The *Client* receives each request and forwards them to the *Storage*.
5. The *Storage* instance responds to each request with the data it has stored at that path. These pieces of data are typically encrypted and cannot be decrypted by the *Storage* instance itself.
6. The *Client* decrypts the encrypted data, then responds to the local request from the browser with the decrypted text in a secure iframe. The user can now see the secure data, but the host website cannot.

2.1 Secure `<iframe>`

Although there already exist several end-to-end encrypted applications in use commercially today, none of them quite solve the problem of endpoint security. These services provide strong security both in transit and at rest using their encryption protocols, but there still remains the issue of displaying the actual data to the user: data must be decrypted and placed on the screen at some point, and existing end-to-end encryption applications maintain control over this process. This means there is a point in the data pipeline where the user must trust the operator of the service not to inject malicious code and exfiltrate plaintext data.

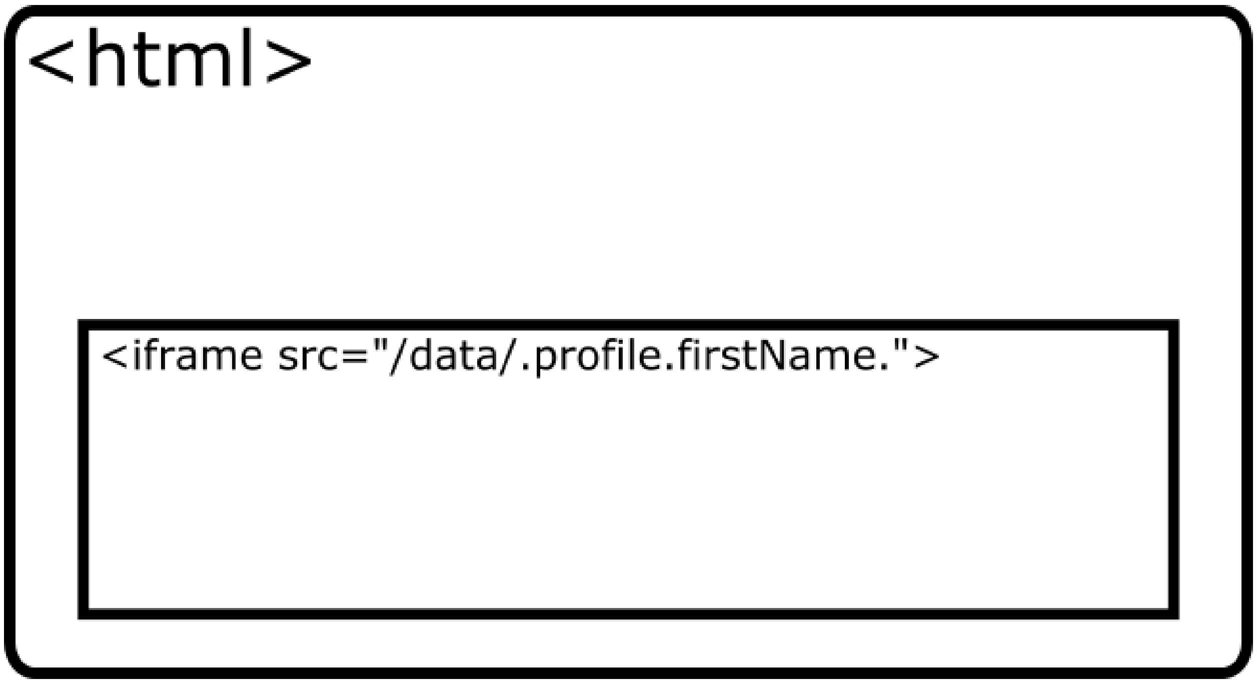
Redact solves this issue by limiting the trusted platform to just the browser, rather than all the websites visited on the browser. The *Client* leverages `<iframe>` elements and CSRF tokens to allow websites to easily interact with protected data without having to, or in fact being able to, actually see it at any point. This technique works using technologies that are well-proven and decades old, and requires no Javascript.

The general flow to request data to be displayed works as such:

1. The website places an `<iframe>` element as a placeholder for the data:

   ```
   <iframe src="/unsecure/data/.profile.firstName."></iframe>
   ```

   There are additional parameters that can be provided which further customize the *Client*'s response, but are left out here for simplicity.



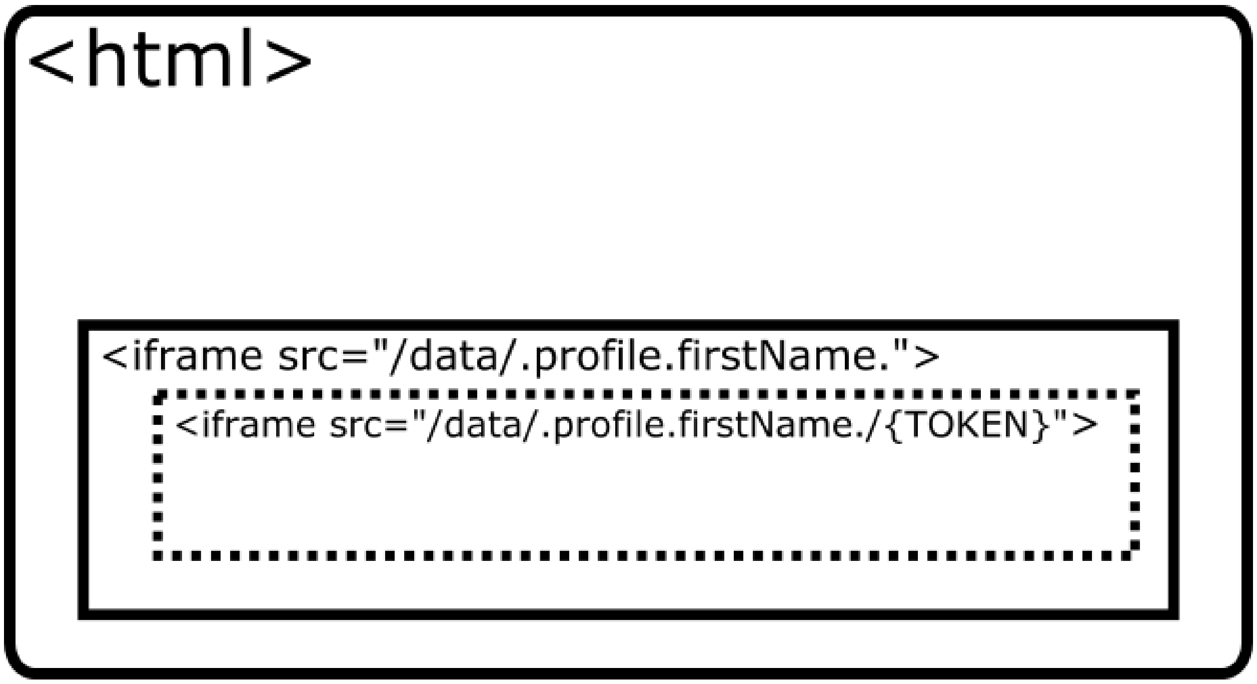
2. The *Client* receives the request and does four things:

   1. Generates a random 256-bit session token.
   2. Attaches a session to the request.
   3. Stores the token in the session.
   4. Responds to the request with a session cookie containing the ID of the request's session and an HTML page containing another `<iframe>`, attaching the token generated in step 1 to the end of the URL path, like this:

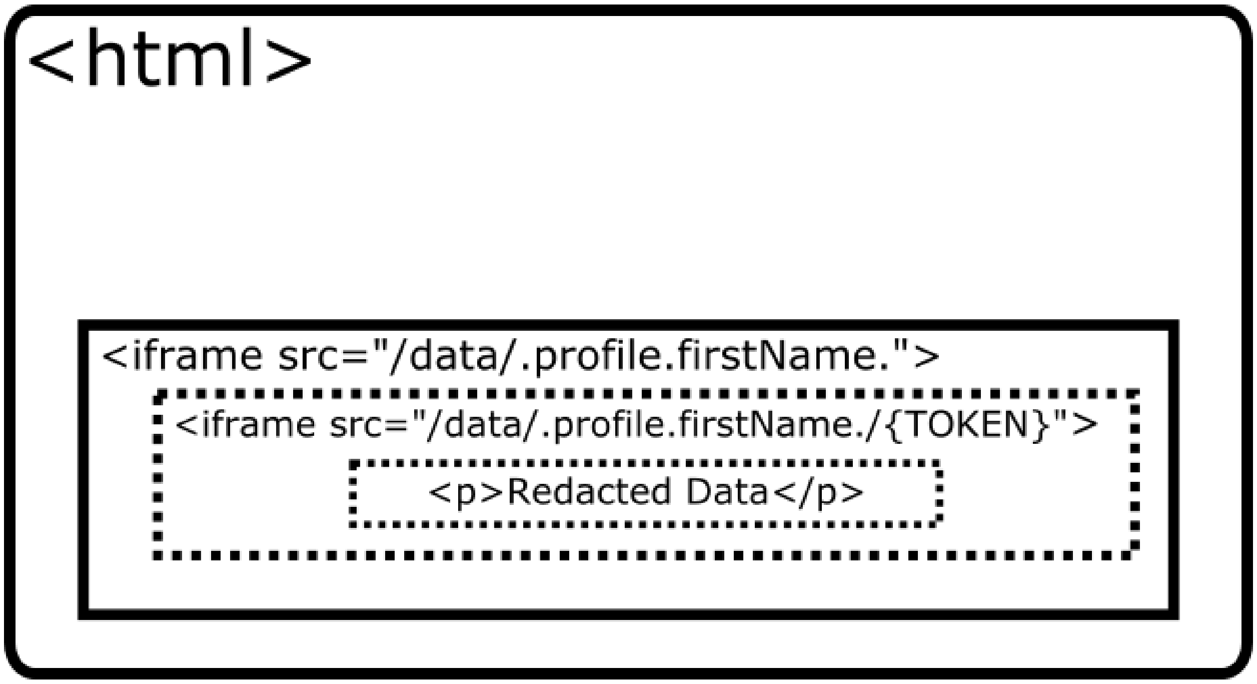
   ```
   <iframe src="/secure/data/.profile.firstName./
   ↪498DF68A39A51DE648799EE13CD26D2163863FC5F43814B8895B78BBA45935A0"></iframe>
   ```

(continues on next page)

(continued from previous page)



3. Upon rendering the iframe response, the browser makes another request to the *Client*, once again according to the `src` of the inner `<iframe>`. The *Client* receives the request and attempts to respond with the decrypted data:
   1. Retrieves the session based on the session ID provided in the request cookie header.
   2. Compare the token in the URL of the request with the token stored in the session.
   3. If the tokens match, it proceeds with fetching the requested data, decrypting it, and responding with an HTML page containing the plaintext. If the tokens do not match, the request is rejected.


<html>
<iframe src="/data/.profile.firstName.">
<iframe src="/data/.profile.firstName./{TOKEN}">
<p>Redacted Data</p>

This process allows the *Client* to ensure that the only time it responds with plaintext data is when the request for the data is coming from itself.

Imagine that a malicious website, acme.com, would like to exfiltrate Alice's redacted data when she visits. In order to do so, they place a script on `acme.com` that runs when Alice visits the website and makes an AJAX request to the Alice's *Client* for data at the path `.profile.firstName.`. This request succeeds and responds with a 200 status code, but the returned HTML just contains an `<iframe>` element, this time with a token appended to the end of the `src` attribute. The script then makes a second AJAX request with the token appended at the end this time.

This request will fail. It isn't enough to provide the correct token at the end of the request path, that request must also be paired with a session that contains the same token. This pairing occurs by attaching the session ID returned by the *Client* in the first request as a cookie header in the second request. Thanks to all modern browsers' cross-origin resource sharing (CORS) protections, it is impossible for a website located at acme.com to fetch the session ID set by a website not at the same domain.

Warning: Cookies set by a domain other than the primary domain being visited are classified as "third-party cookies" by browser vendors. Browsers are increasingly placing limitations on these cookies, the latest being that such cookies MUST be transferred over a TLS connection. Currently the connection between browser and *Client*, both on the user's local device, is not secured by TLS. In the future, it may be necessary to add a self-signed certificate generated by the *Client* to the browser in order to secure that connection.

CHAPTER

THREE

CLIENT

The purpose of the *Client* is to host a small server on the user's device (e.g. phone, laptop) to respond to requests for private data. It listens on a local port, currently defaulted to `::8080`, and responds to these requests with a secure `<iframe>` buffer that displays the data without allowing the website to read it back to itself.

Note: For more information on how the secure `<iframe>` buffers work see *Secure <iframe>*.

3.1 API

3.1.1 GET /unsecure/data/<path>

An unsecure request to the client for data at a given path. The response is not the data itself, but an HTML document with an iframe which makes a secure request to retrieve the data.

Path Parameters

Parameter	Required?	Description	Example
`path`	Required	A jsonpath-style string prepended and appended by a period which represents the path of the data	`.someObj.someVal.`

Query Parameters

Parameter	Required?	Description	Default	Example
`css`	Optional	A URL-encoded CSS block meant to style the displayed data. The HTML to style can be found here.		`iframe%7Bborder%3Anone%3B%7D`
`edit`	Optional	A boolean indicating whether the data will be displayed in an editable form field. If `true`, the value will be displayed in a submittable input box appropriate for its data type.	`false`	`true`
`data_type`	Required	specifies the type of data to expect; this is particularly useful when creating new data that does not yet have a type. The value can be one of: • `Bool` • `U64` • `I64` • `F64` • `String` • **`Media`** – **A Binary file that can be rendered in the browser. Currently supported file types are:** ∗ `image/jpeg` ∗ `image/png` ∗ `image/gif` ∗ `image/apng` ∗ `image/avif` ∗ `image/svg+xml` ∗ `image/webp` ∗ `video/mpeg` ∗ `video/mp4`		`String`
`relay_url`	Optional	the URL endpoint to which a `POST` HTTP request will be sent upon submission of editable data. This would typically be a URL controlled by the host of the Redact-enabled website and used for internal bookkeeping.		`https://foo.com/redact/relay`
`js_message`	Optional	A base64-encoded and URL-encoded message which the editable Redact field will send to the parent page after data is successfully submitted. Refer to *JS Messaging* for more details.		`Y3JlYXRlZA%3D%3D`
`js_height_msg_prefix`	Optional	A base64-encoded and URL-encoded message which a displayed Redact field will prepend to the pixel height of the rendered data, then send to the parent page. This can be used to dynamically adjust the height of a redact iframe on a web page based on the size of the rendered data.		`aGVpZ2h0Oi5hYmMuOg%3D%3D`

3.1.2 `GET /secure/data/<path>/<token>`

An secure request to the client for data at a given path. The response is and HTML document displaying the contents of the data.

Header Parameters

Header Name	Required?	Description
`Cookie sid`	Required	The session ID is used internally by the *Client* to associate the request with a session in its session store.

Path Parameters

Parameter	Required?	Description	Example
`path`	Required	A jsonpath-style string prepended and appended by a period which represents the path of the data	`.someObj.someVal.`
`token`	Required	A random, 256-bit, upper-case alphanumeric CSRF token that is generated and used internally by the *Client*	

Query Parameters

Note: These query parameters are identical to those of `GET /data/<path>` and are typically automatically included in this request by the *Client*.

Parameter	Required?	Description	Default	Example
`css`	Optional	A URL-encoded CSS block meant to style the displayed data. The HTML to style can be found here.		`iframe%7Bborder%3Anone%3B%7D`
`edit`	Optional	A boolean indicating whether the data will be displayed in an editable form field. If `true`, the value will be displayed in a submittable input box appropriate for its data type.	`false`	`true`
`data_type`	Required	specifies the type of data to expect; this is particularly useful when creating new data that does not yet have a type. The value can be one of: • `Bool` • `U64` • `I64` • `F64` • `String` • **`Media`** – **A Binary file that can be rendered in the browser. Currently supported file types are:** ∗ `image/jpeg` ∗ `image/png` ∗ `image/gif` ∗ `image/apng` ∗ `image/avif` ∗ `image/svg+xml` ∗ `image/webp` ∗ `video/mpeg` ∗ `video/mp4`		`String`
`relay_url`	Optional	the URL endpoint to which a `POST` HTTP request will be sent upon submission of editable data. This would typically be a URL controlled by the host of the Redact-enabled website and used for internal bookkeeping.		`https://foo.com/redact/relay`
`js_message`	Optional	A base64-encoded and URL-encoded message which the editable Redact field will send to the parent page after data is successfully submitted. Refer to *JS Messaging* for more details.		`Y3JlYXRlZA%3D%3D`

3.1.3 `POST /secure/data/<token>`

A secure request to the client to update existing data or create new data at a given path.

Header Parameters

Header Name	Required?	Description
`Cookie sid`	Required	The session ID is used internally by the *Client* to associate the request with a session in its session store.
`Content-Type`	Required	• `x-www-form-urlencoded`: For types other than `Media` • `multipart/form-data`: For the `Media` data type.

Path Parameters

Parameter	Required?	Description	Example
`token`	Required	A random, 256-bit, upper-case alphanumeric CSRF token that is generated and used internally by the *Client*	

Query Parameters

Parameter	Required?	Description	Default	Example
`css`	Optional	A URL-encoded CSS block meant to style the displayed data. The HTML to style can be found here.		`iframe%7Bborder%3Anone%3B%7D`
`edit`	Optional	A boolean indicating whether the data will be displayed in an editable form field. If `true`, the value will be displayed in a submittable input box appropriate for its data type.	`false`	`true`
`relay_url`	Optional	the URL endpoint to which a `POST` HTTP request will be sent upon submission of editable data. This would typically be a URL controlled by the host of the Redact-enabled website and used for internal bookkeeping.		`https://foo.com/redact/relay`

Body Parameters

Parameter	Required?	Description	Example
`path`	Required	a jsonpath-style string prepended and appended by a period	`.someObj.someVal.`
`value`	Required	The value of the data being submitted	`String`
`value_type`	Required	Specifies the type of data to expect; this is particularly useful when creating new data that does not yet have a type. The value can be one of: • `Bool` • `U64` • `I64` • `F64` • `String` • **`Media`** – **A Binary file that can be rendered in the browser. Currently supported file types ar** ∗ `image/jpeg` ∗ `image/png` ∗ `image/gif` ∗ `image/apng` ∗ `image/avif` ∗ `image/svg+xml` ∗ `image/webp` ∗ `video/mpeg` ∗ `video/mp4`	`String`

3.1.4 `POST /proxy`

Retrieves the response of a GET request to a given URL, which is made via the client with mutual TLS. The root domain of the URL requested must match the root domain of the request's `Origin` header value. For more information on how to use the Proxy API, see *User Sessions*.

Header Parameters

Header Name	Required?	Description
`Origin`	Required	
`Content-Type`	Required	Must be: `application/json`

Body Parameters

Parameter	Required?	Description	Example
`host_url`	Required	The URL to which to make a GET request	`https://foo.com/redact/session_create`

CHAPTER

FOUR

STORAGE

The *Storage*'s purpose is to provide a public interface for the *Client* to perform CRUD operations on encrypted data. It provides a stable, public API, along with an authentication and authorization layer that allows clients to request or modify the stored data.

An important note is that the owner of the *Storage* server does not need to be trusted. The *Client* encrypts stored data before sending it to storage, meaning that the storage server only ever handles ciphertexts (unless the *Client* is purposefully storing public, plaintext information). This allows operation of a multi-tenant storage service to be delegated to a third-party, reducing the burden on users.

> **Warning:** Currently, the storage interface is only implemented for MongoDB. In the future, other database types will be supported.

4.1 API

4.1.1 `GET /<path>`

Retrieve data at a given path.

Path Parameters

Parameter	Required?	Description	Example
`path`	Required	A jsonpath-style string prepended and appended by a period which represents the path of the data	`.someObj.someVal.`

4.1.2 `POST /`

Upsert data at a given path.

Header Parameters

Header Name	Required?	Description
`Content-Type`	Required	Must be: `application/json`

Body Parameters

The body of the POST request should be a JSON-serialized `Entry` struct. The definition of an `Entry` can be found here.

CHAPTER

FIVE

CRYPTOGRAPHY

In order to secure data and provide authentication and authorization services, Redact leverages several cryptographic systems. Asymmetric keys are used to create unique identities that can be used to identify users or machines across the Redact ecosystem. Symmetric keys are used to encrypt and decrypt user data. By combining these two concepts, Redact can create portable encrypted data that can be consumed by any device owned by a particular user.

Note: All cryptographic algorithms are currently provided by libsodium. Symmetric keys use its xsalsa20poly1305 algorithm. Asymmetric keys use its curve25519xsalsa20poly1305 algorithm.

In the future, other backing cryptographic libraries and algorithms will be supported.

5.1 Identity

Core to the Redact architecture is the concept of identity. In order for a user to give or deny access to a particular piece of data, they must at all times be capable of identifying who is requesting their data and what permissions this user has. In order to avoid centralization and dependance on a managed service to keep track of who has access to what, Redact uses self-sovereign identities in the form of asymmetric keypairs.

At the most basic level, all users have their own keypair with a self-signed client certificate. They can use this keypair and certificate to make anonymous requests to services that support anonymity. These requests are made using mutual TLS to share the identity of the user. For example, when the *Client* requests data from a storage server, or relays a message to a website host, it does so using this client certificate.

In many cases, however, a service may require some form of further authentication. For example, an internal company website may require that only employees of the company be capable of viewing the site. To indicate that the user belongs to a particular group, the keypair can be signed by a different certificate authority. This certificate can also have custom metadata added that can further identify and authorize the user.

5.2 Encryption

Redact leverages encryption to minimize the amount of time and places that your data is available in plaintext. Its encryption framework relies on two crucial invariants:

1. Private data is only decrypted by the *Client*.
2. Private data never exits the *Client* in plaintext.

By following these two rules, Redact ensures that the attack surface for exfiltrating private data remains as small as possible.

CHAPTER

SIX

BUILDING A WEBSITE WITH REDACT

This section describes in detail how to build a simple website with redacted data. The Redact codebase is currently in alpha, and supports displaying and storing strings, numbers, and booleans. Sharing data between Redact users is under development and planned for a future release, but is included in this document because it is a crucial component not only for Redact, but also for Redact-enabled websites.

6.1 Static Web Page

This tutorial begins with a web page that statically displays redacted user data. The website will be a contact page where a user can enter their name, phone number, and social security number, all of which are stored in an encrypted form by Redact. This is admittedly useless as a user can only view their own contact info for now, but it serves as a good example before diving into more advanced topics.

To begin, create an HTML document that contains an iframe pointing to the Redact client.:

```
<iframe src="http://localhost:8080/unsecure/data/.demoapp.name."></iframe>
```

Start the *Client* and *Storage* on your device and open the HTML document in a browser. Examine the `<iframe>` HTML contents and you should see that it has been populated by the client:

```
<iframe src="http://localhost:8080/unsecure/data/.demoapp.name.">    <!-- The <iframe>␣
↪from the source HTML document -->
    <html>    <!-- The response from the Client for the unauthenticated request -->
        ...
        <iframe id="data-frame" src="" title="secure">
            <html>    <!-- the response from the Client for the authenticated request -->
                ...
                <p></p>
                ...
        </iframe>
        ...
    </html>
</iframe>
```

Note: the *src* of the inner iframe is added after page load using javascript. This is a workaround for a iframe caching bug in firefox: https://bugzilla.mozilla.org/show_bug.cgi?id=354176

The data requested at the path `.demoapp.name.` does not yet exist, so go ahead and create a form field so the user can add their name. Modify the iframe `src` to retrieve an editable form field for `.demoapp.name.`:

```
<iframe src="http://localhost:8080/unsecure/data/.demoapp.name.?edit=true"></iframe>
```

`edit=true` has been added as a query parameter to the Redact request, which requests an editable field from the client. The form field is pre-populated with the existing data at the given path, and the HTML input type matches the data-type of the stored item. The submit button triggers a secure POST request to the client, which then updates the data in the *Storage*.

The form field is not visually appealing. It shows the iframes as two large bordered boxes, but the appearance can be modified by passing in custom CSS to the Redact request and styling the outer iframe. Add a CSS stylesheet to the HTML page and style the outer iframe to have no border:

```
iframe {
    border: none;
    height: 66px;
    width: 500px;
}
```

Next, add a CSS query parameter to the Redact request to instruct the Client to apply the CSS to the response:

```
<iframe src="http://localhost:8080/unsecure/data/.demoapp.name.?edit=true&css=iframe
↪{border:none;height:50px}"></iframe>
```

Now, simply add a few more form fields to represent the user's phone number and social security number, and add labels to the page so the user knows what each field represents.:

```
<html>
    <body>
            <p>Name:</p>
            <iframe src="http://localhost:8080/unsecure/data/.demoapp.name?edit=true&
↪css=iframe{border:none;height:50px;}"></iframe>

            <p>Phone Number:</p>
            <iframe src="http://localhost:8080/unsecure/data/.demoapp.phonenumber?
↪edit=true&css=iframe{border:none;height:50px;}"></iframe>

            <p>Social Security Number:</p>
            <iframe src="http://localhost:8080/unsecure/data/.demoapp.
↪socialsecuritynumber?edit=true&css=iframe{border:none;height:50px;}"></iframe>
    </body>
</html>
```

This is a simple HTML page which loads user data from the given paths, and allows the user to edit and update this data on their *Storage*. To create a non-editable, display only version of the page, copy the contents of the existing HTML page, and remove the `edit=true` query parameter from each iframe. Add a link or button which directs the user from the view-only page to the editable page, and vice-versa.

6.2 Modern Web Application

Modern web applications use javascript to respond to user actions and modify the page, and a backend server which responds to HTTP requests for data retrieval and updates. Because Redact data is stored and operated on in a manner which is opaque to the website it is displayed on, the flow of data must be modified to provide a web application the information it needs on the frontend (javascript) as well as on the backend (HTTP server).

Imagine a website that presents an *alert* to a user when they submit data on a form. Normally, the submit button could have an event listener to do this.:

```
<button onclick="alert('Form Submitted')">Submit</button>
```

If the submit button is within a Redact iframe, the web page does not have access to the `<button>` element, and cannot add an event listener in this manner. To solve this limitation, Redact uses JS messaging to securely inform a parent web page that changes have been made to a Redact data field.

To understand how Redact communicates with backend HTTP servers, imagine a traditional website that maintains a list of entries made by the user. A form field allows the user to create a new entry, which will be sent to the server on submission. The entry will then be added to a database, and will be retrieved from the database whenever the user loads their list of entries. With Redact, data entries cannot be directly sent to the HTTP server. They are instead sent to the *Client*, which encrypts them and stores them in the *Storage*. In order to support backend server functionality Redact uses "data relays". Data relays instruct the *Client* to securely send information about a data entry's Redact path to an arbitrary HTTP server.

6.2.1 JS Messaging

The JS Messaging features allows a redacted form field to emit information to the parent page when data within Redact is updated via the page. JS messaging utilizes the postMessage() API. Passing in a `js_message` query parameter to a *Client* request instructs an editable Redact field to send a message to the parent page after data is successfully updated. The contents of the message are simply the value of the `js_message` query parameter.

Note: The `js_message` query parameter must be base64-encoded and URL-encoded.

To accomplish something with the same effect as the following HTML code within a Redact iframe, the `js_message` query parameter must be used. As an example, consider how a traditional website would trigger an alert on submission of a form using events and javascript:

```
<button onclick="alert('Form Submitted')">Submit</button>
```

Use the `js_message` query parameter when retrieving an editable field:

```
<iframe src="http://localhost:8080/unsecure/data/.demoapp.name.?edit=true&js_
↪message=c3VibWl0"></iframe>
```

The *Client* response will contain a form which posts the message `"submit"` to the parent page when the submit button is clicked. Listen for the `window:message` event to trigger the alert javascript:

```
window.onmessage = (event) => {
    try {
        decodedMessage = atob(event.data);
        if (decodedMessage === 'submit') {
            alert("Form Submitted");
```

(continues on next page)

(continued from previous page)

```
        }
    } catch (error) {
        // expected when the event.message is not a base64 encoded string
    }
};
```

6.2.2 Data Relaying

Data relaying allows a Redact-enabled website to link an API endpoint to a Redact data field such that the API endpoint receives a HTTP POST request containing the path of the data when an edit is submitted. When the data at a given path is created or updated, the client orchestrates a request to the endpoint at the given relay URL. If the request succeeds, it is transparent to the user. If the request fails, an error is presented to the user to inform them that, although the data in their *Storage* was updated, the action was not entirely successfully as the backing server did not acknowledge the relay.

To configure a Redact relay, add an endpoint to the backend HTTP server which will receive POST requests with a JSON body representing the path of the data that was updated. A user's *Client* will send an HTTP POST request with a request body in the form:

```
{
    "path": "<DATA PATH>"
}
```

Next, add the relay_url query parameter to the Redact client request within an iframe:

```
<iframe src="http://localhost:8080/unsecure/data/.demoapp.name.?edit=true&relay_url=https
↪%3A%2F%2Ffoo.bar%2Fredact%2Frelay"></iframe>
```

When this data is submitted, a POST request will be made to `https://foo.bar/redact/relay` with the JSON body:

```
{
    "path": ".demoapp.name."
}
```

Note how the request does not have any information identifying a user. Redact users identify themselves using certificates, and relays are no different. The recommended approach for differentiating between users is to establish a mutual TLS connection with incoming relay requests and use attributes of the client cert to identify the user. In Redact, a user can have multiple devices each with a separate key, all signed by the user's key. Therefore, to identify the user use the value of the certificate's Authority Key Identifier. This will uniquely identify the user across multiple devices. For more information on how cryptography is used in Redact, see Cryptography.

6.2.3 User Sessions

Data relays allow a backend server to identify which user is updating their data on a page, but this is not very useful if a website cannot identify which user is visiting the page and making non-relay requests to the server (for example, to retrieve all Redact data entry paths that have been relayed for a given user). The server needs an authenticated method by which to identify a user. This is where user sessions come in handy. They provide a way for a Redact-enabled website to make HTTP requests on behalf of a user identified by their certificate.

User sessions provide a JWT token for a website's frontend to be passed along with HTTP requests to the backend server. The JWT tokens are generated and signed by the website's own backend server upon establishing a mutual TLS connection with the *Client*. This way, the server can validate that a request coming from the UI is coming from the same user that established a mutual TLS between their client and the server on the same device.

This approach to sessions with Redact utilizes the *Client's proxy endpoint*. The proxy endpoint accepts requests directly to the client (as opposed to all other requests which must be requested from within an iframe), and forwards the request as a GET request to a given endpoint. This request is optionally performed with mutual TLS, allowing the given endpoint to uniquely identify the user. The response from the endpoint is then passed back as the response to the proxy request. By responding to this request with a signed JWT token that contains the information needed to identify a user (such as the Authority Key Identifier), the server can verify that subsequent requests with the JWT token are being made on behalf of the same Redact user that is represented in the JWT payload.

CHAPTER

SEVEN

ADVANCED FEATURES

Redact is a set of applications that allows developers to build websites with end-to-end encrypted data. When leveraging Redact, website owners are able to display secure user data on their webpages without having access to that data themselves. Securing websites in this way is beneficial to both the user and the website owner as it protects from data leaks, allows complete control and knowledge over who has access to data, and standardizes secure authentication mechanisms that don't rely on passwords.

Note: The two applications that enable a user to store and retrieve encrypted data are redact-client and redact-store.

The *Client* runs on the user's device and handles requests from the browser for secure data display. The *Storage* is a server that fronts some backing database and provides a stable API for CRUD operations on encrypted data.

The redact-crypto library provides all important cryptographic and storage abstractions.

Redact protects from data leaks as it removes the website's need to store sensitive data entirely. In a traditional website, the website owner maintains a database which allows them to store relevant information that their users submit (such as a phone number or address). When a user visits the website, this information is fetched from the database and displayed. By storing the data of many users in a centralized database, a single breach can expose the data of all users of a site. A user must trust that the website has implemented appropriate protection measures.

In a Redact-enabled website, a user can guarantee that "Redacted" data on a webpage has been secured with strong encryption, is only accessible to those who are explicitly granted access, and can be deleted or updated at any time. In order to operate on the redacted data, websites maintain references to it. For example, in a traditional website, there might be a database field called "name" with the value "Alice Doe". In the Redact-enabled website, the database field would still be called "name" but its value would be something like ".profile.name.". This value is interpreted by the *Client* (installed on the user's device), and is translated into the readable data in a way that makes it impossible for the website owner to read it unless granted access by the user. With no data to steal, the website owner no longer has to worry about the liability of data leaks, and users can rest assured in knowing that the potential attack surface for their data is greatly reduced to a single, higly-secure, encrypted location.

This tight control over data storage then empowers users to make explicit decisions as to who has access to their data. For example, imagine the Redact-enabled website in question is a portal used by healthcare providers to share test results and track prescriptions. A user may want that data to be shared with all treating medical professionals, and would want to make sure they all have access to the most up-to-date set of data. With Redact, all of a user's data can be stored in their personal *Storage* while still being accessible on the health portal. Not only can a user see their own data, they can also grant access to health professionals or institutions so they can view the data in a Redact-enabled portal as well. Furthermore, it's all updated in one place, so everyone always gets the same copy.

In order to identify different users to each other and secure this data storage, Redact is paired with a strong authentication and authorization framework. It employes a public-key infrastructure and assigns keypairs to individual users, and certificate authorities to organizations. At the lowest level, users can authenticate themselves anonymously to any Redact service using mutual-TLS requests with a self-signed *Client* certificate. Depending on the authorization requirements of the request, this request could be accepted, or it could be denied. By augmenting the certificate with metadata and having it be signed by a certificate authority recognized by the server, any arbitrary authorization check can be performed to further approve or deny the request. The management of certificates and secret keys is entirely handled by Redact and eliminates the need for passwords or user-initiated login procedures.

When put together, the components of Redact represent a method for storing and handling user data that fundamentally changes the model for data ownership that has existed since the beginning of the internet. The expectation has always been for users to generate data that is then stored and managed by the website owner. This creates numerous liabilities for both the website owner and user as this data is valuable and prone to theft. In the Redact model, user-generated data remains owned by the user but organized into a coherent interface by the website. The result is the ability to create rich user interfaces and applications without having to implement time-consuming and expensive data protection and authentication systems.