

# Policy-Based Client-Side Encryption in Angular



Robert Pinna [Follow](#)

Sep 28, 2018 · 12 min read



...

Data is exponentially increasing—90% of the world’s data was created in just the last two years. The regulatory climate is shifting—70% of all countries have now adopted comprehensive data privacy laws. As a result, software developers are rethinking their data privacy architecture (or lack thereof).

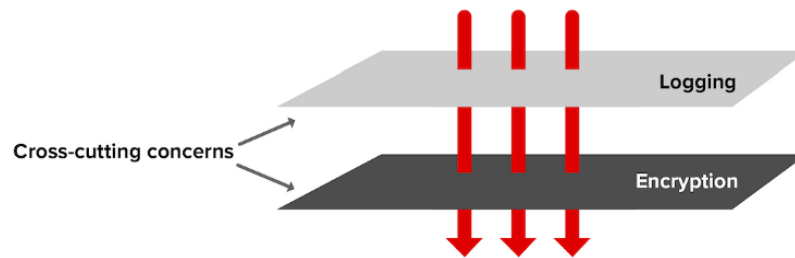
In this article, you will learn how to:

- Implement client-side encryption as part of a data privacy architecture.
- Separate the decision of how to classify data (e.g., *personal health information*) from the decision of who can access that data (e.g., *Seattle Grace Hospital, Dr. Gray*).
- Define a consistent data privacy policy that can be audited and tested.
- Understand symmetric, asymmetric, and transform encryption.

Some of the Angular classes we will use include `HttpInterceptor` and `ClassDecorator`.

# Encryption as a Cross-Cutting Concern

A cross-cutting concern is code that is repeated in many parts of a program, but which is not related to a program's primary function. Classic examples are logging and encryption.



Isolating cross-cutting concerns is an example of *don't repeat yourself* (DRY). It speeds development, keeps the implementation of non-functional requirements (NFRs) consistent across large teams, and simplifies audit and test.

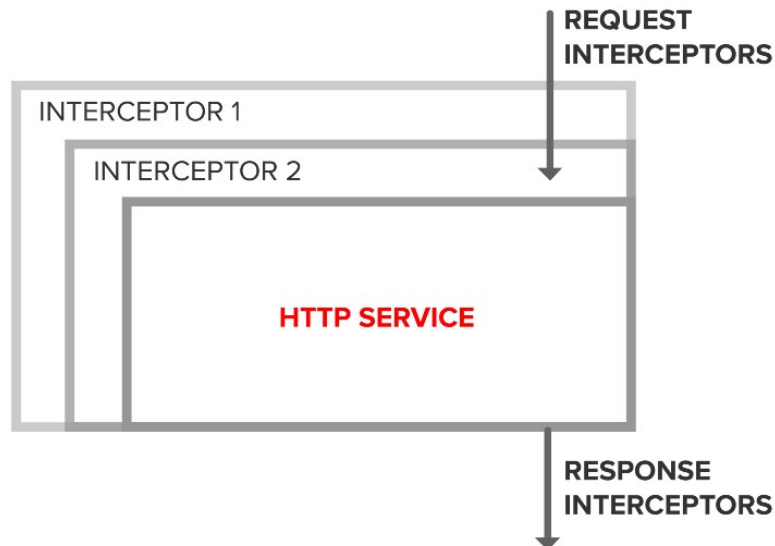
In Angular, we use an `HttpInterceptor` to implement client-side encryption as a cross-cutting concern.

Insider's guide into interceptors and HttpClient mechanics in Angular

You probably know that Angular introduced a new powerful HTTP client in version 4.3...  
[blog.angularindepth.com](http://blog.angularindepth.com)



`HttpInterceptor` is a middleware concept introduced in Angular 4.3 as part of `@angular/common/http`. The framework class `HttpClient` can be configured with one or more `HttpInterceptors` at module initialization. `HttpClient` chains `HttpInterceptors` together based on the order of registration. Each interceptor has an opportunity to modify the `HttpRequest` on the way out and the `HttpResponse` on the way back in. The chain ends by invoking the framework class `HttpService` which sends the `HttpRequest` and listens for an `HttpResponse`.



`HttpInterceptor` provides a clean, well-supported mechanism to insert custom logic into the HTTP pipeline.

## A Logging `HttpInterceptor`

Let's illustrate the concept with an `HttpInterceptor` that simply logs a message to the console and chains to the next handler.

```
1  @Injectable()
2  export class LoggingHttpInterceptor implements HttpInterceptor {
3    intercept(req: HttpRequest<any>, next: HttpHandler): Observable<any> {
4      console.log(`${req.url} has been intercepted!`);
5      return next.handle(req);
6    }
7  }
```

`LoggingInterceptor` is registered in `AppModule` as a provider with `{ provide: HTTP_INTERCEPTORS, useClass: LoggingHttpInterceptor, multi: true }` as shown below.

```

1  @NgModule({
2    imports: [
3      BrowserModule,
4      ...
5    ],
6    declarations: [ AppComponent ],
7    providers: [
8      { provide: HTTP_INTERCEPTORS, useClass: LoggingHttp
9    ]

```

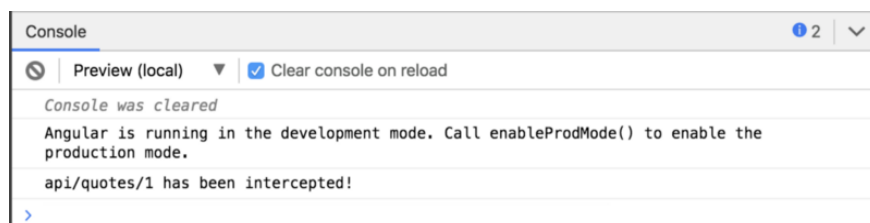
Run the StackBlitz below and click the `Make Request` button.

angular-ironcore-logging-interceptor -  
StackBlitz

Hover and click on URL to run.  
HttpInterceptor that logs to console.  
stackblitz.com



The Interceptor logs the message `api/quotes/1 has been intercepted!` to the console.



## Adding Client-Side Encryption

In this section, we will add an `HttpInterceptor` that encrypts `HttpRequest` data and decrypts `HttpResponse` data.

Implementing the low-level details of encryption and key management is non-trivial. For this reason, we use an audited, vetted third-party encryption library.

*Getting security right is hard for the best teams in the world. It's impossible for most teams.*

— Bruce Schneier

We are going to use the IronCore data privacy platform. IronCore allows us to separate the decision of how to classify data (e.g., *top-secret*, *personal health information*, *personally identifiable information*) from the decision of who can access that data (e.g., *Seattle Grace Hospital*, *Dr. Grey*).

Register `IronHttpInterceptor` as a provider in your `AppModule` .

```
1  @NgModule({
2    imports: [
3      BrowserModule,
4      ...
5    ],
6    declarations: [ AppComponent ],
7    providers: [
8      { provide: HTTP_INTERCEPTORS, useClass: LoggingHttp
9      { provide: HTTP_INTERCEPTORS, useClass: IronHttpIn
```

Run the StackBlitz below and click the Make Request button.

`IronHttpInterceptor` passes through the request and response because the `quote` resource does not have an associated privacy policy.

angular-ironcore-http-interceptor-  
nothing-encrypted - StackBlitz

Hover and click on URL to run. Adds  
IronHttpInterceptor, but no data is...  
stackblitz.com



The `IronHttpInterceptor` will log some messages that show that the request and response should not be encrypted.

```
Console 5
Preview (local) Clear console on reload
Console was cleared
Angular is running in the development mode. Call enableProdMode() to enable the production mode.
api/quotes/1 has been intercepted!
▼ ["pre-decrypt", IronPolicy, HttpRequest, HttpResponse]
  0: "pre-decrypt"
  ► 1: IronPolicy
  ▼ 2: HttpRequest
    body: null
    ► headers: HttpHeaders
    method: "GET"
    ► params: HttpParams
      reportProgress: false
      responseType: "json"
      url: "api/quotes/1"
      urlWithParams: "api/quotes/1"
      withCredentials: false
      __proto__: HttpParams
    __proto__: HttpRequest
  ► 3: HttpResponse
  ► ["not encrypted"]
  ► ["post-decrypt", HttpResponse]
```

## Privacy Policy

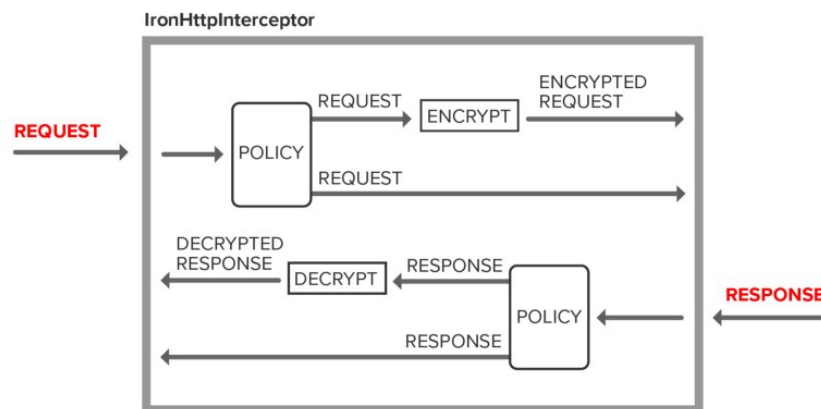
For our example, we're going to define an `Order` class. In our fictitious app, an `Order` is a top-secret message that we must keep secure and private. A REST endpoint persists orders in a (mocked) backend.

We are going to use the `IronEncrypt` class decorator to specify our data privacy policy. Class decorators provide a convenient way to attach metadata to a class. If you've used `@Component` or `@Injectable`, you're familiar with the concept of class decorators.

Implement the `Order` and apply the `IronEncrypt` class decorator with the data classification *top-secret*.

```
1  @IronEncrypt({dataClassId = 'top-secret'})
2  export class Order {
3      public date = new Date();
4      constructor(
5          public title: string,
6          public message: string,
7          public id?: number
8      ) {
```

The interceptor will encrypt the `Order` on HTTP POST and PUT, and decrypt it on HTTP GET. Requests and responses for other data types will pass through the interceptor without being modified.




Start the StackBlitz below and submit an order to encrypt.

angular-ironcore-round-trip - StackBlitz

Hover and click on URL to run. Round trip encrypt, decrypt, to group.

[stackblitz.com](https://stackblitz.com)



The console logs unencrypted and encrypted JSON. The encrypted JSON is generated on the client, using a private key that does not leave the local device.

The StackBlitz will show how `POST` requests are automatically encrypted and `GET` requests are automatically decrypted.

```
Angular is running in the development mode. Call enableProdMode() to enable the production mode.
api/orders has been intercepted!
▶ ["pre-encrypt", IronPolicy, HttpRequest]
▼ ["post-encrypt", EncryptedDocument]
  0: "post-encrypt"
  1: EncryptedDocument
    document:
      "Adi+6THs0kxtzbxMphPzC2VrUwKErV0C1mxUnIGKwQMNYEYcBewiHtD0A8w2I7d+xdIJvmCmJ05rLF+XQ7EbY3LKBjFPELJjr5xRq0yP3j"
      id: 8461686524276831
      __proto__: EncryptedDocument
api/orders/8461686524276831 has been intercepted!
▶ ["pre-decrypt", IronPolicy, HttpResponse]
▶ ["decrypting an item"]
▼ ["post-decrypt", HttpResponse]
  0: "post-decrypt"
  1: HttpResponse
    body: Object
      date: "2018-09-26T17:08:01.795Z"
      id: 8461686524276831
      message: "Set phasers to stun"
      title: "Orders of engagement"
      __proto__: Object
    headers: HttpHeaders
      ok: true
      status: 200
      statusText: "OK"
      type: 4
      url: null
      __proto__: HttpResponse
>
```

## Data Classification

*All human beings have three lives: public, private and secret.*

— Gabriel García Márquez

Most applications have a mix of data with different levels of sensitivity. For example, data classifications might include:

- Protection levels (e.g., restricted, public, private)
- Regulatory classifications (e.g., personal-health-information, personally-identifiable information, pci-card-data).
- Functional groups (e.g., legal, HR, finance).

In a data privacy architecture, we want to make it easy to classify data *declaratively*. Declarative programming is specifying business logic without having to describe control flow. An example of a declarative approach is CSS; we define a CSS selector to *declare* a paragraph as bold, but we don't have to define how bold is implemented. Classifying data declaratively simplifies the consistent application of data privacy rules across large teams of developers with varying skill sets.

We want to be able to classify data when it is created, and decide later who should be able to access specific data classifications.



Here are some examples of how the `IronEncrypt` class decorator can be used to specify data classifications.

```
1 // Classify data by protection level
2 @IronEncrypt({dataClassId = 'private'})
3 export class IntellectualProperty {
4     ...
5 }
6
7 // Classify data by regulatory category
8 @IronEncrypt({dataClassId = 'personal-health-information'})
9 export class MedicalRecord {
10     ...
11 }
12
```

In the examples above, the classification is known at design time. It's often the case where classification is more dynamic. For example, we may want to classify data as *personal-health-information* for the active user. `IronEncrypt` has a number of options to support use cases like this. One simple approach is to define during initialization well-known identifiers (e.g, *personal-health-information*) to serve as lookup keys into a bindings table that holds the actual *surrogate keys*.

```
1 // During initialization
2
3 this.ironPolicyFactory.bindings.set('personal-health-i
4
5     ...
6
7 // Using square brackets to indicate an indirect bindi
8
```

In this article, we're showing how data classification and policy architecture is applied to HTTP with a backend REST endpoint. The same approach is also used to define how data is encrypted in browser local storage or S3 buckets.

## Deciding Who Can Access What Data

| *All problems in computer science can be solved by another level of indirection.*

| — *David Wheeler*

Access control defines which users or system processes are granted access to data, as well as what operations are allowed to be performed on that data.

In most systems, access control is implemented by software algorithms that gate access to the underlying data. Data is only protected when it is accessed through the access control software. Many privacy and security incidents are the result of either bypassing the access control layer to gain direct access to the underlying data or by exploiting inevitable bugs in complicated access control logic.

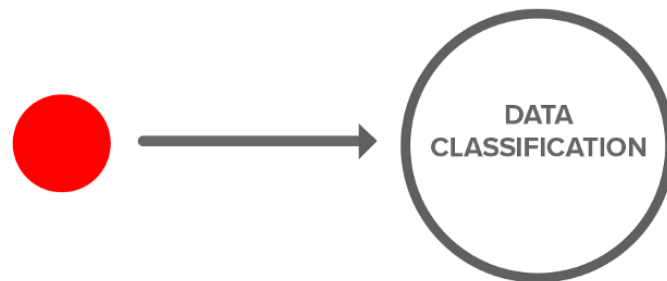
In a data privacy architecture, we want access control to be embedded in the data itself. The important point here is that the access control is *cryptographically enforced*. Encryption algorithms are used to mathematically prove that only users with a valid cryptographic key are able to access data subject to their roles and permissions. The cryptographic implementation is both heavily audited and subject to the scrutiny of open source to provide a higher quality standard.

In this data privacy architecture:

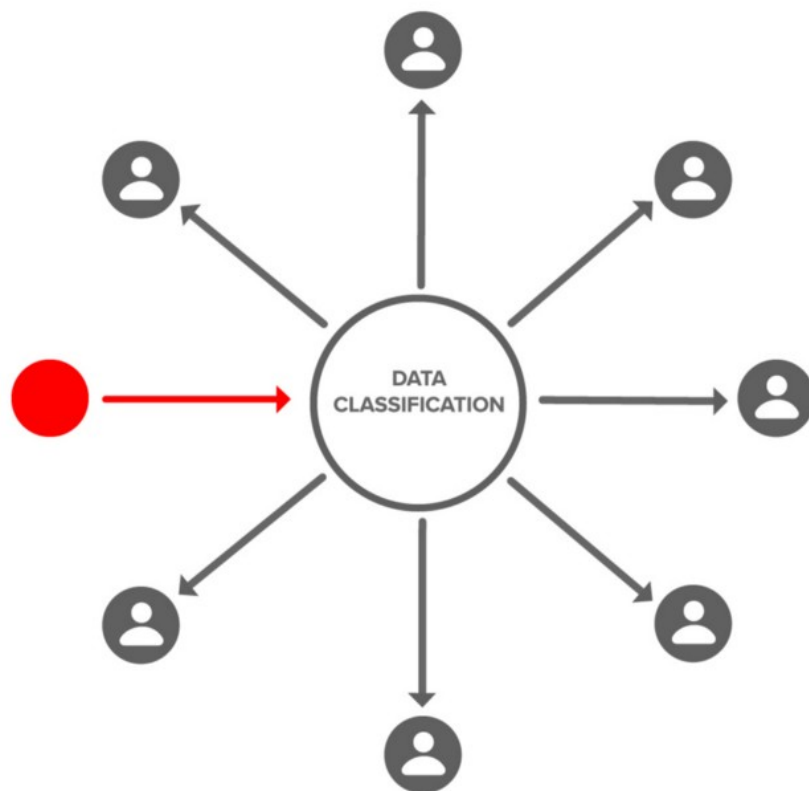
- Data is encrypted at its point of origin.
- Data stays encrypted wherever it lives—in the cloud, on a device, in transit, shared with a partner, in a backup, in a system log, etc.
- Data is decrypted only at the point of use by authorized users or system processes, with access control enforced cryptographically, and all operations logged as part of an audit trail.

To make this system practical and scalable, it's important to separate the decision of how to classify the data (e.g., *personal health information*) from the decision of who can access that data. This is a concept called *orthogonal access control*.

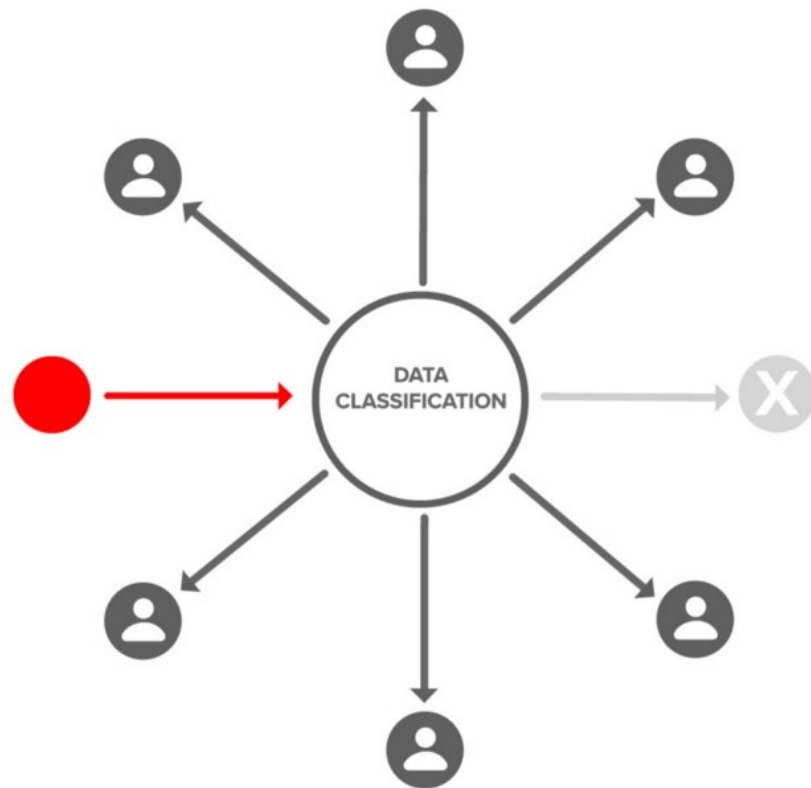
Let's look at some infographics to explain the concept. Here we are encrypting data at its point of origin to a data classification (e.g., personal-health-information).



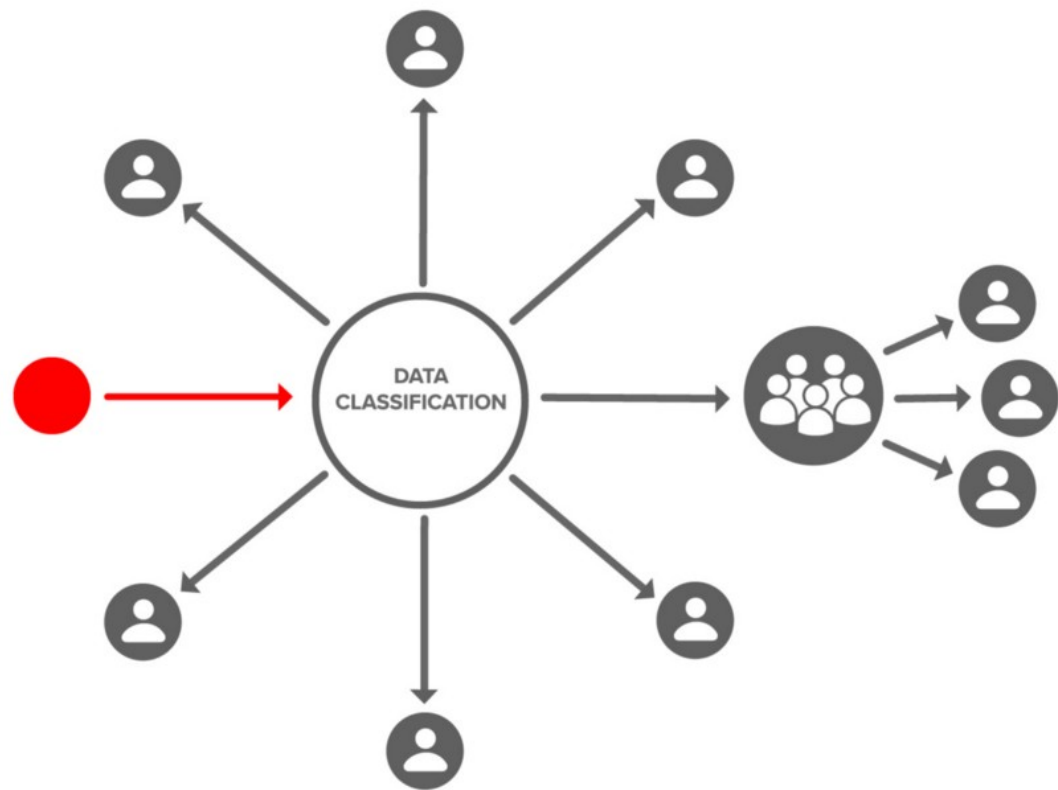
We decide later who has access to that data classification or group through a cryptographic action that adds users to that data classification or group.



There is also a cryptographic action to revoke user access by removing users from a data classification or group.



There is also a cryptographic action to grant access to groups and delegate administration.



While it isn't necessary to understand how cryptographically-backed, orthogonal access control works in order to effectively use it, many developers like to know. Let's explore that below.

## How it Works

Encryption is the process of encoding data, making it unintelligible to anyone but the intended recipient(s). In this section, we're going to explore three types of encryption: symmetric encryption, asymmetric encryption, and transform encryption.

In the descriptions below, we use the terms *plaintext* and *ciphertext*.

Plaintext is the original data that someone wants to secure so it cannot be accessed by anyone but the intended recipients.

Ciphertext is the scrambled result produced when the encryption process is applied to the plaintext.

A good encryption algorithm produces ciphertexts that look like random data and requires a lot of work to recover the plaintext by

anyone that is not authorized. This process of recovering the original plaintext from a ciphertext is called *decryption*.

Most encryption algorithms use a *key* as part of the encryption/decryption process. Possession of the key that is correct for a particular ciphertext serves as a person's authorization to access the plaintext that produced that ciphertext.

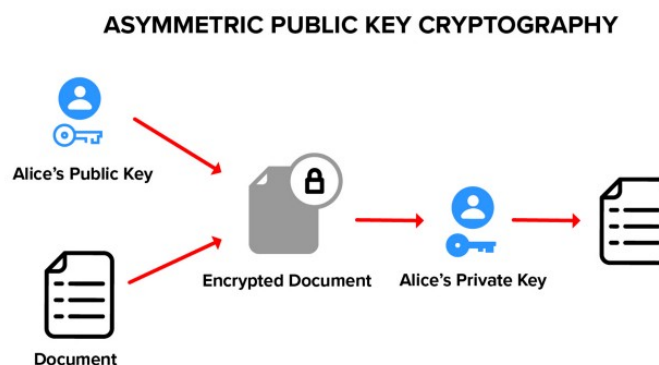
## Symmetric Encryption

Symmetric encryption uses the same key for both encryption and decryption. It's fast and relatively simple, but users must find a way to securely share the key, since it is used for both encryption and decryption.

## Asymmetric Encryption

Asymmetric encryption uses two keys that are mathematically related, generally called a key pair. The plaintext is encrypted with the *public key*, and the ciphertext is decrypted using the corresponding *private key*. Asymmetric encryption is also known as public key encryption because the encryption key can be shared publicly, while the decryption key must be kept private.

If data is encrypted to multiple users, it must be separately encrypted with each user's public key. To revoke user access, we must possess and change the underlying data (and all its copies).

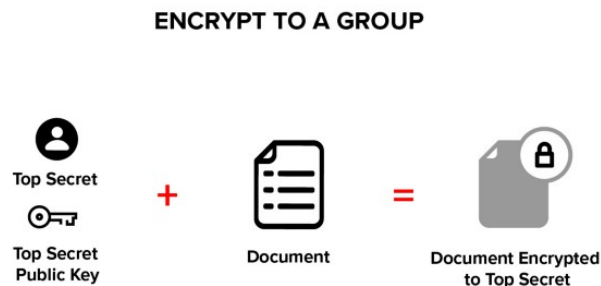


## Transform Encryption

Transform encryption uses two asymmetric key pairs (one for a group and one for a member of the group) and a special *transform key*

derived from these key pairs. Transform encryption is used to create cryptographically-backed, access control groups.

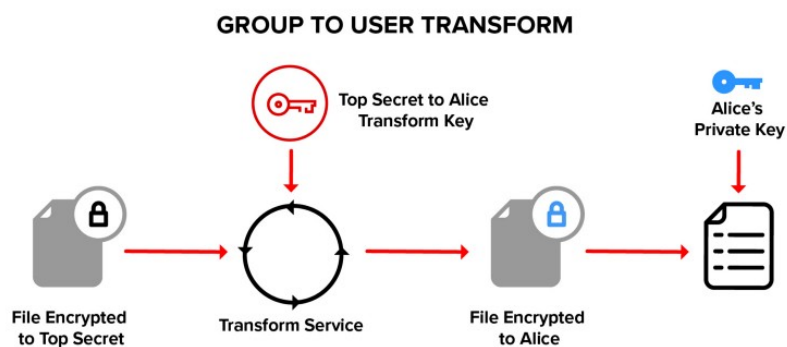
Plaintext is encrypted using a *group public key*, this is standard public key cryptography.



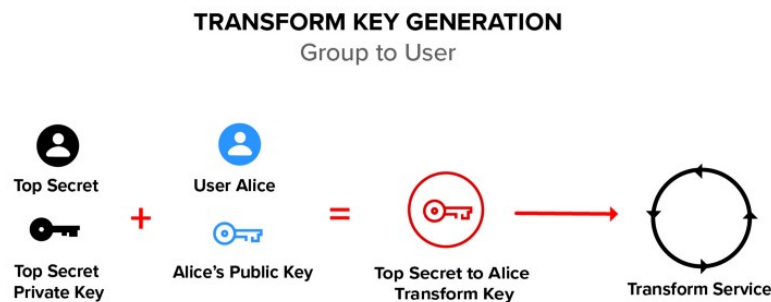
Ciphertext is transformed from group ciphertext to member ciphertext, using a *transform key*. Ciphertext is never decrypted during a transform, allowing the transform to be performed by a blind service.

The transformed member ciphertext is decrypted with the member private key on the member's local client device. The private decryption key does not leave the member's device.

The service provides a natural place to audit access since a transform is required to decrypt data. The service also provides a natural point of revocation because a transform key is required for member access.



Only group administrators can generate a transform key, which is derived from the group private key and the member public key. By generating and deleting transform keys, group administrators are able to add and remove members of the group. These operations may be performed without the need to change or even to possess the underlying data.



Group administrators cannot decrypt data encrypted to the group. This property of being able to administer groups but not decrypt the underlying data is implemented cryptographically with a key augmentation algorithm. See the ACM paper Cryptographically Enforced Orthogonal Access Control for details.

The transformation process does not require (or allow) the transform service to decrypt the ciphertext while transforming it. This allows transforms to be done by a semi-trusted service. The service never gains access to the plaintext and cannot get any information about the private key of either the group or the member.

An important property of transform encryption is that data can be encrypted to a group without knowing who is in the group or who will be added or removed. For example, we can encrypt our medical records and decide later what hospital can access them. When we check out of the hospital, we can remove access. The ability to *decide later* and *change our mind* makes transform encryption ideal for protecting data in the cloud, where data is often shared with a large or dynamic set of users.

*There are only two hard things in Computer Science: cache invalidation and naming things.*

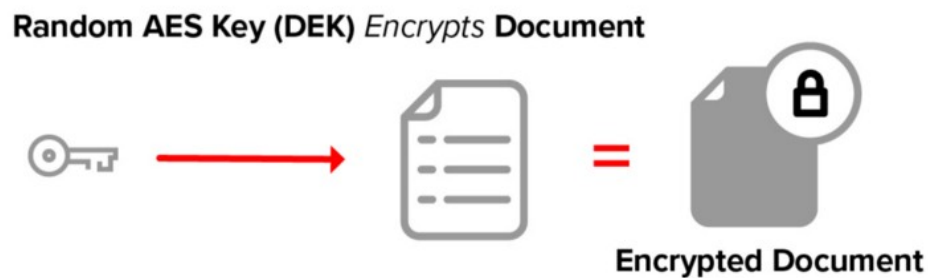


In the academic literature, transform encryption is called Proxy Re-Encryption (PRE). We do not use that term because in other contexts re-encryption implies an encrypt—decrypt—encrypt cycle, which we are **not** doing here. We use the term *transform encryption* to emphasize that the service transforms group ciphertext to member ciphertext, without the need for an intermediate decryption step.

## Envelope Encryption

In the sections above, we simplified our description by assuming that a single type of encryption was applied to the entire data stream. In practice, public key cryptography almost always uses a symmetric inner key, called a document encryption key (DEK), to encrypt the data.

In the encryption process, a random AES key is generated as the document encryption key (DEK). The data is encrypted with this key.

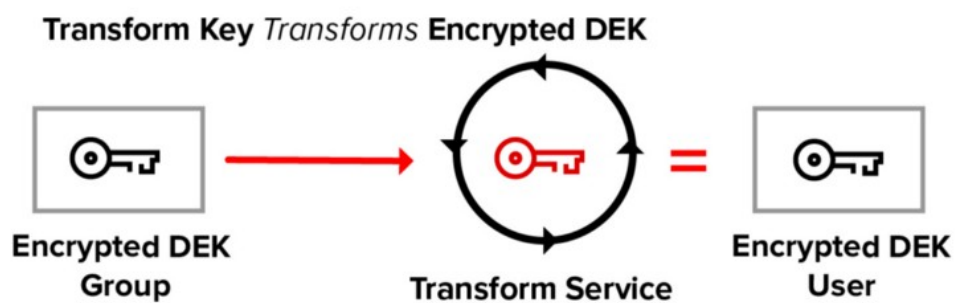


The DEK is then encrypted with a public key. The encrypted DEK may be stored with the data or elsewhere.

### Public Key *Encrypts* DEK

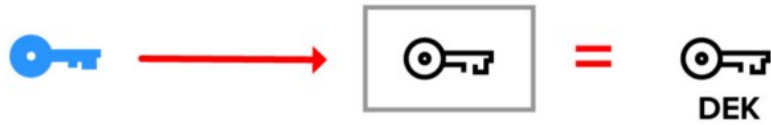


In transform encryption, the service transforms the encrypted DEK from group to user. The transform key is used to perform the “DEK encrypted to group” to “DEK encrypted to user” mapping, the DEK is never decrypted.



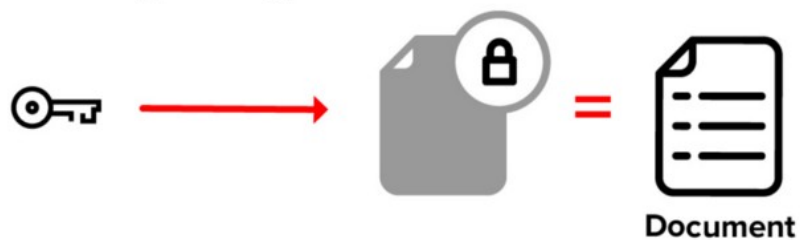
After we have transformed the encrypted DEK, the user's private key is used to recover the DEK symmetric key.

### Private Key *Decrypts* Encrypted DEK



The unencrypted DEK is then used to decrypt the underlying data.

### DEK *Decrypts* Encrypted Document



The encrypted DEK is small, minimizing network traffic and making transforms a lightweight operation. And since the DEK is a symmetric key, decryption is faster.

## Future Topics

We skipped over some important topics related to data privacy. We'll cover those in future articles. If you're interested in following along, watch this blog and/or sign up for our newsletter. Some of the topics we're planning:

- Managing user keys
- Endpoint security

- Identity management and data privacy
- Data privacy with S3
- Data privacy with GraphQL
- Data privacy with Redux and NgRx Store

## Next Steps

A complete sample application is available at  
<https://github.com/IronCoreLabs/getting-started-angular>.

Sign up for your free developer account at  
<https://admin.ironcorelabs.com> and learn more about how the  
IronCore privacy platform makes it easy to implement an advanced  
data privacy architecture.