



JUNE 2, 2022

#### AUTHORS



Manish Ahluwalia  
Head of Customer Security & Privacy



Sean Falconer  
Head of Developer Relations



Daniel Wong

# Demystifying Tokenization: What Every Engineer Should Know



---

## RELATED ARTICLES

Demystifying Tokenization: What Every Engineer Should Know

Skyflow Partners with MuleSoft to Enable API-based Data Privacy, Security, and Compliance across Mulesoft Gateway Integrations

How to Keep Sensitive Data Out of Your Logs: Nine Best Practices

Cloud Developer

Data Privacy Vault

B2B B2C

If you're storing sensitive user data, you're right to be concerned about the potential for compliance and security risks. One method to secure sensitive data is tokenization. In this post, we break down what every engineer should know about tokenization – what it is, how and when to use it, the use cases it enables, and more.

Nearly every business needs to store sensitive data. This includes things like customer names, email addresses, home addresses, and phone numbers – or even highly sensitive data like credit card information and social security numbers. Companies have both business and technical reasons to protect customer data privacy by restricting access to such sensitive information.

Broadly speaking, you can do this in **two ways that aren't mutually exclusive**: secure the infrastructure that handles sensitive data, and secure the data itself. Securing only the infrastructure isn't enough to meet compliance standards or provide robust protection against a data breach, so **you must also secure the data itself**.

Securing the data itself requires some form of obfuscation. This means that instead of storing plaintext values, you would store the obfuscated version instead. If you apply robust controls to the obfuscation and de-obfuscation processes, then only authorized users and processes that have a legitimate need for sensitive data can access plaintext values.

Tokenization is an essential tool to protect sensitive data with obfuscation.

Healthcare

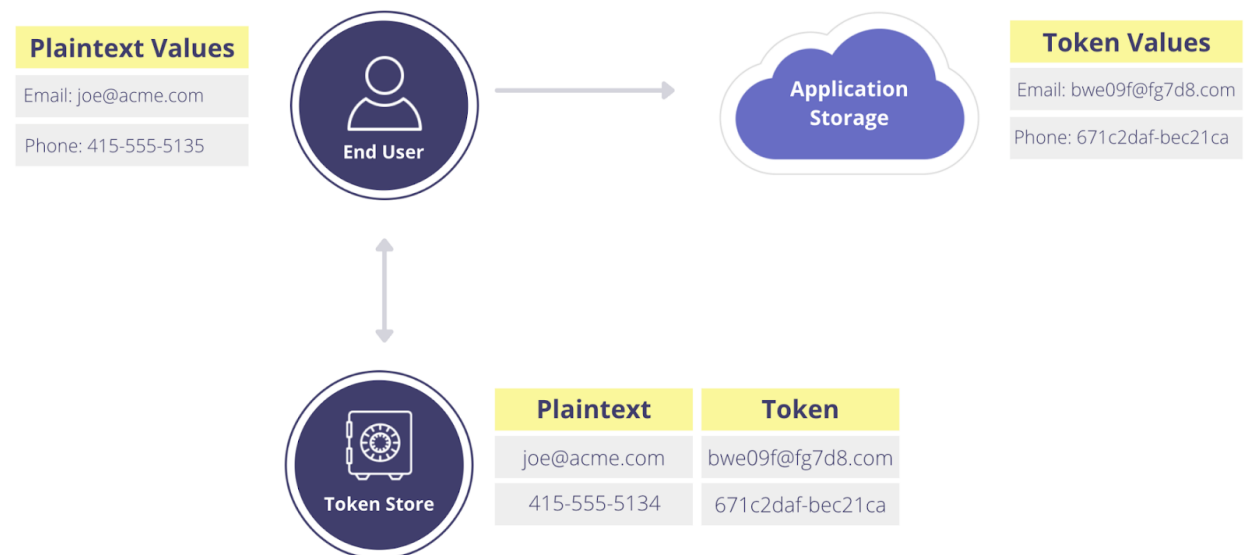
Finance

Data Security

Security Vault

# What is Tokenization?

*Tokenization* is a non-algorithmic approach to data obfuscation that swaps sensitive data for tokens. For example, if you tokenize a customer's name, like "John", it gets replaced by an obfuscated (or *tokenized*) string like "A12KTX". Because there's no mathematical relationship between "John" and "A12KTX", even if someone has the tokenized data, they can't get the original data from tokenized data without access to the tokenization process. This means that even if an environment populated with tokenized data is breached, this doesn't compromise the original data.



*Example of Tokenization: Plaintext-to-token Mapping is Stored in a Secure Data Store*

Because tokenization methods are non-algorithmic, they require some sort of table that maps the original data to the tokenized data (as shown above). Typically this is a database or internal system with tightly-controlled access that exists outside the business's internal application environment. For example, one system for securely storing the original values and managing the token mapping is Skyflow's Data Privacy Vault.

## What is a Token?

In the broadest sense, a token is a pointer that lets you reference something else while providing obfuscation. In the context of data privacy, tokens are data that represent other, more sensitive data. You can think of them as a “stand-in” for the actual sensitive data, such as a social security number or a credit card number.

Tokens are pseudorandom and don't have exploitable value. Instead, they serve as a map to the original value or the cell where it's stored. Generating tokens is a one-way operation that's decoupled from the original value and can't be reversed - like generating a UUID purely from a few random seed values. Tokens can be configured for limited-time use or maintained indefinitely until they are deleted.

## What is Detokenization?

Detokenization is the reverse of tokenization. Instead of exchanging the original sensitive data for a token, detokenization exchanges the token for the original value. Both tokenization and detokenization are controlled by

the tokenization system. Only the tokenization system has access to the token map and can perform these exchanges. And only authorized users and processes have access to the tokenization system.

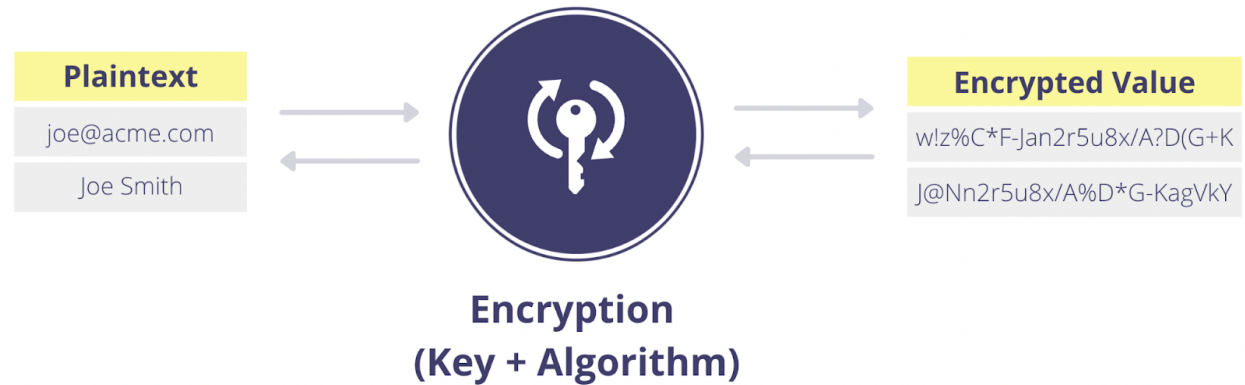
# Tokenization versus Encryption

Another obfuscation approach is encryption. While most readers are no doubt familiar with encryption to some extent, please bear with me because you need to understand encryption to fully understand tokenization. With encryption, you convert plaintext values into ciphertext using well-known algorithms combined with carefully guarded encryption keys. The reverse of this process is decryption, which converts the ciphertext into the original plaintext value, as long as you have the proper decryption key.

Encryption is very important and widely used in security and data privacy. To get encryption right, you must choose the right algorithm, chaining mode (block versus stream), use a strong random number generator and initialization vectors for key generation, and rotate keys. Encryption algorithms are constant targets of cryptanalysis.

The primary difference between tokenization and encryption is: with encryption the ciphertext is still mathematically connected to the original

plaintext value. This means that if someone gets access to the key, either through brute force computation or by stealing it, they can retrieve all of the original data.



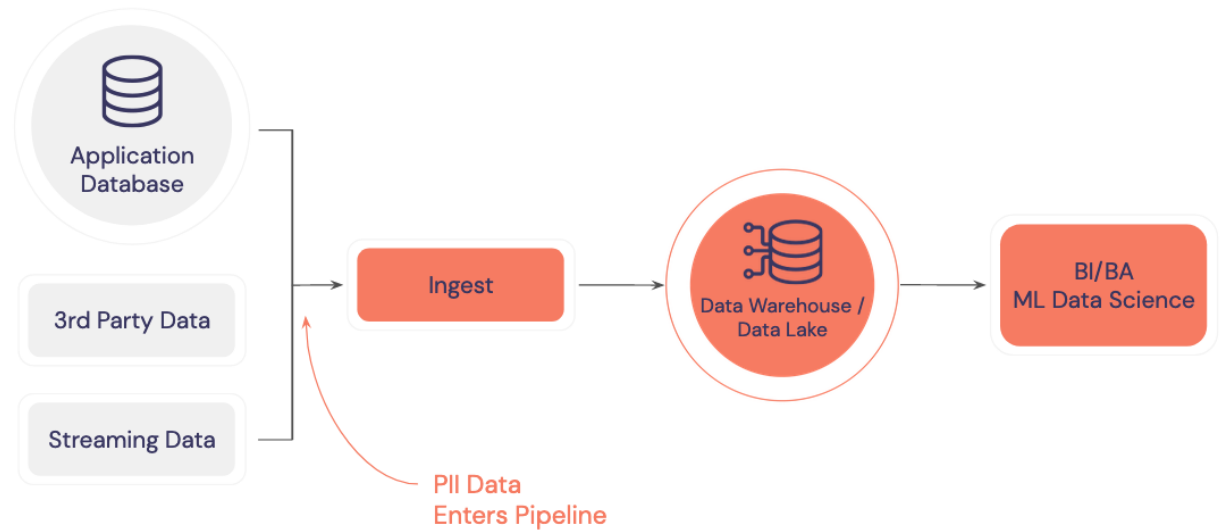
*With Encryption, Anyone with the Key Can Reverse the Encryption Process*

Both encryption and tokenization are necessary to solve challenges around data privacy, but tokenization is better suited to certain use cases than encryption, as described below.

# What Problems Does Tokenization Solve?

Data has a way of proliferating and spreading around different systems through logs, backups, data lakes, data warehouses, etc. When it comes to sensitive user data, replication and fragmentation make compliance and security a nightmare.

For example, consider a simple data pipeline, pictured below. On the left side are different data sources that provide inputs to an ingestion pipeline that copies data downstream to a data warehouse. Each of these data sources likely contains PII data that is being replicated as it's passed along this pipeline to the data warehouse.



*A Common Data Pipeline Architecture Where PII is Being Copied Downstream*

Since PII data ends up in the warehouse/lake - and likely within analytics dashboards - it's a big challenge to disentangle the sensitive data from the non-sensitive data. Companies have to go through painful discovery processes with specialized tools to track down and cleanse their system PII data. This is expensive and complicated.

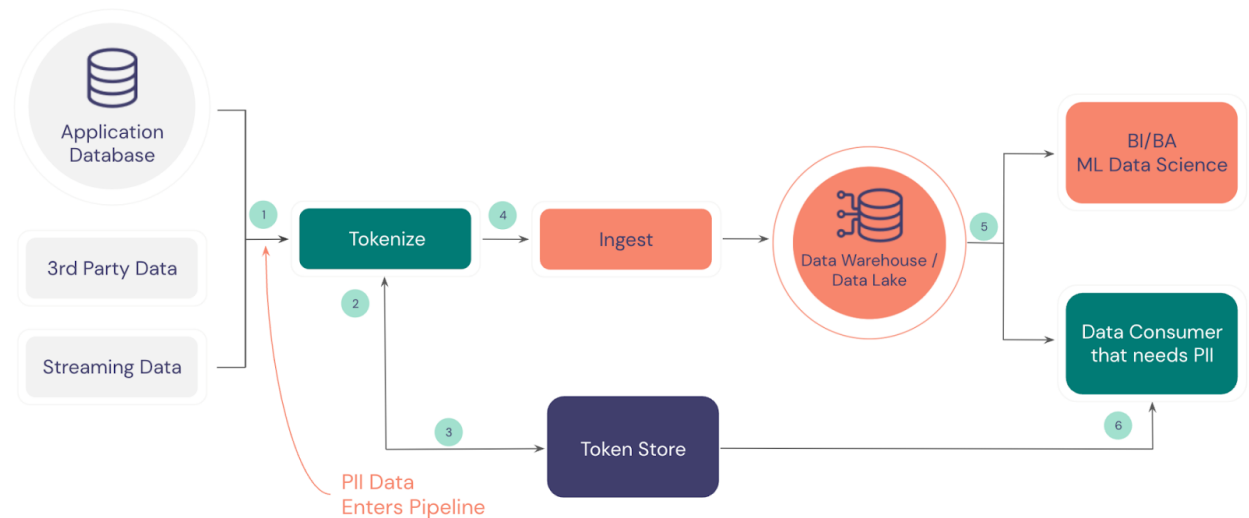
The intermingling of PII with application data also makes it very difficult to meet compliance requirements like data residency. For data residency, you need to extract and regionalize PII based on the customer's country and the areas of the world where you're doing business.

Additionally, access control is another potential challenge in this infrastructure. Each node in the diagram shown above is a target for malicious hackers that needs to be locked down. This example is relatively



simple, so you might be able to handle this node-by-node, but as the pipeline scales, this becomes increasingly difficult. And to make matters worse, any analytics clients that connect to this data warehouse are also potential targets for a data breach.

Tokenization solves this problem, as shown in the following diagram.



*A Common Data Pipeline Where PII Data is Tokenized*

Here's how tokenization works in a common data pipeline:

1. Incoming data, which includes PII, is intercepted so that PII can be tokenized
2. The tokenization process extracts the PII and sends it to the tokenized data store
3. The tokenization store exchanges the PII for tokens
4. The tokenized data is forwarded to the ingestion pipeline

5. The analytics processes and systems can work with the data without risk
6. Authorized data consumers can detokenize the tokens if needed

The data warehouse or data lake no longer stores the original plaintext values, instead it stores tokenized data. Even if a malicious hacker gained full access to the warehouse, all they'd have access to is non-sensitive application data and tokenized forms of sensitive data.

Tokenization also simplifies the deletion of sensitive data. As long as you delete the original value from the token map, there's no longer any connection between the token and the original value.

So the original value is secure, even if you leave all of the tokens in the databases, log files, backups, and your data warehouse. The tokens will never be detokenizable, because the mapping between the tokens and the original values no longer exists.

# Tokenization Approaches and the Features They Enable

The simplest form of tokenization is to exchange plaintext data for a randomly generated value like a UUID. In a simplified world, instead of storing sensitive data, only randomly generated tokens would be stored and transmitted; and only an authorized party would have full authority to view plaintext values. However, in reality only having random tokens isn't enough, because there are real-world requirements that require different types of tokenization.

Some storage and transmission systems, such as APIs, have an expectation that the data they work with is in a certain format. For example, a database system might expect that the “phone\_number” column follows the E.164 standard for international numbers. If a phone number is tokenized as a random number or UUID, the token can't be stored in the database, breaking the existing infrastructure.

In this section, we walk you through a variety of different tokenization approaches and what features they are designed to enable.

## Format-preserving Tokenization

With format-preserving tokenization, a token is exchanged for the original value, but the token respects the format of the original value. If the format of the token is the same as the original value, then the tokenization process is *format-preserving*.

## Format-Preserving Tokenization



*Example of Format-preserving Tokenization*

For example, an 11-digit phone number could be exchanged for a randomly generated 11-digit number following the same format that the database expects to store phone numbers in. Or an email address could be exchanged for a randomly generated string that respects the email format: the username prefix followed by the “@” symbol and a domain (i.e., *username@domain.com*).

## Length-Preserving Tokenization

A length-preserving token has a fixed length or maximum length. In some cases, the format implies the limit on the length. This is the case with a format-preserving phone number, credit card number, or social security number: these tokens are both format-preserving and length-preserving.

On the other hand, an email address requires a format-preserving token, but not necessarily a length-preserving token.

## Random and Consistent Tokenization

Consider the table below. If you want to tokenize the **First Name** column, you'll tokenize the plaintext value "Donald" twice. The question is: does tokenizing "Donald" always generate the same token value?


ID	First Name	Last Name	Favorite Color
1	Donald	Jones	Blue
2	Pluto	Anderson	Red
2	Donald	Smith	Gold

*Sample Table with Sensitive Information*

With encryption, the strongest encryption schemes guarantee that the same plaintext will encrypt to different ciphertexts with each encryption operation. This is a key feature: it means that even if they gain access to these ciphertexts, an attacker gets no information about whether or not two plaintexts are identical. When tokenizing data, you might want a similar guarantee.

If you wanted the same feature with tokenization, you would choose to generate *random tokens* (sometimes called *non-deterministic tokens*). With random tokens, tokenizing the same value twice will yield two

different tokens. The table above might end up looking like the one below, with the **First Name** column replaced with random tokens.



ID	First Name	Last Name	Favorite Color
1	A34TSM3	Jones	Blue
2	R42GV4E	Anderson	Red
2	KT48SR2	Smith	Gold

*Example of Random Tokenization*

However, there are use cases where having a different token value for the same plaintext value is not desirable. You might want to be able to search the tokenized data store for a record where the first name is equal to “Donald” or join on the tokenized first name. A random-token approach can’t support these operations.

In this case, you need a *consistent* tokenization method (sometimes called ‘deterministic tokenization’ because tokens might be generated by a deterministic process). This guarantees that tokenizing “Donald” will always return the same token. Here’s how the original table would look if you consistently tokenize the **First Name** column.

ID	First Name	Last Name	Favorite Color
1	A34TSM3	Jones	Blue
2	R42GV4E	Anderson	Red
2	A34TSM3	Smith	Gold

*Example of Consistent Tokenization*

The consistency of the tokenization process is an important tradeoff for you to consider. It allows you to perform some operations like equality checks, joins, and analytics but also reveals equality relationships in the tokenized dataset that existed in the original dataset.

The opposite of consistent tokenization - *random tokenization* - does not leak any information about data relationships (as shown above) but also doesn't allow querying or joining in the tokenized store. So you should choose between consistent and random tokenization depending on whether it's more important to support operations that rely on data relationships (as might occur when tokenizing identifiers like names) or whether it's more important to keep those relationships obscured to ensure data privacy.

## Value Tokens and Cell Tokens

So far, we've discussed individual values that are sensitive, and using tokens to point to these values, like the first name "Donald". These are *value*

*tokens*: they correspond to a specific value.

However, most of us organize customer data by storing it in records that contain many sensitive fields. So you wouldn't just store the first name, but also last name, phone number, date of birth, government ID and so on.

Instead of creating a token for the first name "Donald", we can create a token that points to a specific cell: in this case, the **First Name** field of the user record being added. These are *cell tokens* - they don't point to the value, but refer to the storage location. If needed, we can still have tokens maintain some of the above constraints like having a fixed length or a specific format.

Cell tokens enable certain optimizations — for example, if a customer's phone number changes, then just the cell referenced by the token needs to be updated. The token itself doesn't change, which frees you from the need to update every tokenized data store when token values change or are deleted.

## Deleting and Updating Values Pointed to by Tokens

Token semantics, like consistent versus random and value tokens versus cell tokens, have implications on how you need to manage update and delete operations.

Let's take a look again at our example table with sensitive information:



ID	First Name	Last Name	Favorite Color
1	Donald	Jones	Blue
2	Pluto	Anderson	Red
2	Donald	Smith	Gold

*A User Data Table With Sensitive Information*

Let's assume we are using random tokens, so our datastore (after tokenization) looks like this:

ID	First Name	Last Name	Favorite Color
1	A34TSM3	Jones	Blue
2	R42GV4E	Anderson	Red
2	KT48SR2	Smith	Gold

*A User Data Table With the First Name Randomly Tokenized*

Now, if “Donald Jones” files a right-to-be-forgotten request, you have the following choices:

- Delete all instances of the (tokenized) value “A34TSM3” from your datastores. This means you delete all replicas. And backups. And logs. This is possible, but very hard to do.

- Delete the token “A34TSM3” from your token table only. You do NOT need to touch your replicas, backups, logs, etc. The string A34TSM3 still exists everywhere but it is meaningless and can never be converted back to “Donald”.

Let’s suppose that the same table was tokenized using consistent token semantics. So the datastore post-tokenization looks like the following:

ID	First Name	Last Name	Favorite Color
1	A34TSM3	Jones	Blue
2	R42GV4E	Anderson	Red
2	A34TSM3	Smith	Gold

*A User Data Table With the First Name Consistently Tokenized*

Note again that the two “Donalds” have the same token for the first name. Now, if “Donald Jones” files a right-to-be-forgotten request, you can’t just delete the value associated with the token “A34TSM3” — if you did, you also delete “Donald Smith’s” first name. In this case, you have no choice but to update your tokenized datastores.

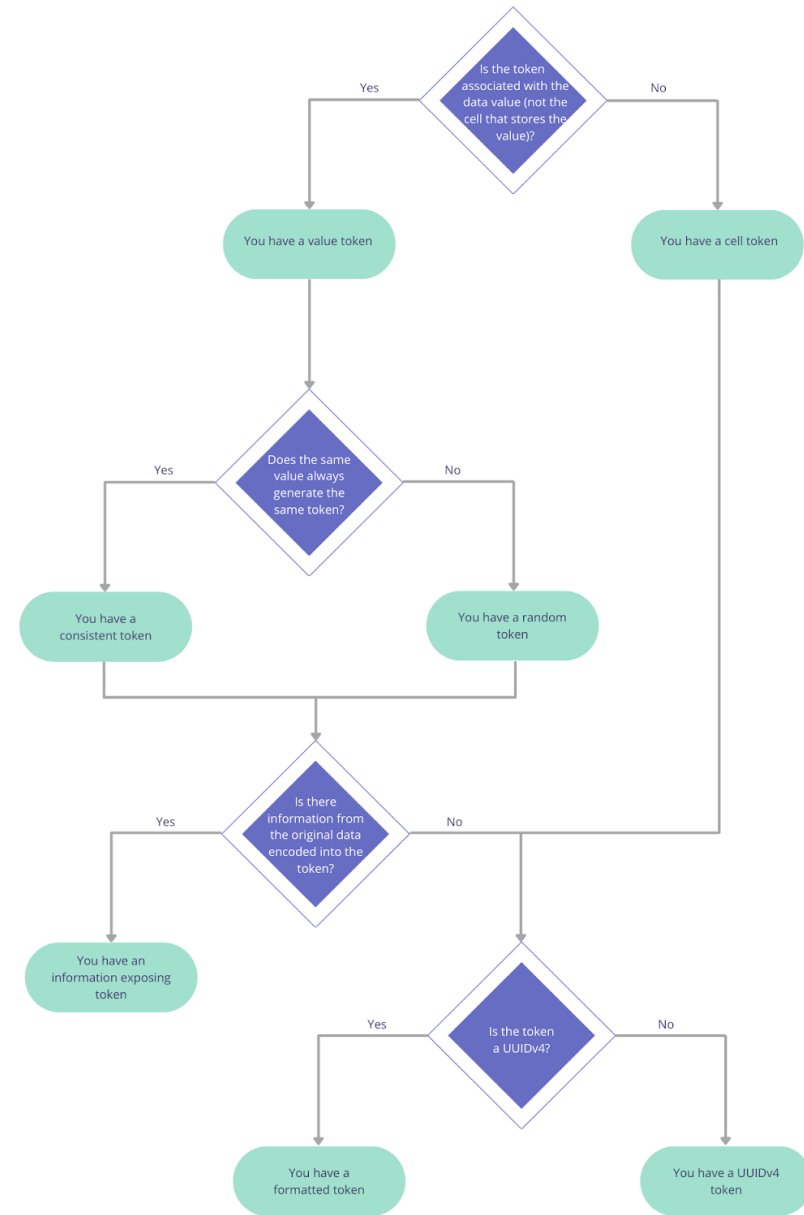
A similar logic applies to updates as well. Without mutating your datastore for deletes and updates, you can’t have all three of the following:

- Consistent tokens
- Multiple rows storing tokens for the same value

- The ability to delete or update a strict subset of the rows

## Token Taxonomy

You need to consider the above token features when choosing how to tokenize a particular type for data. For example, you can have a consistent format-preserving token, or a random format preserving token, or a UUID cell token. See the decision tree below to determine what kind of token you have:



*Decision Tree for Determining the Type of Token You Have*

# Security Best Practices for Tokenization

Tokenization helps keep sensitive data out of your application infrastructure, but you do have to do your homework if you truly want to secure private data. There are several security considerations to be aware of when applying tokenization, which we cover in more detail below:

- Consider Entropy and Space Size
- Restrict Detokenization Permissions
- Treat Tokens as Secrets
- Rate Limit and Monitor

## Consider Entropy and Space Size

When choosing how to tokenize data, it's important to consider the space and entropy of the chosen method as these factors impact how secure the data is.

The total set of valid tokens constitutes the information space from which a tokenization process generates the tokens. Similarly, while the tokenization process doesn't get to choose it, the size of the input values impact the size of the input space to the tokenization process.

The size of the output space (i.e., how many characters are available to encode the token) puts an upper bound on the **entropy** of the tokens. Entropy is an important concept in security since it determines how secure your data is (all other factors being the same, more entropy implies better security). Ideally, you want to maximize entropy, but there is a tradeoff between token entropy and application compatibility. There might be application requirements that impact the token format and constraints.

A crucial thing to note is that certain formats greatly restrict the space from which a tokenization process can pick tokens. For example, if we want tokens to be 10 digits long, as in the format-preserving tokens for phone numbers, then there can only be  $10^{10}$  (or 10 billion) tokens - i.e. the space for the token is all 10 digit numbers.

Certain token formats have an infinite space, for example if there is no restriction on the length of each token. Some spaces are finite, but so large that they are considered practically inexhaustible. For example, UUIDv4 has  $2^{122}$  possibilities. The size of the tokenization space is very important for security reasons and for practical purposes, so it makes sense to give this matter some thought before picking a tokenization approach.

## Restrict Detokenization Permissions

Obviously, if you replace your sensitive data with tokens, and allow anyone to “redeem” (or detokenize) the token, then you haven’t really improved your data security! You should always use the Principle of Least Privilege when granting users or services access to a detokenization service. Ideally,

you should utilize **data governance** to control access. Most applications don't need to detokenize data, so those applications shouldn't have permissions to do so – they should just work with the tokens as-is.

## Rate Limit and Monitor

Rate limiting and monitoring help you catch potential abuse scenarios, like the bruteforce attack mentioned earlier. For example, if a service is trying to detokenize a lot of tokens that don't exist, then this could be suspicious behavior. You want to have monitoring set up to alert you about situations like this.

# Tokenization Alone (Even Done Right) is Not Enough!

As we have shown above, tokenization is a powerful tool. If used correctly as described above, it can make your data security and privacy problems much more tractable. However, like any tool, it only takes you so far.

Arguably the biggest limitation has to do with the fact that we are mostly dealing with each piece of data as if they are independent elements, and

opposed to parts of a record. Let's take an example – let's say you're running a service that caters to monsters.

Your user table looks like this:

ID	Monster	Country
1	Gojira	Japan
2	Dracula	Transylvania
2	The Thing	USA

*A User Data Table With Sensitive Information: Plaintext Monster Names*

Like the rest of us, these fictional monsters value their privacy, so they want us to treat their names as sensitive data. After tokenizing, the monster names look as follows:

ID	Monster	Country
1	ff9-b5-4e-e8d54	Japan
2	4bf-0c-41-f62c3	Transylvania
2	516-f1-4f-e0447	USA





## *A User Data Table With Tokenized Monster Names*

In accordance with local regulations, we want customer support personnel to only access identifying information for monsters in their own locality (so these monsters' PII doesn't leave their country).

To do this, we want to express a rule that says “only allow a token to be detokenized if the country field of the row matches the country of the customer service agent”. You can't evaluate this rule if you only store tokens for the names without any additional context. In other words, when you're trying to detokenize “ff9-b5-4e-e8d54” under the locality-specific rule above, you do not know which country's resident (or monster) this token belongs to.

What you need is a **data privacy vault**, not just a token table. A data privacy vault is much more than just a token table, or just a database. It is a combination of the capabilities of a token table, a database, and a set of governance rules thrown in that lets you evaluate things like the scenario just presented. It isolates and protects sensitive data, while ensuring that sensitive data remains usable for critical workflows.

# Tokenization with Skyflow

Skyflow's Data Privacy Vault has several capabilities that help you to meet your data privacy and security goals. One of the privacy features that

Skyflow supports is tokenization. The Skyflow tokenization solution consists of three parts:

- A Skyflow Data Privacy Vault serves as the trusted third-party cloud service that stores your sensitive data and gives you tokens in return. The vault is your tokenization system.
- Skyflow SDKs help you securely collect sensitive data client-side, insulating your frontend and backend code from handling sensitive data.
- **Skyflow Connections** helps you tokenize data in downstream APIs, so you can pass tokens to services such as Stripe, Twilio, Alloy, etc.

When you create a vault and define a schema in Skyflow, each column you create for your schema has options for tokenization. If you're using built-in **Skyflow data types**, then a tokenization format is already configured for you by default.

For example, in the image below, when creating a column for the built-in email data type, a deterministic format-preserving tokenization method is already set. The format is based on a pre-configured regular expression



Products ▼

Solutions ▼

Company ▼

Blog

Docs

Get a Demo

@ email STRING

**Enable tokenization on this column**

Turning this on will generate tokens for this column. Turning this off means that no tokens will be generated.

Choose the tokenization type for this field

**UUID Deterministic Token**

A persistent UUID token will be generated for a given value. All future occurrences of this value will generate the same token. For example, the value 'johndoe@gmail.com' will always generate the token 'c7db3f3a-5d01-4a98-961e-9cbdb6241b0d' for a given vault.

**UUID Token**

A random token will be generated each time for a given value. For example, a random token for 'johndoe@gmail.com' could be 'c7db3f3a-5d01-4a98-961e-9cbdb6241b0d'. Each future occurrence of the value 'johndoe@gmail.com' will create a different random token.

**Format Preserving Deterministic Token**

A persistent, format preserving token will be generated for a given value. An optional regex can be specified to structure the token format otherwise the format will be inferred based on the value. All future occurrences of this value will generate the same token for a given regex. For example, a format preserving token for 'johndoe@gmail.com' will always generate the token 'bwe09f@fg7d8.com'.

Format preserving regular expression

Cancel

Create Column 

## Tokenization Settings for Skyflow's Predefined **email** Data Type

Tokens are generated automatically when inserting records into your vault based on the tokenization properties you define for the schema.

For example, in the image below, the first name, last name, email, zip code, and phone number of a customer is securely stored in the **customers** table within your Skyflow Vault. The API call returns a JSON object with a tokenized version of the data. The first name, last name, and zip code are tokenized as a UUID while the email and phone number return a format-preserving token.

### Inserting Real Values and Requesting Tokens

```
const skyflowClient = Skyflow.init({
  vaultID: 'my-skyflow-id',
  vaultURL: 'https://my-workspace.vault.skyflowapis.com',
  getBearerToken: generateBearerToken,
});

let response = await skyflowClient.insert({
  records: [
    {
      fields: {
        first_name: 'Jane',
        last_name: 'Doe',
        email: 'jane.doe@gmail.com',
        zip_code: '90210',
        phone_number: '+1 (405) 555-8987'
      },
      table: 'customers'
    }
  ],
  { tokens: true }
});
```

### Customers Table in the Skyflow Vault

skyflow_id	first_name	last_name	email
671c2daf-bec...	J***ne	D***oe	j*****oe@gmail.com

### Tokenized Data

```
{
  "records": [{
    "skyflow_id": "543b3500-200f-44b4-a336-b286bc3a68b0",
    "tokens": {
      "first_name": "80ecd7d2-8d2a-496e-b749-4bb815c7a05b",
      "last_name": "34cb211c-63c5-4b0f-905e-b30e9d0a1228",
      "email": "afdab3ee-ebaa@dsa9998-12b96ad61709.com",
      "zip_code": "f1c64449-17d4-430f-b51e-9281264c3e04",
      "phone_number": "+4 (234) 893-1904"
    }
  }]
}
```

*Example of Inserting Data into a Vault and Receiving Tokens Back*

Additionally, Skyflow's full-featured Data Privacy Vault provides out of the box support for data governance, polymorphic encryption, data residency, and many other features not solved purely by tokenization.

You can read more about Skyflow's support for tokenization in our [developer documentation](#).

## Final Thoughts

Storing sensitive data on your company's infrastructure is a compliance and security burden. The data ends up being copied and passed around, making it nearly impossible to answer questions like: What sensitive data are we storing? Why are we storing it? How do we delete it? And, where is it stored?

Tokenization solves this problem. Tokenization is a method for swapping sensitive data for pseudorandom tokens that don't have exploitable value. The tokenized data is then stored in your application database, while the original sensitive data is stored within a secure, isolated data store like a Skyflow Data Privacy Vault.

By using tokenization, you're insulating your infrastructure from storing sensitive data and greatly reducing your compliance burden. However, it's important to consider the tokenization approach and the potential security tradeoffs of different tokenization approaches with regard to space and entropy.

Finally, tokenization should be just one tool in your data privacy toolbox. A holistic approach requires other tools like data governance, data residency support, encryption, and key rotation – just to name a few.

You could build all of these features yourself, or you could get them all via Skyflow's simple API that makes it easy to protect the privacy of sensitive data. You can try out Skyflow's tokenization and other features when you sign up to [Try Skyflow](#).

Product

Company

[Get a Demo](#)

Docs

Jobs

Blog

Press

Security

Events

Newsletter



[Privacy Policy](#)

[Terms of Service](#)

[Cookie Policy](#)

© 2022 Skyflow, Inc. All rights reserved.