De-identifying sensitive data

Cloud Data Loss Prevention (DLP) can de-identify sensitive data in text content, including text stored in container structures such as tables. De-identification is the process of removing identifying information from data. The API detects sensitive data such as personally identifiable information (PII), and then uses a de-identification transformation to mask, delete, or otherwise obscure the data. For example, de-identification techniques can include any of the following:

- Masking sensitive data by partially or fully replacing characters with a symbol, such as an asterisk (*) or hash (#).
- Replacing each instance of sensitive data with a token, or surrogate, string.
- Encrypting and replacing sensitive data using a randomly generated or pre-determined key.

When you de-identify data using the `CryptoReplaceFfxFpeConfig` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#cryptoreplaceffxfpeconfig) or `CryptoDeterministicConfig` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#cryptodeterministicconfig) infoType transformations, you can re-identify that data, as long as you have the `CryptoKey` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#cryptokey) used to originally de-identify the data.

You can feed information to the API using JSON over HTTPS, as well as the CLI and several programming languages using the DLP client libraries (/dlp/docs/reference/libraries). To set up the CLI, refer to the quickstart (/dlp/docs/quickstart). For more information about submitting information in JSON format, see the JSON quickstart (/dlp/docs/quickstart-json).

API overview

To de-identify sensitive data, use Cloud DLP's `content.deidentify` (/dlp/docs/reference/rest/v2/projects.content/deidentify) method.

There are three parts to a de-identification API call:

- *The data to inspect:* A string or table structure (**`ContentItem`** (/dlp/docs/reference/rest/v2/ContentItem) object) for the API to inspect.
- *What to inspect for:* Detection configuration information (**`InspectConfig`** (/dlp/docs/reference/rest/v2/InspectConfig)) such as what types of data (or *infoTypes* (/dlp/docs/concepts-infotypes)) to look for, whether to filter findings that are above a certain likelihood threshold, whether to return no more than a certain number of results, and so on. Not specifying at least one infoType in an `InspectConfig` argument is equivalent to specifying all built-in infoTypes. Doing so is not recommended, as it can cause decreased performance and increased cost.
- *What to do with the inspection findings:* Configuration information (**`DeidentifyConfig`** (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#DeidentifyTemplate.DeidentifyConfi g)) that defines how you want the sensitive data de-identified. This argument is covered in more detail in the following section.

The API returns the same items you gave it, in the same format, but any text identified as containing sensitive information according to your criteria has been de-identified.

Specifying detection criteria

Information type (or "infoType") detectors are the mechanisms that Cloud DLP uses to find sensitive data.

Cloud DLP includes several kinds of infoType detectors, all of which are summarized here:

- *Built-in infoType detectors* (#built-in) are built into Cloud DLP. They include detectors for country- or region-specific sensitive data types as well as globally applicable data types.
- *Custom infoType detectors* (#custom) are detectors that you create yourself. There are three kinds of custom infoType detectors:
  - *Regular custom dictionary detectors* are simple word lists that Cloud DLP matches on. Use regular custom dictionary detectors when you have a list of up to several tens of thousands of words or phrases. Regular custom dictionary detectors are preferred if you don't anticipate your word list changing significantly.
  - *Stored custom dictionary detectors* are generated by Cloud DLP using large lists of words or phrases stored in either Cloud Storage or BigQuery. Use stored custom

dictionary detectors when you have a large list of words or phrases—up to tens of millions.

- *Regular expressions (regex) detectors* enable Cloud DLP to detect matches based on a regular expression pattern.

In addition, Cloud DLP includes the concept of *inspection rules* (#inspection-rules), which enable you to fine-tune scan results using the following:

- *Exclusion rules* enable you to decrease the number of findings returned by adding rules to a built-in or custom infoType detector.
- *Hotword rules* enable you to increase the quantity or change the likelihood value (/dlp/docs/likelihood) of findings returned by adding rules to a built-in or custom infoType detector.

De-identification transformations

You must specify one or more transformations when you set the de-identification configuration (`DeidentifyConfig` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#DeidentifyTemplate.DeidentifyConfig)). There are two categories of transformations:

- `InfoTypeTransformations` (/dlp/docs/deidentify-sensitive-data#infotype_transformations): Transformations that are only applied to values within submitted text that are identified as a specific infoType.
- `RecordTransformations` (/dlp/docs/deidentify-sensitive-data#record_transformations): Transformations that are only applied to values within submitted tabular text data that are identified as a specific infoType, or on an entire column of tabular data.

InfoType transformations

You can specify one or more infoType transformations per request. Within each `InfoTypeTransformation` object, you specify both of the following:

- One or more infoTypes (/dlp/docs/infotypes-reference) to which a transformation should be applied (the `infoTypes[]` array object).

- A primitive transformation (the `PrimitiveTransformation` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#primitivetransformation) object).

Note that specifying an infoType is optional, but not specifying at least one infoType in an `InspectConfig` (/dlp/docs/reference/rest/v2/InspectConfig) argument causes the transformation to apply to all built-in infoTypes that don't have a transformation provided. Doing so is not recommended, as it can cause decreased performance and increased cost.

Primitive transformations

You must specify at least one primitive transformation to apply to input, regardless of whether you're applying it only to certain infoTypes or to the entire text string. You have several transformation options, which are summarized in the following table. Click the object name for more information.

The full list of possible transformations (/dlp/docs/transformations-reference).

replaceConfig

Setting `replaceConfig` to a `ReplaceValueConfig` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#replacevalueconfig) object replaces matched input values with a value you specify.

For example, suppose you’ve set `replaceConfig` to "`[email-address]`" for all `EMAIL_ADDRESS` infoTypes, and the following string is sent to Cloud DLP:

```
me is Alicia Abernathy, and my email address is aabernathy@example.com.
```

The returned string will be the following:

```
me is Alicia Abernathy, and my email address is [email-address].
```

The following JSON example and code in several languages shows how to form the API request and what the Cloud DLP API returns:

:ant: The code on this page requires that you first set up a Cloud DLP client. For more information about instal
eating a Cloud DLP client, see Cloud DLP client libraries (/dlp/docs/reference/libraries). (Sending JSON to Clou
EST endpoints does not require a client library.)

ProtocolPython (#python)Java (#java)

See the JSON quickstart (/dlp/docs/quickstart-json) for more information about using the Cloud DLP API with JSON.

JSON Input:

```
POST https://dlp.googleapis.com/v2/projects/[PROJECT_ID]/content:deidentify?key={YOU

{
  "item":{
    "value":"My name is Alicia Abernathy, and my email address is aabernathy@example
  },
  "deidentifyConfig":{
    "infoTypeTransformations":{
      "transformations":[
        {
          "infoTypes":[
            {
              "name":"EMAIL_ADDRESS"
            }
          ],
          "primitiveTransformation":{
            "replaceConfig":{
              "newValue":{
                "stringValue":"[email-address]"
              }
            }
          }
        }
      ]
    }
  },
  "inspectConfig":{
    "infoTypes":[
      {
        "name":"EMAIL_ADDRESS"
      }
    ]
  }
}
```

JSON Output:

```
{
  "item":{
    "value":"My name is Alicia Abernathy, and my email address is [email-address]."
  },
  "overview":{
    "transformedBytes":"22",
    "transformationSummaries":[
      {
        "infoType":{
          "name":"EMAIL_ADDRESS"
        },
        "transformation":{
          "replaceConfig":{
            "newValue":{
              "stringValue":"[email-address]"
            }
          }
        },
        "results":[
          {
            "count":"1",
            "code":"SUCCESS"
          }
        ],
        "transformedBytes":"22"
      }
    ]
  }
}
```

redactConfig

Specifying `redactConfig` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#redactconfig) redacts a given value by removing it completely. The `redactConfig` message has no arguments; specifying it enables its transformation.

For example, suppose you've specified `redactConfig` for all `EMAIL_ADDRESS` infoTypes, and the following string is sent to Cloud DLP:

```
me is Alicia Abernathy, and my email address is aabernathy@example.com.
```

The returned string will be the following:

```
me is Alicia Abernathy, and my email address is .
```

The following examples show how to form the API request and what the Cloud DLP API returns:

tant: The code on this page requires that you first set up a Cloud DLP client. For more information about instal eating a Cloud DLP client, see Cloud DLP client libraries (/dlp/docs/reference/libraries). (Sending JSON to Clou EST endpoints does not require a client library.)

ProtocolJava (#java)Python (#python)

JSON Input:

```
POST https://dlp.googleapis.com/v2/projects/[PROJECT_ID]/content:deidentify?key={YOU

{
  "item":{
    "value":"My name is Alicia Abernathy, and my email address is aabernathy@example
  },
  "deidentifyConfig":{
    "infoTypeTransformations":{
      "transformations":[
        {
          "infoTypes":[
            {
              "name":"EMAIL_ADDRESS"
            }
          ],
          "primitiveTransformation":{
            "redactConfig":{

            }
          }
        }
      ]
    }
  },
  "inspectConfig":{
    "infoTypes":[
      {
        "name":"EMAIL_ADDRESS"
      }
    ]
  }
}
```

JSON Output:

```
{
  "item":{
    "value":"My name is Alicia Abernathy, and my email address is ."
  },
  "overview":{
    "transformedBytes":"22",
    "transformationSummaries":[
      {
        "infoType":{
          "name":"EMAIL_ADDRESS"
        },
        "transformation":{
          "redactConfig":{

          }
        },
        "results":[
          {
            "count":"1",
            "code":"SUCCESS"
          }
        ],
        "transformedBytes":"22"
      }
    ]
  }
}
```

characterMaskConfig

Setting `characterMaskConfig` to a `CharacterMaskConfig` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#charactermaskconfig) object partially masks a string by replacing a given number of characters with a fixed character. Masking can start from the beginning or end of the string. This transformation also works with number types such as long integers.

The `CharacterMaskConfig` object has several of its own arguments:

- `maskingCharacter`: The character to use to mask each character of a sensitive value. For example, you could specify an asterisk (*) or hash (#) to mask a series of numbers such as those in a credit card number.
- `numberToMask`: The number of characters to mask. If you don't set this value, all matching characters will be masked.
- `reverseOrder`: Whether to mask characters in reverse order. Setting `reverseOrder` to true causes characters in matched values to be masked from the end toward the beginning of the value. Setting it to false causes masking to begin at the start of the value.
- `charactersToIgnore[]`: One or more characters to skip when masking values. For example, specify a hyphen here to leave the hyphens in place when masking a telephone number. You can also specify a group of common characters (`CharsToIgnore` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#DeidentifyTemplate.CharsToIgnore)
) to ignore when masking.

For example, suppose you've set `characterMaskConfig` to mask with '#' for `EMAIL_ADDRESS` infotypes, except for the '.' and '@' characters. If the following string is sent to Cloud DLP:

```
me is Alicia Abernathy, and my email address is aabernathy@example.com.
```

The returned string will be the following:

```
me is Alicia Abernathy, and my email address is ##########@#######.###.
```

Following are examples that demonstrate how to use the Cloud DLP API to de-identify sensitive data using masking techniques.

:ant: The code on this page requires that you first set up a Cloud DLP client. For more information about instal eating a Cloud DLP client, see Cloud DLP client libraries (/dlp/docs/reference/libraries). (Sending JSON to Clou EST endpoints does not require a client library.)

ProtocolJava (#java)Node.js (#node.js)Python (#python)Go (#go)PHP (#php)C# (#c)

The following JSON example shows how to form the API request and what the Cloud DLP API returns:

JSON Input:

```
POST https://dlp.googleapis.com/v2/projects/[PROJECT_ID]/content:deidentify?key={YOU

{
  "item":{
    "value":"My name is Alicia Abernathy, and my email address is aabernathy@example
  },
  "deidentifyConfig":{
    "infoTypeTransformations":{
      "transformations":[
        {
          "infoTypes":[
            {
              "name":"EMAIL_ADDRESS"
            }
          ],
          "primitiveTransformation":{
            "characterMaskConfig":{
              "maskingCharacter":"#",
              "reverseOrder":false,
              "charactersToIgnore":[
                {
```

```
                "charactersToSkip":".@"
              }
            ]
          }
        }
      }
    ]
  }
},
"inspectConfig":{
  "infoTypes":[
    {
      "name":"EMAIL_ADDRESS"
    }
  ]
}
}
```

JSON Output:

```
{
  "item":{
    "value":"My name is Alicia Abernathy, and my email address is ##########@#######
  },
  "overview":{
    "transformedBytes":"22",
    "transformationSummaries":[
      {
        "infoType":{
          "name":"EMAIL_ADDRESS"
        },
        "transformation":{
          "characterMaskConfig":{
            "maskingCharacter":"#",
            "charactersToIgnore":[
              {
                "charactersToSkip":".@"
              }
            ]
          }
        },
        "results":[
          {
            "count":"1",
            "code":"SUCCESS"
          }
        ],
        "transformedBytes":"22"
      }
    ]
  }
}
```

cryptoHashConfig

Setting `cryptoHashConfig` to a `CryptoHashConfig` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#cryptohashconfig) object performs pseudonymization (/dlp/docs/pseudonymization) on an input value by generating a surrogate value using cryptographic hashing.

This method replaces the input value with an encrypted "digest," or hash value. The digest is computed by taking the SHA-256 hash of the input value. The cryptographic key used to make the hash is a `CryptoKey` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#cryptokey) object, and must be either 32 or 64 bytes in size.

The method outputs a base64-encoded representation of the hashed output. Currently, only string and integer values can be hashed.

For example, suppose you've specified `cryptoHashConfig` for all `EMAIL_ADDRESS` infoTypes, and the `CryptoKey` object consists of a randomly-generated key (a `TransientCryptoKey` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#DeidentifyTemplate.TransientCryptoKey)). Then, the following string is sent to Cloud DLP:

```
me is Alicia Abernathy, and my email address is aabernathy@example.com.
```

The cryptographically generated returned string will look like the following:

```
me is Alicia Abernathy, and my email address is 41D1567F7F99F1DC2A5FAB886DEE5BEE.
```

Of course, the hex string will be cryptographically generated and different from the one shown here.

dateShiftConfig

Setting `dateShiftConfig` to a `DateShiftConfig` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#DeidentifyTemplate.DateShiftConfig) object performs date shifting (/dlp/docs/concepts-date-shifting) on a date input value by shifting the dates by a random number of days.

Date shifting techniques randomly shift a set of dates but preserve the sequence and duration of a period of time. Shifting dates is usually done in context to an individual or an entity. That is, you want to shift all of the dates for a specific individual using the same shift differential, but use a separate shift differential for each other individual.

For more information about date shifting, see the date shifting concept topic (/dlp/docs/concepts-date-shifting).

Following is sample code in several languages that demonstrates how to use the Cloud DLP API to de-identify dates using date shifting.

tant: The code on this page requires that you first set up a Cloud DLP client. For more information about instal eating a Cloud DLP client, see Cloud DLP client libraries (/dlp/docs/reference/libraries). (Sending JSON to Clou EST endpoints does not require a client library.)

JavaNode.js (#node.js)Python (#python)Go (#go)PHP (#php)C# (#c)

gleapis/java-dlp/blob/master/samples/snippets/src/main/java/dlp/snippets/DeIdentifyWithDateShift.java)

```
import com.google.cloud.dlp.v2.DlpServiceClient;
import com.google.common.base.Splitter;
import com.google.privacy.dlp.v2.ContentItem;
import com.google.privacy.dlp.v2.DateShiftConfig;
import com.google.privacy.dlp.v2.DeidentifyConfig;
import com.google.privacy.dlp.v2.DeidentifyContentRequest;
import com.google.privacy.dlp.v2.DeidentifyContentResponse;
import com.google.privacy.dlp.v2.FieldId;
```

```java
import com.google.privacy.dlp.v2.FieldTransformation;
import com.google.privacy.dlp.v2.LocationName;
import com.google.privacy.dlp.v2.PrimitiveTransformation;
import com.google.privacy.dlp.v2.RecordTransformations;
import com.google.privacy.dlp.v2.Table;
import com.google.privacy.dlp.v2.Value;
import com.google.type.Date;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class DeIdentifyWithDateShift {

  public static void main(String[] args) throws Exception {
    // TODO(developer): Replace these variables before running the sample.
    String projectId = "your-project-id";
    Path inputCsvFile = Paths.get("path/to/your/input/file.csv");
    Path outputCsvFile = Paths.get("path/to/your/output/file.csv");
    deIdentifyWithDateShift(projectId, inputCsvFile, outputCsvFile);
  }

  public static void deIdentifyWithDateShift(
      String projectId, Path inputCsvFile, Path outputCsvFile) throws IOException {
    // Initialize client that will be used to send requests. This client only needs
    // once, and can be reused for multiple requests. After completing all of your r
    // the "close" method on the client to safely clean up any remaining background
    try (DlpServiceClient dlp = DlpServiceClient.create()) {
      // Read the contents of the CSV file into a Table
      List<FieldId> headers;
      List<Table.Row> rows;
      try (BufferedReader input = Files.newBufferedReader(inputCsvFile)) {
        // Parse and convert the first line into header names
        headers =
            Arrays.stream(input.readLine().split(","))
                .map(header -> FieldId.newBuilder().setName(header).build())
                .collect(Collectors.toList());
        // Parse the remainder of the file as Table.Rows
        rows =
            input.lines().map(DeIdentifyWithDateShift::parseLineAsRow).collect(Colle
      }
      Table table = Table.newBuilder().addAllHeaders(headers).addAllRows(rows).build
      ContentItem item = ContentItem.newBuilder().setTable(table).build();

      // Set the maximum days to shift dates backwards (lower bound) or forward (upp
      DateShiftConfig dateShiftConfig =
          DateShiftConfig.newBuilder().setLowerBoundDays(5).setUpperBoundDays(5).bui
      PrimitiveTransformation transformation =
          PrimitiveTransformation.newBuilder().setDateShiftConfig(dateShiftConfig).b
      // Specify which fields the DateShift should apply too
      List<FieldId> dateFields = Arrays.asList(headers.get(1), headers.get(3));
      FieldTransformation fieldTransformation =
          FieldTransformation.newBuilder()
              .addAllFields(dateFields)
              .setPrimitiveTransformation(transformation)
              .build();
      RecordTransformations recordTransformations =
          RecordTransformations.newBuilder().addFieldTransformations(fieldTransforma
      // Specify the config for the de-identify request
      DeidentifyConfig deidentifyConfig =
          DeidentifyConfig.newBuilder().setRecordTransformations(recordTransformatio

      // Combine configurations into a request for the service.
      DeidentifyContentRequest request =
          DeidentifyContentRequest.newBuilder()
              .setParent(LocationName.of(projectId, "global").toString())
              .setItem(item)
              .setDeidentifyConfig(deidentifyConfig)
              .build();

      // Send the request and receive response from the service
      DeidentifyContentResponse response = dlp.deidentifyContent(request);

      // Write the results to the target CSV file
      try (BufferedWriter writer = Files.newBufferedWriter(outputCsvFile)) {
        Table outTable = response.getItem().getTable();
        String headerOut =
            outTable.getHeadersList().stream()
                .map(FieldId::getName)
                .collect(Collectors.joining(","));
        writer.write(headerOut + "\n");

        List<String> rowOutput =
            outTable.getRowsList().stream()
                .map(row -> joinRow(row.getValuesList()))
                .collect(Collectors.toList());
        for (String line : rowOutput) {
          writer.write(line + "\n");
        }
        System.out.println("Content written to file: " + outputCsvFile.toString());
      }
    }
  }

  // Convert the string from the csv file into com.google.type.Date
  public static Date parseAsDate(String s) {
    LocalDate date = LocalDate.parse(s, DateTimeFormatter.ofPattern("MM/dd/yyyy"));
    return Date.newBuilder()
        .setDay(date.getDayOfMonth())
        .setMonth(date.getMonthValue())
        .setYear(date.getYear())
        .build();
  }

  // Each row is in the format: Name,BirthDate,CreditCardNumber,RegisterDate
  public static Table.Row parseLineAsRow(String line) {
    List<String> values = Splitter.on(",").splitToList(line);
    Value name = Value.newBuilder().setStringValue(values.get(0)).build();
    Value birthDate = Value.newBuilder().setDateValue(parseAsDate(values.get(1))).bu
    Value creditCardNumber = Value.newBuilder().setStringValue(values.get(2)).build(
    Value registerDate = Value.newBuilder().setDateValue(parseAsDate(values.get(3)))
    return Table.Row.newBuilder()
        .addValues(name)
        .addValues(birthDate)
        .addValues(creditCardNumber)
        .addValues(registerDate)
        .build();
  }

  public static String formatDate(Date d) {
    return String.format("%s/%s/%s", d.getMonth(), d.getDay(), d.getYear());
  }

  public static String joinRow(List<Value> values) {
    String name = values.get(0).getStringValue();
    String birthDate = formatDate(values.get(1).getDateValue());
    String creditCardNumber = values.get(2).getStringValue();
    String registerDate = formatDate(values.get(3).getDateValue());
```

```
    return String.join(",", name, birthDate, creditCardNumber, registerDate);
  }
}
```

cryptoReplaceFfxFpeConfig

Setting `cryptoReplaceFfxFpeConfig` to a `CryptoReplaceFfxFpeConfig` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#cryptoreplaceffxfpeconfig) object performs pseudonymization (/dlp/docs/pseudonymization) on an input value by replacing an input value with a token. This token is:

- The encrypted input value.
- The same length as the input value.
- Computed using format-preserving encryption in FFX mode ("FPE-FFX") keyed on the cryptographic key specified by `cryptoKey`.
- Comprised of the characters specified by `alphabet`. Valid options:
  - `NUMERIC`
  - `HEXADECIMAL`
  - `UPPER_CASE_ALPHA_NUMERIC`
  - `ALPHA_NUMERIC`

The input value:

- Must be at least two characters long (or the empty string).
- Must be comprised of the characters specified by an `alphabet`. The `alphabet` can be comprised of between 2 and 95 characters. (An `alphabet` with 95 characters includes all printable characters in the US-ASCII character set.)

Cloud DLP computes the replacement token using a cryptographic key. You provide this key in one of three ways:

1. By requesting that the Cloud DLP generate it.
2. By embedding it encrypted in the API request. For this option, the key is wrapped (encrypted) by a Cloud Key Management Service (Cloud KMS) key.

3. By embedding it unencrypted in the API request. (Not recommended.)

To create a Cloud KMS wrapped key, you send a request containing a 16-, 24-, or 32-byte `plaintext` field value to the Cloud KMS `projects.locations.keyRings.cryptoKeys.encrypt` (/kms/docs/reference/rest/v1/projects.locations.keyRings.cryptoKeys/encrypt) method. The wrapped key is the value in the `ciphertext` field of the method's response.

The value is a base64-encoded string by default. To set this value in Cloud DLP, it must be decoded into a byte string. The following code snippets highlight how to do this in several languages. End-to-end examples are provided following these snippets.

tant: The code on this page requires that you first set up a Cloud DLP client. For more information about instal eating a Cloud DLP client, see Cloud DLP client libraries (/dlp/docs/reference/libraries). (Sending JSON to Clou EST endpoints does not require a client library.)

JavaPython (#python)PHP (#php)C# (#c)

```
KmsWrappedCryptoKey.newBuilder()
    .setWrappedKey(ByteString.copyFrom(BaseEncoding.base64().decode(wrappedKey)))
```

For more information about encrypting and decrypting data using Cloud KMS, see Encrypting and Decrypting Data (/kms/docs/encrypt-decrypt).

By design, FPE-FFX preserves the length and character set of the input text. This means that it lacks authentic initialization vector, which would cause a length expansion in the output token. Other methods like AES-SIV docs/reference/rest/v2/organizations.deidentifyTemplates#cryptodeterministicconfig) provide these stronge ty guarantees and are recommended for tokenization use cases unless length and character set preservation equirements—for example, for backward compatibility with a legacy data system.

Following is sample code in several languages that demonstrates how to use Cloud DLP to de-identify sensitive data by replacing an input value with a token.

JavaNode.js (#node.js)Python (#python)Go (#go)PHP (#php)C# (#c) ı/googleapis/java-dlp/blob/master/samples/snippets/src/main/java/dlp/snippets/DeIdentifyWithFpe.java)

```
import com.google.cloud.dlp.v2.DlpServiceClient;
import com.google.common.io.BaseEncoding;
import com.google.privacy.dlp.v2.ContentItem;
import com.google.privacy.dlp.v2.CryptoKey;
import com.google.privacy.dlp.v2.CryptoReplaceFfxFpeConfig;
import com.google.privacy.dlp.v2.CryptoReplaceFfxFpeConfig.FfxCommonNativeAlphabet;
import com.google.privacy.dlp.v2.DeidentifyConfig;
import com.google.privacy.dlp.v2.DeidentifyContentRequest;
import com.google.privacy.dlp.v2.DeidentifyContentResponse;
import com.google.privacy.dlp.v2.InfoType;
import com.google.privacy.dlp.v2.InfoTypeTransformations;
import com.google.privacy.dlp.v2.InfoTypeTransformations.InfoTypeTransformation;
import com.google.privacy.dlp.v2.InspectConfig;
import com.google.privacy.dlp.v2.KmsWrappedCryptoKey;
import com.google.privacy.dlp.v2.LocationName;
import com.google.privacy.dlp.v2.PrimitiveTransformation;
import com.google.protobuf.ByteString;
import java.io.IOException;
import java.util.Arrays;

public class DeIdentifyWithFpe {

  public static void main(String[] args) throws Exception {
    // TODO(developer): Replace these variables before running the sample.
    String projectId = "your-project-id";
    String textToDeIdentify = "I'm Gary and my email is gary@example.com";
    String kmsKeyName =
        "projects/YOUR_PROJECT/"
            + "locations/YOUR_KEYRING_REGION/"
            + "keyRings/YOUR_KEYRING_NAME/"
            + "cryptoKeys/YOUR_KEY_NAME";
    String wrappedAesKey = "YOUR_ENCRYPTED_AES_256_KEY";
    deIdentifyWithFpe(projectId, textToDeIdentify, kmsKeyName, wrappedAesKey);
  }

  public static void deIdentifyWithFpe(
      String projectId, String textToDeIdentify, String kmsKeyName, String wrappedAe
      throws IOException {
    // Initialize client that will be used to send requests. This client only needs
    // once, and can be reused for multiple requests. After completing all of your r
    // the "close" method on the client to safely clean up any remaining background
    try (DlpServiceClient dlp = DlpServiceClient.create()) {
      // Specify what content you want the service to DeIdentify
      ContentItem contentItem = ContentItem.newBuilder().setValue(textToDeIdentify).

      // Specify the type of info the inspection will look for.
      // See https://cloud.google.com/dlp/docs/infotypes-reference for complete list
      InfoType infoType = InfoType.newBuilder().setName("US_SOCIAL_SECURITY_NUMBER")
      InspectConfig inspectConfig =
          InspectConfig.newBuilder().addAllInfoTypes(Arrays.asList(infoType)).build(

      // Specify an encrypted AES-256 key and the name of the Cloud KMS key that enc
      KmsWrappedCryptoKey kmsWrappedCryptoKey =
          KmsWrappedCryptoKey.newBuilder()
              .setWrappedKey(ByteString.copyFrom(BaseEncoding.base64().decode(wrappe
              .setCryptoKeyName(kmsKeyName)
              .build();
      CryptoKey cryptoKey = CryptoKey.newBuilder().setKmsWrapped(kmsWrappedCryptoKey

      // Specify how the info from the inspection should be encrypted.
      InfoType surrogateInfoType = InfoType.newBuilder().setName("SSN_TOKEN").build(
      CryptoReplaceFfxFpeConfig cryptoReplaceFfxFpeConfig =
          CryptoReplaceFfxFpeConfig.newBuilder()
              .setCryptoKey(cryptoKey)
              // Set of characters in the input text. For more info, see
              // https://cloud.google.com/dlp/docs/reference/rest/v2/organizations.c
              .setCommonAlphabet(FfxCommonNativeAlphabet.NUMERIC)
              .setSurrogateInfoType(surrogateInfoType)
              .build();
      PrimitiveTransformation primitiveTransformation =
          PrimitiveTransformation.newBuilder()
              .setCryptoReplaceFfxFpeConfig(cryptoReplaceFfxFpeConfig)
              .build();
      InfoTypeTransformation infoTypeTransformation =
          InfoTypeTransformation.newBuilder()
              .setPrimitiveTransformation(primitiveTransformation)
              .build();
      InfoTypeTransformations transformations =
          InfoTypeTransformations.newBuilder().addTransformations(infoTypeTransforma

      DeidentifyConfig deidentifyConfig =
          DeidentifyConfig.newBuilder().setInfoTypeTransformations(transformations).

      // Combine configurations into a request for the service.
      DeidentifyContentRequest request =
          DeidentifyContentRequest.newBuilder()
```

```
              .setParent(LocationName.of(projectId, "global").toString())
              .setItem(contentItem)
              .setInspectConfig(inspectConfig)
              .setDeidentifyConfig(deidentifyConfig)
              .build();

      // Send the request and receive response from the service
      DeidentifyContentResponse response = dlp.deidentifyContent(request);

      // Print the results
      System.out.println(
          "Text after format-preserving encryption: " + response.getItem().getValue(
    }
  }
}
```

fixedSizeBucketingConfig

The bucketing transformations—this one and `bucketingConfig` (#bucketingConfig)—serve to mask numerical data by "bucketing" it into ranges. The resulting number range is a hyphenated string consisting of a lower bound, a hyphen, and an upper bound.

Setting `fixedSizeBucketingConfig` to a `FixedSizeBucketingConfig` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#fixedsizebucketingconfig) object buckets input values based on fixed size ranges. The `FixedSizeBucketingConfig` object consists of the following:

- `lowerBound`: The lower bound value of all of the buckets. Values less than this one are grouped together in a single bucket.
- `upperBound`: The upper bound value of all of the buckets. Values greater than this one are grouped together in a single bucket.
- `bucketSize`: The size of each bucket other than the minimum and maximum buckets.

For example, if `lowerBound` is set to 10, `upperBound` is set to 89, and `bucketSize` is set to 10, then the following buckets would be used: -10, 10-20, 20-30, 30-40, 40-50, 50-60, 60-70, 70-80, 80-89, 89+.

For more information about the concept of bucketing, see Generalization and Bucketing (/dlp/docs/concepts-bucketing).

bucketingConfig

The `bucketingConfig` transformation offers more flexibility than the other bucketing transformation, `fixedSizeBucketingConfig` (#fixedSizeBucketingConfig). Instead of specifying upper and lower bounds and an interval value with which to create equal-sized buckets, you specify the maximum and minimum values for each bucket you want created. Each maximum and minimum value pair must have the same type.

Setting `bucketingConfig` to a `BucketingConfig` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#bucketingconfig) object specifies custom buckets. The `BucketingConfig` object consists of a `buckets[]` array of `Bucket` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#bucket) objects. Each `Bucket` object consists of the following:

- `min`: The lower bound of the bucket's range. Omit this value to create a bucket that has no lower bound.
- `max`: The upper bound of the bucket's range. Omit this value to create a bucket that has no upper bound.
- `replacementValue`: The value with which to replace values that fall within the lower and upper bounds. If you don't provide a `replacementValue`, a hyphenated `min-max` range will be used instead.

If a value falls outside of the defined ranges, the `TransformationSummary` (/dlp/docs/reference/rest/v2/TransformationOverview#transformationsummary) returned will contain an error message.

For example, consider the following configuration for the `bucketingConfig` transformation:

```
etingConfig":{
ckets":[

 "min":{
   "integerValue":"1"
 },
 "max":{
   "integerValue":"30"
 },
 "replacementValue":{
   "stringValue":"LOW"
 }
,

 "min":{
   "integerValue":"31"
 },
 "max":{
   "integerValue":"65"
 },
 "replacementValue":{
   "stringValue":"MEDIUM"
 }
,

 "min":{
   "integerValue":"66"
 },
 "max":{
   "integerValue":"100"
 },
 "replacementValue":{
   "stringValue":"HIGH"
 }
```

This defines the following behavior:

- Integer values falling between 1 and 30 are masked by being replaced with `LOW`.
- Integer values falling between 31-65 are masked by being replaced with `MEDIUM`.
- Integer values falling between 66-100 are masked by being replaced with `HIGH`.

For more information about the concept of bucketing, see Generalization and Bucketing (/dlp/docs/concepts-bucketing).

replaceWithInfoTypeConfig

Specifying `replaceWithInfoTypeConfig` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#replacewithinfotypeconfig) replaces each matched value with the name of the infoType. The `replaceWithInfoTypeConfig` message has no arguments; specifying it enables its transformation.

For example, suppose you've specified `replaceWithInfoTypeConfig` for all `EMAIL_ADDRESS` infoTypes, and the following string is sent to Cloud DLP:

```
me is Alicia Abernathy, and my email address is aabernathy@example.com.
```

The returned string will be the following:

```
me is Alicia Abernathy, and my email address is EMAIL_ADDRESS.
```

timePartConfig

Setting `timePartConfig` to a `TimePartConfig` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#timepartconfig) object preserves a portion of a matched value that includes `Date`, `Timestamp`, and `TimeOfDay` values. The `TimePartConfig` object consists of a `partToExtract` argument, which can be set to any of the `TimePart` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#timepart) enumerated values, including year, month, day of the month, and so on.

For example, suppose you've configured a `timePartConfig` transformation by setting `partToExtract` to `YEAR`. After sending the data in the first column below to Cloud DLP, you'd end up with the transformed values in the second column:

Original values	Transformed values
9/21/1976	1976
6/7/1945	1945
1/20/2009	2009
7/4/1776	1776
8/1/1984	1984
4/21/1982	1982

Record transformations

Record transformations (the **`RecordTransformations`** (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#recordtransformations) object) are only applied to values within tabular data that are identified as a specific infoType. Within `RecordTransformations`, there are two further subcategories of transformations:

- **`fieldTransformations[]`** (/dlp/docs/deidentify-sensitive-data#field_transformations): Transformations that apply various field transformations.
- **`recordSuppressions[]`** (/dlp/docs/deidentify-sensitive-data#record_suppressions): Rules defining which records get suppressed completely. Records that match any suppression rule within `recordSuppressions[]` are omitted from the output.

Field transformations

Each **`FieldTransformation`** (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#fieldtransformation) object includes three arguments:

- `fields`: One or more input fields (**`FieldID`** (/dlp/docs/reference/rest/v2/FieldId) objects) to apply the transformation to.
- `condition`: A condition (a **`RecordCondition`** (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#recordcondition) object) that must evaluate to true for the transformation to be applied. For example, apply a bucket transformation to an age column of a record only if the ZIP code column for the same record is within a specific range. Or, redact a field only if the birthdate field puts a person's age at 85 or above.
- One of the following two transformation type arguments. Specifying one is required:
  - `infoTypeTransformations`: Treat the contents of the field as free text, and apply a **`PrimitiveTransformation`** (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#primitivetransformation) only to content that matches an **`InfoType`** (/dlp/docs/reference/rest/v2/InfoType). These transformations were discussed earlier in this topic (/dlp/docs/deidentify-sensitive-data#infotype_transformations).
  - `primitiveTransformation`: Apply the specified primitive transformation (**`PrimitiveTransformation`**

(/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#primitivetransformation) object) to the entire field. These transformations were discussed earlier in this topic (/dlp/docs/deidentify-sensitive-data#primitive_transformations).

Record suppressions

In addition to applying transformations to field data, you can also instruct Cloud DLP to de-identify data by simply suppressing records when certain suppression conditions evaluate to true. You can apply both field transformations and record suppressions in the same request.

You set the `recordSuppressions` message of the `RecordTransformations` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#recordtransformations) object to an array of one or more `RecordSuppression` objects.

Each `RecordSuppression` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#recordsuppression) object contains a single `RecordCondition` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#recordcondition) object, which in turn contains a single `Expressions` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#expressions) object.

An `Expressions` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#expressions) object contains:

- `logicalOperator`: One of the `LogicalOperator` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#logicaloperator) enumerated types.
- `conditions`: A `Conditions` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#conditions) object, containing an array of one or more `Condition` (/dlp/docs/reference/rest/v2/organizations.deidentifyTemplates#condition) objects. A `Condition` is a comparison of a field value and another value, both of which be of type `string`, `boolean`, `integer`, `double`, `Timestamp`, or `TimeofDay`.

If the comparison evaluates to true, the record is suppressed, and vice-versa. If the compared values are not the same type, a warning is given and the condition evaluates to false.