SentinelDB Documentation Documentation

Release latest

Apr 03, 2019

SentinelDB documentation

CHAPTER 1

Getting Started

1.1 Overview

SentinelDB is a cloud-based datastore that ensures data protection and compliance with data-protection regulations like GDPR. It implements all data-protection best practices like encryption (per record), secure audit trail, anonymization and pseudonymization of data, two-factor authentication, fraud detection, as well as relying on a secure server infrastructure.

1.2 Main concepts

1.2.1 Personal data

Personal data is any data that can be associated with a person that can be identified. In technical terms this means that once you have a "users" table, any record in any other table that has a "userId" reference, is also personal data (in addition to all the data in the "users" table).

1.2.2 Personal data store

A personal data store is a module that is tasked with storing all personal data in a secure and compliant way. Many organizations choose to organize their internal systems around such a personal data store so that there is a single, well-protected place to keep personal data as opposed to having multiple systems that each has to implement all data protection measures. The personal data store is then accessed by all other systems when personal data is needed, but other systems do not store any personal data. SentinelDB can be seen as a Personal-data-store-as-a-service.

1.2.3 Data de-identification

Data about a natural person, including medical data, should generally be protected. However, systems can choose to remove any characteristics from the data, which can be used to identify a natural person. That way data becomes anonymous and can be used for various purposes. More importantly, it doesn't fall into the category of "personal

data" when regulations (e.g. GDPR and HIPAA) are concerned. HIPAA Article 164.514(b) details how to de-identify medical data. One option is to remove all possible identifiers which renders the data unlinkable to a person.

Data de-identification is a useful mechanism for compliant data architecture. Organizations can store all personal information (all the identifiers that can be used to pinpoint a particular person) in a well-protected personal data store, and store de-identified data in multiple systems outside of that store. In simplified technical words, you keep only the user ID in the "users" table of each system and every other field is moved to a personal data store, where it can be retrieved via that ID.

1.2.4 User profile

User profiles are the core entities of a data-protection focused system. The "users" table (or "user" object) holds all information that can be used to identify a person (names, email, personal identifier, passport number, SSN, address, etc.). Sometimes the structure is more complex, e.g. a user can have multiple addresses, which can be a separate table, but it's part of the same user hierarchy.

1.2.5 Record

Records are pieces of data, parts of a personal data store, that are related to (or "owned by") a user and should also be protected because they may contain identifying or sensitive information. A record can contain no identifying information and still be stored if the organization considers the data to be of high risk and in need of more thorough protection.

Some examples: direct messages can contain any information about the people involved, including identifying information, so it may be considered as a candidate for storage. Scanned documents where the contents are not technically parseable (except via OCR) can be stored as binary records. CVs usually contain personally identifiable information (PII) which cannot be easily stripped, so they can also be stored as records in a personal data store. Any other structured or unstructured data that may contain identifiable information can be stored as a record, associated with a given user.

The hard part that depends on the business case and the particular data in use is what entities to put in the personal data store as records associated with a given user and what to leave in existing systems, outside the personal data store. There is no recipe for this and decisions should be taken on a case-by-case basis.

1.2.6 Pseudonymization

Pseudonymization is a mechanism recommended by GDPR. In involves transforming data to a form that data subjects (people) cannot be identified, but that is reversible with the possession of a certain secret (e.g. a key). One approach would be to encrypt identifiers when exporting data. A personal data store can be queried for data and the response can be pseudonymized with a key, provided by the system that executes the query. Then the result can be provided to a third party (e.g. for analysis). After the analysis is done and the data is returned, the data can be de-pseudonymized by using the secret key.

1.2.7 Anonymization

Anonymization is the process of stripping all identifying personal data while keeping all non-identifying data. If a personal data store is used to store non-identifying data in the form of records, anonymization can mean removing all attributes of a user, but keeping the association between the user and their records. Anonymization can be done automatically after a certain period of time, which is a good practice – you don't lose data that can yield business insights, but the identifying information is removed. Anonymization can sometimes be used to implement the right to be forgotten, but extra care should be taken, as records are preserved in that case. If all non-identifying data is stored

outside the personal data store, then simply removing a user from the personal data store renders the data in other systems effectively anonymous.

1.2.8 Two-factor authentication

Two-factor authentication (2FA) is a mechanism for providing strong user authentication. It introduces an additional "factor" (element) to a traditional username/password authentication, thus reducing the risk of leaked credentials. 2FA schemes include the following factors: "something you know", "something you have" and "something you are". Usernames and passwords are "something you know", smartphones and hardware tokens are "something you have", and biometric data is "something you are". In the context of a typical web application, a 2nd factor can be a smartphone with an OTP (one-time password) generator. Each time a user logs in, they should type their username and password and an additional 6-digit code to confirm that they possess the smartphone that was originally used to enroll the user in the 2FA scheme.

Strong authentication is key to data protection as it greatly reduces the risk of compromising personal accounts and extracting potentially sensitive information from them.

1.3 Authentication

In order to use the API, you have to authenticate your calls. For that you need you need to register and obtain API credentials . Then you should pass an Authorization header with each request:

```
Authorization: Basic <base64(organizationId:secret)>
```

In case you have mobile or desktop applications and want to store user-specific tokens, you can use OAuth to exchange a user's username and password for an access token which is then passed using:

```
Authorization: Bearer <token>Storing data
```

Once you have authentication credentials, you can experiment through the API console . Below is a step-by-step example using curl (we've trimmed the authentication header for better readability):

1.4 Search schema

Data in SentinelDB is encrypted per record so in order to be able to search in the encrypted data, you have to define a schema. You can define it via the search schema UI or via an API call. For example:

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/
↪json'
--header 'Authorization: Basic NTBmOGNkYWYtNGUzZS00NDhhLWJmMDctOWRhODk5Nz...' -d '[ \
  { \
        "name": "customerName", \
        "analyzed": true, \
        "indexed": true, \
        "visibilityLevel": "PUBLIC" \
  } \
 ]' 'https://api.db.logsentinel.com/api/search-schema/bc3f863b-796b-4ecc-96aa-
↪abf0acea04a4/RECORD?recordType=Order'
```

This creates a schema for a record of type `Order` with one field – the `customerName`. You will be able to search by the customer name after inserting records. If you want to be able to search users by attributes other then their email, ID or username, you can define a search schema for your users as well.

Note: Search schemas may appear flat, as you can only specify a list of fields, but in case you are storing a nested structure (e.g. `address.primary.streetName`), you can use the dot notation to specify that a nested field is indexed and searchable

You can create and modify search schemas by API calls or via a dedicated UI from the dashboard. Each schema field has the following properties:

- `name` - the name of the field. In flat structures it is a simple name, but a dot notation can also be used in case of nested structures (as shown above, for example `address.primary.streetName`)
- `indexed` - specifies whether the field is indexed and therefore searchable.
- `analyzed` - specifies whether keywords should be extracted from the field (useful for full-text fields). Note that due to the encrypted nature of the search, stemming or other keyword analysis is not performed.
- `visibility` - `PUBLIC`, `PRIVATE` or `PROTECTED`. The visibility is a property that instructs SentinelDB what fields to return for queries. Only public ones are returned by default. If private or protected fields are required to be returned, this should be explicitly specified in the query. This is useful for protecting sensitive data from accidentally being fetched and displayed on public pages.

1.5 Inserting data

First, we create a user. The `bc3f863b-796b-4ecc-96aa-abf0acea04a4` parameter is the datastore ID in which we want to store the user (and then the record). We supply an arbitrary JSON for attributes as well as a few predefined fields like email and password:

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/
↪json'
--header 'Authorization: Basic NTBmOGNkYWYtNGUzZS00NDhhLWJmMDctOWRhODk5Nz...' -d '{ \
   "attributes": { \
         "firstName": "John", \
         "lastName": "Smith", \
         "city": "London" \
   }, \
   "email": "john.smith%40example.com", \
   "password": "password", \
   "username": "john.smith%40example.com" \
 }' 'https://db.logsentinel.com/api/user/datastore/bc3f863b-796b-4ecc-96aa-
↪abf0acea04a4'
```

Then we create a new record (which in this case is a simple order). Note that we specify the type=Order parameter as well as the ownerId parameter. The ID of the owner is the ID that the "create user" query generated.

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/
↪json'
--header 'Authorization: Basic NTBmOGNkYWYtNGUzZS00NDhhLWJmMDctOWRhODk5Nz...' -d '{ \
          "customerName": "John Smith", \
          "value": 120, \
          "items": ["pizza"] \
       } \
 }' 'https://db.logsentinel.com/api/record/datastore/bc3f863b-796b-4ecc-96aa-
↪abf0acea04a4?ownerId=ab3f863b-796b-4ecc-96aa-abf0acea05a4&type=Order'
```

Now we have a user with one order. If we want to retrieve all order for a given user, we just query SentinelDB. In this case we don't specify any additional filtering criteria, so the body is an empty query `{}`. (If we wanted to search by

an indexed field (e.g. `customerName`, we would be able to do that by specifying a list of key-value filters)

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/
↪json'
--header 'Authorization: Basic NTBmOGNkYWYtNGUzZS00NDhhLWJmMDctOWRhODk5Nz...'
-d '{}' 'https://db.logsentinel.com/api/search/records/Order/datastore/bc3f863b-796b-
↪4ecc-96aa-abf0acea04a4?pageSize=20'
```

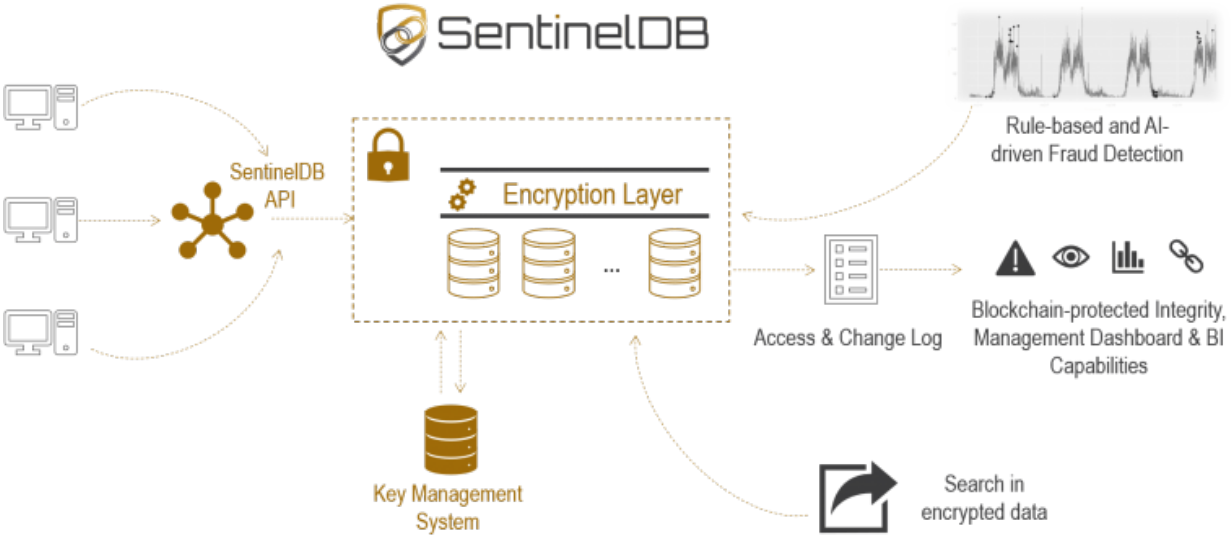
CHAPTER 2

Security Architecture

SentinelDB relies on five major security components:

- **Encryption per record** – we encrypt every record with a separate key, using AES-256, thus ensuring that no data breach can occur even in case access is compromised. We do not store any data unencrypted even in our search engine
- **Secure key hierarchy** – we have a multi-level hierarchy of encryption keys and a master key stored in a hardware security module (HSM). Each key is wrapped by its parent key and no key is stored in plain-text
- **Blockchain audit trail** – leveraging our award-winning Sentinel Trails product, we store an unmodifiable audit trail of all events related to data access and modification. We also periodically compare the current state of the database with the audit trail to guarantee the integrity of data
- **Fraud detection** – multiple layers of fraud detection are employed to detect and block any potential breaches, disguised as legitimate traffic, through the RESTful API
- **Hardended infrastructure security** – we rely on Amazon Web Services for a secure and compliant underlying infrastructure, but we improve on that by utilizing multi-factor authentication as well as secure audit trail for each administrative access. All data is encrypted in transit and at rest.

Below is an overview of the SentinelDB architecture:


SentinelDB
SentinelDB API
Encryption Layer
Rule-based and AI-driven Fraud Detection
Access & Change Log
Blockchain-protected Integrity, Management Dashboard & BI Capabilities
Key Management System
Search in encrypted data

CHAPTER 3

API Reference

The full API reference is available here

CHAPTER 4

Features

SentinelDB offers a wide variety of features, mostly through our RESTful API . Below you can find a full list of the features and their descriptions:

- **Encryption per record** – we encrypt every record with a separate key. You get this feature out of the box and don't have to do anything additional. We also do regular re-encryption of data, following the best practices in the field.
- **Datastore management** – you can have multiple datastores (conceptually equivalent to a "database", "keystore" or "schema" in popular database solutions), and you can manage them via our dashboard or through an API. Each datastore has its own wrapped encryption key and its own separate audit trail.
- **User management and authentication** – you can store all your users and their personal data in SentinelDB. Each user can have a username and password which allows SentinelDB to be used for authenticating users. Whenever authentication succeeds, SentinelDB issues an OAuth-compliant token which can be used to make further API calls to user-specific endpoints. This is useful when you have to store the authentication in a mobile app or a desktop application, rather than using the general API credentials by a backend system.
- **Two-factor authentication** - each user stored in SentinelDB can be enrolled for two-factor authentication. Then each time username/password authentication is attempted, the user also has to provide the 2FA code in order to obtain a token.
- **Record store** - each user can have a number of records that are owned by them. You can store different types of records, with different fields. Records can be binary (encoded as Base64) in order to accommodate scanned documents and other non-structured files.
- **Record schema validation** - SentinelDB is schemaless, but you can define a custom JSON-schema and upon each insert or update, the data is validated against the schema.
- **Search in encrypted data** - records and users can be searched by any of their fields (provided they are declared to be searcheable in a search schema). Searches can be done by exact match or by keyword.
- **SQL support** - we support a subset of the SQL syntax for querying, inserting and updating records and users in the database. You can *read more here*
- **Search schema** - each user and each record type can have a search schema defined so that SentinelDB knows which fields to index and make searchable. Schemas are flexible and can be modified if new fields are added.

- **Version history** - the full history of modification for each user and each record is preserved and you can fetch any previous version for a given user or record.
- **Audit trail** - all reads and writes are logged at our blockchain-based audit trail service – Sentinel Trails. You can login to the trails dashboard and drill down in the activity with flexible time-dependent queries. In order to have a more useful audit trail, you may have to provide an actorId for some of the SentinelDB API operations, indicating who was the user performing an action on a particular record (or another user). In most cases that would be the owning user, so no additional parameters are required.
- **Fraud detection** - we allow you to define fraud-detection rules based on the audit trail which, when triggered, will notify you for potential data breaches. We have a default, built-in set of rules in SentinelDB that would automatically block the extraction of data. Ontop of that, we also have a machine-learning based anomaly detection that is trained to detect usage pattern anomalies and report them.
- **"Forget user"** - implements the right to erasure (as per GDPR) by deleting all data about a user and keeping only a record that erasure has been performed.
- **Pseudonymization** - if you need to provide a sample of your data to a third party for analysis, you can use pseudonymization to protect the user identities. SentinelDB has that feature built-in – you just provide a pseudonymization key and we export pseudonymized data.
- **Anonymization** - you can choose to manually or automatically (after a period of time) anonymize the records. Anonymization is not "erasure", but it renders all the records anonymous, thus making them unlinkable to a particular person (data subject). This feature can be useful if you need to keep historical, business-relevant data and you don't need it to be attached to a particular person (user).
- **Attribute visibility configuration** - some data attributes are public, others should be visible to some types of authenticated users (e.g. partners), and others should be strictly private. SentinelDB allows the configuration of attribute visibility which then forces your application to explicitly request private and protected data. This can prevent accidental displaying of private and protected data on public pages or leaking it through public API endpoints.
- **Custom master key provider** - by default we use AWS KMS for secure management of keys. However, you can choose to supply your own master key management. In order to do that, you have to expose several endpoints (for wrapping and unwrapping keys) and register your provider with us.
- **Automatic scalability** - you don't have to worry about scaling your database. SentinelDB and its underlying storage mechanism handle that for you. All you need to do is store and retrieve data.

CHAPTER 5

SQL Support

SentinelDB supports a subset of the SQL syntax. Below are a few examples of how to use SQL queries. They are sent to a special SQL API endpoint, or can be used in the dashboard.

5.1 Select statements

5.1.1 Example

```
SELECT *, foo, bar
FROM "3aa42837-6de4-4203-a8c9-b9696d4408eb".records.Order
WHERE foo=aaa AND bar=bbb
ORDER BY foo DESC, bar
LIMIT 10
```

5.1.2 Explanation

- The `SELECT` clause can contain * and any other literal. If object is record special select words are ‘ ‘ , *‘id’, ‘ownerid’, ‘updated’. If object is user special select words are “, ‘id’, ‘updated’, ‘username’, ‘password’, ’email’, ‘status’*“ extracts the whole user/record represented as json.these words extract fields of the object itself and not its body that comes from the client.All other select words extract info from the body of the object. As it is json dot notation can be used. Example: `foo.bar.bass`
- The `FROM` clause must be in the form `.users` or `.records`. The default type name is None (when no type is provided None is used)
- The `WHERE` clause supports list of conditions separated by AND word. Each condition is in the form field=value. As in SELECT clause special words conditions are checked against the object fields and not against the body fields.Other fields conditions are checked against object’s body only if they are made searchable (with search schema)
- The `ORDER BY` clause can contain list of fields separated by ‘,’ ASC and DESC sorting is allowed for every field

- The `LIMIT` clause expects numeric value for limiting results

5.2 Update statements

5.2.1 Example

```
UPDATE "3aa42837-6de4-4203-a8c9-b9696d4408eb".records.Order
   SET 'some.thing'=something
   WHERE id='953df9c1-5917-4ff0-a5e0-604c07071324'
```

5.2.2 Explanation

- The `UPDATE` clause is the same as FROM clause from Select statement
- The `SET` clause contains list of key-value pairs separated by `,`. It uses the same special words as the select clause. If fields are not special dot notation creates new json structure in the body `SET 'foo.bar'=green` results in `{"foo":{"bar" : "green"}}`
- The `WHERE` clause is the same as in the Select statement. If it contains only id, it is fast, otherwise all matching objects are matched so it might be slower.

5.3 Delete statements

5.3.1 Example

```
DELETE "3aa42837-6de4-4203-a8c9-b9696d4408eb".records.Order
    WHERE id='953df9c1-5917-4ff0-a5e0-604c07071324'
```

5.3.2 Explanation

- The `DELETE` clause is the same as in the UPDATE and FROM cases
- The `WHERE` is the same as in the UPDATE statement including fast processing by id

5.4 Insert statements

5.4.1 Example

```
INSERT INTO "3aa42837-6de4-4203-a8c9-b9696d4408eb".records.Order
    (ownerId, foo, bar)
    VALUES ('4aa42837-6de4-4203-a8c9-b9696d4408eb', green, yellow)
```

5.4.2 Explanation

- The `INSERT` clause is the same like UPDATE and FROM + added list of field names. Special field names are the same as in other clauses
- The `VALUES` caluse contains the values of the fields in the INSERT clause

5.5 Common syntax

When field or value contains `-`, `.` or other special characters it must be quoted with `"` or `'`. In other cases quotes are not mandatory, but can be used.

5.6 Prepared statements

All SQL API endpoints (select, update, delete, insert) support prepared statements. To keep APIs stateless, parameter values are always expected with the query itself. The number of `?` and the number of params must be the same.

Examples:

5.6.1 Select

```
{"query":"select  id,foo,bar, updated, ownerId from '3aa42837-6de4-4203-a8c9-
↪b9696d4408eb'.records.Order where id=?  and updated > ?",
 "params":["953df9c1-5917-4ff0-a5e0-604c07071324", ""]
}
```

5.6.2 Update

```
{"query":"update '3aa42837-6de4-4203-a8c9-b9696d4408eb'.records.Order set 'some.thing
↪'=? where id=?",
 "params" :["foo","5095dfb5-5bbf-4a15-8d7d-12d66a3a4f2c"]
}
```

5.6.3 Delete

```
{"query":"delete '3aa42837-6de4-4203-a8c9-b9696d4408eb'.records.Order where id=?",
 "params":["5095dfb5-5bbf-4a15-8d7d-12d66a3a4f2c"]
}
```

5.6.4 Insert

```
{"query":"insert into '3aa42837-6de4-4203-a8c9-b9696d4408eb'.records.Order (foo, bar,
↪'bass.bar') values (?,?,?) ",
 "params":["pink", "green", "yellow"]
}
```

CHAPTER 6

Compliance FAQ

Q: Is SentinelDB GDPR compliant and why?

A: Yes. SentinelDB implements all the technical requirements of GDPR, following the "privacy by design" principle, extensively utilizing encryption and providing functionality to implement the right to erasure and pseudonymization. If you store all personal data inside SentinelDB and only non-personal or non-identifiable data outside, you are covered in terms of data storage when GDPR is concerned. Note, however, that GDPR contains not only technical requirements, but organizational ones as well – even though an organization is storing personal data properly, this doesn't mean that the organization can breach GDPR with its practices and procedures. Full compliance strongly depends on technical compliance, but is not limited to it.

Q: Is SentinelDB HIPAA compliant and why?

A: Yes. SentinelDB implements all the technical requirements of HIPAA by providing a way to de-identify your data and store the sensitive parts in a very secure, encrypted datastore. Our underlying infrastructure (AWS) is HIPAA-compliant as well. Additionally, we offer BAAs (Business Associate Agreements), as per HIPAA requirements. As with GDPR, HIPAA compliance is not limited to the technical aspects of data storage and your organization has to account for that.

Q: If another data protection legislation is applicable, is SentinelDB compliant?

A: Generally – yes. Local legislation can have some specifics (e.g. location requirements) which we don't cover out-of-the-box, but the general principles underlying all data protection regulations are covered by SentinelDB. If you have concerns about particular data protection legislation, contact us and we will do the required legal analysis.

CHAPTER 7

Mobile Backend as a Service

SentinelDB can be used as MBaaS (Mobile Backend as a Service). That roughly means you can directly invoke the SentinelDB APIs from your mobile applications rather than going through your own backend.

To do that, you should obtain the MBaaS credentials from the API Crednetials page – these credentials can only be used to register users and nothing else, so if someone manages to extract them from the mobile application, they won't be able to do much.

After you register a user and log them in, store their token in the device (prefer secure storage), and then authenticate each request with the token stored:

```
Authorization: Bearer <token>
```

The user would only be allowed to obtain their own records.

Such a setup is ideal for applications that need to store personal or even medical data in the cloud and don't want to take any risks in terms of compliance.

7.1 Obtaining the token

The token is obtained by invoking the `/api/oauth/token` endpoint via POST and passing `username`, `password` and `datastoreId` as request parameters.