VOTING Ausmittlung

Event Signature

Author	Abraxas Informatik AG
Classification	public
Version	1.0
Date	3. Jun 2024

For digital Switzerland. For Sure.

Contents

1. Introduction

The voting data processed by VOTING Ausmittlung must be always verifiable. Voting data includes all political transactions of a contest. A non-challengeable authorship and integrity of the data must be guaranteed throughout the entire runtime and beyond. This additional security layer requires the introduction of a signature on critical data. With the signature, all voting data classified as critical should be stored in a signed form. This concept describes the requirements and the technical design for the introduction of event signatures.

2. Initial situation

Today, the voting data is stored as representative JSON blobs in the EventStore. The EventStore already offers various security-relevant functions, such as the idempotency or immutability of the recorded event data. The threat analysis carried out has shown that the immutability is not 100% guaranteed due to the lack of signatures. An additional security layer in the form of a signature has been implemented so that the authorship and integrity of the data can be ensured when evaluating voting data.

3. System overview

The system architecture can be found in the VOTING Basis & Ausmittling Building Block View document.

3.1 Context delimitation

The signatures are applied to the event data classified as critical for a vote or election. This includes all event data that is written to the EventStore from the start to the end time of a contest by VOTING Ausmittlung. This includes all political transactions. After the end of the test phase, all data must be signed until the end of a contest. Event data from the test phase of VOTING Ausmittlung are not subject to the requirement of a signature.

3.1.1 Definitions

The signature is an integral part of various phases that a vote or election goes through in its life cycle. For the most important definitions and principles see "VOTING-Services – Glossary.pdf". The following table explains the most important definitions and principles and shows the relevance to the signature. The following definitions do not necessarily reflect the professional circumstances within a canton but are used to illustrate the technical logic from an application point of view.

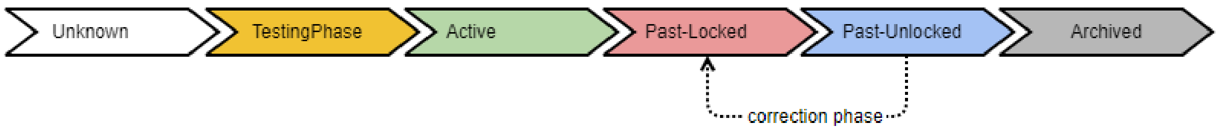
Term	Definition	Relevance for signature
contest	• A contest can consist of several votes and elections (0 to n).	• The number of possible parallel contests influences the design for the

Term	Definition	Relevance for signature
	• Depending on the constellation of the master data, several contests can take place in parallel on an election and voting day. For example, contest A takes place in the canton-A, contest B in the canton-B. In addition, it is possible that several contests are held if they are not on the same domain of influence type (For example, a contest with the domain of influence types (doi-type) CH, CT, BZ, MU, SK can be held in parallel with a contest with the doi-types SC, KI, OG, KO, AN). • If only contests are held at the same domain of influence type, they are treated as individual contests in VOTING. Municipality A conducts contest X, municipality B conducts contest Y. If only contests are held at the cantonal level, these are treated as individual contests. • If municipal votes and elections take place in parallel with cantonal vote and elections (doi-type CT), the municipalities (doi-type MU) are subordinated to the cantons and are thus not conducted as independent contests. • When votes and elections at the cantonal level takes place in parallel with federal business, cantonal votes and elections are subordinated to the federal votes and elections and are not administered in a separate contest. • On blank contest dates (predefined voting and election days by the federal government), the doi-types CT, BZ, MU, SK are attached to the federal contest (doi-type CH). Only one contest is active per canton.	management and application of signature keys. • Maximum number of parallel contests: ○ Municipal level: ~2148 ○ Cantonal level: 26 ○ Federal level: 26 (Cantons are responsible to manage and execute the votes and elections at federal level.)
vote or election	• A vote or election is always part of a contest. • One domain of influence is assigned to each vote or election (1 to 1). • A municipality or canton may conduct 1 to n votes or elections on the day of the contest. However, it is also possible that	Since the validity context of signatory keys is mapped to the contest level, the number of votes or elections is not relevant.

Term	Definition	Relevance for signature
	only cantonal or only municipal votes take place. • From a technical point of view, a vote or election is mapped as an aggregate in the EventStore. • To simplify some terms, it is possible that "votes and elections" are summarized as "political business" in the source code.	
Domain of influence type	A domain of influence takes place on one domain of influence type at a time. The following domain of influence types exist according to eCH-0155: • CH (Federal) • CT (Canton) • BZ (District) • MU (Municipality) • SK (City District) • SC (School Community) • KI (Parish) • OG (Local Citizens' Assembly) • KO (Corporation) • AN (Other)	All business levels are relevant for the formation of signatures.
Settings	Within VOTING Basis, cantonal settings can be made. This allows cantonal differences in the attributes to be configured. These have a direct influence on the evaluation or plausibility of the final result.	Cantonal settings are not considered critical to the final outcome of a contest and are excluded from the signature.

3.1.2 Lifecycle of a contest

A contest passes through several statuses in its lifecycle, which are an important indication of the application of signatures. In the following, the lifecycle of a contest is shown:

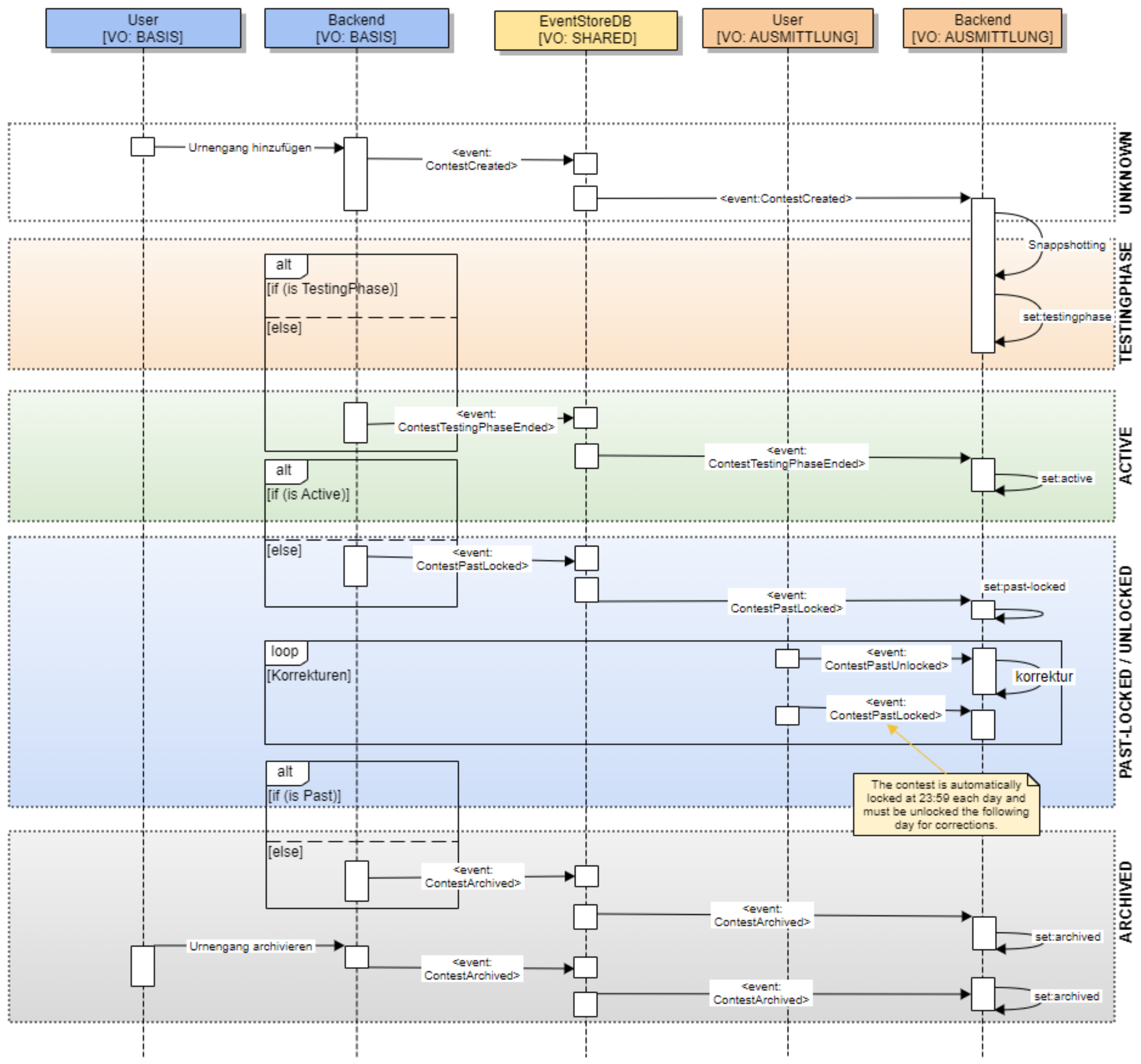


Status	Description
Unknown	Until a contest is added in VOTING Basis, it will not be available in VOTING Ausmittlung.
TestingPhase	After a contest is added, it is automatically created via Events Subscriptions in VOTING and given the status *TestingPhase.* The authority performs a test with the configuration in order to ensure, that there is no errors.
Active	After the end of the test phase, the contest is automatically set to *Active* status and retains this until the end of the voting and election day.
Past-Locked	At the end of the voting and election day, the status of the contest is automatically *set from Active* to *Past-Locked.* In this status, the contest is locked and no corrections can be made. This status can change several times until archiving, when correction phases take place.
Past-Unlocked	In the period after the voting and election day and before archiving, corrections can be made to the contest. If a correction is to be made, the contest is set to Past-Unlocked status. 0..n such correction phases can be carried out until archiving.
Archived	Once the contest is archived via VOTING Basis, no more changes can be made. Archiving is done manually via VOTING Basis or automatically via a previously defined date.

The signature is applied according to the table in the status "Active" and "Past-Locked" and "Past-Unlocked".

3.1.3 Signature start/end

For signing, the start and end time of a contest must be clearly identifiable on a technical level. The technical limitation to active contests and the temporal delimitation is an important security feature for the signature in order to limit the attack vector to a minimum. A signature is always associated with the use of a secret key, which must be short-lived and may only be used for the signing in a temporally closed time window. The following sequence diagram shows the different phases that are run through in the two relevant states ACTIVE, PAST-LOCKED and PAST-UNLOCKED.
In the ACTIVE phase, the time window is defined by the two events ContestTestingPhaseEnded and ContestPastLocked. In the PAST-LOCKED/UNLOCKED phases, 0..n corrections can be made over several defined time windows. These time windows are defined via the events ContextPastLocked and ContestPastUnlocked.



Phases	Definition time	Event / Action	Time / Duration	Description
Start time phase contest *Status: Active*	The start time of a contest begins when the test phase ends.	Event: ContestTestingPhaseEnded	**Time**: Date end of test phase **Duration min**: 1 day **Duration max**: End of test phase until end of contest.	On blank contest dates, the contest normally starts on the Saturday evening/night before the Sunday of the voting and election day, as soon as the test period ends. A cron job in VOTING Basis checks at cyclical intervals whether the test phase has ended (time-based comparison). As soon as the test phase is finished,

Phases	Definition time	Event / Action	Time / Duration	Description
				the event *ContestTestingPhaseEnded is* triggered, which triggers the clean-up work of the test phase in VOTING Ausmittlung. As a result, the Read Model is cleaned up and prepared for the active contest.
end time Phase contest *Status: Active*	From a temporal point of view, the end time of a voting and election day is clearly defined as the end of the day (23:59).	Event: ContestPast	**Time**: 23:59 (Local Time CH)	A contest is closed at the end of the voting and election day (midnight 00:00) and changes from the status "Active" to "Past-Locked". A cron job in VOTING Basis checks at cyclical intervals whether the end of the voting and election day has been reached (time comparison). The event ContestPast is used to inform VOTING Basis about the end time of the voting and election day.
Start time for corrections *Status: Past*	The start time of a correction phase begins when the contest is set to Past-Unlocked status by the user.	Event: ContestPastUnlocked	**Time:** Anytime from the start to the end of a day **Duration**: Max. 24h until end of day 23:59 (Local Time CH)	Until it is archived, an original contest can be set to Past-Unlocked status at any time after the Active phase and released for corrections. The correction must be released via a user action at the contest level. The status of the contest is independent of the status of the associated votes and elections.
End time for corrections *Status: Past*	The end time of a correction phase is automatically ended at 23:59 on the same day of the start time.	Event: ContestPastLocked	**Time**: 23:59 (Local Time CH) on the day of the correction	For security reasons, each correction phase is automatically ended by the system at 23:59 (UTC+1) on the same day. If further corrections are necessary, the contest must be set to correction again on the following day. The correction status of the contest is independent of

Phases	Definition time	Event / Action	Time / Duration	Description
				the status of related votes and elections.

3.2 Events to be signed

All events (according to the chapter Context delimitation) which are sent from the VOTING Ausmittlung backend to the EventStore must be signed. The events that store the public key used for the signature in the EventStore must be provided with a signature by the Hardware Security Module (HSM).

3.3 Signature payload

3.3.1 Signature data structure

All data (properties) that are important for the signature must be prepared via a defined data specification. This specification must be reproducible and idempotent at all times. A user must be able to perform a signature validation based on the data required for the validation.
Event Publisher: https://github.com/abraxas-labs/voting-library-dotnet/blob/main/src/Voting.Lib.Eventing/Persistence/EventPublisher.cs#L40

Specification Signature Version 1:

```
byte[] bytesToSign = CONCAT(SignatureVersion, EventId, Name, Payload,
UrnengangId, HostId, KeyId, Timestamp)
```

Codeblock 1 Signatur Byte Array Blob

Property	Position	Data type	Data type (original)	Value range (original)	Description	Normalisation
SignatureVersion	1	Byte (fix 0x01)	byte	1...255	The signature version is also written to the metadata so that the correct signature algorithms and models can be used for signature creation and validation.	-
EventId	2	Byte array	String	n/a	The event ID is generated by VOTING itself as a unique key per event. The EventId must be normalised for the signature and converted	1. UTF8 2. Lower-Case

Property	Position	Data type	Data type (original)	Value range (original)	Description	Normalisation
					into a byte array representation.	
Name	3	Byte array	String	Application-dependent	The name corresponds to the FQDN of the proto aggregate used. The name must be normalised for the signature and converted into a byte array representation.	1. UTF8
Payload	4	Byte array	Byte array	Application-dependent	The payload contains all business-relevant event data. The event payload is specified by the event publisher as a byte array and must be used unchanged for the signature.	-
UrnengangId	5	Byte array	String	n/a	The contest Id is the unique reference to the associated contest. The contest ID must be normalised for the signature and converted into a byte array representation.	1. UTF8
HostId	6	Byte array	String	n/a	The Host Id is the identification of the host (Hostname / Pod Id) on which the Backend Service instance is running. The Pod Id consists of the Pod name and a unique UID suffix. Further information on the naming convention: https://kubernetes.io/docs/concepts/overview/working-with-objects/names/#uids Command: System.Environment.MachineName	1.UTF8

Property	Position	Data type	Data type (original)	Value range (original)	Description	Normalisation
					The host ID must be normalised for the signature and converted into a byte array representation.	
KeyId	7	Byte array	String	128 bit	The Key Id corresponds to the unique identifier from the public signatory key for the given host. The Key ID must be normalised for the signature and converted into a byte array representation.	1. UTF8
Timestamp	8	Byte array	Int64 (unix epoch)	64 bit	The timestamp is specified as UTC in milliseconds and is used for traceability. The timestamp corresponds to the time at which the event was created before it was published.	1. Big-Endian

Important notes:

- For the payload, it is ensured that the bytes have not been manipulated. However, it is not ensured how these bytes must be interpreted (encoding, data type, ...). The interpretation is not part of the signature.
- The Guid data type must be normalised with ToLowerCase() before ToByteArray(), because it is not guaranteed that the EventStore returns this property in the same character casing for the event when reading it. Furthermore, the ToByteArray() method of the Guid data type does not follow the RFC4122 standard (see: https://docs.microsoft.com/en-us/dotnet/api/system.guid.tobytearray?view=net-6.0).

3.4 Signature metadata + persistence

Specification Metadata Version 1:
The serialisation can be freely selected. It must be ensured that all signature-relevant data types can be read again according to the specification.

Property	Data type
SignatureVersion	Byte
UrnengangId	String (UTF8)
HostId	String (UTF8)

Property	Data type
KeyId	String (UTF8)
Signature	Byte array
Timestamp	Int64

3.5 Signature algorithm

The signature payload must be encrypted with a secure signature procedure. The specifications for the signature key are defined in the following table.

	Algorithm	Basis for decision-making
Signature algorithm	ECDSA	The ECDSA algorithm fulfils all security requirements for the application and cryptography policies. ECDSA is significantly faster when signing (.NET Support)
Elliptic Curve	Curve P-521	NIST Specified Curve (.NET Support)
Hash algorithm	SHA2-512	SHA2-512 has no known weaknesses and meets the requirements of ECDSA Curve P-521

3.6 Private/Public Key Management

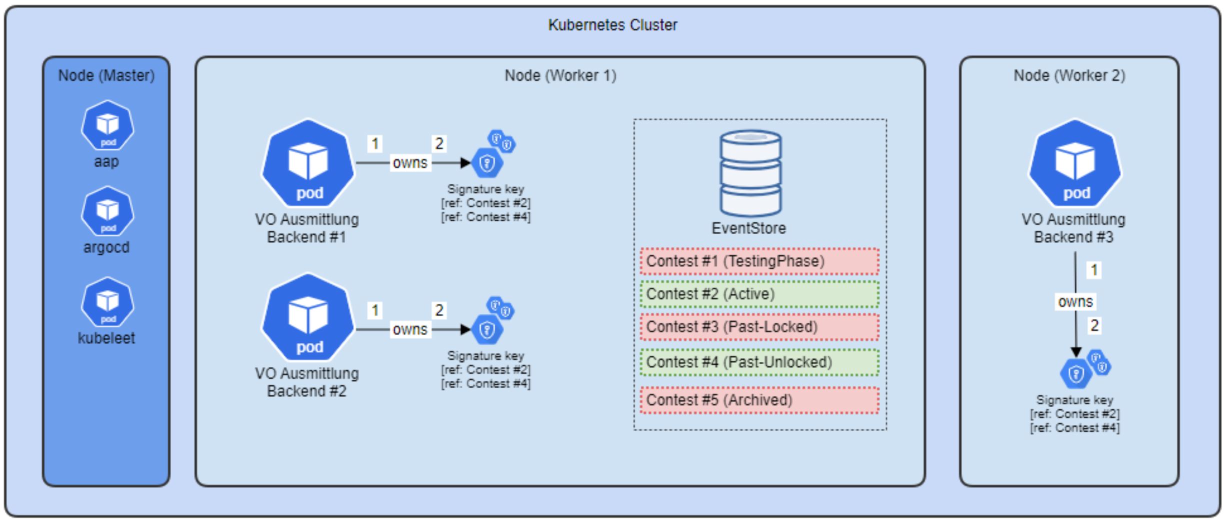
3.6.1 Validity range

The validity range of a key pair for signing is defined by the contest itself. For each contest and backend instance, there is a maximum of one key pair consisting of a public and a private key.

3.7 Private/Public Key Management

3.7.1 Validity range

The validity range of a key pair for signing is defined by the contest itself. For each contest and backend instance, there is a maximum of one key pair consisting of a public and a private key.


Kubernetes Cluster
Node (Master)
aap
argocd
kubeleet
Node (Worker 1)
VO Ausmittlung Backend #1
owns
Signature key [ref: Contest #2] [ref: Contest #4]
VO Ausmittlung Backend #2
EventStore
Contest #1 (TestingPhase)
Contest #2 (Active)
Contest #3 (Past-Locked)
Contest #4 (Past-Unlocked)
Contest #5 (Archived)
Node (Worker 2)
VO Ausmittlung Backend #3
pod

3.7.2 Validity period

Period of validity for signing

- The validity period of the signature in the different phases is defined in the chapter "Signature start/end".
- The period of validity is defined at the technical level on the basis of the contest statuses and is ensured at the application level. At the technical level of the private key, there is no restriction on the validity period.

Validity period for signature validation

- The public key used for encryption remains valid indefinitely.
- Only events that are within the validity range of the key at the time of signing may be declared valid.
- Each key has a certain maximum validity period, which is initiated by the time of issue and ends at the end of the day (election and voting day / correction day). The exact times are defined in the chapter "Signature Start/End".

3.7.3 Key pair creation

Each backend instance is responsible for providing valid keys for signing. The creation of a key pair can be done in two ways:

- Passive: The backend instance is informed via a notification when new keys are needed for a contest.
- Active: The backend instance must check itself at a restart whether keys are needed for current contests.

3.8 HSM Public Key Signing

To confirm the authorship of generated key pairs, the public keys used for signature validation must be signed by a trustworthy authority. The public key must be deposited with the signature in the EventStore and verified before each signature validation. Abraxas Informatik

AG has a self-hosted HSM service, which can be connected to VOTING Ausmittlung for this purpose.

3.8.1 HSM Service Interface

The HSM service interface has the following features:

Owner	Abraxas Informatik AG (IT Services)
Manufacturer	Utimaco CryptoServer HSM
Service	Independent geo-redundant service
Documentation	HSM Services (Abraxas internal documentation)
Interface	PKCS#11 (Alternative: CSP/CNG/SQLEKM/JCE)
Authentication	Basic (Password / Username)
SLA	• Standby duty • Geo redundancy • Maintenance window with failover mechanism
Signature creation	Signing of any payload
Signature validation	• Variant 1: Validation of signatures according to PKSC#11 Interface description • Variant 2: Query of public certificate for manual validation of the signature at application level.
Signature Timestamp	Not supported.

3.8.2 HSM authentication

Authentication against the HSM is done via Basic Authentication with username and password. This authentication data is managed separately per environment in GoPass by VOTING and made available to all backend service instances (pods) via declarative GitOps standard mechanisms. If authentication against the HSM fails, the backend instance must be designated as "unhealthy" to Kubernetes because it is unable to sign the public keys.

3.8.3 HSM PKCS interface

The connection to the HSM is made via the standardised PKCS#11 interface. For .NET 6, the following variants are available for connecting PKCS#11 (as of 08.12.2021):

- Managed Library
- Unmanaged Library
- Native interopability

Solution approach:

- Use of the Pkcs11Interop Library which enables the connection to the PKCS#11 interface installed on the operating system level via a native interopability wrapper.
- With Pkcs11Interop, the PKCS#11 API can be used in .NET managed code.
- Pkcs11Interop Reference: https://github.com/Pkcs11Interop/Pkcs11Interop

3.8.4 Procedure HSM signing

The public key used for validating the signature must be signed by the Hardware Security Module after it has been generated. This ensures the authorship of the generated public key for future validations. The signature by the HSM runs as follows:

1. The VOTING Ausmittlung backend creates a new key pair consisting of a public and private key for each contest and instance (Pod).
2. The public key is transferred to the HSM service for signing.
3. The HSM service creates a signature and returns it to the VOTING Ausmittlung backend.
4. The public key is stored together with the generated signature in the EventStore and is available for future signature validations.

3.8.5 Signature Public Key via HSM

The public key is also signed with all relevant data (property) using the same procedure as already described for the signing of the event payload in the chapter "Signature data structure".
Specification Signature Version 1:

```
byte[] bytesToSign = CONCAT(SignatureVersion, UrnengangId, HostId, KeyId,
PublicKey, ValidFrom, ValidTo)
```

Codeblock 2 Signatur Byte Array Blob

Property	Position	Data type	Data type (original)	Value range (original)	Description	Normalisation
SignatureVersion	1	Byte (fix 0x01)	byte	1...255	The signature version is also written into the metadata so that the correct signature algorithms and models can be used for validation.	none
UrnengangId	2	Byte array	String	128 bit	The contest Id is the unique reference to the associated contest.	1. UTF8

Property	Position	Data type	Data type (original)	Value range (original)	Description	Normalisation
					The Urn Gang Id must be normalised for the signature and converted into a byte array representation.	
HostId	3	Byte array	String	1...255	The Host Id is the identification of the host (container) on which the backend service instance is running. The host ID must be normalised for the signature and converted into a byte array representation.	1. UTF8
KeyId	4	Byte array	String	1128 bit	The Key Id corresponds to the unique identifier of the public signature key (Public Key Id). The Key ID must be normalised for the signature and converted into a byte array representation.	1. UTF8
PublicKey	5	Byte array	Byte array	Hash from key	The public key is used for the verification of signed events.	1. SHA512 hash(PubicKey [bytes])
ValidFrom	6	Byte array	Int64 (unix epoch)	n/a	The time from which the public key may be used for signature validations.	1. Big-Endian
ValidTo	7	Byte array	Int64 (unix epoch)	n/a	The time until which the public key may be used for signature validations.	1. Big-Endian

3.8.6 HSM Public Key Event

All data (properties) that are important for the signature must be prepared via a defined data specification. This specification must be reproducible and idempotent at all times. A user must be able to perform a signature validation based on the data required for the validation.
Each public key generated and signed by the HSM must be persisted in the EventStore for future signature validations. For full traceability and validation, the following data must be stored as part of this event. The public keys and all associated data (properties) must be retrievable at any time.

Specification Public Key Event Version 1:

Property	**Data type**	**Data type (original)**	**Description**	**Normalisation**
SignatureVersion	Byte (fix 0x01)	byte	The version corresponds to the version of the model.	No normalisation
UrnengangId	Byte array	String	See description chapter "Signature Public Key via HSM".	1. UTF8
HostId	Byte array	String	See description chapter "Signature Public Key via HSM".	1. UTF8
KeyId	Byte array	String	See description chapter "Signature Public Key via HSM".	1. UTF8
PublicKey	Byte array	Byte array	See description chapter "Signature Public Key via HSM".	No normalisation
HsmSignature	Byte array	Byte array	The signature generated by the HSM.	No normalisation
ValidFrom	Byte array	Int64 (unix epoch)	See description chapter "Signature Public Key via HSM".	1. Big-Endian
ValidTo	Byte array	Int64 (unix epoch)	See description chapter "Signature Public Key via HSM".	1. Big-Endian

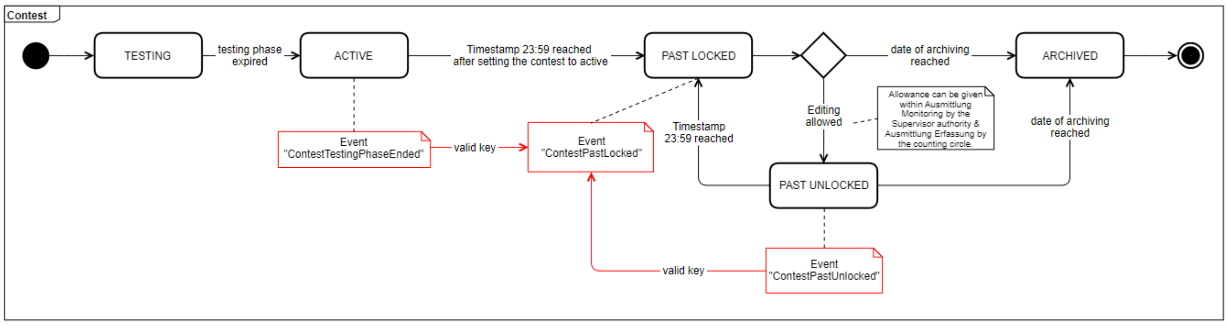
3.8.7 Procedure HSM validation

The public key used for validating the signature must be signed by the HSM after it has been generated. This ensures the authorship of the generated public key for future validations. The signature by the HSM runs as follows:

1. The metadata from the signed event must be read out.
2. The corresponding public key must be read from the EventStore. The following metadata must match the public key event:
   - UrnengangId
   - HostId
   - KeyId
3. The validation of the PublicKey with the associated HsmSignature must be checked against the PKCS#11 validation interface by the HSM.

  - Alternatively, the public certificate can be queried by the HSM and the signature check performed manually.

3.9 Signature process flow


Contest
TESTING
testing phase expired
ACTIVE
Timestamp 23:59 reached after setting the contest to active
PAST LOCKED
date of archiving reached
ARCHIVED
Event "ContestTestingPhaseEnded"
valid key
Event "ContestPastLocked"
Timestamp 23:59 reached
Editing allowed
Allowance can be given within Ausmittlung Monitoring by the Supervisor authority & Ausmittlung Erfassung by the counting circle.
date of archiving reached
PAST UNLOCKED
valid key
Event "ContestPastUnlocked"

3.10 Backend notification

3.10.1 Initial situation

All running backend instances must be informed about relevant status changes of contests. In order for the backend instances to be able to generate individual key pairs for contests and sign events, a corresponding notification mechanism is required.

3.10.2 Requirements

The requirements for the notification mechanism are defined below:

Definitions: Backend instances = Consumer

Id	**Request**	**Status**
AC01	Consumers must be notified of changes in the status of contests at runtime.	**required**
AC02	Notification must be asynchronous. Polling of the contest status is not permitted.	**required**
AC03	The consumers must be able to register again to the notifications after successful start-up and also after connection interruptions.	**required**
AC04	Only consumers whose pods are in a healthy state may be notified of status changes via contest. In Kubernetes, these are defined by the Healthy and Ready states.	**required**
AC05	All successfully collected notifications must be confirmed by the consumers via an acknowledgement.	**optional**
AC06	Each successfully received notification must be recorded in the log.	**required**
AC07	Failed notifications must be output in the log as errors and in the alerting.	**required**

3.10.3 Solution: EventStore

Event Store is used for the notification of backend instances. The EventStore is the core component in the VOTING system and is responsible for processing contest status changes. With the catch-up strategy, the requirements can be implemented with this core component. The integration can be carried out based on existing know-how and technical principles.

In addition to the persistent subscriptions already used today, the EventStore offers the possibility of implementing broadcasting by means of catch-up subscriptions. Several consumers can register independently of each other via a subscription to one or more streams and receive the events in sequential order. This variant offers the following advantages:

- The EventStore is an integral part of the VOTING architecture and manages the relevant status changes of contests.
- Catch-up subscriptions can be used to achieve broadcasting at the consumer level.
- A consumer can define the start position itself at the time of the subscription. Events can thus be run through in replay mode over a defined period of time.

The consumer must remember the in-memory position after the initial subscription and all subsequent events received. This is necessary so that the entire history of events is not replayed when a new subscription is made.

Statement: Catch-up Subscriptions
You can have multiple subscribers for the same stream and all those subscribers will get all the events from that stream. Subscriptions have no influence on each other and can run on their own pace.
ref: https://developers.eventstore.com/clients/dotnet/21.2/subscriptions.html#catch-up-subscriptions