Ockam

Design Review

November 22, 2023

Prepared for:
Mrinal Wadhwa
Ockam Inc.

Prepared by: **Scott Arciszewski, Fredrik Dahlgren, Marc Ilunga, and Joop van de Pol**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

The sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through any source other than that page may have been modified and should not be considered authentic.

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineering director was associated with this project:

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Scott Arciszewski, Consultant
scott.arciszewski@trailofbits.com

Fredrik Dahlgren, Consultant
fredrik.dahlgren@trailofbits.com

Marc Ilunga, Consultant
marc.ilunga@trailofbits.com

Joop van de Pol, Consultant
joop.vandepol@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
September 28, 2023	Pre-project kickoff call
October 10, 2023	Status update meeting #1
October 16, 2023	Status update meeting #2
October 20, 2023	Status update meeting #3
November 1, 2023	Status update meeting #4
November 6, 2023	Delivery of report draft
November 6, 2023	Report readout meeting
November 22, 2023	Delivery of comprehensive report

Executive Summary

Engagement Overview

Ockam engaged Trail of Bits to review the security of Ockam, which is a set of protocols and managed infrastructure. Ockam's protocols aim to enable secure end-to-end communication between endpoints across various topologies as if they were connected locally, without any modification of the endpoints themselves. Moreover, users may deploy the protocols on their premises, thereby not relying on Ockam's managed infrastructure and obviating the need to trust Ockam or its infrastructure.

A team of four consultants conducted the review from October 2 to November 3, 2023, for a total of 11 engineer-weeks of effort. Our testing efforts focused on the security of Ockam's protocols in the context of two specific use cases: TCP Portals and Kafka Portals. With full access to design documentation, we reviewed the design of the protocols, focusing on the two use cases in scope. We conducted the review using automated and manual processes. While we had access to the current implementation of Ockam's protocols to aid in understanding the design, the implementation itself was not in scope for the review.

Observations and Impact

Ockam's protocols use robust cryptographic primitives according to industry best practices. None of the identified issues pose an immediate risk to the confidentiality and integrity of data handled by the system in the context of the two in-scope use cases. The majority of identified issues relate to information that should be added to the design documentation, such as threat model details (TOB-OCK-5, and TOB-OCK-1) and increased specification for certain aspects (TOB-OCK-2, TOB-OCK-4, and TOB-OCK-6).

Regular question-and-answer sessions were held during the engagement to clarify any questions about the protocols and to provide information that was missing from the design documentation. These were very helpful to understanding the overall system as well as any design considerations made by the developers.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Ockam take the following steps:

- **Remediate the findings disclosed in this report.** Although no findings indicate any immediate or high-severity risk to the Ockam system, the findings should be addressed as part of a direct remediation or as part of any redesign that may occur when addressing other recommendations.

- **Incorporate all relevant information from the Q&A sessions into the design documentation.** The questions asked during this design review correspond to potential system issues that are not excluded by the current design documentation. While the answers showed that the actual system does not suffer from these specific issues, incorporating the answers into the design documentation would help ensure that this becomes an explicit, rather than implicit, part of the design.

- **Consider a security audit of the implementation of Ockam's protocols.** Ultimately, the implementation of a protocol determines the risk associated with the protocol, as a secure design does not imply a secure implementation. Issues in the deployment of a protocol may arise from discrepancies between the design and the implementation or from specific implementation choices that violate the assumptions in the design. A review of the implementation of Ockam's protocols can uncover such issues, helping the deployment of Ockam to be as safe as possible. For specific use cases, it may be necessary to audit the corresponding configuration of third-party infrastructure, such as GitHub and AWS.

- **Consider further formal analysis of Ockam's protocols.** In our review, we used automated tooling to assist with the analysis in the allotted time. Formal models provide a high level of assurance in the targeted protocol and are also a good starting point when trying to understand how a small change or update would impact the overall security guarantees provided by the protocol.

  Moreover, if the threat model and expected security guarantees are documented as recommended in this report, a formal analysis may be valuable in understanding the guarantees provided by the system. For this reason, we recommend that Ockam consider additional formal analyses of its protocols. The section on Coverage Limitations provides some example protocol aspects that could benefit from formal modeling.

Finding Severities and Categories

The following tables provide the number of findings by severity, difficulty, and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	3
Low	0
Informational	3
Undetermined	0

Difficulty	*Count*
Low	0
Medium	0
High	4
Undetermined	0
Not Applicable	2

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	6

Project Goals

The engagement was scoped to provide a security assessment of Ockam's protocols with respect to the TCP Portals and Kafka Portals use cases. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are there any critical design flaws that could result in the compromise of data confidentiality or integrity?
- Is the design vulnerable to any known cryptographic attacks?
- Does the system use any weak cryptographic primitives?
- Is the trust infrastructure sufficiently secure?
- Does the system follow best practices for the generation, distribution, storage, and destruction of cryptographic keys?
- Are handshakes conducted in a cryptographically secure manner?
- Are secure channel protocols used effectively?
- Are access controls, policies, and credentials cryptographically secure?
- Is the Noise framework securely used within the design?
- Does the design of the system lend itself well to improvements and iterations over time?
- Within the design, is end-to-end encryption guaranteed?
- Does the design enable the deployment of the system and onboarding of new clients in a secure manner?

Project Targets

The engagement involved a review and testing of the following target.

Ockam design documentation

Repository	https://app.gitbook.com/o/bSIWJRTRgnhbW4jNMcHT/
Version	`2023-11-03`
Type	Documentation

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **Manual review of the design documentation.** The review focused on the following topics:
    - **Secure channels.** Secure end-to-end communication is implemented using the Noise framework, a well-known framework for designing secure channel protocols. Ockam Secure Channels are based on the XX pattern of the Noise protocol. We reviewed the use of secure channels in Ockam, focusing on whether the Noise protocol is properly instantiated and used effectively to provide end-to-end security. We paid special attention to the effect of unreliable delivery on the security of secure channels. We also investigated authentication guarantees of the secure channels when used in a post-specified setting where nodes may establish secure channels with other nodes whose identities are not known in advance.
    - **Routing and transport mechanisms.** Within Ockam's infrastructure, routing and transport mechanisms deliver messages with no reliability guarantees out of the box, unlike protocols such as TCP or QUIC. Additional mechanisms are used in the system to provide some forms of reliability, but these are currently not documented in the design. We investigated the routing and transport mechanisms used in Ockam, focusing on their impact on the security of secure channels. Moreover, we examined potential concerns leading to denial of service or waste of resources.
    - **Identities and credentials.** In the Ockam system, each participating entity has one or more Ockam Identities. These are cryptographically verifiable digital identities that allow other entities to verify the authenticity of digital signature keys used to authenticate users. To bootstrap trust in these identities, the Ockam system supports the use of credentials, which are signed attestations of identities provided by an issuer. We investigated the cryptographic primitives used to realize these concepts, as well as the applicability of known cryptographic attacks.
    - **Access controls and policies.** The Ockam system relies on various cryptographic techniques to guarantee authenticity and confidentiality. Each particular use case in the Ockam system can use access controls and access control policies to ensure that certain checks are performed, such as the verification of credentials from a specified issuer. We investigated whether

parsing issues could circumvent these security controls and whether they were correctly applied in the two in-scope use cases.

  - **TCP Portals and Kafka Portals use cases.** These use cases focus on enabling end-to-end encrypted communication between endpoints over any network topology. The TCP Portals use case focuses on enabling applications to communicate with each other as if they were making a local TCP connection, whereas the Kafka Portals use case achieves the same for applications that consider they are talking to Kafka servers via a local connection. We investigated the use of third-party infrastructure (such as GitHub, AWS, OAuth, etc.), the correct use of different aspects supported by the overall Ockam system, and whether any issues in specific system components could be used to compromise confidentiality or integrity of user data.

- **Q&A sessions with the designers.** In cases where the documentation did not provide sufficient information to determine the applicability or impact of potential attacks, we asked questions of the designers, who provided answers.

- **Formal modeling of protocol aspects.** We used Verifpal and CryptoVerif to model the following aspects:

  - **Modeling with Verifpal.** We used Verifpal to model the cryptographic core of secure channels. We focused on confidentiality and authentication guarantees. We also modeled identities in Ockam to understand whether the intended authentication guarantees are provided.

  - **Modeling with CryptoVerif.** We modeled identities in CryptoVerif, which allowed us to further understand the exact security properties needed to ensure that the design provided sufficient guarantees for each intended use case.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. We did not formally model all aspects of the system. Specifically, the following may warrant further modeling (although all were considered as part of the manual review):

- **Identity binding guarantees of secure channels**. The secure channel protocol allows post-specified peers, where communicating parties may learn about the peer's identity upon completion of the protocol. In such scenarios, the protocol must properly bind identities to the derived session key. In particular, the protocol must protect against identity misbinding attacks, also known as unknown key-share attacks. Ockam builds upon Noise's public key binding to bind identities to session keys. Due to time limitations, we did not formally model whether such binding was done securely.

- **Complete modeling of identities and credentials.** In our formal modeling work, we focused on modeling the cryptographic core mechanisms underpinning identities, change histories, and credentials. In doing so, we abstracted some elements, focusing only on cryptographically relevant aspects. Nevertheless, more complete modeling will provide valuable insights into the security of the different components used in Ockam.

- **Post-compromise security of various keys.** We did not model any post-compromise security of identity keys, purpose keys, or session keys. It is worth investigating what guarantees can still be provided when specific keys are compromised.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software and protocols. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software. We further use open-source formal modeling tools to perform automated testing of cryptographic protocols and system designs.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Verifpal	An open-source symbolic protocol analyzer that focuses on ease of use, allowing rapid modeling and verification of cryptographic protocols	Appendix C.1
CryptoVerif	An open-source computational protocol verifier and proof assistant used to create models that state precise assumptions on the primitives used in the protocol and the desired security properties to be proven	Appendix C.2

Areas of Focus

Our automated testing and verification work focused on the following system properties:

- Authenticity of Ockam Identities
- Confidentiality and authenticity of data exchanged during the secure channel handshake

The results of this focused testing are detailed below.

Authenticity of Ockam Identities. We used Verifpal to create a model of Ockam Identities in a scenario where one user communicates two `Change` blocks to another user that has access to the first user's identifier. We also used CryptoVerif to create a model of change generation in a scenario where an adversary can obtain signatures on arbitrary messages under the primary identity keys.

Property	Tool	Result
Authenticity of first `Change` block based on Ockam Identifier	Verifpal	**Passed**

Property	Tool	Result
Authenticity of second Change block based on Ockam Identifier	Verifpal	**Passed**
Existential unforgeability of Change blocks	CryptoVerif	**Passed**
Strong unforgeability of Change blocks	CryptoVerif	**TOB-OCK-6**

The linked finding above is informational severity and not currently exploitable; please see the finding itself for further details.

Ockam Secure Channels. We used Verifpal to model the cryptographic core of Ockam Secure Channels, including but not limited to the handshake.

Property	Tool	Result
Secrecy of session keys	Verifpal	**Passed**
Confidentiality and authenticity of transported messages	Verifpal	**Passed**

The design principles of the Noise framework enable the XX pattern to provide further security guarantees. For instance, the XX pattern is designed to be resistant to key compromise impersonation (KCI) attacks. KCI resistance guarantees that even when a user's long-term secret key is compromised, the adversary should not be able to impersonate other users towards the user whose key was compromised. This result has been further confirmed by Dowling, Rösler, and Schwenk in their analysis of the Noise framework. Therefore, we did not verify these guarantees using our formal model.

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	The system is vulnerable to SNDL attacks by quantum computers	Cryptography	**Medium**
2	Serialized VersionedData struct's data is ambiguous	Cryptography	**Informational**
3	Truncating ChangeHistory hash to 160 bits introduces risk of collisions	Cryptography	**Informational**
4	The meanings of the primary key fields created_at and expires_at are undocumented	Cryptography	**Medium**
5	Insufficient threat model documentation	Cryptography	**Medium**
6	The supported signature schemes have different security properties	Cryptography	**Informational**

Detailed Findings

1. The system is vulnerable to SNDL attacks by quantum computers	
Severity: **Medium**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-OCK-1
Target: Keys and Vaults section	

Description

The Ockam system uses the Noise XX handshake pattern with ECDH based on X25519, which could be broken using a large-scale quantum computer. Currently, no quantum computer capable of doing this exists. Although Ockam could be quickly updated to resist quantum attacks, current handshakes would still be vulnerable to "store now, decrypt later" (SNDL) attacks using a quantum computer, as described in the exploit scenario below. The design should address whether this is considered as part of the threat model.

Exploit Scenario

An attacker captures and stores the full transcript of a handshake and the subsequent encrypted communications. Once a large-scale quantum computer becomes available, the attacker recovers all ECDH private keys using the quantum computer. They use this to obtain the derived keys from the transcript and decrypt the communications.

Recommendations

Short term, document whether SNDL attacks using a quantum computer are applicable to the threat model for different use cases so that users can consider this in the context of their own risk management. If the attacks are applicable, investigate the feasibility of incorporating post-quantum secure alternatives into the system.

Because the goal is to prevent SNDL attacks, it is not necessary to upgrade all cryptographic primitives to be secure against a quantum computer. However, at least one of the contributions to the key derivation must come from a PQC KEM (e.g., Signal's PQXDH). As described in the PQNoise paper, it is straightforward to update the Noise ee pattern using a PQC KEM to achieve the same round complexity. Instead of replacing the classical DH, we propose adding a PQC KEM to achieve a hybrid solution (e.g., the IETF draft on hybrid key exchange in TLS).

Long term, if attacks using a quantum computer are part of the threat model for Ockam use cases, migrate both the key exchanges and digital signatures to a hybrid solution incorporating both classical and post-quantum resistant primitives (e.g., using PQNoise with suitable hybrid KEMs).

2. Serialized VersionedData struct's data is ambiguous	
Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-OCK-2
Target: Identities and Credentials section	

Description

Identity private keys are used to sign both `Change` blocks, which are used to rotate the identity keys, and `PurposeKeyAttestation` blocks, which are used to attest to purpose keys. Before being signed, the data inside these blocks is serialized using the Concise Binary Object Representation (CBOR) data format and included in a `VersionedData` struct.

However, this serialization will not add different labels for these different data types by itself, which means that the `data` field of a `VersionedData` instance does not indicate which type it contains (i.e., `ChangeData` or `PurposeKeyAttestationData`). Therefore, it is possible to provide a signed `VersionedData` instance containing a `PurposeKeyAttestationData` block where the receiver expects it to contain a `ChangeData` block. The receiver will potentially accept the signature as valid as long as it is created using the expected identity key.

Whether this confusion can be exploited depends on implementation details that are not in scope for this design review. Specifically, it depends on what happens when a `PurposeKeyAttestationData` type is deserialized as a `ChangeData` type. Currently, it seems likely that this will fail due to differences in the structure of the types, but it is not possible to determine the full behavior from the design description alone. Even if the current implementation rejects a serialized `PurposeKeyAttestationData` instance as invalid when it attempts to deserialize the byte vector as a `ChangeData` structure, a future version of the protocol may accept it as valid if either of the two types change.

Exploit Scenario

A node creates and attests to various purpose keys to outsource the handling of the purpose to other instances, which do not have access to the identity key. An attacker compromises one of the instances and obtains the `PurposeKeyAttestation` block containing a signed `VersionedData` instance.

The attacker communicates a new `Change` to another node, while providing the signed `VersionedData` from the `PurposeKeyAttestation` block. The `previous_signature` field inside the `Change` block is set to the `signature` from the `PurposeKeyAttestation` block (i.e., the signature on the `VersionedData` instance containing the

`PurposeKeyAttestationData`). The other node will accept the `previous_signature` on the `VersionedData` because it is a signature under the same identity primary key.

For this exploit to be useful, the `data` field of the `VersionedData` instance, which is a serialized `PurposeKeyAttestationData` type, must deserialize to a `ChangeData` type with a public key for which the attacker knows the corresponding private key. Otherwise, the attacker will be unable to provide a proper `signature` for the `Change` block.

Recommendations

Short term, specify in the design that the `data` field inside the `VersionedData` struct to be signed using identity primary keys must be unambiguous. For example, this field could be a single `enum` type that contains the `PurposeKeyAttestationData` and `ChangeData` types with different CBOR labels. Alternatively, add an additional label to the `VersionedData` struct to indicate which data type it contains.

Long term, ensure that all different types of private keys used to create digital signatures sign only a single unambiguous type (with different labels for each of the possible subtypes or contained types).

3. Truncating ChangeHistory hash to 160 bits introduces risk of collisions	
Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-OCK-3
Target: Identities and Credentials section	

Description

The `Identifier` and `ChangeHash` types are defined as the first 160 bits of a SHA-256 hash of the `Change` record inside of a `ChangeHistory` chain. A collision attack against SHA-256 truncated to 160 bits is possible with a time cost of approximately 2^{80} queries using standard collision-finding techniques based on Pollard's Rho method. These figures are on the upper end of feasibility but still possible to exploit.

Exploit Scenario

Mallory, a user with an existing Ockam Identity wants to cause disagreement about her current primary public key between different nodes, who are relying on the latest `ChangeHash` to identify the user's primary public key.

She uses Pollard's Rho method to find two `ChangeData` records with the same `ChangeHash`, which allows her to create two distinct `ChangeHistory` chains where the final entries disagree on the primary public key but still have the same `ChangeHash`.

Computing 2^{80} SHA-256 hashes is within the reach of botnets today. For comparison, the Bitcoin mining network computes about 2^{68} SHA-256 hashes per second (or about 2^{92} per year). The equivalent amount of computational resources can cross the 2^{80} threshold in a little over 1 hour.

Once a collision is found, Mallory then selectively shares different `Change` records to different partitions of the network.

Recommendations

Short term, increase the length of the truncated SHA-256 hash from 160 bits (20 bytes) to at least 192 bits (24 bytes) for the `ChangeHash`. With this change, the cost of finding a collision becomes 2^{96} rather than 2^{80}, which multiplies the time required to find a collision by a factor of 65,536. Consequently, the collision attack is no longer practical.

Long term, whenever reducing the security margin of a cryptographic primitive (e.g., truncating hashes in this case), document why this is done and why the impact on security is considered acceptable.

4. The meanings of the primary key fields created_at and expires_at are undocumented	
Severity: **Medium**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-OCK-4
Target: Identities and Credentials section	

Description

The meanings of the `created_at` and `expires_at` fields on the `ChangeData` type are not sufficiently explained by the design documentation. This may lead users to make security-critical decisions based on a flawed interpretation of these fields.

The `ChangeData` type is used to update a user's primary key. The type's two fields, `created_at` and `expires_at`, define the lifetime of the new key. The Identities and Credentials section mentions that the `expires_at` timestamp indicates "when the `primary_public_key` included in the change should stop being relied on as the primary public key of this identity."

However, the documentation does not specify what happens if a user allows their primary key to expire without signing and broadcasting a new change. Intuitively, it is easy to assume that nodes accept changes only from live keys, so an expired key can no longer sign new changes. However, this is not described in the documentation and there is currently no check in the code performing change history validation to ensure this behavior.

The meaning of the `created_at` field is also not sufficiently explained. It is currently unclear how this field should be validated or acted on by other nodes. In fact, the implementation explicitly allows changes where `created_at` is greater than `expires_at`. This means that the `created_at` field cannot be relied on to define an overall lifetime or validity period for the key.

Exploit Scenario

Alice uses Ockam to set up a network of nodes. One of the nodes is taken offline, and eventually the primary key used for the node expires. Since the key has expired, Alice believes that it is no longer sensitive and does not take proper precautions to either protect or delete the key.

Mallory, a malicious user, gains access to the node and obtains the expired key. She can now create a new change based on the expired key. She broadcasts the updated change history to other nodes in the network. Since nodes do not check the `expires_at` field when the change history is validated, the new change is accepted as valid by other nodes, allowing Mallory to gain access to the network.

Recommendations

Short term, document the expected meanings of the `created_at` and `expires_at` fields, and specify how these fields should be validated. Ensure that changes signed by expired keys are rejected by all nodes. Alternatively, if the lifetime is meant to be enforced only for purpose key attestations, document this restriction and rename the two fields (e.g., to `attestations_not_valid_before` and `attestations_not_valid_after`) to make this clear.

Additionally, clearly specify the full life cycle of secrets and credentials, including any applicable revocation or expiration mechanisms.

Long term, ensure that the documentation always reflects the proper meaning of each value specified by the protocol. In particular, if values have unintuitive or surprising meanings, they should always be documented.

5. Insufficient threat model documentation	
Severity: **Medium**	Difficulty: **Not Applicable**
Type: Cryptography	Finding ID: TOB-OCK-5
Target: All sections	

Description

The threat model for Ockam's protocols is not specified in the documentation. There is no description of the different actors in the system, the assets they value, the other actors they trust, and the security controls for achieving security goals. From the threat model, it should be clear what aspects of the assets are important, including but not limited to confidentiality, integrity or authenticity, and availability.

The design review included weekly discussions between auditors and developers, who provided all relevant threat modeling information for the two in-scope use cases. However, in the absence of a documented general threat model for the protocol, users and developers must make assumptions about the security goals of each component and how these goals are met in the implementation. If any of these assumptions are false, this could lead to surprising behavior and potentially real security issues when users deploy the protocol.

There is no specific exploit scenario for this finding, so the difficulty rating is not applicable.

Recommendations

Short term, add an informal threat model to each use case and section of the documentation to ensure no gaps exist.

Long term, develop a formal threat model that applies to Ockam's protocols. Explicitly state any assumptions that must be true for the protocol to be secure. Define different types of threat actors and specify how the protocol deals with them.

Once a general threat model is in place, each use case and section of the protocol needs a threat model section that describes only the ways in which they deviate from the general model for the overall protocol.

6. The supported signature schemes have different security properties	
Severity: **Informational**	Difficulty: **Not Applicable**
Type: Cryptography	Finding ID: TOB-OCK-6
Target: All sections	

Description

Ockam's supported signature schemes, EdDSA and ECDSA, offer different security guarantees but are used interchangeably. Although this issue is not currently exploitable, if Ockam modifies the system in the future, it could become necessary to rely on additional security guarantees.

The signed change history in Ockam allows an identity to rotate its primary key and attest to the validity of this change. Each rotation is associated with a `Change` data structure that is hashed and then signed by the current primary key and the previous primary key (if it exists). The documentation specifies two signing algorithms: `EdDSACurve25519Signature` and `ECDSASHA256CurveP256Signature`. The former is the preferred algorithm, as the latter algorithm is supported only due to the lack of support for EdDSA in cloud hardware security modules (HSMs). A formal modeling of the security properties of the signature scheme with CryptoVerif reveals a discrepancy between the security guarantees offered by the two schemes.

In terms of security guarantees, EdDSA and ECDSA are equivalent only in that they are both believed to guarantee Existential Unforgeability under Chosen-Message Attacks (EUF-CMA). This means that an attacker who has several valid signatures on various messages will be unable to create a valid signature for a new message.

However, EdDSA, as instantiated, provides additional guarantees that ECDSA does not. In particular, EdDSA provides Strong Unforgeability under Chosen-Message Attacks (SUF-CMA), so an attacker who has a number of valid signatures on various messages will be unable to create a valid new message-signature pair.

ECDSA does not provide this guarantee because for any valid ECDSA signature (`r,s`), the signature (`r,-s`) is also valid. This means that an attacker could take a `Change` block with associated ECDSA signatures and share the same `Change` block with modified (but valid) signatures. When modeling change histories with CryptoVerif, only the strong unforgeability of the first change can be proven.

The issue described above does not pose a direct threat in the current deployment of Ockam in the context of the two in-scope use cases, which do not rely on strong unforgeability. However, from a design perspective, it is desirable to have a clear understanding of what security guarantees are expected from different components, irrespective of their concrete instantiations. Furthermore, any future extension to the protocol might involve beyond unforgeability guarantees like exclusive ownership (which means that *any* valid signature can be created only from the private key corresponding to the public key). Fortunately, the current signing mechanism in Ockam is close to the BUFF construction, which provides beyond unforgeability security.

There is no specific exploit scenario for this finding, so the difficulty rating is not applicable.

Recommendations

Short term, consider adding a brief description of the security properties of the signed change histories and other signed data structures. Then document how the instantiations of different components provide the expected security guarantees. Consider all use cases of signatures in the system and whether any beyond unforgeability guarantees might be expected. If SUF-CMA security is desired, ECDSA can be modified by restricting the s component of the signature to the upper or lower half of its range.

Long term, specify all the cryptographic assumptions each system component is expected to meet for the protocol to be secure. Document how each instantiation of a specific primitive meets the required assumption.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Not Applicable	We did not identify a specific exploit scenario for this finding.
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Design Quality Findings

We identified the following design quality issues by reading the design documentation. These are generally minor inconsistencies and typos that do not correspond to security vulnerabilities. However, resolving them is recommended in case parts of the design documentation are reused in customer-facing documentation:

- The description of the derivation of Ockam Identities is not consistent with the implementation. It states that the identifier is a hash of a `Change` block, which includes a signature. However, the identifier is a hash of only the included `ChangeData` type.
- The image in the section on Purpose Key Attestations is missing a purpose in the `PurposeKeyAttestationData` type (cf. the next image in the Credentials section).
- The image on the Noise handshake in the section on Authenticated Key Establishment does not describe the construction of the payloads containing the `IdentityAndCredentials` type of both the responder and initiator.
- The details of the `Rekey` operation are not specified in the design documentation. The specification mentions that a rekey is done via an encryption operation. However, the details of what is being encrypted are missing, and the specification links only to the Noise specification for rekeying. For completeness, include a description of rekeying in the Ockam documentation as well. This is also more consistent with the description of the handshake, which includes more details from the Noise specification.
- The sliding window mechanism is used to mitigate replay attacks. However, the exact details of how this works are not included in the description of the sliding window mechanism but are covered only implicitly by the examples.
- The title for the Trust Contexts section is misspelled.
- In the TCP Portals use case description, there are three occurrences of the typo "idenity."
- In the Kafka Portals use case description on creating a Kafka producer, the text states that the command to create a Kafka Producer "creates a sidecar to a Kafka Consumer." It further states that certain components and responses "work exactly the same as in the producer sidecar and the consumer sidecar above," but it should refer only to "the consumer sidecar above."

C. Formal Modeling

During the engagement, we used the symbolic model checker Verifpal to formally model different aspects of the protocol. This work and the results obtained are described in this section.

C.1 Verifpal Modeling

Verifpal is a symbolic model checker, much like Tamarin and ProVerif. Its main advantage over other tools is that it focuses on usability first, which allows the user to quickly model new protocols and updates to existing protocols.

Symbolic model checkers like Verifpal model cryptographic primitives as black boxes. Protocol participants and an attacker can perform computations using the defined cryptographic primitives, and algebraic properties of these primitives are modeled using equations. The attacker is allowed to run the protocol an unbounded number of times to observe (in the case of a passive attacker) and modify (in the case of an active attacker) messages in flight. (This is sometimes referred to as the Dolev-Yao model.) In this model it is possible to prove equivalence between terms (e.g., a shared secret computed individually by two protocol participants), as well as model common security properties like confidentiality and authenticity of messages between participants. It is also possible to analyze more complex security properties like forward secrecy, post-compromise security, and security against KCI attacks.

However, there are limits to what can be expressed in the symbolic model. Since cryptographic primitives are modeled in an idealized way, there is no way to analyze attacker advantage or asymptotic security properties. (These types of properties are more naturally expressed in the computational model using tools like CryptoVerif.)

Modeling Ockam Identities Using Verifpal

We used Verifpal to model Ockam Identities and change history validation. The goal of this exercise was to formally prove that an attacker could not tamper with updates to the change history without it being discovered. The following model implements change history validation under the assumption that identities are distributed out of band. For simplicity, we disregard protocol versioning, purpose key revocation, and primary key lifetimes because these fields are irrelevant for the security property we are trying to model.

```
attacker[active]

// Alice creates her first primary key pair.
principal Alice [
  knows private secret_key_0

  // Generate `ChangeData` 0.
  public_key_0 = G^secret_key_0
```

```
  change_data_0 = CONCAT(nil, public_key_0)

  // Compute identity as the hash of the first `ChangeData` and sign the first hash.
  change_hash_0 = HASH(change_data_0)
  signature_0 = SIGN(secret_key_0, change_hash_0)
]

// Alice updates her primary key pair.
principal Alice [
  knows private secret_key_1

  // Generate `ChangeData` 1 and sign its hash.
  public_key_1 = G^secret_key_1
  change_data_1 = CONCAT(change_hash_0, public_key_1)
  change_hash_1 = HASH(change_data_1)
  signature_1 = SIGN(secret_key_1, change_hash_1)
  previous_signature_1 = SIGN(secret_key_0, change_hash_1)
]

// Alice now sends her `ChangeHistory` to Bob.
// Alice's identity (`change_hash_0`) is shared out of band.
Alice -> Bob: [change_hash_0], change_data_0, signature_0  // nil
Alice -> Bob:  change_hash_1,  change_data_1, signature_1, previous_signature_1
```

Figure C.1.1: Alice creates two changes, which are then sent to Bob.

```
// Bob verifies Alice's first `Change` against her identity.
principal Bob [
  // Verify the first `ChangeHash` against Alice's known identity.
  valid_0 = ASSERT(HASH(change_data_0), change_hash_0)?

  // Verify the `ChangeHash` signature against the current key.
  alice_previous_change_0, alice_public_key_0 = SPLIT(change_data_0)
  _ = ASSERT(alice_previous_change_0, nil)?
  _ = SIGNVERIF(alice_public_key_0, change_hash_0, signature_0)?
]

// Dummy message signaling that the `ChangeData` received by Bob was accepted.
Bob -> Alice: [valid_0]

// Bob verifies Alice's second `Change` against her first primary public key.
principal Bob [
  // Verify `ChangeHash` signature against the previous key.
  valid_1 = SIGNVERIF(alice_public_key_0, change_hash_1, previous_signature_1)?
  _ = ASSERT(HASH(change_data_1), change_hash_1)?

  // Verify `ChangeHash` signature against the current key.
  alice_previous_change_1, alice_public_key_1 = SPLIT(change_data_1)
  _ = ASSERT(alice_previous_change_1, change_hash_0)?
  _ = SIGNVERIF(alice_public_key_1, change_hash_1, signature_1)?
]

// Dummy message signaling that the `ChangeData` received by Bob was accepted.
```

```
Bob -> Alice: [valid_1]

// Alice has to use all messages received in order for the model to compile.
principal Alice [
  _ = HASH(valid_0)
  _ = HASH(valid_1)
]
```

Figure C.1.2: Bob validates the two changes against Alice's identity, which is sent out of band.

```
queries [
  // The attacker cannot have tampered with Alice's first `Change` under the
  // assumption that Bob accepts the first `ChangeData` as valid.
  authentication? Alice -> Bob: change_data_0 [
    precondition [Bob -> Alice: valid_0]
  ]
  // The attacker cannot have tampered with Alice's second `Change` under the
  // assumption that Bob accepts the second `ChangeData` as valid.
  authentication? Alice -> Bob: change_data_1 [
    precondition [Bob -> Alice: valid_1]
  ]
]
```

Figure C.1.3: The model checks whether the attacker could have tampered with either of the two `ChangeData` messages under the assumption that Bob accepted the message as valid.

We modeled the protocol using an active attacker under the assumption that Alice's identity is shared out of band with Bob. In Verifpal, this can be achieved using guarded messages (denoted using square brackets). Since each change is validated in multiple steps, we had to indicate to the prover that Bob may compute on untrusted data as part of the validation process. For this reason, we modeled authentication under the additional assumption that Bob successfully validates the corresponding change, which is indicated by sending a dummy message (denoted `valid_0` and `valid_1` above) to Alice.

Running the prover on this model did not identify any attacks on the protocol. This offers a high level of confidence that the design is secure against active attacks.

When not including the additional assumption and the dummy messages, Verifpal describes the following hypothetical attack:

1. An attacker records the first Change and its self-signature.

2. The attacker replaces the second Change with the first Change, setting both signatures to be the recorded self-signature.

3. Now Bob will verify the signature on the second Change using the public key of the first Change, which will be correct since it is the self-signature.

4. Next, Bob will SPLIT the Change data, which Verifpal considers a computation on attacker-provided (and hence non-authentic) data.

It is clear that this attack does not translate to Ockam Identities because the next step of the verification would check the `previous_change_hash` in the Change data against the hash of the first Change, which will fail in this case. However, the attack does highlight a small issue in the design, which is that the `previous_signature` and `signature` fields of a Change are not domain separated and an attacker can exchange or copy them. It is worth considering fixing this issue, although it is currently not exploitable.

To reanalyze the model using Verifpal, save the model defined in figures C.1.1 - C.1.3 above as a single file called `ockam-identities.vp`, and then run Verifpal on the model using the following command:

```
verifpal verify ockam-identities.vp
```

C.2 CryptoVerif Modeling

CryptoVerif is a cryptographic proof assistant and protocol verifier that operates in the computation model. Unlike symbolic verifiers, in CryptoVerif, messages are bitstrings, cryptographic primitives are functions manipulating bitstrings, and the adversary is modeled as a probabilistic polynomial-time Turing machine. Cryptographic proofs are then carried out in the game-playing framework pioneered by Bellare and Rogaway (among others). Concretely, the security of a protocol is analyzed through a game played between a challenger and an adversary. The adversary is given access to several oracles that capture the adversary's capabilities (e.g., the adversary may obtain signatures over arbitrary messages). Finally, a winning condition defines what action the adversary must perform for a given security property to be broken (e.g., the adversary must produce a signature forgery for a new message).

CryptoVerif proves security using sequences of games where the starting game is the protocol that is modeled and the final game is an ideal protocol where the security is given by construction. To deem a protocol secure, the sequence of games must result from small, consecutive modifications that are mainly unnoticeable up to a certain negligible probability. The probability of noticing divergences between games is argued based on the security of a component used in the protocol (e.g., the unforgeability of a signature scheme). Therefore, the output of CryptoVerif when the protocol is secure is a probability bounding the adversary's ability to break the protocol.

Modeling Ockam Identities Using CryptoVerif

We used CryptoVerif to model some aspects of identities in Ockam. Modeling with CryptoVerif is much more time consuming. Therefore, we made several simplifications in our model. The model below analyzes the security of identities upon creation and upon creation of the next primary key.

At a high level, we consider an existing (honest) identity, and we demand that the adversary not be able to claim that identity. Furthermore, we also demand that the adversary may not produce a subsequent primary key that will be accepted. In other words, the adversary should not be able to create a valid signature for the initial change data associated with the identity nor for the subsequent primary key. As we discussed in TOB-OCK-6, an insight from this model is that a valid signature on a change does not imply that it was created using the corresponding private key unless the signature scheme is SUF-CMA secure.

We describe the model below. First, we define parameters and the custom types used in the model (figure C.2.1).

```
(* Parameters *)
type keyseed [large,fixed]. (* Type for seed used for key generation *)
type pkey [bounded]. (* Type for public key *)
type skey [bounded]. (* Type for secret key *)
type signature [bounded]. (* Type for signatures *)
type hashoutput [fixed, large]. (* Type for hash output *)
type hashkey_t [fixed]. (* Type for hash key to model a random Oracle *)
(* NIL constant input in the first change data *)
const NIL: hashoutput.
```

Figure C.2.1: Type definitions and constants (`ockam-identities.ocv:5–14`)

Next, we define the cryptographic primitives we use—namely, a signature scheme and a hash function. To experiment with different assumptions for the signature scheme, we may replace the `SUF_CMA_det_signature` macro with the `UF_CMA_det_signature` macro. In the second case, we assume only EUF-CMA security.

```
(* A hash function modeled as a random oracle *)
expand ROM_hash_2(hashkey_t, hashoutput, pkey, hashoutput, RO, RO_Oracle, qRom).
(* Signatures *)
proba Psign.     (* Breaking the (S)UF-CMA property *)
proba Psigncoll. (* Collision between independently generated keys *)
expand SUF_CMA_det_signature(
  keyseed,     (* Seed used for key generation *)
  pkey,        (* Public key *)
  skey,        (* Secret key *)
  hashoutput,  (* Input space of signature scheme *)
  signature,   (* Signature (output) space of signature scheme *)
  skgen,       (* Secret key generation algorithm *)
  pkgen,       (* Public key generation algorithm *)
  sign,        (* Signature algorithm *)
  verify,      (* Verification algorithm *)
  Psign,
  Psigncoll
).
```

Figure C.2.2: Definition of cryptographic primitives (`ockam-identities.ocv:16–34`)

Next, we define helper functions.

```
(* Helper functions to serialize change data *)
fun serialize_change_1(pkey, pkey, hashoutput, signature, signature, signature):
bitstring [data].
(* Helper function to simplify signature key generation *)
letfun keygen() =
  rk <-R keyseed;
  sk <- skgen(rk);
  pk <- pkgen(rk);
  (sk, pk).
```

Figure C.2.3: Serialization and key generation helpers (`ockam-identities.ocv:36–43`)

Then we define events and a query that captures the security property that we want to analyze—namely, the unforgeability of change histories.

```
(*
  Events and corresponding queries capture the desired properties of the protocol.
  We consider two users, A & B. B knows A's identity and engages in a protocol with
  A,where A proves knowledge of the initial primary public key. In a second
  interaction, A generates a new primary public key and sends the complete change
  history to B.

  `make_change_0` is set to true when A generates a signature over the initial
  change data
*)
event make_change_0(pkey, signature).
(*
  `make_change_1` is set to true when A generates a signature over the new change
  data
*)
event make_change_1(pkey, pkey, hashoutput, signature, signature, signature).
(* `make_change_0` is set to true if B accepts the initial primary key *)
event accept_change_0(pkey, signature).
(* `make_change_1` is set to true if B accepts the new primary key *)
event accept_change_1(pkey, pkey, hashoutput, signature, signature, signature).
(*
  The first query asserts that if B accepts the initial change, then A must have
  generated the signature that B accepted. In other words, any `accept_change_0`
  must correspond to a `make_change_0` event. The query is *injective*, meaning we
  expect each event to happen once as modeled.
*)
query pk: pkey, sig: signature; inj-event(accept_change_0(pk, sig) ) ==>
inj-event(make_change_0(pk, sig)).
(*
  The second query asserts that if B accepts the new change, then A must have
  generated the signature that B accepted. In other words, any `accept_change_1`
  must correspond to a `make_change_1` event. The query is *injective*, meaning we
  expect each event to happen once as modeled.
*)
```

```
query pk_0: pkey, pk_1:pkey, change_hash_1:hashoutput, sig_0: signature, sig_1_0:
signature, sig_1_1: signature; inj-event(accept_change_1(pk_0, pk_1, change_hash_1,
sig_0, sig_1_0, sig_1_1)) ==> inj-event(make_change_1(pk_0, pk_1, change_hash_1,
sig_0, sig_1_0, sig_1_1)).
```

Figure C.2.4: Events and queries (`ockam-identities.ocv:46–73`)

Now we define the process that models users' actions—that is, the generation and verification of signed change histories.

```
(*
  `processB` knows an identity `change_hash0`. The adversary can submit **in
  parallel** a signed change for either the initial primary key or the new primary
  key.
*)
let processB(hf:hashkey_t, change_hash0: hashoutput, prim_pk_0: pkey) =
   (
      (*
        B receives a public key and signature proving as initial change data. B
        verifies the signature and accepts the new identity if the verification
        succeeds. In that case, B sets the event `accept_change_0` to true.
      *)
      OinitB(pk: pkey, sig: signature) :=
      change_hash <- RO(hf, NIL, pk);
      if change_hash = change_hash0 && verify(change_hash0, pk, sig) then
      (
        event accept_change_0(pk, sig);
        return(true)
      )
      else
      (
        return(false)
      )
   )
   |
   (
     (*
       B receives change corresponding to a new change data. B verifies the signed
       history and sets `accept_change_1` to true if verification succeeds.
     *)
     Overfy_change1(new_change: bitstring) :=
      (
        let serialize_change_1(pk_0, pk_1, change_hash_1, sig_0, sig_1_0, sig_1_1) =
new_change in
        computed_change_hash0 <- RO(hf, NIL, pk_0);
        computed_change_hash1 <- RO(hf, change_hash0, pk_1);
        computed_change_hash0_is_valid <- computed_change_hash0 = change_hash0 &&
verify(change_hash0, pk_0, sig_0);
        computed_change_hash1_is_valid <- computed_change_hash1 = change_hash_1 &&
verify(change_hash_1, pk_0, sig_1_0) && verify(change_hash_1, pk_1, sig_1_1);
        if computed_change_hash0_is_valid && computed_change_hash1_is_valid then
        (
```

```
        event accept_change_1(pk_0, pk_1, change_hash_1, sig_0, sig_1_0, sig_1_1);
        return(true)
      )
      else
      (
        return(false)
      )
    )
  ).
```

Figure C.2.5: User processes (`ockam-identities.ocv:101–141`)

Finally, we define the main process that runs the game and exposes the oracles to the adversary. We also define a helper function that generates signing and verification key pairs.

```
process
  Ostart():=
    (* `hf` is a hash key, selecting a hash function at random. *)
    hf <-R hashkey_t;
    (*
      The primary key is generated and the corresponding identity `change_hash0`
      is also computed
    *)
    let (prim_sk_0: skey, prim_pk_0: pkey) = keygen() in
    change_hash0 <- RO(hf, NIL, prim_pk_0);

    return(change_hash0); (* `change_hash0` is returned to the adversary *)

    (*
      The game exposes the oracles defined by `processA` and `processB` to the
      Adversary. The adversary can also query the random oracle via `RO_Oracle`.
    *)
    (
      run processA(hf, prim_sk_0, prim_pk_0)
      |
      run processB(hf, change_hash0, prim_pk_0)
      |
      run RO_Oracle(hf)
    )
```

Figure C.2.6: The main process (`ockam-identities.ocv:144–164`)

We can run CryptoVerif with the following command:

`CryptoVerif ockam-identities.ocv,`

A successful proof shows that all queries have been proved. The output also contains the probability that the adversary may invalidate each query and hence break the security of change histories.

At a high level, the result below shows that breaking the security of change histories is possible only with a small probability related to forging signatures of a secure signature scheme and the probability of creating hash collisions.

```
[...]
RESULT Proved forall
    sig_1_1, sig_1_0_0, sig_0_0: signature,
    change_hash_1: hashoutput,
    pk_1, pk_0_0: pkey;
inj-event(
    accept_change_1(pk_0_0, pk_1, change_hash_1, sig_0_0, sig_1_0_0, sig_1_1)
) ==>
inj-event(
    make_change_1(pk_0_0, pk_1, change_hash_1, sig_0_0, sig_1_0_0, sig_1_1)
)
up to probability
    Psign(time_2, 1) + Psign(time_1, 2) + (1.5 * qRom^2 + 29 + 12 * qRom) /
    |hashoutput| + 10 * Psigncoll
[...]
RESULT Proved forall
    sig: signature,
    pk: pkey;
inj-event(accept_change_0(pk, sig)) ==> inj-event(make_change_0(pk, sig))
up to probability
    Psign(time_1, 2) + (1.5 * qRom^2 + 10 * qRom + 26) /
    |hashoutput| + 7 * Psigncoll
[...]
```

Figure C.2.7: CryptoVerif output excerpt

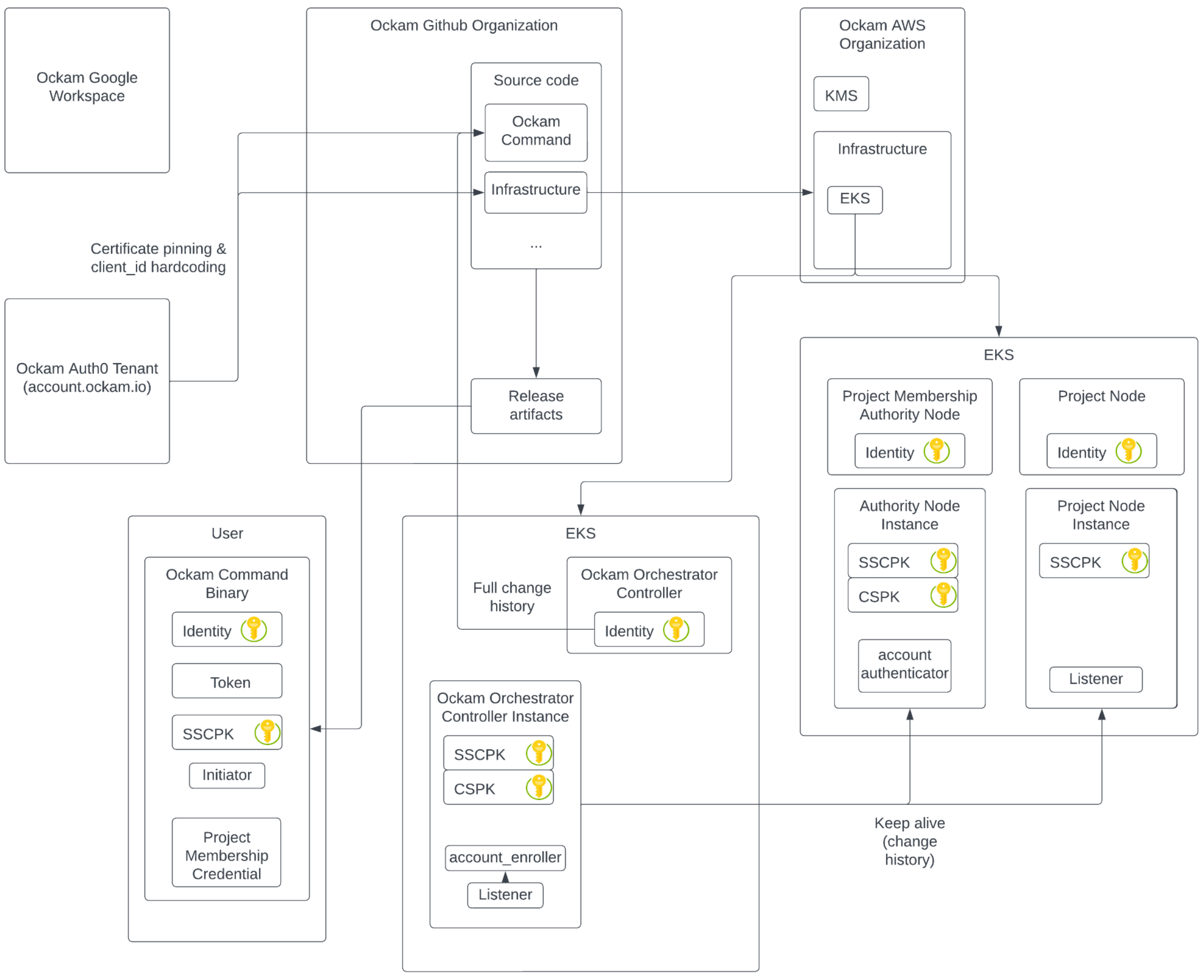
Here, we used the random oracle model for ease of modeling; however, a much weaker collision resistance property would suffice to prove the result.

D. Use Case Diagrams

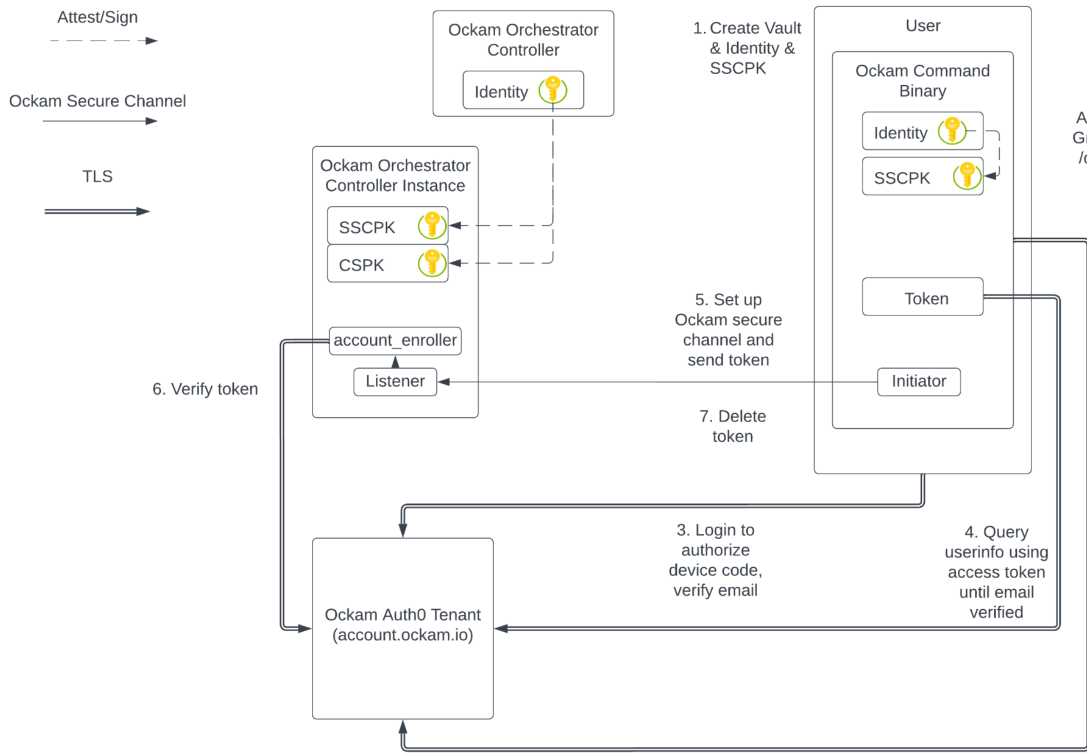
The following diagrams show the various steps taken for the TCP Portals and Kafka Portals use cases, including the relationships between various keys and storage items. Section D.1 contains general diagrams that show processes common to both use cases, while section D.2 contains diagrams for the TCP Portals use case, and section D.3 contains diagrams for the Kafka Portals use case.

D.1 General

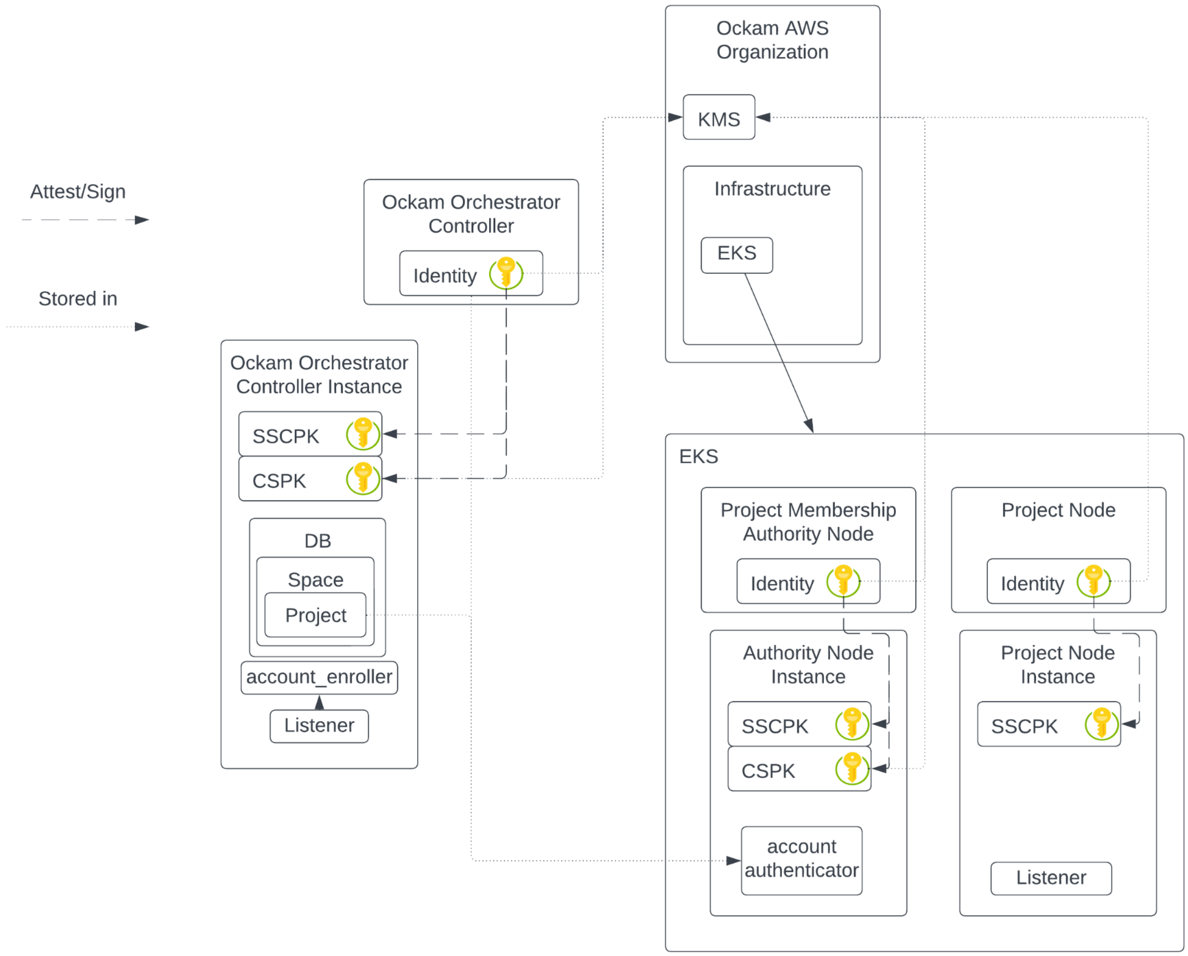
Overview



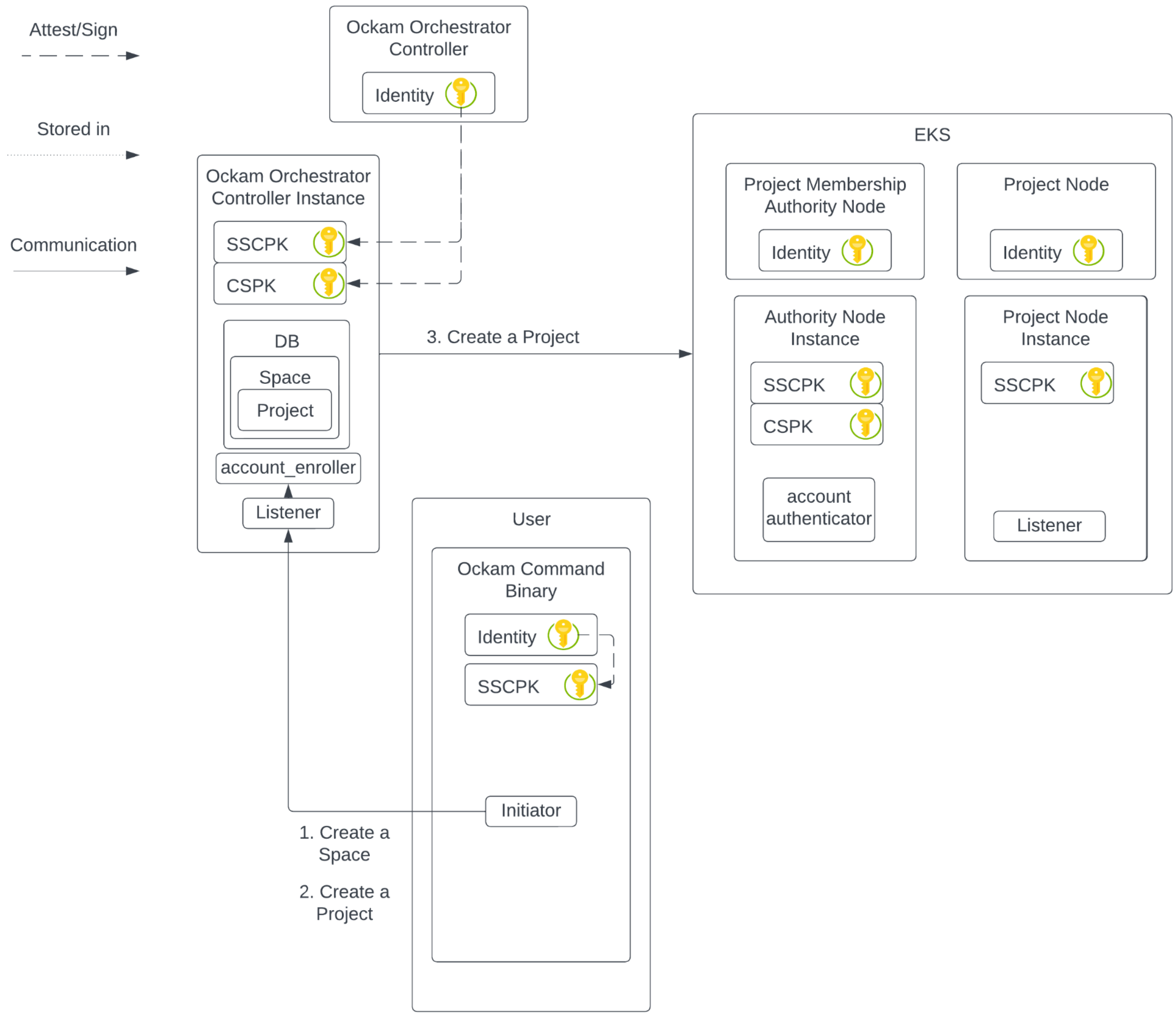
Enroll with the Orchestrator Controller



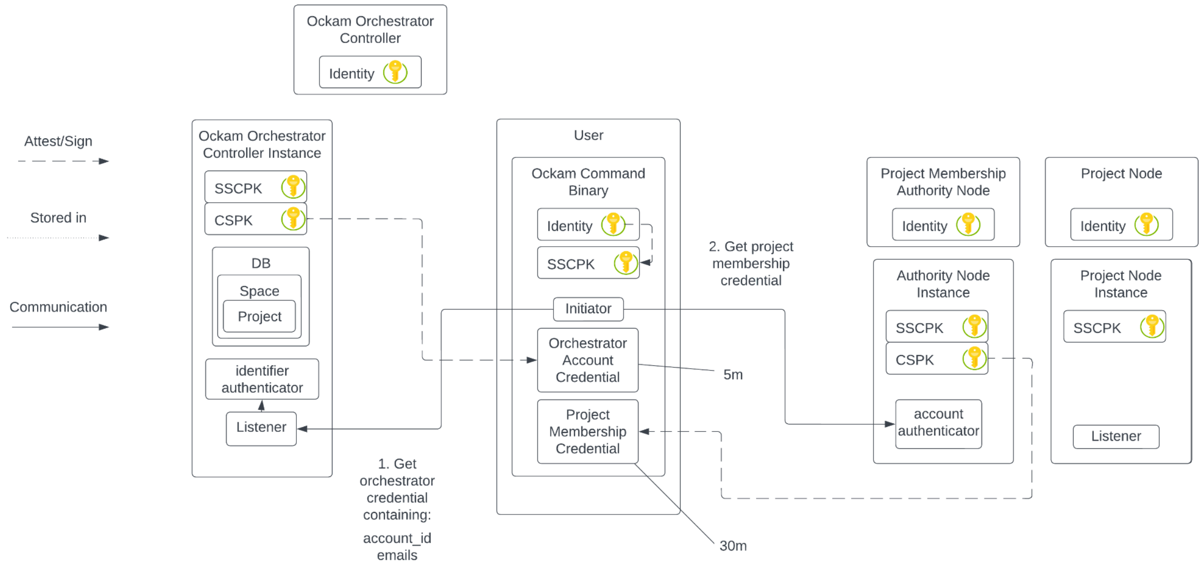
Project Node Hierarchy



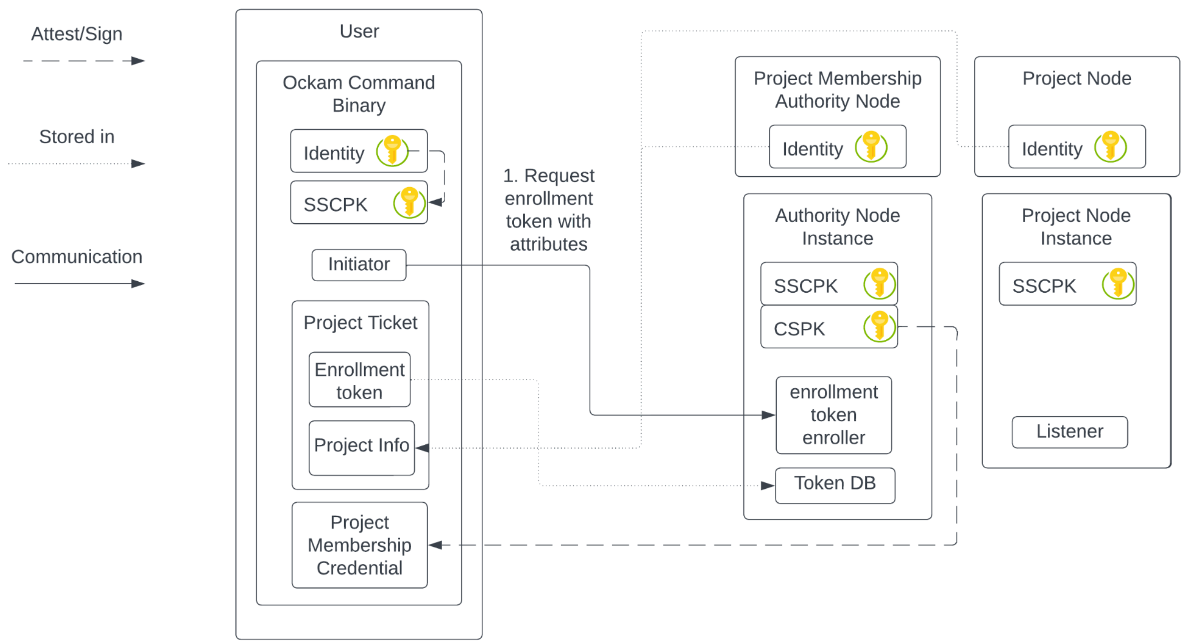
Create Space and Project



Get Project Credentials

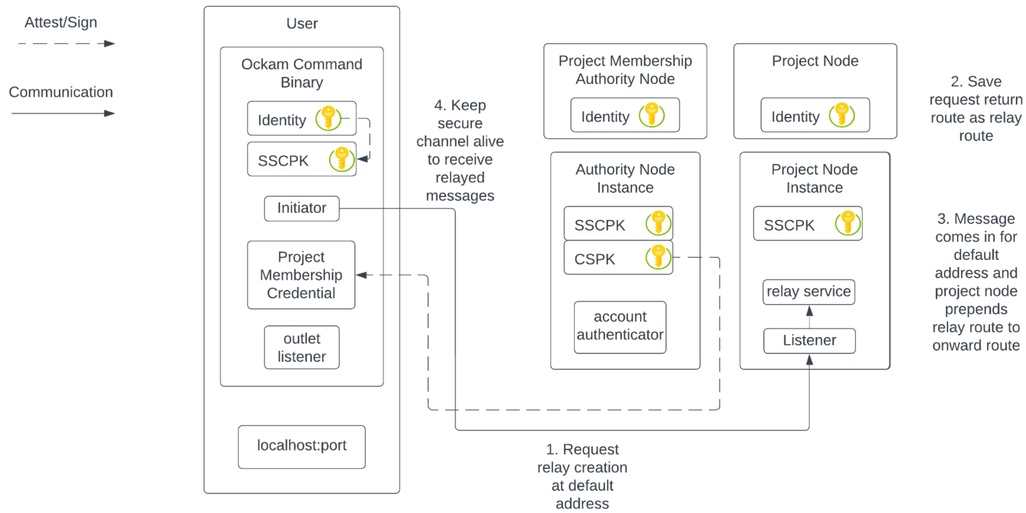


Get Project Ticket

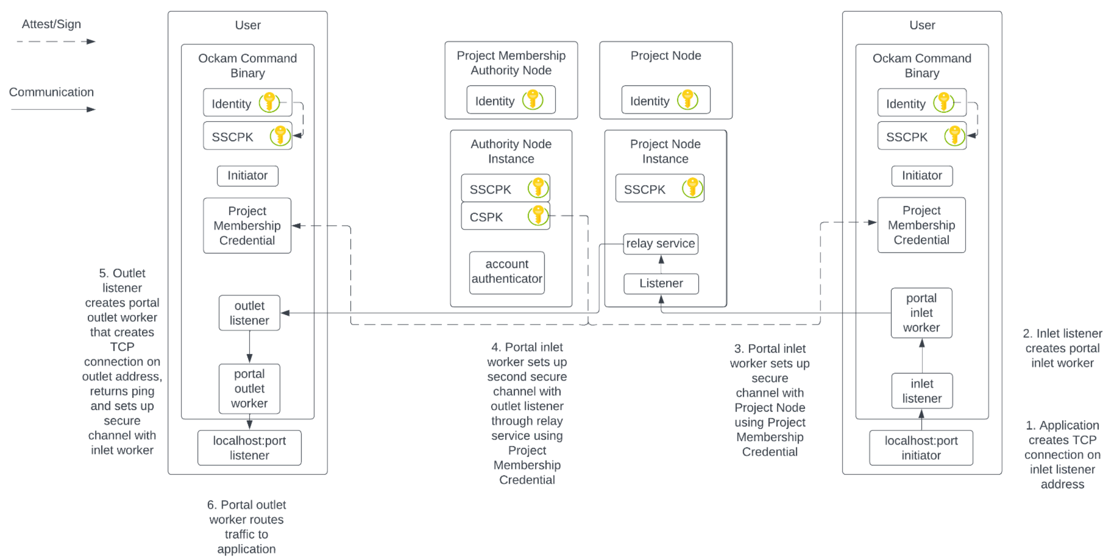


D.2 TCP Portals

Outlet and Relay Creation

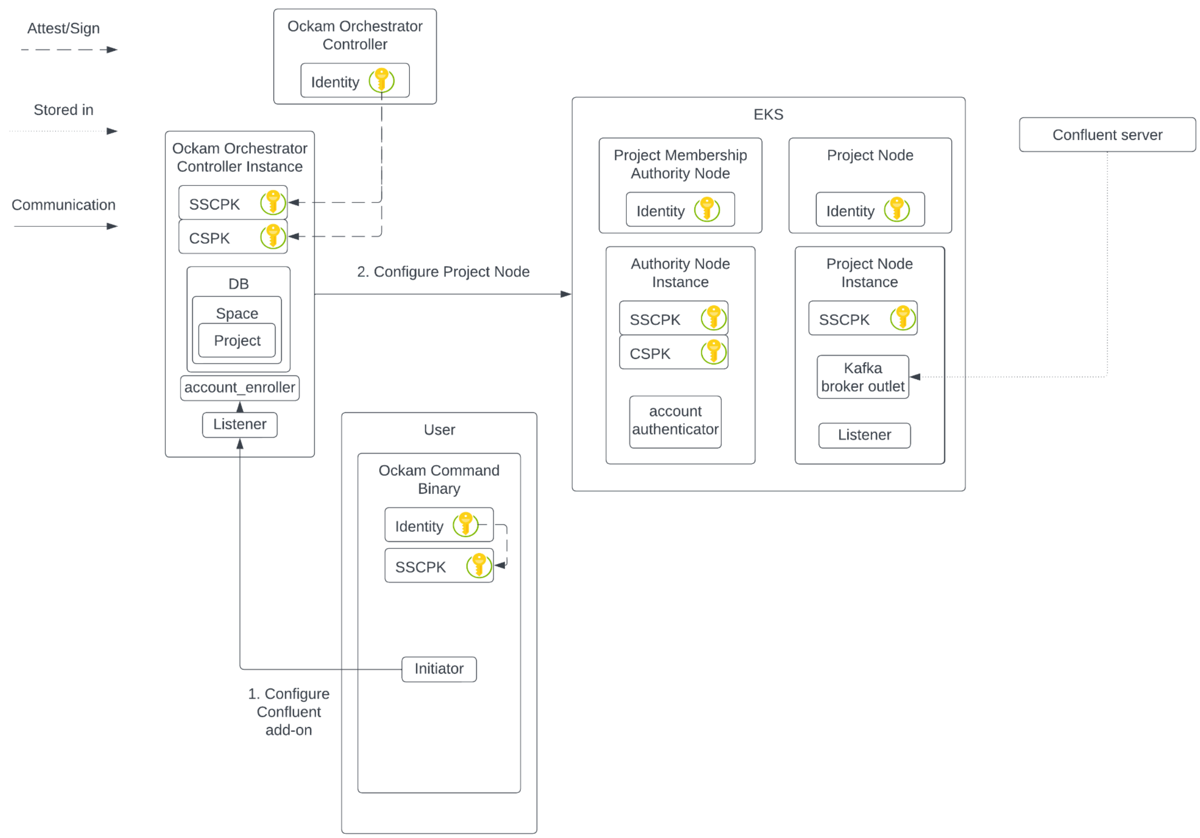


Inlet Creation and Portal

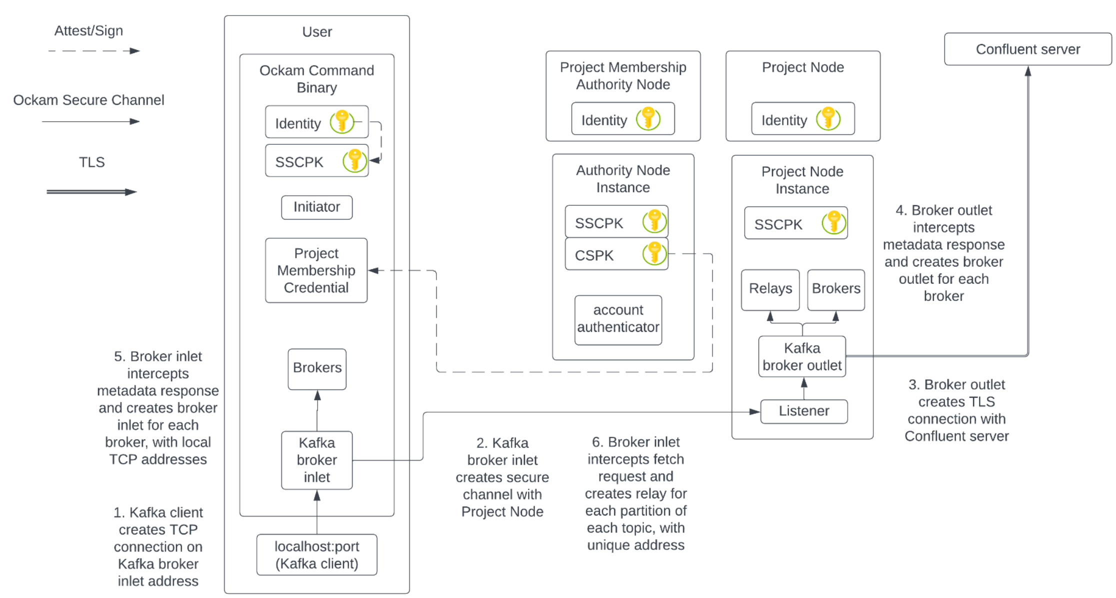


D.3 Kafka Portals

Kafka Config



Kafka Consumer Creation



Kafka Producer Creation and Portal