Security Audit Report

Anoma Q4 2025: AnomaPay (Phase I)

Authors:
Ivan Gavran,
Aleksandar Stojanovic,
Carlos Rodriguez

Last Revised:
19.12.2025

Contents

Audit Overview

The Project

In December 2025, Anoma engaged with Informal Systems to conduct a security audit of some components of the AnomaPay application for private token transfers in the Anoma ecosystem.

Scope of this report

The audit focused on evaluating the correctness and security properties of the resource logic token transfer ZK circuit, the forwarder contracts to wrap, unwrap and migrate resources, and the front-end key management system.

Audit plan

The audit was conducted between December 1st, 2025 and December 16ht, 2025 by the following personnel:

- Ivan Gavran
- Aleksandar Stojanovic
- Carlos Rodriguez

Conclusions

The Heliax team has demonstrated strong engineering practices and a mature security mindset in building a complex privacy-preserving payment system that brings private token transfers for EVM-compatible chains. The codebase exhibits careful attention to cryptographic primitives, thoughtful API design for witness construction, and a well-structured separation between V1 and V2 implementations that enables protocol upgrades through resource migration.

The witness construction libraries build on top of the RISC0 implementation of the Anoma Resource Machine (ARM), with comprehensive validation of resource fields, proper authorization signature verification, and careful handling of ephemeral versus persistent resource lifecycle management. The team's responsiveness to audit findings—particularly the rapid patching of the critical fee-on-transfer token vulnerability through balance verification checks and the constructor validation for migration safety—reflects a proactive security posture. The integration with Uniswap's canonical Permit2 contract for gasless ERC20 approvals demonstrates awareness of ecosystem best practices, while the emergency stop mechanisms and upgradability patterns show consideration for operational safety and long-term maintainability.

The audit identified several significant findings that required immediate attention, the most critical being the fee-on-transfer and rebasing ERC20 token vulnerability that could cause a mismatch between on-chain resource quantities and actual token balances locked in the forwarder contract. This issue, which could lead to either under-collateralized resources (fee-on-transfer case) or transaction failures during unwrap operations (rebasing case), was successfully mitigated through a combination of contract-level balance verification (for fee-on-transfer tokens) and frontend whitelisting (for rebasing tokens, where no optimal contract-level solution exists).

The second medium-severity finding regarding the `ERC20ForwarderV2` constructor storing an intermediate commitment tree root if deployed prematurely—before Protocol Adapter V1 is emergency-stopped—highlighted the importance of deployment ordering to prevent resource stranding during migration. The team addressed this through constructor validation that enforces on-chain verification of V1's stopped status.

The third medium-severity finding revealed that the frontend application was incorrectly encoding cryptographic signatures as UTF-8 text rather than proper hexadecimal bytes when deriving the Key Encryption Key, reducing entropy from 8 bits per byte to approximately 4 bits and making the encrypted keyring vulnerable to offline brute-force attacks if an attacker achieves XSS and exfiltrates the vault from IndexedDB.

It is important to emphasize that the majority of AnomaPay's fundamental security guarantees—particularly protection against double-spending, unauthorized resource consumption, quantity manipulation, and cross-token attacks—are enforced through the ARM RISC Zero implementation and the EVM Protocol Adapter's delta proof verification, rather than through witness construction or forwarder contract logic alone. The ARM's delta proof system ensures transaction balance by enforcing that for any given resource kind (determined by the combination of `logic_ref` and `label_ref` values), the total quantity consumed must exactly equal the total quantity created. This cryptographic invariant prevents inflation or deflation attacks at the protocol level: an attacker cannot mint more resources than they deposit tokens for (Property AP-01), cannot burn resources to receive more tokens than the resource quantity (Property AP-02), and cannot transfer resources in a way that changes the total supply (Property AP-03). For trivial padding resources used to satisfy action tree structure requirements, the resource logic enforces that the quantity field must be zero, making them inherently balanced since zero consumption equals zero creation. The transfer circuit's enforcement of nullifier key ownership and authorization signature verification prevents unauthorized consumption of others' resources (Property AP-02, Threat f; Property AP-03, Threat g), while the Protocol Adapter's nullifier set enforcement prevents double-spending by ensuring each resource can only be consumed once, making these authorization checks effective for the lifetime of each resource. This security architecture means that while the witness construction code analyzed in this audit is critical for generating valid transaction proofs with proper resource field bindings and authorization signatures, the ultimate security boundaries are cryptographically enforced by the Protocol Adapter's on-chain proof verification, which we previously audited and is assumed correct per the threat model assumptions.

Finally, the security of the AnomaPay frontend application—which is responsible for user's keyring management and Permit2 signing—depends on defenses against cross-site scripting (XSS) and JavaScript injection attacks. Any XSS vulnerability could allow attackers to exfiltrate encrypted vaults from IndexedDB, intercept private keys during cryptographic operations, or inject malicious transaction parameters before signing. The signature encoding bug demonstrates this cascading risk: what begins as an encoding error reduces KEK entropy and makes brute-force attacks feasible, completely bypassing the ARM's cryptographic protections if an attacker achieves XSS. The frontend must employ defense-in-depth strategies including strict Content Security Policy (CSP) headers to prevent script injection, and rigorous input sanitization for all user-provided data. Any frontend compromise through XSS or JavaScript injection could allow attackers to exfiltrate private keys and directly steal user funds. An attacker with access to a user's `nk_key` and `auth_sk` can consume all of the user's persistent resources by generating valid zkVM proofs that transfer those resources to attacker-controlled addresses, completely bypassing the ARM's cryptographic protections since the attacker possesses legitimate credentials.

System Overview

AnomaPay is an application for enabling private token transfers in the Anoma ecosystem. It enables users to transfer ERC20 tokens while maintaining privacy through the ARM (Anoma Resource Model) framework, using RISC Zero for ZK circuit proving and verification. The system supports three main operations: **Wrap** (minting ERC20 tokens into private resources using Permit2 signatures), **Unwrap** (burning private resources back to ERC20 tokens), and **Transfer** (private token transfers between users without revealing amounts or participants). Version 2 adds a **Migrate** operation that allows users to upgrade V1 resources to V2 when the V1 Protocol Adapter needs to be deprecated. The backend application orchestrates witness construction, compliance proof generation, logic proof generation via RISC Zero circuits, and transaction submission to the Anoma network, with REST API endpoints for fee estimation and transaction construction.

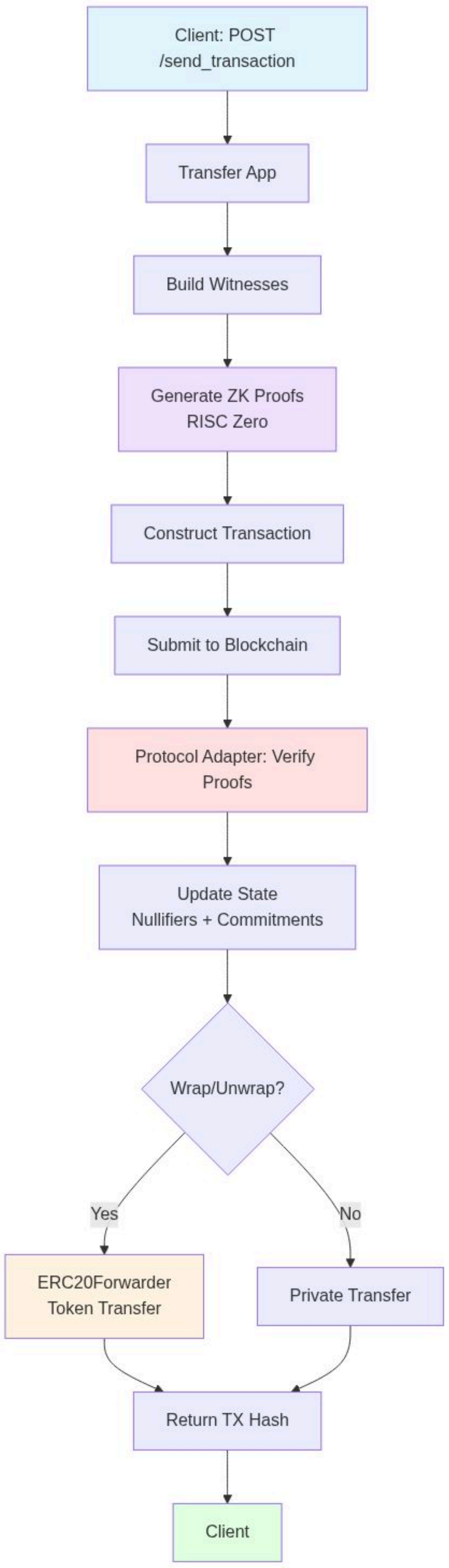
Client: POST /send_transaction
Transfer App
Build Witnesses
Generate ZK Proofs RISC Zero
Construct Transaction
Submit to Blockchain
Protocol Adapter: Verify Proofs
Update State Nullifiers + Commitments
Wrap/Unwrap?
Yes
No
ERC20Forwarder Token Transfer
Private Transfer
Return TX Hash
Client

Figure 2: E2E flow

In the following sections we describe in more detail the components under scope for this engagement.

Transfer library

The transfer libraries for v1 and v2 implement the resource logic and proof generation infrastructure. These libraries define the `TransferLogic` (v1) and `TransferLogicV2` (v2) structs that wrap around their respective witness types and implement the `LogicProver` trait. The libraries provide factory functions for creating resource logics for different operations:

- `consume_persistent_resource_logic()` for consuming existing resources,
- `create_persistent_resource_logic()` for creating new resources,
- `mint_resource_logic_with_permit()` for wrapping ERC20 tokens, and
- `burn_resource_logic()` for unwrapping back to ERC20.

The v2 library extends v1 functionality with migration support through the `migrate_resource_logic()` function, which allows users to migrate a persistent v1 resource to v2 by validating the v1 resource's merkle path of commitment tree and authorization signature. Additionally, v2 includes provides a helper function `construct_migrate_tx()` for building migration transactions. The libraries reference their respective RISC Zero circuit binaries (`TOKEN_TRANSFER_ELF` and `TOKEN_TRANSFER_V2_ELF`) and image IDs (`TOKEN_TRANSFER_ID` and `TOKEN_TRANSFER_V2_ID`) for proof generation.

Transfer witness

The transfer witnesses for v1 and v2 define the witness data structures that contain all information needed for ZK proof generation. The core struct `TokenTransferWitness` (and its v2 counterpart) includes the ARM resource, a boolean indicating whether it's consumed or created, the action tree root, nullifier key, authorization signature, encryption information, forwarder information, label information (forwarder and token addresses), and value information (authorization and encryption public keys). These witnesses implement the `LogicCircuit` trait with a `constrain()` method that generates the `LogicInstance` used in ZK proofs.

The v2 witness enhances v1 with migration capabilities through `ForwarderInfoV2` and `MigrateInfo` structs. The `MigrateInfo` struct contains the migrated v1 resource, its nullifier key, merkle path proving membership in the commitment tree, authorization signature, value information, and both v1 and v2 forwarder addresses. This allows the witness to validate that a v1 resource is properly migrated to v2 during the migration operation.

Integration with application

The witness and library components integrate into the overall AnomaPay application through the REST API orchestrator. When a transfer request arrives, the application's request handlers construct witness data. These witnesses are wrapped in `TransferLogic` or `TransferLogicV2` structs, which are then passed to the proof generation pipeline.

Front-end key management

A client-side cryptographic vault system following Anoma's semi-deterministic key hierarchy, that manages four types of key pairs per user: **authority keys** (express ownership and authorize resource consumption), **nullifier keys** (reflect the right to nullify/spend resources), **encryption keys** (produce resource encryption keys for private state), and **discovery keys** (produce discovery encryption keys for transaction scanning). These keys are generated randomly using HMAC-SHA256, and secp256k1/SHA-256 for public key derivation.

A **Key Encryption Key (KEK)** (derived with HKDF from of a signature of the user's wallet) is used to encrypt the entire keyring with AES-256-GCM. The encrypted vault is stored in the browser's IndexedDB, identified by a SHA-256 hash of the wallet address.

Additionally, the application provides an interface for users to sign **Permit2** data (EIP-712 signatures binding token approvals to specific Anoma action trees), which is then submitted to the Anoma protocol to wrap ERC-20 tokens into resources. The signature authorizes the token transfer without requiring an on-chain approval transaction.

Audit Dashboard

Target Summary

- **Type:** Protocol and implementation
- **Platform:** Rust, Solidity, TypeScript
- **Artifacts:**
  - `anomapay-backend` (at commit `03e60b6`):
    - `contracts/src` (excluding `drafts/ERC20ForwarderV3.sol`).
    - `simple_transfer/tranfer_witness` and `simple_transfer/transfer_witness_v2`.
    - `simple_transfer/tranfer_library` and `simple_transfer/transfer_library_v2`.
  - `pay-interface-app` (at commit `957e8bf`):
    - `src/hooks/useAccountVault.tsx`
    - `src/lib/permit2.ts`
    - `src/domain/keys/models.ts`
    - `src/domain/keys/services.ts`
    - `src/domain/crypto/services.ts`

Engagement Summary

- **Dates**: December 1st, 2025 ⟶ December 16th, 2025
- **Method**: Threat modelling, manual code review

Severity Summary

Finding Severity	Number
Critical	1
High	0
Medium	2
Low	0
Informational	5
Total	**8**

Table 1: Identified Security Findings

Threat Model

Backend

Definitions

Mint request

A mint request wraps existing ERC20 tokens into private Anoma resources by depositing tokens into the forwarder contract via Permit2.

Sample mint request structure:

```
consumed_resources: [
    1× Ephemeral Resource {
      logic_ref: <TRANSFER_LOGIC_IMAGE_ID>,
      label_ref: hash(<FORWARDER_ADDRESS> || <ERC20_TOKEN_ADDRESS>),
      value_ref: hash(<MINTER_ADDRESS>),
      quantity: Q,
      is_ephemeral: true,
      nonce: <NONCE>,
      nk_commitment: commit(<MINTER_NF_KEY>),
      rand_seed: <RANDOM_SEED>,
    }
]

nullifier_key: <MINTER_NF_KEY>

created_resources: [
    1 × Persistent Resource {
      logic_ref: <TRANSFER_LOGIC_IMAGE_ID>,
      label_ref: hash(<FORWARDER_ADDRESS> || <ERC20_TOKEN_ADDRESS>),
      value_ref: hash(<RECEIVER_AUTH_PK || RECEIVER_ENCRYPTION_PK>),
      quantity: Q,
      is_ephemeral: false,
      nonce: <NONCE>,
      nk_commitment: commit(<RECEIVER_NF_KEY>),
      rand_seed: <RANDOM_SEED>,
    }
]

witness_data (for ephemeral consumed): {
    sender_wallet_address: <MINTER_ADDRESS>,
    token_contract_address: <ERC20_TOKEN_ADDRESS>,
    permit2_data: {
      deadline: <PERMIT2_DEADLINE>,
      nonce: <PERMIT2_NONCE>,
      permit2_signature: <PERMIT2_SIGNATURE>,
    }
}

witness_data (for persistent created): {
    receiver_discovery_public_key: <RECEIVER_DISCOVERY_PK>,
    receiver_authorization_verifying_key: <RECEIVER_AUTH_PK>
    receiver_encryption_public_key: <RECEIVER_ENCRYPTION_PK>,
    token_contract_address: <ERC20_TOKEN_ADDRESS>,
}
```

Invariants:

1. `consumed[0].is_ephemeral == true`
2. `created[0].is_ephemeral == false`
3. `consumed[0].quantity == created[0].quantity`
4. `logic_ref` should be the same (`<TRANSFER_LOGIC_IMAGE_ID>`) and match the correct circuit image ID
5. `permit2_signature` signs over `(Q, <ERC20_TOKEN_ADDRESS>, action_tree_root)`, where `action_tree_root` is the root of the action tree with the resources
6. `label_ref` matches `hash(<FORWARDER_ADDRESS> || <ERC20_TOKEN_ADDRESS>)` for both resources
7. `token_contract_address` of witness matches for both resources and matches the token contract address in `label_ref`

Mint requests in general may have [1..n] consumed ephemeral resources and [1..m] created persistent resources. For a given fungibility domain:

- 1 consumed ephemeral resource may be balanced by more than 1 created persistent resources, as long as the total quantity of the created resources matches the quantity of the consumed resource.
- Multiple consumed ephemeral resources may be balanced by 1 created persistent resource, as long as the total quantity of the consumed resources matches the quantity of the created resource.

Sample burn request

A burn request unwraps private ARM resources back into ERC20 tokens by releasing tokens from the forwarder contract to a specified recipient.

Sample burn request structure:

```
consumed_resources: [
    1 × Persistent Resource {
      logic_ref: <TRANSFER_LOGIC_IMAGE_ID>,
      label_ref: hash(<FORWARDER_ADDRESS> || <ERC20_TOKEN_ADDRESS>),
      value_ref: hash(<BURNER_AUTH_PK || BURNER_ENCRYPTION_PK>),
      quantity: Q,
      is_ephemeral: false,
      nonce: <NONCE>,
      nk_commitment: commit(<BURNER_NF_KEY>),
      rand_seed: <RANDOM_SEED>,
    }
]

nullifier_key: <BURNER_NF_KEY>

created_resources: [
    1 × Ephemeral Resource {
      logic_ref: <TRANSFER_LOGIC_IMAGE_ID>,
      label_ref: hash(<FORWARDER_ADDRESS> || <ERC20_TOKEN_ADDRESS>),
      value_ref: hash(<RECEIVER_ADDRESS>),
      quantity: Q,
      is_ephemeral: true,
      nonce: <NONCE>,
      nk_commitment: commit(<RECEIVER_NF_KEY>),
      rand_seed: <RANDOM_SEED>,
    }
]

witness_data (for persistent consumed): {
    sender_authorization_verifying_key: <BURNER_AUTH_PK>,
    sender_encryption_public_key: <BURNER_ENCRYPTION_PK>,
    sender_authorization_signature: <SIGNATURE_OVER_ACTION_TREE_ROOT>,
}

witness_data (for ephemeral created): {
    token_contract_address: <ERC20_TOKEN_ADDRESS>,
    receiver_wallet_address: <RECEIVER_ADDRESS>,
}
```

Invariants:

1. `consumed[0].is_ephemeral == false`
2. `created[0].is_ephemeral == true`
3. `consumed[0].quantity == created[0].quantity`
4. `logic_ref` should be the same (`<TRANSFER_LOGIC_IMAGE_ID>`) and match the correct circuit image ID
5. `label_ref` matches `hash(<FORWARDER_ADDRESS> || <ERC20_TOKEN_ADDRESS>)` for both resources
6. `sender_authorization_signature` exists with valid signature over `action_tree_root` (root of the action tree with the resources)
7. `sender_authorization_verifying_key` matches authorization verifying key in `value_ref`
8. `sender_encryption_public_key` matches authorization verifying key in `value_ref`
9. `receiver_wallet_address` matches the address in `value_ref`

10. `token_contract_address` matches the token contract address in `label_ref`

Burn requests in general may have [1..n] consumed persistent resources and [1..m] created ephemeral resources. For a given fungibility domain:

- 1 consumed persistent resource may be balanced by more than 1 created ephemeral resources, as long as the total quantity of the created resources matches the quantity of the consumed resource.
- Multiple consumed persistent resources may be balanced by 1 created ephemeral resource, as long as the total quantity of the consumed resources matches the quantity of the created resource.

Sample transfer request

A transfer request moves wrapped tokens from one user to another entirely within the ARM privacy layer, without interacting with the forwarder contract.

Sample transfer request structure:

```
consumed_resources: [
    1× Persistent Resource {
      logic_ref: <TRANSFER_LOGIC_IMAGE_ID>,
      label_ref: hash(<FORWARDER_ADDRESS> || <ERC20_TOKEN_ADDRESS>),
      value_ref: hash(<SENDER_AUTH_PK || SENDER_ENCRYPTION_PK>),
      quantity: Q,
      is_ephemeral: false,
      nonce: <NONCE>,
      nk_commitment: commit(<SENDER_NF_KEY>),
      rand_seed: <RANDOM_SEED>,
    }
]

nullifier_key: <SENDER_NF_KEY>

created_resources: [
    1 × Persistent Resource {
      logic_ref: <TRANSFER_LOGIC_IMAGE_ID>,
      label_ref: hash(<FORWARDER_ADDRESS> || <ERC20_TOKEN_ADDRESS>),
      value_ref: hash(<RECEIVER_AUTH_PK || RECEIVER_ENCRYPTION_PK>),
      quantity: Q,
      is_ephemeral: false,
      nonce: <NONCE>,
      nk_commitment: commit(<RECEIVER_NF_KEY>),
      rand_seed: <RANDOM_SEED>,
    }
]

witness_data (for persistent consumed): {
    sender_authorization_verifying_key: <SENDER_AUTH_PK>,
    sender_encryption_public_key: <SENDER_ENCRYPTION_PK>,
    sender_authorization_signature: <SIGNATURE_OVER_ACTION_TREE_ROOT>,
}

witness_data (for persistent created): {
    receiver_discovery_public_key: <RECEIVER_DISCOVERY_PK>,
    receiver_authorization_verifying_key: <RECEIVER_AUTH_PK>
    receiver_encryption_public_key: <RECEIVER_ENCRYPTION_PK>,
    token_contract_address: <ERC20_TOKEN_ADDRESS>,
}
```

Invariants:

1. `consumed[0].is_ephemeral == false`
2. `created[0].is_ephemeral == false`
3. `consumed[0].quantity == created[0].quantity`
4. `logic_ref` should be the same (`<TRANSFER_LOGIC_IMAGE_ID>`) and match the correct circuit image ID
5. `label_ref` matches `hash(<FORWARDER_ADDRESS> || <ERC20_TOKEN_ADDRESS>)` for both resources
6. `sender_authorization_signature` exists with valid signature over `action_tree_root` (root of the action tree with the resources)
7. `sender_authorization_verifying_key` matches authorization verifying key (`<SENDER_AUTH_PK>`) in `value_ref` in the consumed persistent resource
8. `sender_encryption_public_key` matches encryption key (`<SENDER_ENCRYPTION_PK>`) in `value_ref` in the consumed persistent resource
9. `receiver_authorization_verifying_key` matches authorization verifying key (`<RECEIVER_AUTH_PK>`) in `value_ref` in the created persistent resource
10. `receiver_encryption_public_key` matches encryption key (`<RECEIVER_ENCRYPTION_KEY>`) in `value_ref` in the created persistent resource
11. `token_contract_address` matches the token contract address in `label_ref`

Transfer requests in general may have [1..n] consumed persistent resources and [1..m] created persistent resources. For a given fungibility domain:

- 1 consumed persistent resource may be balanced by more than 1 created persistent resources, as long as the total quantity of the created resources matches the quantity of the consumed resource.
- Multiple consumed persistent resources may be balanced by 1 created persistent resource, as long as the total quantity of the consumed resources matches the quantity of the created resource.

Assumptions

- The EVM-compatible blockchain (Ethereum) operates normally with expected block production times, finality guarantees, and network liveness. Transactions submitted with sufficient gas are processed and included in blocks within a reasonable timeframe without significant delays, and the chain does not experience consensus failures or extended periods of downtime.
- The Protocol Adapter correctly prevents double-spending by enforcing nullifier uniqueness. Each resource can be consumed at most once across all transactions. Once a resource's nullifier is added to

the global nullifier set on-chain, that resource becomes permanently unspendable and any subsequent transaction attempting to consume the same resource will be rejected.
- The canonical Uniswap Permit2 contract correctly validates signatures, enforces nonce uniqueness to prevent signature replay attacks, validates that deadlines are not in the past, and prevents signature reuse across different transactions through its nonce bitmap mechanism.

V1 and V2 logic circuit checks

For ephemeral consumed resources (user is minter):

- Check a: Forwarder call type is `CallType::Wrap`.

  For v1: code ref

  For v2: code ref

- Check b: `label_ref` matches `hash(<FORWARDER_CONTRACT> || <ERC20_TOKEN_ADDRESS>)`.

  For v1: code ref

  For v2: code ref

- Check c: Instance of logic circuit includes `<USER_ADDRESS>`, `<ERC20_TOKEN_ADDRESS>`, quantity in ephemeral consumed resource, root of action tree with resources, Permit2 nonce, deadline and signature.

  For v1: code ref

  For v2: code ref

For ephemeral created resources (user is burner):

- Check d: Forwarder call type is `CallType::Unwrap`.

  For v1: code ref

  For v2: code ref

- Check e: `label_ref` matches `hash(<FORWARDER_CONTRACT> || <ERC20_TOKEN_ADDRESS>)`.

  For v1: code ref

  For v2: code ref

- Check f: `value_ref` matches `hash(<USER_ADDRESS>)`.

  For v1: code ref

  For v2: code ref

- Check g: Instance of logic circuit includes `<USER_ADDRESS>`, `<ERC20_TOKEN_ADDRESS>`, quantity in ephemeral created resource.

For v1: code ref

For v2: code ref

For persistent consumed resources (user is either burner or sender):

- Check h: `value_ref` matches `hash(<USER_AUTH_PK> || <USER_ENCRYPTION_PK>)`.

  For v1: code ref

  For v2: code ref

- Check i: Signature of root of action tree with resources is verified with `<USER_AUTH_PK>`.

  For v1: code ref

  For v2: code ref

For persistent created resources (user is either minter or recipient):

- Check j: `label_ref` matches `hash(<FORWARDER_CONTRACT> || <ERC20_TOKEN_ADDRESS>)`.

  For v1: code ref

  For v2: code ref

- Check k: Instance of logic circuit includes ciphertext of (persistent created resource, `<FORWARDER_ADDRESS>`, `<ERC20_TOKEN_ADDRESS>`) encrypted with Diffie-Hellman key of (minter secret key, `<USER_ENCRYPTION_PK>`).

  For v1: code ref

  For v2: code ref

- Check l: Instance of logic circuit includes ciphertext of empty vector encrypted with Diffie-Hellman key of (minter secret discovery key, `<USER_DISCOVERY_PK>`).

  For v1: code ref

  For v2: code ref

Migration V2 logic circuit checks

Additionally to the V1 logic circuit checks, for ephemeral consumed resources during migration:

- Check a: Forwarder call type is `CallType::Migrate`.

  For v2: code ref

- Check b: `label_ref` matches `hash(<FORWARDER_CONTRACT> || <ERC20_TOKEN_ADDRESS>)`.

  For v2: code ref

- Check c: Migrated resource `is_ephemeral` is false.

  For v2: code ref

- Check d: Migrated resource `label_ref` matches `hash(<FORWARDER_V1_CONTRACT> || <ERC20_TOKEN_ADDRESS>)`.

  For v2: code ref

- Check e: Migrated resource `value_ref` matches `hash(<USER_AUTH_PK || USER_ENCRYPTION_PK>)`.

  For v2: code ref

- Check f: Signature of root of action tree with resources is verified with `<USER_AUTH_PK>`.

  For v2: code ref

- Check g: Migrated resource `quantity` matches ephemeral resource quantity.

  For v2: code ref

- Check h: Instance of logic circuit includes `<ERC20_TOKEN_ADDRESS>`, quantity in ephemeral consumed resource, migrated resource nullifier, commitmment tree root, migrated resource `logic_ref`, and `<FORWARDER_V1_CONTRACT>`.

  For v2: code ref

Protocol properties

Protocol properties

Property AP-01

> If a user submits a mint request with a valid Permit2 signature authorizing the transfer of Q tokens, then after the Anoma state transitions are confirmed, the specified recipient(s) will have received usable persistent resource(s) representing Q wrapped ERC20 tokens, and the `ERC20Forwarder` contract will hold Q tokens as backing.

Preconditions**:**

1. User has Q > 0 tokens of ERC20 token at `<MINTER_ADDRESS>`.
2. User has approved Permit2 contract to spend their tokens.
3. User has created valid Permit2 signature with:
   - `token`: ERC20 token address
   - `amount`: Q
   - `nonce`: Unique, unused Permit2 nonce
   - `deadline`: Future timestamp
   - `action_tree_root`: Root of action tree for resources in request

4. Protocol Adapter is deployed and functioning.
5. `ERC20Forwarder` is deployed and registered with PA.
6. zkVM circuits (Transfer Logic) are correctly deployed with matching Image IDs.

Postconditions**:**

1. On-chain state (Protocol Adapter):
   - Commitment for created persistent resource(s) added to commitment tree.
   - Nullifier for ephemeral consumed resource(s) added to nullifier set.
2. On-chain state (ERC20 Token):
   - User's balance decreased: `IERC20(token).balanceOf(<MINTER_ADDRESS>) -= Q`.
   - Forwarder's balance increased: `IERC20(token).balanceOf(forwarder) += Q`.
3. User state (off-chain):
   - Recipient possesses `nk_commitment` binding for created resource(s): `resource.nk_commitment = commit(<RECEIVER_NF_KEY>)`.
   - Recipient possesses `value_ref` binding for created resource: `resource.value_ref = hash(<RECEIVER_AUTH_PK || RECEIVER_ENCRYPTION_PK>)`.

Threats:

- Threat a: Attacker deposits ERC20 tokens of type A, but receives resource(s) for ERC20 token of type B.

  Threat does not hold. The token address is cryptographically bound through three layers: (1) Permit2 signature validation requires the exact token contract address (code ref), (2) the ZK circuit validates that `label_ref = hash(forwarder, token_address)` matches the ephemeral resource's label (v1 code ref, v2 code ref), and (3) the action tree root (signed in Permit2) commits to all resource data including the token-specific `label_ref`. Any resource substitution causes failure. Finally, the match between the deposited token and the received resource is checked through the delta proof connecting the consumed ephemeral resource and created persistent resource.

- Threat b: Attacker deposits Q ERC20 tokens of type A, but the total quantity of persistent resource(s) created is Q', such that Q' != Q.

  Threat does not hold except for the case of fee-on-transfer and rebasing ERC20 tokens (see finding "*Fee-on-transfer and rebasing ERC20 tokens cause resource quantity mismatch*")**.** The Permit2 signature binds the exact amount Q that the user authorizes (code ref). The ZK circuit logic for consumed ephemeral resource logic proofs reconstructs permit data with amount matching the resource quantity (v1 code ref, v2 code ref), and the external payload encodes this permit for forwarder verification. When the forwarder calls Permit2 contract, signature verification ensures the permit amount matches what the user signed (code ref). The delta proof enforces that created persistent resource(s) total quantity equals the consumed ephemeral quantity, ensuring Q tokens deposited results in exactly Q resource quantity created.

- Threat c: User deposits Q ERC20 tokens, but no persistent resource(s) created, or created with 0 amount.

  Threat does not hold. The ephemeral resource will be consumed whenever we trigger wrapping of the resource (v1 code ref, v2 code ref). The created forwarder reconstructs permit data to have that exact quantity (v1 code ref, v2 code ref). The proof, thus, will only be verified if the ephemeral consumed resource matches in quantity the user-signed ERC20 token. Because of the delta proof, it implies a creation of the matching quantity of persistent resources.

The Protocol Adapter's atomic transaction model ensures both token deposit and commitment inclusion in tree succeed or both fail: PA verifies proofs, executes external forwarder calls triggering token transfer via Permit2, then processes logic for each resource adding nullifiers and commitments and updates the commitment tree root. If any step fails, the entire Ethereum transaction reverts, rolling back all state changes including token transfers.

- Threat d: User deposits Q ERC20 tokens, but created persistent resource(s) are not usable (not transferable, not burnable).

  Threat does not hold. Whenever tokens are deposited, a one or more persistent resources are created with `logicRef` corresponding to the token transfer circuit, and the `labelRef` corresponding to the ERC20 token address and the forwarder contract address. As long as the recipient of the created resource(s) holds the keys (authority, encryption, discovery) and the nullifiers for them, they will be able to use the resource. With those keys, the user may spend the resources, for as long as the transfer logic allows it. The logic will check for whether the `value_ref` in the resource and in the witness match (v1 code ref, v2 code ref), and whether the action tree was signed (v1 code ref, v2 code ref) by the owner of the resource.

  The resource payload is encrypted (v1 code ref, v2 code ref) using the key from `valueRef`. Thus, decrypting the information is possible with the same key (to inspect the contents of the resource).

- Threat e: User deposits Q ERC20 tokens, but created persistent resource(s) received by non-intended user.

  Threat does not hold. The resource commitment binds ownership through `value_ref = hash(auth_pk || encryption_pk)` validated in circuit (v1 code ref, v2 code ref), ensuring only the intended recipient with matching keys (as provided in the witness) can later consume the resource. The Protocol Adapter adds this commitment to the commitment tree without inspecting its contents —ownership binding happens cryptographically in the proof. An attacker cannot modify `value_ref` without invalidating the ZK proof that PA verifies before accepting the commitment.

- Threat f**:** Permit2 signature doesn't sign over the `action_tree_root` that includes all intended resources.

  Threat does not hold. The Permit2 signature cryptographically commits to the action tree root through the witness mechanism: the EIP-712 witness type string defines `Witness(bytes32 actionTreeRoot)` structure (code ref) and the forwarder passes the witness hash to Permit2 via `witness: ERC20ForwarderPermit2.Witness({actionTreeRoot: actionTreeRoot}).hash()` (code ref).

- Threat g: Ephemeral resources trigger other external calls besides the call to `forwardCall()` from Protocol Adapter.

  Threat does not hold. The Protocol Adapter processes external payloads exclusively through `_executeForwarderCalls` invoked in `_processLogic` (code ref), which iterates through each resource's external payload (code ref). Each forwarder call executes `IForwarder(untrustedForwarder).forwardCall` (code ref) with output validation (code ref). The `ERC20Forwarder` restricts operations to Wrap/Unwrap via `CallType` enum (code ref), and `ERC20ForwarderV2` adds MigrateV1 as the only additional operation type.

This is the only code path where ephemeral resources trigger external calls—no other mechanisms exist in the Protocol Adapter for resource—initiated external execution.

Property AP-02

If a user submits a valid burn request with a valid authorization signature, then after the Anoma state transitions are confirmed, the user will have consumed their persistent resource(s) representing Q wrapped ERC20 tokens, the `ERC20Forwarder` contract will have transferred Q tokens to the specified recipient address(es), and the forwarder's balance will be reduced by Q tokens.

Preconditions:

1. User possesses a valid persistent resource with:
   - `logic_ref = <TRANSFER_LOGIC_IMAGE_ID>`
   - `label_ref = hash(<FORWARDER_ADDRESS> || <ERC20_TOKEN_ADDRESS>)`
   - `value_ref = hash(<BURNER_AUTH_PK || BURNER_ENCRYPTION_PK>)`
   - `quantity = Q`
   - `is_ephemeral = false`
2. User knows the nullifier key: `resource.nk_commitment = commit(<USER_NF_KEY>)`.
3. User signs valid authorization signature over `action_tree_root`.
4. Forwarder contract has ≥ Q tokens of the ERC20 token.
5. Protocol Adapter is deployed and functioning.
6. `ERC20Forwarder` is deployed and registered with Protocol Adapter.
7. zkVM circuits (Transfer Logic) are correctly deployed with matching Image IDs.

Postconditions:

1. On-chain state (Protocol Adapter):
   - Nullifier for consumed persistent resource(s) added to nullifier set.
   - Commitment for created ephemeral resource(s) added to commitment tree.
2. On-chain state (ERC20 Token):
   - Recipient's balance increased: `IERC20(token).balanceOf(<BURNER_ADDRESS>) += Q`.
   - Forwarder's balance decreased: `IERC20(token).balanceOf(forwarder) -= Q`.
3. User state (off-chain):
   - Persistent resource(s) is spent (nullifier published on-chain).
   - User no longer possesses usable resource(s).

Threats:

- Threat a: Attacker burns resource for ERC20 token type A, but receives ERC20 tokens of type B from the forwarder.

  Threat does not hold. The circuit requires that for a created ephemeral resource, the call of the forwarder information is of type `Unwrap` (v1 code ref, v2 code ref).

  The ephemeral resource's `labelRef` contains the ERC20 address (v1 code ref, v2 code ref), and the same address is included in the forwarder data (v1 code ref, v2 code ref).

The delta proof enforces that consumed persistent resource (token type A) and created ephemeral resource (token type A) must have matching `label_ref` (as this is part of the resource kind), preventing cross-token burning. The forwarder's `_unwrap` function transfers the exact token specified in the external payload (code ref).

- Threat b: Attacker burns persistent resources with total quantity Q, but recipient receives a quantity of Q' ERC20 tokens, such that Q' != Q.

  Threat does not hold except for the case of fee-on-transfer and rebasing ERC20 tokens (see finding "*Fee-on-transfer and rebasing ERC20 tokens cause resource quantity mismatch*")**.** Because of the delta proof, the ephemeral resource created and the burned persistent resource will only be balanced if Q' == Q. This quantity will match the quantity (v1 code ref, v2 code ref) defined in the unwrap call.

  The forwarder's `_unwrap` function transfers exactly the amount specified in the external payload via `safeTransfer` (code ref), which reverts on failure. The Protocol Adapter's atomic execution ensures both nullifier addition and token transfer succeed or both revert, preventing partial state updates.

- Threat c: User burns persistent resource(s) with quantity Q, but forwarder fails to transfer tokens, or transfers 0 tokens to recipient.

  Threat does not hold. The circuit requires that for a created ephemeral resource, the unwrap call is defined with the matching quantity (v1 code ref, v2 code ref). The delta proof will bound the amount in the ephemeral with total amounts in burned persistent resources.

  The Protocol Adapter's atomic transaction model ensures both persistent resource consumption and token transfer succeed or both fail: PA verifies all proofs, executes external forwarder calls (code ref) triggering token transfer via `safeTransfer` (code ref), then adds nullifiers to the set. The `safeTransfer` call reverts if the forwarder has insufficient balance or if the transfer fails for any reason. If any step fails, the entire Ethereum transaction reverts, rolling back nullifier addition.

- Threat d: User burns persistent resource(s) but tokens are sent to a non-intended recipient address.

  Threat does not hold. The circuit checking the ephemeral resource creation (which encodes the Unwrap call) makes sure that the `user_addr` encoded in the forwarder information of the witness matches the address in the resource (v1 code ref, v2 code ref).

  Then the forwarder's `_unwrap` function transfers the tokens to the account specified in the external payload via `safeTransfer` (code ref), which reverts on failure.

- Threat e: User burns persistent resource(s) when authorization signature is missing or doesn't sign over the `action_tree_root` that includes all intended resources.

  Threat does not hold. There is an explicit check (v1 code ref, v2 code ref) for the signature over `action_tree_root`.

  The Protocol Adapter computes the action tree root from all nullifiers and commitments in the action (code ref), then assembles the logic instance with this computed root (code ref) and sends it to the verifier (code ref). A malicious user signing over a different action tree root would cause the ZK proof verification to fail, as the proof was generated with a different action tree root than what PA computed from the actual transaction resources.

- Threat f: Attacker without knowledge of the nullifier key or authorization signing key is able to burn someone else's persistent resource.

  Threat does not hold. The `tag` can only be calculated (v1 code ref, v2 code ref) from the knowledge of the nullifier key. Without it, the attacker cannot produce the correct commitment (this is a general property of spending resources in Anoma). The circuit logic for consumed persistent resources validates that the `value_ref` of the resource is constructed with the authority and encryption public keys of the user that can consume the resource, and only they (by possessing the corresponding private keys) can decrypt the resource information and produce the necessary signature to consume it. The `auth_pk` is then used to verify the signature (v1 code ref, v2 code ref) over the `action_root`.

- Threat g: Persistent resources trigger call to `forwardCall()` from Protocol Adapter.

  Threat does not hold. The Protocol Adapter only executes external forwarder calls for resources with non-empty external payloads (code ref). In the circuit logic, only ephemeral resources generate external payloads (v1 code ref, v2 code ref), while persistent resources return empty external payloads both when consumed (v1 code ref, v2 code ref) and when created (v1 code ref, v2 code ref). Therefore, persistent resources cannot trigger `forwardCall` as they never generate external payloads that PA would execute.

Property AP-03

> If a user submits a transfer request with valid authorization signature, then after the Anoma state transitions are confirmed, the user will have consumed their persistent resource(s) representing Q wrapped ERC20 tokens, and the specified receiver(s) will possess new persistent resource(s) that represent in total Q wrapped ERC20 tokens of the same token type, with no forwarder contract interaction.

Preconditions:

1. User possesses a valid persistent resource with:
   - `logic_ref = <TRANSFER_LOGIC_IMAGE_ID>`
   - `label_ref = hash(<FORWARDER_ADDRESS> || <ERC20_TOKEN_ADDRESS>)`
   - `value_ref = hash(<SENDER_AUTH_PK || SENDER_ENCRYPTION_PK>)`
   - `quantity = Q`
   - `is_ephemeral = false`
2. User knows the nullifier key: `resource.nk_commitment = commit(<SENDER_NF_KEY>)`.
3. User signs valid authorization signature over `action_tree_root`.
4. Protocol Adapter is deployed and functioning.
5. zkVM circuits (Transfer Logic) are correctly deployed with matching Image IDs.

Postconditions:

1. On-chain state (Protocol Adapter):
   - Nullifier for consumed persistent resource(s) added to nullifier set.
   - Commitment for created persistent resource(s) added to commitment tree.
2. Sender state (off-chain):
   - Sender's persistent resource(s) is spent (nullifier published on-chain).
   - Sender no longer possesses usable resource(s).
3. Receiver state (off-chain):

- Receiver possesses `nk_commitment` binding for created resource(s): `resource.nk_commitment = commit(<RECEIVER_NF_KEY>)`.
- Receiver possesses `value_ref` binding for created resource(s): `resource.value_ref = hash(<RECEIVER_AUTH_PK || RECEIVER_ENCRYPTION_PK>)`.
- Receiver can decrypt resource details using their encryption key.

Threats:

- Threat a: Attacker transfers persistent resource(s) of ERC20 token type A, but recipient receives persistent resource(s) for ERC20 token type B.

  Threat does not hold. If the consumed resources and the created resources would represent different ERC20 token contracts (by having different contract addresses included in the `value_ref`) then these resources would have non-matching kinds (code ref) and as a result the transaction would not be balanced and the verification of the delta proof would fail. The verification of the delta proof in the Protocol Adapter makes sure consumed and created resources are in balance per resource kind.

- Threat b: Attacker transfers persistent resource(s) with quantity Q, but recipients receive persistent resource(s) with total quantity Q' where Q' != Q.

  Threat does not hold. If Q' != Q, then transaction would not be balanced. For the same number of consumed and created resources, if the total quantity of all consumed resources is Q, and the total quantity of all created resources is Q' (with Q' != Q) then the delta value (code ref) would not reduce to `ProjectivePoint::GENERATOR * rcv_scalar` and the verification of the delta proof in the Protocol Adapter would fail.

- Threat c: Sender transfers Q tokens, but recipient does not receive persistent resource(s), or receives persistent resource(s) with 0 amount.

  Threat does not hold. If the created resource(s) have a `value_ref` correctly constructed with the authority and encryption public keys of the intended recipient(s) and a commitment of the nullifier key of the intended recipient(s), then the intended recipient will be able to inspect the contents of the resource and consume it. In case of errors in the construction of these values, the resource might become unusable, but there is nothing that the protocol can do to prevent this. If the total quantity of consumed resources is Q, the total quantity of created resources cannot be 0, as the delta proof verification would fail (following the same argument as explained in the previous threat).

- Threat d: Sender transfers Q tokens, but created persistent resource(s) not usable by receiver (not transferable, not burnable).

  Threat does not hold. The created resource is bound to the witness-supplied forwarder contract through `label_ref` constraint (v1 code ref, v2 code ref). In the `value_ref` constraint (v1 code ref, v2 code ref), the resource is bound to an authorization key (using which we will be able to spend it at a later point).

- Threat e: Sender transfers resource but created resource is received by a non-intended user receiver.

  Threat does not hold. The `value_ref` constraint (v1 code ref, v2 code ref) at resource creation, binds the resource to an authorization key. At resource consumption, there is a constraint that the owner of the spent user has signed over the whole action tree (v1 code ref, v2 code ref) (and, thus, the receiver is intended).

- Threat f: Sender transfers resource when authorization signature is missing or doesn't sign over the `action_tree_root` that includes all intended resources.

  Threat does not hold. At resource consumption, there is a constraint that the owner of the consumed resource has signed over the whole action tree (v1 code ref, v2 code ref) (and, thus, the receiver is intended).

- Threat g: Attacker without knowledge of the nullifier key or authorization signing key is able to transfer someone else's persistent resource.

  Threat does not hold. When consuming persistent resource(s) the `tag` can only be calculated (v1 code ref, v2 code ref) from the knowledge of the nullifier key. Without it, the attacker cannot produce the correct commitment (this is a general property of spending resources in Anoma). The circuit logic for consumed persistent resources validates that the `value_ref` of the resource matches `hash(auth_pk || encryption_pk)` with the authority and encryption public keys of the user that can consume the resource. The `auth_pk` is then used to verify the signature (v1 code ref, v2 code ref) over the `action_root`.

Property AP-04

> When a new version of Protocol Adapter and Forwarder contract are live, users can migrate only once their persistent (non-consumed) resources whose commitments are present in the final commitment tree root of the previous version of Protocol Adapter.

Preconditions:

1. PAv1 has stopped operating and published its final commitment tree root.
2. User owns a valid persistent (non-ephemeral) resource in PAv1's final state.
3. User has the merkle path proving their resource commitment exists in V1's commitment tree.
4. User has valid authorization signature over the migration action tree root.
5. `ERC20ForwarderV2` contract is deployed and operational.
6. `ERC20ForwarderV2` address is set as the `_emergencyCaller` of `ERC20Forwarder`:
   1. The emergency committee (multisig) sets it after PAv1 stops (code ref).
   2. Without this precondition, every migration attempt fails because V1 would reject the emergency call from V2 (code ref1, ref2).

Postconditions:

1. PAv1 resource nullifier is recorded in V2 contract to prevent double-migration.
2. New V2 resource is created with same quantity as V1 resource.
3. V2 resource has correct `label_ref` for V2 forwarder.
4. User retains ownership via same `auth_pk` and `encryption_pk` in V2 resource.

Threats:

- Threat a: Migration of non-existent PAv1 resources.

  Threat does not hold. Inside the circuit, a Merkle tree root is calculated for the resource commitment from the witness path (code ref). Equality of that root with the final `V1` root ensures the resource commitment exists in PAv1's final tree. At the forwarder level, FWv2 validates migration metadata consistency by checking `rootV1`, `logicRefV1`, and `forwarderV1` match the expected values stored at deployment (code ref). Additionally, the forwarder verifies the nullifier is not in PAv1's nullifier set, confirming the resource was not consumed before V1 stopped (code ref).

- Threat b: Double migration of same PAv1 resource.

  Threat does not hold. The forwarder implements double migration prevention through nullifier tracking: it first verifies the nullifier is not in PAv1's nullifier set, confirming the resource was not consumed before V1 stopped (code ref), then adds the nullifier to FWv2's internal `migratedNullifiers` set (code ref). The `_addNullifier` function reverts with `PreExistingNullifier` if the nullifier already exists (code ref), making it impossible to migrate the same resource twice.

- Threat c: User is unable to migrate PAv1 resource.

  Threat does not hold except for the case of fee-on-transfer and rebasing ERC20 tokens (see finding "*Fee-on-transfer and rebasing ERC20 tokens cause resource quantity mismatch*"). If preconditions are met and the user provides valid migration parameters, the forwarder allows migration by validating metadata consistency (root, logic reference, forwarder address) and preventing double migration through nullifier checks (code ref). Migration succeeds if FWv1 holds sufficient token balance for the transfer (code ref). The circuit will impose an additional constraint that the owner's `value_info.auth_pk` validates the signature over the v2 authorization domain and the action tree root (code ref).

- Threat d: PAv1 resources remain locked and are not possible to migrate.

  Threat does not hold. Tokens in FWv1 remain accessible for migration through the emergency call mechanism. During migration, FWv2 calls `forwardEmergencyCall` on FWv1 with an unwrap instruction (code ref), which executes `_unwrap` to transfer tokens from FWv1 to FWv2 (code ref). This requires V2 to be set as the emergency caller on V1 (stated in preconditions). There is no lockup mechanism preventing token withdrawal from FWv1 during migration.

- Threat e: PAv2 resource is unusable (not transferable, not burnable) after migration.

  Threat does not hold. The forwarder's role in migration is limited to validating metadata, transferring tokens from FWv1 to FWv2, and emitting the `Wrapped` event to signal tokens are backing V2 resources. Resource creation in PAv2 with `label_ref`, `value_ref`, and `nk_commitment` constructed with the information of the intended recipient of the V2 resource is handled by the Protocol Adapter after verifying the ZK proof (ensuring the created resource is transferable and burnable). If the information of the intended user is correct and PA stores commitment for the rescue, the intended user will be able to use the resource.

- Threat f: Non legitimate owners of PAv1 resources are able to migrate them.

  Threat does not hold. The forwarder does not verify resource ownership—the `_migrateV1` function receives only token address, amount, nullifier, and metadata (code ref) without any authorization signature or ownership proof. Authorization is validated by the ZK circuit, which requires the user to sign over the action tree root with the authorization key matching the one used in the migrated resource's `value_ref` (code ref). The Protocol Adapter verifies this proof before invoking the forwarder. The circuit requires that the key `value_info.auth_pk` validates the v2 signature over the whole action tree. An attacker without the correct authorization key cannot generate a valid proof, preventing illegitimate migration attempts from reaching the forwarder.

- Threat g: Total quantity of resource(s) in PAv2 is != as quantity of migrated resource in PAv1.

  Threat does not hold except for the case of fee-on-transfer and rebasing ERC20 tokens (see finding “*Fee-on-transfer and rebasing ERC20 tokens cause resource quantity mismatch*”). At the forwarder level, the same `amount` value is used consistently: FWv2 emits `Wrapped` with this amount (code ref), passes it to FWv1's emergency call (code ref), which emits `Unwrapped` and transfers the tokens (code ref). The forwarder does not verify the actual received balance matches the specified amount. The ZK circuit enforces that the V2 ephemeral resource quantity equals the V1 persistent resource quantity, and the delta proof ensures that ephemeral resource quantity equals the created persistent resource quantity. which PA verifies before calling the forwarder.

- Threat h: Resource migration can be executed when PAv1 has not been emergency stopped.

  Threat does not hold. Migration requires emergency calls from FWv2 to FWv1, which enforces that PAv1 must be stopped. During migration, FWv2 calls `forwardEmergencyCall` on FWv1 (code ref), which validates that PAv1 is emergency-stopped by calling `_checkEmergencyStopped()` (code ref). This check calls `isEmergencyStopped()` on the protocol adapter (code ref) and reverts with `ProtocolAdapterNotStopped` if PAv1 is still operating. Therefore, migration cannot execute while PAv1 is running—the emergency call mechanism enforces precondition 1 on-chain.

  Note: While migration execution is blocked, premature V2 deployment before PAv1 stops can cause resources to be stranded due to root mismatch—see finding “*ERC20ForwarderV2 constructor does not verify PAv1 is emergency-stopped*”.

- Threat i: Forwarder V2 completes migration operation but fails to emit `Wrapped` event or emits it with incorrect parameters (wrong token address, sender, or amount).

  Threat does not hold except for the case of fee-on-transfer and rebasing ERC20 tokens (see finding “*Fee-on-transfer and rebasing ERC20 tokens cause resource quantity mismatch*”)**.** The `_migrateV1` function unconditionally emits the `Wrapped` event with correct parameters from the migration input: `token` address, `from` set to `address(_ERC20_FORWARDER_V1)` indicating tokens are coming from V1, and `amount` (code ref). The event emission occurs before the emergency call to transfer tokens from FWv1 (code ref). If the token transfer fails, the entire transaction reverts, rolling back the event emission. This ensures the `Wrapped` event is only persisted when migration completes successfully.

- Threat j: Forwarder V1 completes migration operation but fails to emit `Unwrapped` event or emits it with incorrect parameters (wrong token address, recipient address, or amount).

  Threat does not hold except for the case of fee-on-transfer and rebasing ERC20 tokens (see finding “*Fee-on-transfer and rebasing ERC20 tokens cause resource quantity mismatch*”)**.** During migration, FWv2 calls `forwardEmergencyCall` on FWv1 with an unwrap instruction encoding `token`, `receiver = address(FWv2)`, and `amount` (code ref). This triggers FWv1's `_unwrap` function (code ref), which unconditionally emits the `Unwrapped` event with parameters decoded from the input: `token` address, `to` set to FWv2's address (the recipient), and `amount` (code ref). The event emission occurs before the `safeTransfer` (code ref). If the token transfer fails, the entire transaction reverts, rolling back the event emission. This ensures the `Unwrapped` event is only persisted when migration completes successfully.

Implementation properties

Property AP-05

The `ERC20Forwarder` contract implements proper access control, properly validates input parameters, uses safe ERC20 operations to handle non-standard tokens, correctly handles special token types, and follows Checks-Effects-Interactions pattern to prevent reentrancy attacks.

Threats:

- Threat a: Attacker succeeds to call `forwardCall()` directly bypassing Protocol Adapter.

  Threat does not hold. The `forwardCall` function enforces that only the Protocol Adapter can invoke it through caller validation (code ref). The `_checkCaller` function verifies `msg.sender == _PROTOCOL_ADAPTER` and reverts with `UnauthorizedCaller` if the caller is not the authorized Protocol Adapter (code ref). An attacker calling this function directly would be rejected.

- Threat b: Protocol Adapter can execute `forwardCall()` after being emergency stopped.

  Threat does not hold. The `forwardCall` function does not check if the Protocol Adapter is emergency-stopped (this is by design). When the PA is stopped, it cannot process transactions or invoke forwarders. The emergency stop happens at the PA level, preventing PA from calling `forwardCall` in the first place. The forwarder does not need to redundantly check PA's stopped state since a stopped PA cannot execute transactions that would trigger forwarder calls.

- Threat c: Attacker succeeds to `_wrap()` or `_unwrap()` functions directly bypassing Protocol Adapter.

  Threat does not hold. Both `_wrap` and `_unwrap` are internal functions that cannot be called externally. They are only accessible through `_forwardCall` (code ref), which itself is only called by `forwardCall` that enforces Protocol Adapter authorization (code ref).

- Threat d: Attacker succeeds to call `forwardEmergencyCall()` directly bypassing emergency caller.

  Threat does not hold. The `forwardEmergencyCall` function validates that only the authorized emergency caller can invoke it (code ref). The function first checks that `_emergencyCaller` is set (code ref), then uses `_checkCaller(_emergencyCaller)` to verify the caller matches, and additionally validates that the Protocol Adapter is emergency stopped (code ref). An unauthorized caller would be rejected with `UnauthorizedCaller`.

- Threat e: Attacker succeeds to call `setEmergencyCaller()` directly bypassing emergency comittee.

  Threat does not hold. The `setEmergencyCaller` function enforces that only the emergency committee can set the emergency caller (code ref). The function validates the caller matches `_EMERGENCY_COMMITTEE` and additionally checks that the Protocol Adapter is emergency-stopped (code ref), and prevents setting the emergency caller more than once (code ref). An attacker cannot bypass this authorization.

- Threat f: Forwarder doesn't validate input data from `forwardCall()` and accepts malformed calldata.

  Threat does not hold. The forwarder validates input data through Solidity's `abi.decode` which reverts on malformed data. In `_wrap`, the function decodes a specific structure requiring `CallType`, `address`, `PermitTransferFrom`, `bytes32`, and `bytes` (code ref). In `_unwrap`, it decodes `CallType`, `address`, `address`, and `uint128` (code ref). If calldata doesn't match the expected structure, `abi.decode` reverts. Additionally, wrap validates amount fits in `uint128` (code ref).

- Threat g: Forwarder doesn't correctly extract `CallType` from input or doesn't validate it.

  Threat does not hold. The `CallType` is extracted from byte 31 of the input (code ref) and used to route to the appropriate function. While the extracted value is not explicitly validated against the enum range, Solidity enums revert on invalid values during the cast. If byte 31 contains a value outside the valid enum range (0 or 1 for `ERC20Forwarder`, 0-2 for `ERC20ForwarderV2`), the enum cast would fail.

- Threat h: Forwarder doesn't validate addresses are non-zero.

  Threat does not hold. The forwarder validates critical addresses during construction: it checks that `protocolAdapter` and `logicRef` are non-zero (code ref), and that `emergencyCommittee` is non-zero (code ref). For runtime operations, user-provided addresses (token, owner, receiver) are validated by the operations themselves: Permit2 validates the owner/token addresses during signature verification, and `safeTransfer` validates the recipient address by attempting the transfer which would revert for zero address.

- Threat i: Forwarder makes external calls before updating state and external contract can reenter.

  Threat does not hold. The forwarder follows the CEI pattern. In wrap operations, the event is emitted before the external call to Permit2 (code ref). In unwrap operations, the event is emitted before the `safeTransfer` (code ref). For migration in V2, the nullifier is added to the internal set before the external emergency call (code ref). The forwarder does not modify any internal state during wrap/unwrap operations that could be exploited through reentrancy— token balance changes occur in external ERC20 contracts. Authorization is handled by the Protocol Adapter before the forwarder is invoked, and Permit2 nonces prevent signature replay. Even if a malicious token reentered during transfer, there is no exploitable state manipulation since the forwarder's authorization model relies on PA's pre-call verification rather than internal state checks.

- Threat j: Forwarder doesn't properly handle the behaviour of non-standard ERC20 tokens.

  Threat holds. See finding "*Fee-on-transfer and rebasing ERC20 tokens cause resource quantity mismatch*".

- Threat k: If ERC20 token is deflationary (burns portion of tokens on transfer), then fewer tokens arrive than expected.

  Threat holds. See finding "*Fee-on-transfer and rebasing ERC20 tokens cause resource quantity mismatch*".

- Threat l: If ERC20 token is rebasing (balance changes over time without transfers) and forwarder doesn't account for this, then forwarder balance could increase/decrease independently of wrap/unwrap operations.

  Threat holds. See finding "*Fee-on-transfer and rebasing ERC20 tokens cause resource quantity mismatch*".

- Threat m: If ERC20 token is fee-on-transfer and, then Q tokens are authorized but only (Q - fee) tokens arrive at forwarder.

  Threat holds. See finding “*Fee-on-transfer and rebasing ERC20 tokens cause resource quantity mismatch*”.

- Threat n: Forwarder completes wrap operation (receives tokens via Permit2) but fails to emit `Wrapped` event or emits it with incorrect parameters (wrong token address, sender, or amount).

  Threat does not hold except for the case of fee-on-transfer and rebasing ERC20 tokens (see finding “*Fee-on-transfer and rebasing ERC20 tokens cause resource quantity mismatch*”). The `_wrap` function unconditionally emits the `Wrapped` event with correct parameters: token address from the permit, sender (owner), and amount (code ref). The event emission occurs before the Permit2 transfer (code ref). If the Permit2 transfer fails, the entire transaction reverts, rolling back the event emission. This ensures the event is only persisted when the wrap operation completes successfully.

- Threat o: Forwarder emits `Wrapped` event but wrap operation fails (tokens not transferred).

  Threat does not hold. The event emission occurs before the token transfer (code ref), but Ethereum’s atomic transaction execution ensures that if the subsequent Permit2 call fails (code ref), the entire transaction including the event emission is reverted. There is no scenario where the event persists without successful token transfer.

- Threat p: Forwarder completes unwrap operation (transfers tokens to recipient) but fails to emit `Unwrapped` event or emits it with incorrect parameters (wrong token address, recipient, or amount).

  Threat does not hold except for the case of fee-on-transfer and rebasing ERC20 tokens (see finding “*Fee-on-transfer and rebasing ERC20 tokens cause resource quantity mismatch*”). The `_unwrap` function unconditionally emits the `Unwrapped` event with correct parameters: token address, recipient, and amount (code ref). The event emission occurs before the `safeTransfer` (code ref). If the token transfer fails, the entire transaction reverts, rolling back the event emission. This ensures the event is only persisted when the unwrap operation completes successfully.

- Threat q: Forwarder emits `Unwrapped` event but unwrap operation fails (tokens not transferred).

  Threat does not hold. The event emission occurs before the token transfer (code ref), but Ethereum’s atomic transaction execution ensures that if the subsequent `safeTransfer` fails (code ref), the entire transaction including the event emission is reverted. There is no scenario where the event persists without successful token transfer.

- Threat r: Forwarder contract emits events, logs, or observable on-chain data beyond the minimum necessary (token address, public sender/recipient addresses, and amounts), allowing observers to correlate wrap/unwrap operations with specific persistent resource commitments/nullifiers or link users’ addresses to their private resource activity.

  Threat does not hold. The forwarder emits only the minimum necessary information: `Wrapped(token, from, amount)` and `Unwrapped(token, to, amount)` (code ref). These events reveal public addresses and token amounts but do not expose resource commitments, nullifiers, or any zero-knowledge proof data that would link on-chain activity to specific private resources. The `from` address in wrap events is the token owner (public Ethereum address), and the `to` address in unwrap events is the recipient specified in the external payload. Resource creation and consumption occur at the Protocol Adapter level with encrypted discovery and resource payloads, maintaining privacy of resource ownership while allowing public tracking of forwarder token movements for accounting purposes.

Integration properties

Property AP-06

The `ERC20Forwarder` correctly integrates with Permit2 for wrap operations, correct witness type string formatting, accurate witness data binding, and prevention of signature replay attacks across different transactions.

Threats:

- Threat a: Forwarder contract doesn't use canonical Permit2 address (`0x000000000022D473030F116dDEE9F6B43aC78BA3`).

  Threat does not hold. The forwarder uses the canonical Permit2 address `0x000000000022D473030F116dDEE9F6B43aC78BA3` hardcoded as an immutable constant (code ref). This is the official Uniswap Permit2 deployment address used across all supported chains, ensuring compatibility with user signatures created for the canonical contract.

- Threat b: Forwarder uses wrong or malformed `witnessTypeString` that causes Permit2 EIP-712 signature verification to fail.

  Threat does not hold. The forwarder uses the correct witness type string respecting Permit2's requirements for alphabetical ordering of structs (code ref). The witness type string `"Witness witness)TokenPermissions(address token,uint256 amount)Witness(bytes32 actionTreeRoot)"` is passed to Permit2's `permitWitnessTransferFrom` (code ref) and matches the format documented in Uniswap's Permit2 reference for witness-based signature transfers.

- Threat c: The action tree root in transaction payload is different to the action tree root that user signed in Permit2 signature.

  Threat does not hold. The action tree root from the transaction input is cryptographically bound to the Permit2 signature through the witness mechanism. The forwarder decodes the `actionTreeRoot` from the input (code ref), constructs a witness struct with this root, computes its EIP-712 struct hash (code ref), and passes it to Permit2 for signature verification (code ref). If the action tree root in the transaction differs from what the user signed, Permit2's signature verification fails and the transaction reverts.

- Threat d: Witness hash doesn't match the hash format expected by Permit2.

  Threat does not hold. The witness hash is computed using the correct EIP-712 struct hash format expected by Permit2. The hash function encodes the witness type hash and action tree root using `keccak256(abi.encode(_WITNESS_TYPEHASH, actionTreeRoot))` (code ref), where `_WITNESS_TYPEHASH = keccak256("Witness(bytes32 actionTreeRoot)")` (code ref). This matches the EIP-712 typed structured data hashing standard that Permit2 uses for witness validation.

Key management system

Assumptions

- The `signature` used for KEK derivation is generated from a fixed, known message that the user signed intentionally.
- Browser's Web Crypto API implementation is correct and secure.
- SHA-256 and HMAC implementation in `@noble/hashes` are correct.
- secp256k1 curve is correctly implemented in `@noble/secp256k1`.
- User's private keys are securely stored in Heliax servers and, in the event of users clearing browser data and erasing the contents of IndexedDB, they can recover the keys from the servers.
- Threat a: Keys are generated with random seed instead of deterministic seed, making recovery impossible after vault deletion.
- Threat b: Deterministic key generation uses weak or predictable seed source, compromising key security for recoverability.
- Threat c: Vault stored in IndexedDB is deleted (browser data cleared, browser reinstall) with no recovery mechanism, causing permanent key loss.
- The input parameters to function `signPermit()` are correct and validated upstream:
  - The values of `props` argument (of type `Permit2Props`) for `signPermit()` are correct and validated:
    - `permit2Address` is the legitimate Permit2 contract address on the target chain.
    - `spenderAddress` is the legitimate forwarder contract address.
    - `token` address corresponds to a legitimate ERC20 token.
    - `amount` does not exceed the user's actual balance.
    - `deadline` is a reasonable timestamp.
    - `nonce` is a random value.
    - `actionTreeRoot` is a valid root of the action tree with the resources of the transaction.
    - `chainId` matches the network the user is connected to.
  - `walletClient` is from a trusted wallet provider.
  - `ownerAddress` matches the connected wallet's address.

Properties

Property AP-07: User signatures used for KEK (key encryption key) derivation are treated as secrets (never persisted, logged, or transmitted).

- Threat a: Signature is inadvertently logged in console logs, error tracking services, or server logs.

  Threat does not hold. The functions—`createVaultAccount()` (code ref), `unlockVault()` (code ref) and `wrapSignatureAsCryptoKey()` (code ref)—through which the signature passes do not log or persist it (in localStorage or cookies).

  However, the exception that is re-thrown when there are JSON parsing errors in `fromJSON` (code ref) could leak sensitive information key material, as the error message includes `e.message` which may contain JSON fragments with hex-encoded private keys. These errors might potentially be exposed in console logging or user-facing error messages, so we would recommend to sanitize the error message to prevent any leakage of sensitive information.

- Threat b: Signature is included in API responses, events, or telemetry.

  Threat does not hold. The functions—`createVaultAccount()` (code ref), `unlockVault()` (code ref) and `wrapSignatureAsCryptoKey()` (code ref)—through which the signature passes do not include it in API responses or events.

- Threat c: XSS vulnerability allows attacker to inject JavaScript that intercepts signature.

  Threat might hold. In addition to robust server-side validation and sanitisation of all user input, we recommend enforcing a strict Content Security Policy (CSP) to significantly reduce the impact of any residual XSS issues. At a minimum, the policy should:

  - Restrict the execution of JavaScript to the application's own origin (`'self'`) and a limited set of explicitly trusted third-party domains from which scripts are intentionally loaded.
  - Avoid the use of unsafe directives such as `'unsafe-inline'` and `'unsafe-eval'`.
  - Restrict the sources for styles, images, fonts, network connections, and frames to only those required by the application.
  - Prevent the application from being embedded in frames on untrusted origins.

  A suitable template policy (that can act as a strarting point) with placeholders would be:

```
Content-Security-Policy:
  default-src 'self';
  script-src 'self' <TRUSTED_SCRIPT_DOMAINS> 'nonce-{RANDOM}';
  style-src 'self' <TRUSTED_STYLE_DOMAINS> 'nonce-{RANDOM}';
  img-src 'self' data: <TRUSTED_IMAGE_DOMAINS>;
  font-src 'self' <TRUSTED_FONT_DOMAINS>;
  connect-src 'self' <TRUSTED_API_DOMAINS>;
  frame-src 'none';
  frame-ancestors 'none';
  base-uri 'self';
  form-action 'self';
  object-src 'none';
```

  `<TRUSTED_*>` should be replaced with a minimal allow-list of external domains that are required for application functionality (e.g. CDNs, monitoring services, analytics), and no others. It should be rolled out in `Content-Security-Policy-Report-Only` mode first to identify violations before enforcing it.

  We also would recommend implementing regular dependency security audits using `npm audit` and `npm outdated` to detect vulnerable and outdated packages. Run `npm audit` before each deployment and use `npm audit --audit-level=high` in CI/CD pipelines to fail builds on critical issues. Run `npm outdated` regularly to identify packages with available updates, as outdated packages often contain unpatched security issues. Integrate both commands into GitHub Actions/GitLab CI to prevent merging vulnerable dependencies. When vulnerabilities are found, use `npm audit fix` for automatic patches or manually upgrade packages when breaking changes are involved.

Property AP-08: Private keys (authority, nullifier, encryption, discovery) are never logged, transmitted or stored in plaintext.

- Threat a: Private keys are inadvertently logged in console logs, error tracking services, or server logs.

  Threat does not hold. There are no console logging statements that could expose keyring data. Error handling in `decrypt()` (code ref) and `fromJson()` (code ref) use a catch block without logging sensitive cryptographic material (code ref, code ref). Additionally, error tracking services are integrated in the application per the `package.json` analysis, eliminating the risk of automatic error transmission to external services.

- Threat b: Private keys are included in API responses, events, or telemetry.

  Threat does not hold. None of the files in scope make API calls that could leak private key material, and no error tracking or telemetry services are present.

- Threat c: Private keys are not encrypted with the KEK when stored in the vault.

  Threat does not hold. Inspection confirms that private keys are properly encrypted before vault storage. The `createVaultAccount()` function (code ref) follows a secure flow: (1) derives the Key Encryption Key (KEK) from the user's signature using HKDF (code ref), (2) serializes the keyring containing all private keys (code ref), (3) encrypts the serialized keyring using AES-GCM with the KEK (code ref), and (4) stores the ciphertext and IV in the vault entry (code ref). Private keys never exist in plaintext within the vault structure persisted to IndexedDB—only the encrypted ciphertext is stored, which can only be decrypted using the correct user signature to re-derive the KEK.

Property AP-09: Private keys are randomly generated.

- Threat a: Keys are generated using deterministic derivation with a seed parameter.

  Threat does not hold. The function `createUserKeyring()` is called without any seed parameter (code ref), resulting in random key generation. The `generatePrivateKey()` function uses `generateRandomBytes()` when no seed is provided (code ref), which leverages `crypto.getRandomValues()` for cryptographically secure randomness. As confirmed by the development team, the purpose of the optional seed parameter in function `createUserKeyring()` to allow the keys to be deterministically generated for testing. Since the intention is that private keys in production are randomly generated, no seed will be passed.

- Threat b: Identical private keys are produced for authority, nullifier, encryption, and discovery.

  Threat does not hold. In production the function `createUserKeyring()` is called without a seed parameter, which triggers random key generation for each key pair via `generateRandomBytes()`. Each invocation of `generateRandomBytes()` produces a different cryptographically secure random value, ensuring that authority, encryption, discovery, and nullifier private keys are all distinct and independent. When a seed parameter is provided (for testing), the implementation calls `KeyPair.create(seed)` three times without domain separation, producing identical authority, encryption, and discovery private keys. Only the nullifier key would differ because `NullifierKeyPair.create` uses the "Nullifier" domain (code ref). However, the development team has informed us that domain separators will be removed completely, as keys are generated randomly and domain separation does not provide any additional security.

Property AP-10: Public keys are always correctly derived from their corresponding private keys using specified algorithms.

- Threat a: HMAC-SHA256 output produces invalid secp256k1 private key (value ≥ curve order), causing key generation to fail or produce weak keys.

  Threat holds. The function `generatePrivateKey()` directly returns raw HMAC-SHA256 output without validation (code ref). The secp256k1 curve order is `n = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141`, and HMAC-SHA256 produces 256-bit outputs uniformly distributed in `[0, 2^256)`, which includes values ≥ n with probability approximately `2^-128` (extremely rare but non-zero). When such invalid values are passed to `secp256k1.getPublicKey()`, the `@noble/secp256k1` library throws an error (code ref). This would cause key generation to completely fail, preventing vault creation. As a recommendation, validation could be added in `generatePrivateKey()` to ensure `0 < privateKey < n`, and if invalid, re-derive the private key-

- Threat b: Nullifier public key derivation uses wrong hash function or incorrect input, producing keys that don't match protocol expectations-

  Threat does not hold- The function `NullifierKeyPair-derivePublicKey()` correctly derives the nullifier commitment (public key) by hashing the private key with SHA-256 (code ref)-

- Threat c: Authority, encryption and discovery public keys are derived incorrectly (wrong curve operation, wrong base point), causing authentication or decryption failures-

  Threat does not hold- The function `KeyPair-derivePublicKey()` correctly uses `secp256k1-getPublicKey(privateKey, true)` from the `@noble/secp256k1` library- This function performs the standard secp256k1 elliptic curve operation `publicKey = privateKey × G` where G is the generator point- The second parameter `true` specifies compressed format (33 bytes), which is the standard compact representation- The library internally validates that private keys are in the valid range `[1, n)` before performing scalar multiplication, preventing mathematical errors from invalid inputs-

Property AP-11: The keyring in a user's encrypted vault can only be recovered with the correct user signature-

- Threat a: HKDF produces weak or predictable KEK that can be brute-forced or predicted-

  Threat does not hold- The function `deriveKeyEncryptionKey()` (code ref) correctly uses the Web Crypto API's HKDF implementation with SHA-256 as the hash function- The HKDF derivation takes a high-entropy input key material (the user's signature, which is cryptographically unpredictable and depends on the user's private wallet key)- The derived KEK has 256-bit length (code ref), providing security against brute-force attacks- The only vulnerability would be if the user signature itself is weak or predictable, but this is a wallet implementation concern outside of the scope-

- Threat b: AES-GCM IV reuse with same KEK breaks encryption security-

  Threat does not hold- The function `encrypt()` (code ref) generates a new cryptographically random 12-byte IV for each encryption operation using `generateRandomBytes(12)` (code ref)- The `generateRandomBytes()` utility uses `crypto-getRandomValues()` which provides cryptographically secure random values from the browser's CSPRNG- This ensures that even if the same KEK is used multiple times, each encryption operation uses a different IV, preventing IV reuse-

Property AP-12: Keyring serialization maintains integrity and can be correctly deserialized

- Threat a: Data is corrupted during text encoding (JSON)-

  Threat does not hold- Keyring serialization uses `convertUserKeyringToJson` and `converUserKeyringFromJson`, which handle JSON serialization/deserialization- Serialization is performed by `KeyPairSerializer-toJson`, which uses `JSON-stringify` to serialize private/public key pairs that have already been converted to hex strings by `toHex` (code ref)- Deserialization is performed by `KeyPairSerializer-fromJson`, which uses `JSON-parse` to reconstruct the keyring structure (code ref)- JavaScript's `JSON-stringify` and `JSON-parse` are deterministic and reliable for this data structure—hex strings contain only ASCII characters `[0-9a-f]`, which have no special meaning in JSON and don't require escaping- Any malformed JSON would cause `JSON-parse` to throw an exception, which would be caught and handled, preventing silent corruption-

- Threat b: Serialization/deserialization assigns keys to wrong purposes (authority <-> encryption swap).

  Threat does not hold. Serialization and deserialization use explicit field names that prevent key misassignment. During serialization, each key pair is explicitly assigned to its named field: `authorityKeyPair`, `nullifierKeyPair`, `discoveryKeyPair`, and `encryptionKeyPair` (code ref). During deserialization, the fields are explicitly read by name from the parsed object (`keyringObj.authorityKeyPair`, `keyringObj.nullifierKeyPair`, etc.) and assigned to the corresponding properties in the returned `UserKeyring` (code ref). JSON serialization preserves field names, so `authorityKeyPair` will always deserialize back to `authorityKeyPair`. The only way keys could be swapped is through a programming error like writing `authorityKeyPair: KeyPairSerializer.fromJson(KeyPair, keyringObj.encryptionKeyPair)`, but inspection confirms all field names are correctly matched. Additionally, the `NullifierKeyPair` uses different field names (`nk`, `cnk` instead of `privateKey`, `publicKey`), which would cause deserialization to fail if incorrectly assigned to a regular `KeyPair`, providing type-level protection against nullifier key swaps.

- Threat c: Deserialization does not recover original keys.

  Threat does not hold. Inspection of the serialization/deserialization round-trip confirms that keys are correctly recovered. The serialization flow at keys/services.ts:49-56 converts each `KeyPair` to JSON using `KeyPairSerializer.toJson`, which extracts the raw `Uint8Array` private and public keys and converts them to hex strings using `toHex()` (code ref). The deserialization flow reverses this process using `KeyPairSerializer.fromJson`, which parses the JSON, extracts the hex strings, and converts them back to `Uint8Array` using `fromHex()` (code ref), then reconstructs the `KeyPair` with the original byte arrays. The `toHex` and `fromHex` utilities are deterministic bijective functions—`fromHex(toHex(bytes))` always equals `bytes`. JSON serialization of hex strings is lossless since hex characters `[0-9a-f]` have no special meaning in JSON. Therefore, the deserialized keys are byte-for-byte identical to the original keys.

Property AP-13: Each user's vault is cryptographically isolated.

- Threat a: Vault ID collision causes one user's vault to overwrite another user's vault in IndexedDB.

  Threat does not hold. Vault IDs are hashed using `hashVaultId` (ref), which applies SHA-256 to produce a 256-bit hash (code ref). IndexedDB is sandboxed per-origin by the browser's same-origin policy, meaning each website's IndexedDB is completely isolated from other websites. An attacker cannot access or manipulate another origin's IndexedDB without executing malicious JavaScript within that origin (which would be an XSS vulnerability). Within the same origin (same application), all users share the same IndexedDB database, but each user has a distinct wallet address, which produces a distinct SHA-256 hash. The probability of accidental collision between different wallet addresses is cryptographically negligible. The IndexedDB schema uses the hashed ID as a unique primary key, so even if a collision occurred, the `add()` operation (code ref) would fail with a constraint violation error rather than silently overwriting. Therefore, vault ID collisions cannot occur under normal operation, and adversarial collisions would require prior compromise of the application (XSS), at which point the attacker already has full access to all vault operations.

- Threat b: Same signature can be used to decrypt multiple users' vaults if vault IDs are predictable or signature is not bound to specific vault.

  Threat does not hold. Each user's signature is cryptographically unique to their wallet's private key. When a user signs a message to unlock their vault, the signature is generated using their wallet's private key, producing a signature that only that specific private key can generate. Even if two users sign the identical challenge message, their signatures will be completely different because they use different private keys. The signature serves as the input key material for HKDF, so different users produce different KEKs, which decrypt different vaults.

- Threat c: Vault ID hashing collision (different IDs producing same hash) causes vault storage conflicts.

  Threat does not hold. `hashVaultId` uses SHA-256 to hash the vault ID (typically a wallet address). SHA-256 provides strong collision resistance, so the probability of finding two different wallet addresses that produce the same SHA-256 hash is negligible. The IndexedDB schema uses the hashed ID as the unique primary key (code ref), so if a collision somehow occurred, the `add()` operation would fail with a constraint violation. In the unlikely event of a natural collision, the second user would be unable to create their vault (the operation would fail). Therefore, SHA-256 collision resistance provides sufficient protection against vault storage conflicts.

Property AP-14: All cryptographic libraries and Web Crypto APIs are used correctly according to specifications.

- Threat a: `SubtleCrypto.importKey()` called with incorrect or weak parameters, or errors not properly handled.

  Threat holds. `wrapSignatureAsCryptoKey` correctly uses `window.crypto.subtle.importKey()` with appropriate parameters (code ref). The function imports the signature as `"raw"` format, which is the correct format for importing arbitrary byte sequences as key material. The algorithm is specified as `{ name: "HKDF" }`, which correctly designates the key as input material for HKDF key derivation. The `extractable` parameter is set to `false`, which is a security best practice preventing the key from being exported after import. The `keyUsages` is set to `["deriveKey"]`, which correctly restricts the key to only be used for key derivation operations. However, error handling is absent throughout the call chain—the function does not have explicit try-catch blocks, meaning errors from `importKey()` would propagate through `createVaultAccount` (code ref) and `create` (code ref) to the caller without specific error context. Web Crypto API errors are typically DOM exceptions with generic messages, which are intentionally vague to avoid information leakage. We would recommend though to add try-catch blocks with more descriptive error messages to distinguish between import failures, derivation failures, and encryption failures for better debugging and user experience.

- Threat b: `SubtleCrypto.deriveKey()` called with incorrect or weak parameters, or errors not properly handled.

Threat holds. `deriveKeyEncryptionKey` correctly uses `window.crypto.subtle.deriveKey()` with appropriate parameters for HKDF (code ref). The derivation algorithm specifies `name: "HKDF"`, `hash: "SHA-256"`, `salt` using `VAULT_DOMAIN_SALT`, and `info` using `VAULT_DOMAIN_INFO`. The derived key algorithm is `{ name: "AES-GCM", length: 256 }`, specifying 256-bit AES keys for strong encryption. The key is non-extractable (`extractable: false`) and restricted to `["encrypt", "decrypt"]` operations. However, error handling is absent—the function does not have explicit try-catch blocks, meaning errors from `deriveKey()` would propagate through `createVaultAccount` / `unlockVault` and `create` / `unlock` to the caller. Even though these errors are generic DOM exceptions that don't expose sensitive information, we would recommend adding try-catch blocks to provide clearer error context distinguishing derivation failures from other vault operation failures.

- Threat c: `SubtleCrypto.encrypt()` / `decrypt()` called with incorrect or weak parameters, or errors not properly handled.

  Threat holds. `encrypt` correctly uses `window.crypto.subtle.encrypt()` with appropriate AES-GCM parameters (code ref): `name: "AES-GCM"` and a freshly generated 12-byte IV (the recommended size for AES-GCM). The `decrypt` function uses matching parameters (code ref): `name: "AES-GCM"` with the stored IV. Error handling for decrypt is present—the function wraps `decrypt()` in a try-catch block and returns a safe generic error message (code ref), preventing information leakage about why decryption failed. However, encrypt has no error handling—errors from `encrypt()` would propagate through `createVaultAccount` to the caller. The decrypt error handling is exemplary, thus we would recommend adding try-catch to `encrypt` for consistency, though the security risk is minimal.

- Threat d: `@noble/hashes` (`sha256()`, `hmac()`) called with incorrect or weak parameters, or errors not properly handled.

  Threat does not hold. The `sha256` function is used to derive nullifier commitments (code ref): `sha256(nk)`, which takes a `Uint8Array` as input and returns a 32-byte hash output. The `hmac` function is used for private key generation (code ref): `hmac(sha256, actualSeed, domainBytes)`, where `sha256` specifies the hash algorithm, `actualSeed` is the key material (either provided seed or random bytes from `generateRandomBytes()`), and `domainBytes` is the message. Even though the functions may throw exceptions, since the inputs are not untrusted or from user-provided input, it is not necessary to wrap the calls in try-catch blocks.

- Threat e: `@noble/secp256k1` (`getPublicKey()`) called with incorrect parameters, or errors not properly handled.

  Threat does not hold. `KeyPair.derivePublicKey` correctly calls `secp256k1.getPublicKey(privateKey, true)` with appropriate parameters (code ref): the private key as a `Uint8Array` and `true` for compressed format (33 bytes). Aas previously established in the analysis of Property AP-10, Threat a, the `secp256k1.getPublicKey()` performs strict validation and throws an error if the private key is outside the valid range `[1, n)` where n is the curve order. The validation occurs in the library's `secretKeyToScalar` function which calls `arange(num, 1n, N, 'invalid secret key: outside of range')` (code ref). There is no error handling around `getPublicKey()`, so if the private key generated by HMAC-SHA256 happens to be $\geq$ n, the error would propagate uncaught, causing vault creation to fail completely for the affected user. However, given the negligible probability that the private is not valid, we do not consider necessary to add error handling in `derivePublicKey` to detect invalid private keys.

Property AP-15: All cryptographic inputs requiring randomness must be generated from non-deterministic, cryptographically secure random sources.

- Threat a: `generateRandomBytes()` uses weak source of non-determinism.

  Threat does not hold. `generateRandomBytes` correctly uses `crypto.getRandomValues(uint8Array)` (code ref), which is the Web Crypto API's cryptographically secure pseudorandom number generator (CSPRNG). The function is used correctly for generating private key material when no seed is provided (code ref), and for generating AES-GCM initialization vectors (code ref).

- Threat b: IV reuse when calling `encrypt()` multiple times.

  Threat does not hold. The `encrypt` function generates a fresh random IV for every encryption operation by calling `generateRandomBytes(12)` (code ref). Each time `encrypt` is called, a new cryptographically secure 12-byte IV is generated using `crypto.getRandomValues()`, ensuring that the same (KEK, IV) pair is never reused. The IV is returned alongside the ciphertext and stored in the `VaultEntry` structure. When `createVaultAccount` is called multiple times, each call to `encrypt` generates a distinct IV, preventing IV reuse even with the same KEK.

Property AP-16: Permit2 signatures are correctly constructed with valid parameters and match contract expectations

- Threat a: EIP-712 typed data structure (types, domain, witness) doesn't match Permit2 contract expectations.

  Threat does not hold. The call to `SignatureTransfer.getPermitData()` is provided by the official Uniswap Permit2 SDK, which guarantees that the returned `domain`, `types`, and `values` match the Permit2 contract's expected EIP-712 structure. The function receives: the `permitTransferFrom` structure that includes all required fields—`permitted` (token and amount), `spender`, `nonce`, and `deadline`—; the address of the Permit2 contract; the chain ID of the Ethereum network; and the custom witness information containing the `actionTreeRoot` field as `bytes32`.

- Threat b: Signature components (r, s, v) are incorrectly split or encoded.

  Threat does not hold. The `signPermit` function correctly splits the EIP-712 signature into its components. The signature returned by `walletClient.signTypedData()` is a 65-byte hex string following the standard Ethereum signature format (code ref): 32 bytes for `r`, 32 bytes for `s`, and 1 byte for `v`. The splitting logic uses `viem`'s `slice()` function to extract the components correctly: `r = slice(signature, 0, 32)` extracts bytes 0-31, `s = slice(signature, 32, 64)` extracts bytes 32-63, and `v = slice(signature, 64, 65)` extracts byte 64. The `v` component is converted from a `Hex` byte to a numeric value using `hexToNumber(v)`.

- Threat c: `witnessType` format doesn't match Forwarder's contract expected format.

  Threat does not hold. The `witnessType` information (code ref) matches the expected witness type string in Forwarder's contract (code ref).

Findings

Name	Type	Severity	Status
Fee-on-transfer and rebasing ERC20 tokens cause resource quantity mismatch	Implementation	4 - Critical	Resolved
ERC20ForwarderV2 constructor does not verify PAv1 is emergency stopped	Implementation	2 - Medium	Resolved
Signature incorrectly encoded as UTF-8 Text	Implementation	2 - Medium	Resolved
Emergency caller can only be set once	Implementation	0 - Informational	Acknowledged
Double JSON encoding of Keyring	Implementation	0 - Informational	Acknowledged
Insufficient error handling in Web Crypto API operations	Implementation	0 - Informational	Resolved
XSS prevention as security control	Implementation	0 - Informational	Acknowledged
Miscellaneous code improvements	Implementation	0 - Informational	Resolved

Table 2: Identified Security Findings

Fee-on-transfer and rebasing ERC20 tokens cause resource quantity mismatch

Severity Critical **Exploitability** High **Status** Resolved

Type Implementation **Impact** High

Involved artifacts

- `anomapay-backend`: `contracts/src/ERC20Forwarder.sol`
- `anomapay-backend`: `contracts/src/drafts/ERC20ForwarderV2.sol`

Description

Since the ERC20 interface allows arbitrary `transfer` / `transferFrom` implementations and also can have internal rebasing logic which is out of our control, transferring these tokens could cause a mismatch between the resource quantity and the actual amount locked in the forwarder contract. This affects wrap, unwrap, and migrate operations.

Problem scenarios

Fee-on-transfer token scenario:

- User signs Permit2 signature authorizing transfer of 100 USDT-FoT (hypothetical fee-on-transfer variant)
- Backend generates proof creating persistent resource with quantity = 100
- Forwarder calls Permit2 which transfers tokens from user
- 5% fee deducted during transfer, forwarder receives only 95 tokens
- PA commitment tree stores commitment of resource with quantity 100, but forwarder holds only 95 tokens

Rebasing token scenario:

- User wraps 100 RB tokens, forwarder holds 100 RB, resource quantity = 100
- RB executes negative rebase reducing all balances by 10%
- Forwarder balance becomes 90 RB, but resource still shows quantity = 100
- User attempts to unwrap 100 RB, transaction reverts due to insufficient balance

Recommendation

For fee-on-transfer tokens, add balance checks after ERC20 transfers in wrap, unwrap, and migrate operations. The forwarder should query its token balance before and after the transfer, verify the balance increased or decreased by exactly the expected amount, and revert if the amounts don't match.

For rebasing tokens, no optimal contract-level solution exists. A blacklist approach is not practical since anyone can deploy ERC20 tokens, making it impossible to maintain a comprehensive list. A whitelist would severely restrict user experience by limiting which tokens can be wrapped. The fundamental issue is that rebasing balance changes occur completely independently of the Protocol Adapter's and Forwarder's function calls, making them undetectable and unpreventable at the contract level. The system must accept this limitation and rely on application-layer mitigation.

Mitigation

The team has addressed these issues as follows:

Fee-on-transfer tokens: Fixed in commit 2de7930 with balance checks added after transfers.

Rebasing tokens: Accepted risk with frontend whitelist mitigation. The frontend will maintain a list of supported tokens, preventing users from wrapping rebasing tokens through the UI. Users interacting directly with contracts are responsible for understanding these risks.

Resolution

The development team addressed this finding in PR #174.

ERC20ForwarderV2 constructor does not verify PAv1 is emergency stopped

Severity Medium **Exploitability** Low **Status** Resolved

Type Implementation **Impact** High

Involved artifacts

- `anomapay-backend`: `contracts/src/drafts/ERC20ForwarderV2.sol`

Description

The `ERC20ForwarderV2` constructor captures the Protocol Adapter V1's final commitment tree root at deployment time (code ref) but does not verify that PAv1 has been emergency stopped. If V2 is deployed while V1 is still operating, the constructor captures an intermediate root rather than the actual final root. V1 continues processing transactions and adding commitments to the tree after V2's deployment, resulting in a different final root when V1 eventually stops. During migration, FWv2 validates that the user-provided root matches the stored root from deployment. Resources created in V1 after V2's premature deployment have merkle proofs against the real final root, which V2 rejects because it expects the outdated intermediate root captured at deployment. These resources become permanently stranded - they cannot migrate to V2 (root mismatch) and cannot be used in V1 (stopped). While the `forwardEmergencyCall` mechanism prevents migration execution until V1 stops, it does not prevent the constructor from capturing a non-final root that makes later migrations impossible for some resources.

Recommendation

This issue can be addressed through either on-chain enforcement or careful deployment procedures:

Option 1: Constructor validation (recommended)

Add a validation check in the `ERC20ForwarderV2` constructor to verify that Protocol Adapter V1 is emergency-stopped before capturing the commitment tree root:

```solidity
constructor(
  address protocolAdapterV2,
  bytes32 logicRefV2,
  address emergencyCommittee,
  ERC20Forwarder erc20ForwarderV1
) ERC20Forwarder(protocolAdapterV2, logicRefV2, emergencyCommittee) {
  if (address(erc20ForwarderV1) == address(0)) {
    revert ZeroNotAllowed();
  }

  _ERC20_FORWARDER_V1 = erc20ForwarderV1;
```

```
    _PROTOCOL_ADAPTER_V1 = erc20ForwarderV1.getProtocolAdapter();

    // Verify PAv1 is emergency-stopped before capturing final root
    if (!IProtocolAdapter(_PROTOCOL_ADAPTER_V1).isEmergencyStopped()) {
      revert ProtocolAdapterV1NotStopped(_PROTOCOL_ADAPTER_V1);
    }

    _COMMITMENT_TREE_ROOT_V1 =
    ICommitmentTree(_PROTOCOL_ADAPTER_V1).latestCommitmentTreeRoot();
    _LOGIC_REFERENCE_V1 = erc20ForwarderV1.getLogicRef();
}
```

This ensures that V2 can only be deployed after V1 has been properly stopped, making the captured commitment tree root truly final and preventing resources from being stranded.

Option 2: Careful deployment process

Alternatively, rely on governance to ensure correct deployment order:

1. Emergency stop PAv1 first
2. Wait for confirmation that PAv1 has stopped processing
3. Deploy `ERC20ForwarderV2` (which captures the now final root)
4. Set FWv2 as the emergency caller on FWv1 via the emergency committee

This approach requires careful coordination, similar to the existing requirement that **FWv2 must be set as the emergency caller for FWv1** before migrations can execute. However, Option 1 is preferred as it provides on-chain enforcement rather than relying solely on operational procedures.

Resolution

The development team addressed this finding in PR #204.

Signature incorrectly encoded as UTF-8 Text

Severity Medium **Exploitability** Low **Status** Resolved

Type Implementation **Impact** High

Involved artifacts

- `pay-interface-app`: `src/hooks/useSignUp.tsx`

Description

The application incorrectly converts cryptographic signatures from hexadecimal strings to bytes using UTF-8 text encoding (code ref) instead of proper hexadecimal encoding. This encoding error occurs when deriving the Key Encryption Key (KEK) used to protect the user's cryptographic keyring.

The `signature` parameter is a `Hex` type (e.g., `"0x1234567890abcdef..."`), which is a hexadecimal string representation of a cryptographic signature. The `TextEncoder().encode()` method treats this as a regular string and converts each **character** to its UTF-8 byte representation, rather than converting the **hexadecimal string** to the actual signature bytes it represents.

Input	Method	Output	Interpretation
`"0x1234..."`	`TextEncoder().encode()`	`[48, 120, 49, 50, 51, 52, ...]`	ASCII codes for characters '0', 'x', '1', '2', '3', '4', …
`"0x1234..."`	`hexToBytes()` (correct)	`[18, 52, ...]`	Actual signature bytes (0x12, 0x34, …)

The wrong encoding produces:

- ASCII text representation of the hex string.
- Values constrained to only 17 possible byte values: `[48-57, 97-102, 120]` representing `['0'-'9', 'a'-'f', 'x']`.
- First two bytes always `[48, 120]` (ASCII for "0x").
- Highly predictable and structured data.

The correct encoding produces:

- Actual cryptographic signature bytes.
- Values spanning the full byte range (0-255).
- Cryptographically random distribution.
- Unpredictable data derived from the signature algorithm.

Impact on KEK derivation:

The signature bytes are used as input material for HKDF to derive the Key Encryption Key (code ref). This means the KEK protecting all user keys is derived from predictable ASCII text patterns rather than cryptographically random signature bytes, fundamentally undermining the security model.

Problem scenario

This vulnerability requires XSS to be exploited. An attacker must first:

1. Achieve Cross-Site Scripting (XSS) on the application.
2. Use XSS to exfiltrate the encrypted vault from IndexedDB.

The attacker does not need to intercept the signature. This vulnerability enables offline brute-force attacks against the vault alone (i.e. they only need to brute-force ASCII text patterns). Without this bug, an attacker would need to brute-force the actual cryptographic signature space.

Entropy comparison:

Metric	Wrong encoding	Correct encoding	Impact
Possible byte values	17 (ASCII hex)	256 (full range)	15x reduction
Entropy per byte	~4 bits	8 bits	50% reduction
First 2 bytes	Always [48, 120]	Random	100% predictable
Pattern	Highly structured	Random	Exploitable

An attacker analyzing the KEK derivation knows:

- The input is ASCII text of a hex string.
- It always starts with “0x”.
- Every byte is in the range [48-57] (digits) or [97-102] (a-f) or 120 (x).
- The pattern follows hexadecimal text structure.
- Only 17 possible values per byte instead of 256.

This transforms the problem from attacking a cryptographic signature to attacking a predictable text pattern.

Recommendation

Replace UTF-8 text encoding with proper hex decoding:

```tsx
// src/hooks/useSignUp.tsx
import { useMutation } from "@tanstack/react-query";
import { useAccountVault } from "hooks/useAccountVault";
import type { VaultEntry } from "types";
import type { Address, Hex } from "viem";
import { hexToBytes } from "viem";

export const useSignUp = () => {
  const { create, retrieve, unlock } = useAccountVault();

  return useMutation({
    mutationFn: async ({
      walletAddress,
      signature,
    }: {
      walletAddress: Address;
      signature: Hex;
    }) => {
      const existingVault = await retrieve(walletAddress);
      const signatureBytes = hexToBytes(signature);

      // ... rest of code unchanged
    },
  });
};
```

Resolution

The development team addressed this finding in PR #115.

Emergency caller can only be set once

Severity Informational **Exploitability** N/A **Status** Acknowledged

Type Implementation **Impact** N/A

Involved artifacts

- `anomapay-backend`: `contracts/src/bases/EmergencyMigratableForwarderBase.sol`

Description

The `EmergencyMigratableForwarderBase` contract enforces that the emergency caller can only be set once through the `setEmergencyCaller` function. Once `_emergencyCaller` is set to a non-zero address (code ref), any subsequent attempt to change it will revert with `EmergencyCallerAlreadySet`. We assume this is an intentional design decision that provides certain security guarantees during the migration phase.

Design rationale

The immutability of the emergency caller after initial setting serves several purposes:

1. **Prevents post-migration manipulation**: Once migration begins, the emergency caller cannot be changed, ensuring users know exactly which contract (e.g., FWv2) can access FWv1's tokens throughout the entire migration period.
2. **Simplifies trust model:** Users only need to verify the emergency caller once when it's set by the emergency committee. There is no need to monitor for subsequent changes that could redirect migrated funds.
3. **Enforces careful vetting:** The emergency committee must thoroughly test and validate the new forwarder version before setting it as the emergency caller, since this decision is irreversible.

Implications

This design accepts certain trade-offs:

1. **Cannot correct mistakes:** If the emergency committee accidentally sets an incorrect address as the emergency caller, there is no way to fix this error.
2. **Cannot upgrade if bugs found:** If a critical bug is discovered in FWv2 after it's been set as the emergency caller, there is no way to designate a patched FWv2.1 contract. Users attempting to migrate would be forced to use the buggy contract.
3. **Cannot respond to compromises:** If the emergency caller contract is compromised or its private keys are leaked, the emergency committee cannot designate a new secure contract as the emergency caller.

Double JSON encoding of Keyring

Severity Informational **Exploitability** N/A **Status** Acknowledged

Type Implementation **Impact** N/A

Involved artifacts

- `pay-interface-app`: `src/domain/keys/services.ts`

Description

The application performs double JSON encoding when serializing the user keyring. Each individual key pair is first serialized to a JSON string using `KeyPairSerializer.toJson()` (code ref), and then the entire keyring object (containing these JSON strings) is serialized again using `JSON.stringify()` (code ref).

Current output format:

```json
{
  "authorityKeyPair": "{\"privateKey\":\"a1b2c3d4e5f6...\",\"publicKey\":
  \"f6e5d4c3b2a1...\"}",
  "nullifierKeyPair": "{\"nk\":\"1234567890ab...\",\"cnk\":\"abcdef123456...\"}",
  "discoveryKeyPair": "{\"privateKey\":\"9876543210fe...\",\"publicKey\":
  \"fedcba098765...\"}",
  "encryptionKeyPair": "{\"privateKey\":\"abcdef123456...\",\"publicKey\":
  \"654321fedcba...\"}"
}
```

This pattern works correctly (deserialization handles double decoding properly), but has some downsides:

- Each `"` inside nested JSON becomes `\"`.
- Escape characters add ~10-15% size overhead.
- The serialization logic is split across two layers unnecessarily

For encrypted vaults in IndexedDB, this means slightly larger storage footprint and network transfer sizes.

Recommendation

Refactor to perform single-level JSON encoding that produces nested objects directly.

Change `KeyPairSerializer` to return objects:

```tsx
// src/domain/keys/models.ts
export class KeyPairSerializer {
  /**
   * Serializes a {@link KeyPairBase} subclass into a plain object.
   * @param keyPair Instance to serialize
   * @returns Plain object with hex-encoded keys
   */
  static toObject<T extends KeyPairBase>(keyPair: T): Record<string, string> {
    const ctor = keyPair.constructor as KeyPairConstructor<T>;
    const { privateKey, publicKey } = ctor.keysName;

    return {
      [privateKey]: toHex(keyPair.keys.privateKey),
      [publicKey]: toHex(keyPair.keys.publicKey),
    };
  }

  /**
   * Deserializes a plain object produced by {@link toObject} back into a key pair.
   * @param constructor Key pair subclass to instantiate
   * @param obj Plain object with key fields matching {@link KeyPairBase.keysName}
   * @returns Key pair with private/public keys restored from hex
   * @example  KeyPairSerializer.fromObject(NullifierKeyPair,
   keyringObj.nullifierKeyPair)
   */
  static fromObject<T extends KeyPairBase>(
    constructor: KeyPairConstructor<T>,
    obj: Record<string, string>
  ): T {
      try {
        const { privateKey, publicKey } = constructor.keysName;

        const privHex = obj[privateKey];
        const pubHex = obj[publicKey];

        if (typeof privHex !== "string" || typeof pubHex !== "string") {
          throw new Error("Missing key fields in object");
        }

        return new constructor(fromHex(privHex), fromHex(pubHex));
      } catch (e) {
      throw new Error(
        `Invalid JSON: ${e instanceof Error ? e.message : String(e)}`
      );
    }
  }
}
```

Update keyring serialization:

```tsx
// src/domain/keys/services.ts
export const convertUserKeyringToJson = (userKeyring: UserKeyring): string => {
  return JSON.stringify({
    authorityKeyPair: KeyPairSerializer.toObject(userKeyring.authorityKeyPair),
    nullifierKeyPair: KeyPairSerializer.toObject(userKeyring.nullifierKeyPair),
    discoveryKeyPair: KeyPairSerializer.toObject(userKeyring.discoveryKeyPair),
    encryptionKeyPair: KeyPairSerializer.toObject(userKeyring.encryptionKeyPair),
  });
};

export const convertUserKeyringFromJson = (json: string): UserKeyring => {
  const keyringObj = JSON.parse(json);
  return {
    authorityKeyPair: KeyPairSerializer.fromObject(
      KeyPair,
      keyringObj.authorityKeyPair
    ),
    nullifierKeyPair: KeyPairSerializer.fromObject(
      NullifierKeyPair,
      keyringObj.nullifierKeyPair
    ),
    discoveryKeyPair: KeyPairSerializer.fromObject(
      KeyPair,
      keyringObj.discoveryKeyPair
    ),
    encryptionKeyPair: KeyPairSerializer.fromObject(
      KeyPair,
      keyringObj.encryptionKeyPair
    ),
  };
};
```

Improved output format:

```json
{
  "authorityKeyPair": {
    "privateKey": "a1b2c3d4e5f6...",
    "publicKey": "f6e5d4c3b2a1..."
  },
  "nullifierKeyPair": {
    "nk": "1234567890ab...",
    "cnk": "abcdef123456..."
  },
  "discoveryKeyPair": {
    "privateKey": "9876543210fe...",
    "publicKey": "fedcba098765..."
  },
  "encryptionKeyPair": {
    "privateKey": "abcdef123456...",
    "publicKey": "654321fedcba..."
  }
}
```

Insufficient error handling in Web Crypto API operations

Severity Informational **Exploitability** N/A **Status** Resolved

Type Implementation **Impact** N/A

Involved artifacts

- `pay-interface-app`: `src/domain/crypto/services.ts`

Description

The application correctly uses Web Crypto API functions (`SubtleCrypto.importKey()`, `SubtleCrypto.deriveKey()`, and `SubtleCrypto.encrypt()`) with appropriate cryptographic parameters. However, most of these operations lack explicit error handling through try-catch blocks, causing generic DOM exception errors to propagate to callers without specific context.

Functions with missing error handling:

1. `importKey()` call in `wrapSignatureAsCryptoKey()` (code ref)
2. `deriveKey()` call in `deriveKeyEncryptionKey()` (code ref)
3. `encrypt()` call in `encrypt()` (code ref)

Function with proper error handling:

1. `decrypt()` call in `decrypt()` (code ref)
   - Has try-catch block
   - Returns safe generic error message
   - Prevents information leakage

While the security impact is minimal because Web Crypto API errors are intentionally generic (DOM exceptions don't leak sensitive information by design), adding error handling provides:

1. Better debugging: Distinguish between import failures, derivation failures, and encryption failures.
2. Improved user experience: Show meaningful messages instead of generic errors.
3. Consistency: Match the pattern already used in `decrypt()`.
4. Future-proofing: If error handling requirements change, the infrastructure is in place.

Recommendation

Follow the pattern established in `decrypt()` and add try-catch blocks to the other three functions.

1. Update `wrapSignatureAsCryptoKey()`

```tsx
// src/domain/crypto/services.ts
```

```
export const wrapSignatureAsCryptoKey = async (
  signatureBytes: Uint8Array
): Promise<CryptoKey> => {
  try {
      return window.crypto.subtle.importKey(
        "raw",
        signature,
        { name: "HKDF" },
        false,
        ["deriveKey"]
      );
  } catch (e) {
    throw new Error("Failed to prepare signature for key derivation");
  }
};

export const deriveKeyEncryptionKey = async (
  signatureCryptoKey: CryptoKey
): Promise<CryptoKey> => {
  try {
      const encoder = new TextEncoder();
      return window.crypto.subtle.deriveKey(
        {
          name: "HKDF",
          hash: "SHA-256",
          salt: encoder.encode(VAULT_DOMAIN_SALT),
          info: encoder.encode(VAULT_DOMAIN_INFO),
        },
        key,
        { name: "AES-GCM", length: 256 },
        false,
        ["encrypt", "decrypt"]
      );
  } catch (e) {
    throw new Error("Failed to derive encryption key");
  }
};

export const encrypt = async (
  kek: CryptoKey,
  plaintext: string
): Promise<EncryptionResult> => {
  try {
        const iv = generateRandomBytes(12);
      const ciphertext = await window.crypto.subtle.encrypt(
        {
          name: "AES-GCM",
          iv,
        },
        keyEncryptionKey,
        new TextEncoder().encode(serializedKeyring)
      );
      return {
        iv,
        ciphertext,
      };
  } catch (e) {
    throw new Error("Failed to encrypt data");
  }
};
```

Resolution

The development team addressed this finding in PR #115.

XSS prevention as security control

Severity Informational **Exploitability** N/A **Status** Acknowledged

Type Implementation **Impact** N/A

Involved artifacts

- `pay-interface-app`

Description

XSS prevention is an important security control for this application. This application manages cryptographic keys in the browser with architectural characteristics that create unique risks:

1. Permanent signature: Static signature with no nonce/timestamp acts as permanent "master password" to derive encryption keys.
2. Memory exposure: JavaScript provides no secure memory management; decrypted keys exist as plaintext in memory.
3. Client-side storage: Encrypted vaults stored in IndexedDB are accessible to any same-origin JavaScript.

However, all of these risks require code execution to exploit. An attacker needs XSS to:

- Intercept the signature during use.
- Exfiltrate the encrypted vault from IndexedDB.
- Read decrypted keys from memory.

Recommendations

Implement XSS prevention through multiple defensive layers:

1. Strict Content Security Policy (CSP)

Implement a strict CSP that only allows scripts from your own origin, with no exceptions for inline scripts or eval.

```
Content-Security-Policy:
  default-src 'self';
  script-src 'self' <TRUSTED_SCRIPT_DOMAINS> 'nonce-{RANDOM}';
  style-src 'self' <TRUSTED_STYLE_DOMAINS> 'nonce-{RANDOM}';
  img-src 'self' data: <TRUSTED_IMAGE_DOMAINS>;
  font-src 'self' <TRUSTED_FONT_DOMAINS>;
  connect-src 'self' <TRUSTED_API_DOMAINS>;
  frame-src 'none';
  frame-ancestors 'none';
  base-uri 'self';
```

```
form-action 'self';
object-src 'none';
```

`<TRUSTED_*>` should be replaced with a minimal allow-list of external domains that are required for application functionality (e.g. CDNs, monitoring services, analytics), and no others. It should be rolled out in `Content-Security-Policy-Report-Only` mode first to identify violations before enforcing it.

Critical rules:

- Never use `unsafe-inline` or `unsafe-eval`.
- Never use wildcards in `script-src`.
- Configure CSP violation reporting to monitor attempts.
- Test CSP in report-only mode first, then enforce.

2. Aggressive input sanitization

Treat all data as potentially malicious and sanitize any HTML before rendering.

3. Framework-level protections

React automatically escapes JSX content, preventing most XSS attacks. Avoid `dangerouslySetInnerHTML`, validate URLs before rendering links, and never use `eval()` or string-based `setTimeout()`.

4. Supply chain security

Continuously monitor dependencies, update promptly, and audit package scripts (e.g. `npm audit` and `npm outdated`).

5. Subresource integrity (SRI)

If you load scripts from CDNs, use SRI hashes to ensure the files haven't been tampered with. If the CDN is compromised or serves different content, the browser will refuse to execute the script.

Miscellaneous code improvements

Severity N/A **Exploitability** N/A **Status** Reported

Type Implementation **Impact** N/A

- The current implementation in `TokenTransferWitnessV2::constrain()` manually computes the tag by calculating the commitment and conditionally computing the nullifier based on whether the resource is consumed (code ref). This logic duplicates the existing `tag()` method (code ref), which already encapsulates the same computation. Replacing the manual implementation with a simple `let tag = self.tag()?;` call would eliminate code duplication, and bring consistency with the V1 implementation (code ref), which already uses this pattern. This is a low-risk refactoring with no behavioral changes, as both approaches are functionally equivalent—they produce the same nullifier for consumed resources and commitment for created resources.

  The development team addressed this recommendation in PR #210.

- While `rand::random()` is generally secure (here, here), explicitly using `OsRng` for generating encryption nonces would improve security. The change is minimal: adding `use rand::rngs::OsRng;` and replacing `rand::random()` with explicit `OsRng` calls.

  The development team addressed this recommendation in PR #210.

- The forwarder contracts follow the CEI pattern by emitting events before making external calls. For migration operation, this causes FWv2 to emit the `Wrapped` event before calling FWv1, which then emits the `Unwrapped` event, resulting in logs showing "Wrapped → Unwrapped" when the logical flow is "unwrap from V1 → wrap into V2". While emitting the `Wrapped` event after the emergency call would improve log readability and chronological clarity, it would break the CEI pattern. However, the security risk is minimal since events don't modify state. The current implementation prioritizes consistent security patterns over log readability, which is an acceptable tradeoff, though reordering events specifically for migration could be considered if log clarity is preferred.

Appendix: Vulnerability Classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of Common Vulnerability Scoring System (CVSS) v3.1, which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the Impact score, and the Exploitability score. The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ CVSS Qualitative Severity Rating Scale, and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

Impact Score	Examples
🟠 **High**	Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic
🟡 **Medium**	Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x)
🟢 **Low**	Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction)
🔵 **None**	Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation

Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be
  - *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
  - *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for < 3%; *medium* for 3-10%; *large* for 10-33%, *all* for >33%. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small

capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

Exploitability Score	Examples
🟠 **High**	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
🟡 **Medium**	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
🟢 **Low**	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
🔵 **None**	illegitimate actions taken in a coordinated fashion by all actors

Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.

As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

Severity Score	Examples
🔴 **Critical**	Halting of chain via a submission of a specially crafted transaction
🟠 **High**	Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers
🟡 **Medium**	Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users
🟢 **Low**	2x increase in node computational requirements via coordinated withdrawal of all user tokens
🔵 **Informational**	Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability, etc.) set forth in the associated Services Agreement. This report provided in connection with the Services set forth in the Services Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This audit report is provided on an “as is” basis, with no guarantee of the completeness, accuracy, timeliness or of the results obtained by use of the information provided. Informal has relied upon information and data provided by the client, and is not responsible for any errors or omissions in such information and data or results obtained from the use of that information or conclusions in this report. Informal makes no warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of this report. This report should not be considered or utilized as a complete assessment of the overall utility, security or bugfree status of the code.

This audit report contains confidential information and is only intended for use by the client. Reuse or republication of the audit report other than as authorized by the client is prohibited.

This report is not, nor should it be considered, an “endorsement”, “approval” or “disapproval” of any particular project or team. This report is not, nor should it be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts with Informal to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor does it provide any indication of the client’s business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should it be leveraged as investment advice of any sort.

Blockchain technology and cryptographic assets in general and by definition present a high level of ongoing risk. Client is responsible for its own due diligence and continuing security in this regard.