PUBLIC

Code Assessment

of the SRG Smart Contracts

April 04, 2023

Produced for

Blockswap LABS

by

CHAINSECURITY

Contents

1 Executive Summary 3
2 Assessment Overview 5
3 Limitations and use of report 13
4 Terminology 14
5 Findings 15
6 Resolved Findings 19
7 Informational 23
8 Notes 26

1 Executive Summary

Dear BlockSwap,

Thank you for trusting us to help Blockswap with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of SRG according to Scope to support you in forming an opinion on their security risks.

Blockswap implements State Replication Gateway - a cross chain state portability system, that allows the extension of a smart contract states between EVM-compatible chains.

The most critical subjects covered in our audit are functional correctness, asset solvency and signature handling. Security regarding Functional correctness and asset solvency is good. Signature handling is improvable, see Problems Related to Consent and ConsentVerification.

The general subjects covered are event handling and gas efficiency. Gas efficiency is improvable, see Gas Optimisation. Event handling can be improved as well, see Pausing and Unpausing Emit Misleading Events.

In summary, we find that the codebase provides an improvable level of security.

Many of the issues we identified during our assessment, which you have acknowledged without taking action, have the potential to cause human errors and other negative impacts. It is important to address these issues promptly to ensure the overall safety and reliability of your system.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	5
• Code Corrected	4
• Acknowledged	1
Low-Severity Findings	11
• Code Corrected	3
• Specification Changed	2
• Acknowledged	6

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed based on the source code files in `contracts/modular-gateway` folder from the SRG repository. The main source of the documentation is the `README.md` and `savETHImplementation.md` files from the SRG repository. Details about the specification were clarified by the Blockswap during direct conversations over communication platforms. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	16 January 2023	d0f119b9b6eeede733d18f578b554fe8db14ce13	Initial Version
2	7 March 2023	6258be552fa5afcc0e2c9a04666f79fc2e2b6fa4	Version with fixes
3	3 April 2023	097a53bb4146c95cc80a50d66efe7b4db4b9ca9f	Version with fixes

For the solidity smart contracts, the compiler version `0.8.13` was chosen.

2.1.1 Excluded from scope

Any imported libraries and contracts that are not mentioned in the Scope are excluded from the assessment. The processes of registering, managing and monitoring the Endorsers in the `EndorserRegistry` contracts are outside the scope of this assessment.

2.1.2 Assumptions

Certain things regarding this assessment are assumed to be always correct:

1. Assessed system is made of multiple modules that need to be connected with proper configuration. We assume that the setup of the system is always correct, meaning the deployed system is capable to perform its functions. All the correct contracts are assumed to be deployed and initialized correctly. We assume no bugs might arise due to misconfiguration.
2. We assume, that the ERC20 tokens the SRG will deal with, do not have any special behaviors: reentrancies, rebasing(balance changes without transfers), fees on transfer. Special wrapper ERC20 contracts that protect the assessed system from such behaviors are assumed to be used when needed.
3. Endorsers registered in the `EndorserRegistry` contracts assumed to be well-behaving.
4. Anyone can get ownership of KNOT from the savETHRegistry open index. Users can isolate the KNOT from the open index just before the fee mint and return KNOT back to the open index after the mint. This way the mint fee will go to the user. This arbitrage opportunity is assumed to be welcomed. Users, who put their KNOTs to open index, are assumed to know about this.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview. The information about components that are out-of-scope for this assessment (e.g. behaviour of Endorsers) is taken directly from the documentation provided by the client and assumed to be correct.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Blockswap implements a Blockswap State Replication Gateway (SRG) - a system that enables cross-chain state replications. Such a process of creating a state representation from one chain(a.k.a. original domain) on the another domain (destination domain) is called a state extension. In the assessed version of the code the SRG supports the ERC20 token balances extension and savETH Registry dETH balances extension. To extend a state from one domain to the other, a **deposit** transaction needs to be performed on the originating domain. Then, the Endorsers, privileged actors, need to attest to the data integrity of a Gateway. Later **push** transaction, containing references to the corresponding deposit transaction need to be made on destination domain.

2.2.1 Terminology of use

In this system overview, we use origin chain / mainnet and destination chain / Optimism interchangeably. In general, the origin is not always mainnet and destination is not always Optimism. Origin is the chain where the state originates and the destination is where the state is created and directly tied to the original domain. The user on the origin chain is called Alice, while the user on the destination chain is called Bob.

2.2.2 General setup of the SRG system

When a state originating on one domain needs to be extended to multiple destination domains a set of contracts need to be deployed on both destination and origin chains:

1. `Gateway`: Depending on the nature of the bridged token, this can be `ERC20Gateway` or `savETHGateway` contract.
2. `Accumulator`: Contains the implementation of incremental Merkle tree data structure. It collects the state change receipts for further attestation.
3. `Ingestor`: Handles the "muting" or burning of the assets on the originating domain during the deposit transaction.
4. `Dispenser`: Manages the "unmuting" or minting of the assets on the destination domain during the push transaction. Each Dispenser inherits the `dispenseViaRecovery` function. This function receives Merkle proof and calls the internal `_performDispense` function if the proof is valid for some unclaimed leaf in the Merkle tree with `recoveryMerkleRoot`.
5. `EndorserRegistry`: Contains the Endorsers and manages their statuses. Its owner can modify the list of valid endorsers and their status.
6. `RPBSVerificationLibrary`: Checks the validity of the endorsements. Each endorsement is a Restricted Partially Blinded Signature (RPBS). Common (unblinded) information contains the accumulator root, deposit leaf and its index, endorsement expiration, destination gateway address, and origin gateway address. Hash of Merkle proof is however blinded. Any validator can guess the data inside the blinded part, however, the Blockswap stated that the RPBS gives them flexibility for future features.
7. `ConsentVerification`: Initiator of the deposit submits consent - ECDSA signature that the asset receiver uses on the destination domain. The ConsentVerification contract validates this ECDSA signature.

A new instance of Gateway needs to be deployed for any asset that needs to be bridged from the chains, in any direction. Each asset or state is native to a single chain, called the origin. Origin gateway handles

extensions for all destination domains. However, the destination gateways can only extend their state back to the origin gateway and not to other destination chains.

2.2.3 General Deposit Flow

1. Alice signs a consent - EIP712-compliant (according to the specification. See Problems Related to Consent and ConsentVerification related issue) hash of the following struct:

```
Consent(
    uint256 consentId,
    address paramOne,
    uint256 paramTwo,
    bytes paramThree
)
```

   consentId in this case acts as a nonce that increases for Alice every time she makes a state extension on a chain.

2. On chain X Alice calls `deposit` or `batchDeposit` function on a Gateway responsible for the desired contract with proper input parameters.
3. Gateway performs various checks of the deposit parameters. First of all, it checks that the correct consentId(alias for nonce) is used by Alice to issue the consent and the caller is Alice herself. Then, by using the internal function `_assertConsumptionAuthorised`, the validity of the transaction summary is checked.
4. Function `consume` called on the Ingestor contract
5. Gateway then validates the amount extended to the destination chain against `batchStakingRule`.
6. `totalExtended` to the destination domain is increased by the amount of the deposit made by Alice.
7. `DepositEvent` is emitted.
8. Deposit transaction summary, which is a hash of transaction parameters, is injected via a call to the `Accumulator.injectVectorCommitment` function. `Accumulator` contract is an incremental Merkle tree implementation, that collects state change receipts for further attestation.
9. Alice's `consentId` gets incremented for further deposits.

2.2.4 General Push flow

To finalize the extension on the destination, Bob needs to collect `Accumulator` RPBS endorsements from the Endorsers. Once the endorsements are gathered, Bob can proceed in the following sequence:

1. Bob calls the Gateway `push` or `batchPush` function on the destination chain, providing deposit details, endorsements, Merkle tree proof and push transaction summary.
2. `push` function checks the validity of the origin chain gateway, destination chain Id, and gateway. If pushing takes place without recovery mode enabled, the destination gateway verifies the validity of RPBS endorsements by calling `RPBSVerificationLibrary`.
3. A subsequent call to the internal function `_dispense` is made. If recovery has not been enabled happen:

    3.1. Destination Gateway verifies the validity of Alice's consent signature by calling `ConsentVerification.validateConsentSignature` function.

    3.2. A call to `_assertDispenseAuthorised` verifies the validity of the transaction summary

3.3. Finally, `Dispenser.dispense()` gets called, which calls the internal `_performDispense` function, which is overridden, depending on the gateway type.

3.4. This UTXO is marked as spent.

3.5. Push transaction summary, which is a hash of transaction parameters, is injected via a call to the `Accumulator.injectVectorCommitment` function.

If recovery has been enabled:

3.1. Recovery checkpoint, as well as Merkle root for recovery, should already be injected

3.2. When the previous checks passed, the Gateway calls into the `Dispenser` module `dispenseViaRecovery` function.

4. The extended amount should satisfy the `batchStakingRule`.
5. `totalExtended` amount of the origin chain gets increased by the amount extended.
6. A `Push` event gets emitted.

2.2.5 ERC20Gateway

To handle the bridging of ERC20 token balances, specialized versions of the contracts from the General setup of the SRG system have been implemented:

- ERC20Gateway - this version of the Gateway contract is intended to be used with a fixed token address. In the `_assertConsumptionAuthorised` function this contract checks that the deposit transaction summary matches the hash of deposit parameters. In the `_assertDispenseAuthorised` function this contract checks the validity of Merkle tree proof.
- GatewayToken - this contract acts as a representation of the token on the destination domain. ERC20Gateway of the destination domain uses this token to mint assets that were bridged from the origin domain.
- ERC20OriginIngestor - this contract acts as ingestor on the chain, where token originates from. The `consume` function of this contract `safeTransferFrom` funds from the user to the `ERC20OriginDispenser`.
- ERC20OriginDispenser - during `push` operations on `ERC20Gateway` contracts, the `ERC20OriginDispenser._performDispense` function transfers the tokens to the specified receiver.
- ERC20DestinationDispenser - during `push` operations on `ERC20Gateway` contracts, the `ERC20DestinationDispenser._performDispense` mints new `GatewayToken` assets.
- ERC20DestinationIngestor - the `consume` function of this contract burns the `GatewayToken` tokens.

2.2.5.1 ERC20 Deposit and Push Flows

The ERC20 token follows the General Deposit Flow and the General Push Flow. To deposit, the owner of the tokens needs to sign the hash of Consent struct. The fields have the following meaning:

- `paramOne` - recipient address on the destination domain
- `paramTwo` - amount of tokens
- `paramThree` - optional data. Used in DepositEvent and contributes to the deposit leaf hash.

Differences described in the list above Explain how specialized contracts alter the flows.

2.2.6 savETHGateway

2.2.6.1 Brief savETH Registry overview

Users can stake a 32 ETH validator with the Ethereum Deposit Contract through the StakeHouse registry and in exchange get 24 dETH and 8 SLOT token. SLOT tokens are not relevant to this assessment and won't be covered in this System Overview. 24 dETH token by default belongs to a KNOT, that is registered in `savETHRegistry` contract. Each KNOT is identified by 48 bytes BLS pub key of the validator. KNOTs belong to a certain index inside the `savETHRegistry`. Each index is controlled by a single user address and multiple KNOTs can be attached to the same index. When Beacon chain validator performs its duties, the rewards are generated. An authorized minter can mint the reward dETH to a specified KNOT.

KNOTs can be added to an open index - a special kind of index that allows index owners to exit into liquid ERC20 savETH tokens. savETH represents the share of dETH tokens owned by KNOTs in an open index. Rewards increase the exchange rate of savETH to dETH. Any savETH ERC20 holder can withdraw - burn the savETH and mint actual liquid dETH that are not associated with any KNOT.

When savETH state needs to be extended to the foreign domain, the owner of the index needs to provide the 48 bytes KNOT pubkey as an argument for the `deposit` function.

2.2.6.2 savETH system setup

To handle the bridging of savETH KNOTs, specialized versions of the contracts from the General setup of the SRG system have been implemented:

- savETHGateway - this gateway contract works with a fixed `StakeHouseUniverse` - registry of stakehouse contracts. In the `_assertConsumptionAuthorised` function this contract checks that the deposit transaction summary matches the hash of deposit parameters. It also requires that the KNOT's index owner is msg.sender. Only KNOTs that hold at least 24 ether are valid. In the `_assertDispenseAuthorised` function this contract checks the validity of Merle tree proof. It also requires that the receiver index owner is msg.sender. This contract is abstract and two separate contracts that extend it are meant to be deployed: savETHDestinationGateway and savETHOriginGateway. Their specific functionality is described in savETH poke and balanceIncrease.
- dETHOriginIngestor - the `consume` function of this contract transfers the deposited KNOT from the user to the designated gateway index.
- dETHOriginDispenser - the `_performDispense` function of this contract transfers the KNOT from the designated gateway index to the destination index.
- dETHDestinationDispenser - the `_performDispense` function of this contract creates a new KNOT on the foreign domain and sets the balance of this KNOT to the same value as on the origin domain.
- savETHRegistryDestinationGateway - this is a specialized version of the `savETHRegistry` contract. It has a new `rageQuitAndCleanUp` function, which completely deletes the data about the KNOT from the registry. A KNOT can be recreated again after a call to this function.
- dETHDestinationIngestor - the `consume` function of this contract transfers the deposited KNOT to the gateway index and calls `savETHDestinationGateway.rageQuitAndCleanUp`.
- savETHDestinationReporter - this contract is an authorized minted for the `savETHRegistryDestinationGateway`. The `balanceIncrease` function of it is described in savETH poke and balanceIncrease.
- StakeHouseUniverseDestinationGateway - a miniature version of the full StakeHouseUniverse needed to operate the savETH registry on the destination domain.

2.2.6.3 savETH Deposit Flow

The extension of the savETH KNOTs follows the General Deposit Flow. To deposit, the owner of the index that holds the KNOT needs to sign the hash of the Consent struct. The fields have the following meaning:

- `paramOne` - address of StakeHouse contract associated with KNOT being moved.
- `paramTwo` - destination savETH index.
- `paramThree` - KNOT's BLS pub key. The full dETH amount in this knot will be extended to the other domain

While in ERC20 Deposit and Push Flows the ERC20OriginDispenser takes ownership of tokens on the origin side, in savETH case KNOT is moved into an unspendable index known by the Gateway. Similar to mint/burn procedures in the foreign domain, the KNOTs are created and `rageQuitAndCleanUp`. Differences described in the list above Explain how specialized contracts alter the deposit flow.

2.2.6.4 savETH Push Flow

The extension of the savETH KNOTs follows the General Push Flow. The `savETHDestinationGateway` has 2 additional push functions: `pushAndWithdrawDETHFromOpenIndex` and `batchPushAndWithdrawDETHFromOpenIndex`. They can be seen as a combination of the following sequence: regular savETH push of the KNOT, move of the KNOT to open index, withdrawal of savETH ERC20 into dETH ERC20 token.

Differences described in the list above Explain how specialized contracts alter the push flow.

2.2.6.5 savETH poke and balanceIncrease

On the origin chain an authorized miner can mint new dETH inflation rewards to KNOTs when the beacon chain reports such rewards. The `savETHOriginGateway` contract has special `pokeLatestBalance` and `batchPokeLatestBalances` functions that are used to propagate the origin registry balance changes to the destination domain. On the foreign domain, these poke actions are finalized with the help `balanceIncrease` and `batchBalanceIncrease` functions of the `savETHDestinationGateway` contract. Poke actions are not authorized, meaning anyone can poke any KNOT that was migrated to the foreign domain. No consent is needed. RPBS endorsements need to be passed to the `balanceIncrease` function.

Flow of poke in `savETHOriginGateway` contract:

1. Knot balance in the gateway is fetched from `savETHRegistry`.
2. Check this poking action is permissible by assuring that the Knot is present in the gateway, valid, and active.
3. Poke event is emitted.
4. Poke leaf hash is inserted into the accumulator of the destination domain.

Now, finalize the poke the `savETHDestinationGateway` uses the following `balanceIncrease` function on the foreign chain:

1. Origin gateway, destination chain Id, and gateway are validated.
2. RPBS endorsements are validated.
3. Check that the Knot is already migrated to the current(foreign) domain
4. Validate the Merkle proof for the poke leaf.
5. A call to `savETHDestinationReporter.increaseBalance()` is made. It mints dETH for the KNOT.
6. Balance increase leaf is injected into the accumulator

2.2.7 Pausability and Recovery

The following special functions can be used by privileged users to kill the functionality of the gateways:

- Gateway.pauseGateway - pauses the Gateway. All actions such as push, deposit, poke, balanceIncrease are disabled for all the domains. Can only be called by the privileged GatewayOperator role holder. Can be undone by unpauseGateway function.
- Gateway.pauseDomain - Disables all actions for a given domain. Can only be called by the privileged GatewayOperator role holder. Can be undone by unpauseDomain function.
- Gateway.triggerKillSwitch - triggers the activation of the recovery assets on a specific domain. Always disables the deposit, poke and balanceIncrease actions. If specified, push actions can be disabled as well. Can only be called by the privileged GatewayOperator role holder. Cannot be undone.
- IGatewayOperatorControlCentre - a special contract that Gateway uses to determine if a specific domain is killed.

The recovery process for a domain:

1. `triggerKillSwitch` function is activated by the GatewayOperator. Push actions should not be killed.
2. `injectForeignDomainCheckpoint` function is called by the GatewayOperator. While the assigned checkpoint value is not immediately used on-chain, it will be used off-chain for reconcilliation.
3. `injectRecoveryMerkleRoot` function is called by the operator. It sets the recovery Merkle tree root that push actions can later use.

After this process, the dispenser modules associated with the killed domain is put into special recovery mode, where the General Push flow has the following changes:

- The RPBS endorsements are not checked by the Gateway
- The accumulator proof needs to contain a Merkle proof for the Recovery Merkle tree root, instead of the Accumulator tree root.
- `Dispenser.dispenseViaRecovery` is executed instead of step 3 in General Push flow. Dispensers in this mode check the validity of the recovery Merkle proof and execute `_performDispense`.

2.2.8 Roles and Trust model

Several contracts of SRG system are deployed via proxy, including `Gateway`, `GatewayToken`, `Accumulator`, `StakeHouseUniverseDestinationGateway`, and origin and destination Ingestors/Dispensers. The proxy admin is considered fully trusted. In general, we assume the deployers and accounts with the owner role as fully trusted and they configure contracts with the correct parameters.

All contracts implementing or facilitating the implementation of the gateway functionality as well as `savETHRegistry`, `Universe`, `BalanceReporter`, etc. have privileged accounts that need to be trusted to behave correctly for the bridge to function as expected.

Accumulator Managers: As they are the only entities privileged to insert new leaves into the incremental Accumulator Merkle tree, they should be trusted.

Endorsers: They play a critical role in verifying the data fetched from the origin/destination chain indicating a deposit action is not tampered with. Hence, they are assumed to be fully trusted.

Gateway Operators: These users should be treated as fully trusted, because they are entitled to add new domains, trigger kill switch, inject checkpoint and Merkle tree root to the dispenser modules, pause/unpause their gateway and domain.

Gateway Admins: Those users holding this role are considered to be fully trusted, as they are privileged to add new admins and operators.

Token Minter/Burner: Addresses marked as minter/burner, as the name suggests, can burn and mint `GatewayToken`'s. As a result, they should be fully trusted.

Token Admins: They are entitled to add new token admins and token minters/burners. Hence, they have to be trusted.

Users: Users interacting with the system are considered untrusted.

2.2.9 Version 2 changes

In Version 2 following changes were made:

`GatewayToken` has a single minter and single burner address. These addresses are assumed to be `ERC20DestinationDispenser` and `ERC20DestinationIngestor` correspondingly. There are no token admins anymore on `GatewayToken`. These new minter and burner roles are fully trusted or, in the case of smart contracts, assumed to be correctly set to contracts that cannot mint/burn a wrong number of tokens.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- Problems Related to Consent and ConsentVerification Acknowledged

Low-Severity Findings	6

- Finalization of Extension Can Stuck Acknowledged
- Missing Status by EndorserLifecycleStatusUpdated Acknowledged
- Pausing and Unpausing Emit Misleading Events Acknowledged
- RPBS _isOnCurve Infinity Point Acknowledged
- Repeated Pokes Acknowledged
- saveETHManager.init Is Not Defined in ISavETHManager Acknowledged

5.1 Problems Related to Consent and ConsentVerification

Design Medium Version 1 Acknowledged

The consent ECDSA signature that the depositor needs to provide has the following design issues:

1. When Alice wants to deposit 100 wei ERC20 token from Optimism to Bob on mainnet, she has to sign EIP712 hash of `Consent(Bob, 100, "")`. However, the same signature she has to provide if Alice wants to deposit 100 ERC20 token from Polygon to Bob. This signature does not contain any information about origin domain. Only address of ConsentVerification contract on the destination chain is taken into account.
2. Only EOA addresses are able to sign data. Multisig wallets or any other smart contract addresses won't be able to provide a consent signature. This limits potential integrations of SRG with the other systems
3. Consent itself is redundant. Both ERC20Gateway and savETHGateway allow only deposits when `msg.sender == ownerGivingConsent`. Thus, only Alice will be able to insert the deposit leaf into Accumulator. Check that deposit is inserted into the Accumulator tree can be seen as an "Alice wanted to transfer funds to Bob" check. In addition, deposit leaf insertion contains more information and thus is a stronger constraint.
4. CONSENT_TYPEHASH violates the EIP712 specification. The type of a struct must be encoded as name || "(" || member_1 || "," || member_2 || "," || … || member_n ")" where each member is written as type || " " || name. The CONSENT_TYPEHASH doesn't have member names.

5. ConsentVerification.computeTypedStructHash violates the EIP712 specification. Each encoded member value must be exactly 32-byte long. abi.encodePacked will encode address _paramOne as 20-byte long.

Acknowledged:

Blockswap responded:

> We will be addressing these as part of the transportation layer upgrade we mentioned in the call

5.2 Finalization of Extension Can Stuck

Design | Low | Version 1 | Acknowledged

Once `deposit` or `pokeLatestBalance` functions get called, they need to be finalized on the destination domain, using `push` and `balanceIncrease` functions. However, this second call can be "stuck". The SRG system does not offer users any way to recover stuck funds. Some reasons can be user mistakes, e.g. user deposited to addresses nobody has control over on their destination domain. However, more serious are issues where origin domain functions do not perform strict enough verification of the parameters.

For example:

1. Consent verification relies on OZ library that enforces s-value of v,r,s tuple to be in the lower range of secp256k1n curve. In general, ecrecover percompile supports both high and low s-value signatures. During the deposit function call, only the v-values are constrained.
2. If batch staking rule constant is misconfigured, a check on deposit might be satisfied, while during the push the `amountExtended` value can be too small.

Acknowledged:

Blockswap responded:

> Users should always be able to verify their transactions before signing to ensure funds are not locked much like when interacting with a single blockchain and ensuring things like recipient are correct etc.

5.3 Missing Status by EndorserLifecycleStatusUpdated

Design | Low | Version 1 | Acknowledged

When the event `EndorserLifecycleStatusUpdated` gets emitted, it just indicates the status of a given endorser has been updated, without containing any further information about its current status. As a better practice, the current status of the endorser can be embodied in this event.

Acknowledged:

Blockswap responded:

> Indexers can read the state of the contract at the time of event emission. We will address this later

5.4 Pausing and Unpausing Emit Misleading Events

Design Low Version 1 Acknowledged

Some functions, which perform the state transitions of the Gateway contract do not check, whether the contract is already in the needed state:

- pauseGateway
- unpauseGateway
- pauseDomain
- unpauseDomain

As a result, repeated calls to these functions will have no effects on the state but will trigger an event. Some of these functions also do not check, whether the killswitch was triggered for this domain. Thus, `pauseGateway` after `triggerKillSwitch` can be called.

Acknowledged:

Blockswap has acknowledged the issue without fixing it, responding:

> We are ok with this. contract storage should always be checked for the source of truth

5.5 RPBS _isOnCurve Infinity Point

Correctness Low Version 1 Acknowledged

Most of the bn128 libraries consider that the infinity point belongs to the curve. The `_isOnCurve` function does not.

Reference:

- Ethereum
- Clearmatic bn256

Acknowledged:

Blockswap has acknowledged that their implementation does not follow the implementation of bn128. Blockswap responded:

> We will be addressing these as part of the transportation layer upgrade.

5.6 Repeated Pokes

Design Low Version 1 Acknowledged

For a single `pokeLatestBalance` action, the `balanceIncrease` finalization can be done multiple times. Only the first `balanceIncrease` will have an effect. All other calls will result in no balance update, however, the poke leaf will still be inserted in the Accumulator. In theory, an attacker can spam this transaction to deplete all the leaves in the Merkle tree.

Acknowledged:

Blockswap responded:

> We will add a spam mitigation strategy later.

5.7 `saveETHManager.init` Is Not Defined in `ISavETHManager`

Design | Low | Version 1 | Acknowledged

In `StakeHouseUniverseDestinationGateway.init` an ERC1967 Proxy gets deployed for the logic contract `_saveETHManagerLogic`. However, when encoding `_data` input field for the constructor of ERC1967, it assumes

1. `init` function is implemented for `savETHManager`, although not defined in `ISavETHManager`
2. `init` function of `savETHManager` has the exact same layout as `savETHDestinationReporter.init`, which might be invalid assumption.

These assumptions can be wrong.

Acknowledged:

Blockswap has acknowledged it, claiming that calling this function from external users is not encouraged.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	4

- Cost of bytesToHex Code Corrected
- ERC20 Token Decimals Code Corrected
- Inconsistent DepositEvent Amount Units Code Corrected
- batchDeposit Reverts if the Lengths of Input Arrays Match Code Corrected

Low-Severity Findings	5

- EIP165 Interface Implementation Check Is Not Fully Correct Code Corrected
- Sandwich Attack Without MEV Services Specification Changed
- Deployed Event of Gateway Is Not Informative Specification Changed
- whenGatewayAndPushNotKilled Specification Mistmatch Code Corrected
- dETH Dispensers Are Not a IsavETHDispenser Code Corrected

6.1 Cost of bytesToHex

Design Medium Version 1 Code Corrected

The `bytesToHex` is used to convert bytes to hex string. The only reason for this is to be compliant with the elliptic-js library. The `bytesToHex` is an extremely inefficient on-chain. Such conversions on-chain are strongly discouraged. They are computationally expensive and may lead to gas exhaustion. This does not pose a direct security risk, but it lowers the overall usability and scalability of the contract.

Code corrected:

The `RPBS-sol` package now operates with the bytes representation directly, without conversion to hex. The new function `encodePoint` is used in the assessed contracts instead of `encodePointHex` function.

6.2 ERC20 Token Decimals

Design Medium Version 1 Code Corrected

The `GatewayToken` contract inherits from `ERC20Upgradeable` fixed decimals of 18. In general, this might be not the same as the original token decimals. As a result, this might break UIs that will deal with such bridged tokens. Also, protocols that rely on decimals might have problems with compatibility.

Code corrected:

The `GatewayToken.decimals` now returns a variable that can be set in the `init` function.

6.3 Inconsistent DepositEvent Amount Units

Design Medium Version 1 Code Corrected

The `ERC20Gateway.deposit` function users provide the amount of tokens to extend as a `BaseInputParams.paramTwo` in wei. This param in wei will contribute to the deposit leaf hash. Same `paramTwo` in wei will be emitted in `DepositEvent`. On the destination domain recipient will need to specify the same paramTwo in `_depositMetadata.baseDepositInfo.paramTwo`.

However, this is not consistent with `savETHGateway`. In `savETHGateway.deposit` the user provides the KNOT that he wants to migrate. The `savETHRegistry.knotDETHBalanceInIndex` in gwei of this KNOT will contribute to the deposit leaf hash. But the `knotDETHBalanceInIndex` in **wei** will be emitted in Deposit tx. On push, `_depositMetadata.amount` in gwei will be converted to wei and the savETH on the destination domain will be minted. In summary, inconsistency is that units of event do not match the value from `_depositMetadata.amount` and the deposit leaf hash. Assuming that the endorsers will be querying the depositMetadata for the attestation, an extra conversion of deposit event values will be needed for one of these cases to compute the hash of the RPBS info.

Code corrected:

Blockswap has successfully resolved this inconsistency in various parts of the codebase (both dispensers and ingestors).

6.4 batchDeposit Reverts if the Lengths of Input Arrays Match

Correctness Medium Version 1 Code Corrected

To batch deposit a set of transactions to their corresponding domains given suitable input params for each deposit, the input arrays should have the same length. However, in the implementation:

```
uint256 numOfElements = _transactionSummaries.length;
if (numOfElements == 0) revert EmptyArray();
if (numOfElements == _domainIds.length) revert InconsistentArrayLengths();
if (numOfElements == _baseParams.length) revert InconsistentArrayLengths();
```

Which means a correctly formed input will not be handled. The functionality of all functions must be tested before deployment.

Code corrected:

Conditions were fixed. Now all 3 input arrays of the `batchDeposit` function required to be of the same length.

6.5 EIP165 Interface Implementation Check Is Not Fully Correct

Correctness Low Version 2 Code Corrected

According to eip-165, to detect that contract implements ERC-165, the source contract needs to make 2 calls. First call - to check that IERC165 is supported. Second call - to check that the invalid interface 0xffffffff is not supported.

However, the `Gateway._assertModuleAdheresToERC165Interface` function only performs the first call.

Source: https://eips.ethereum.org/EIPS/eip-165#how-to-detect-if-a-contract-implements-erc-165

Code corrected:

A check has been added to ensure that a module supporting IERC165 does not support an invalid interface.

6.6 Sandwich Attack Without MEV Services

Design Low Version 1 Specification Changed

The savETHRegistry has the following arbitrage opportunity, that bots can profit from: if a bot sees fees being to a KNOT, that belongs to an open index, a bot can sandwich this `mintDETHReserves` function call by 2 transactions: `isolateKnotFromOpenIndex` and `addKnotToOpenIndex`. This way, bot will get ownership of the KNOT that will have fees minted. As a result, the savETH rate won't increase. To execute this sandwich attack on the mainnet, the bot needs access to MEV service. However, on the destination domains, with the help of the poke function bots can steal fees from the open index without such services. The bot just needs to have a smart contract that sandwiches `balanceIncrease` the same way as `mintDETHReserves`. Since `balanceIncrease` in a `mintDETHReserves` on a destination domain without access control, this is possible.

Specification corrected:

Users, who put the KNOT into an open index voluntarily, accept the lower fees (potentially 0). In exchange, they get liquid assets that can be traded. Any actor is encouraged to perform balance updates (such as `mintDETHReserves`). The MEV sandwich described in this issue is seen as an arbitrage from that perspective. Hence, no fixes are needed.

6.7 `Deployed` Event of Gateway Is Not Informative

Design Low Version 1 Specification Changed

After initialization of a gateway through `__Gateway_init`, a mere event `Deployed` is emitted without any arguments. This event does not contain any parameters. Since it is used in the proxy initialization, this event increases the bytecode size of the deployed contract.

Specification corrected:

Blockswap responded:

After an event is emitted, all contract states can be read directly from a node saving deployment costs from not emitting data in events.

6.8 whenGatewayAndPushNotKilled Specification Mistmatch

Correctness Low Version 1 Code Corrected

As the name of the modifier suggests, not only should the gateway be operational, but also push for the foreign domain should not be killed. However, this modifier performs the following checks:

```
if (!domainMetadata.operational && isPushKilled[_domainId]) revert DomainOperationsArePausedOrKilled();
```

It means the scenarios, in which either domain is not operational or is killed, the modifier reverts. To make it comply with the specification, an OR operator instead of AND should be used.

Code corrected:

The `&&` operator was replaced by the `||` in the if condition above. Now the `whenGatewayAndPushNotKilled` will not revert only when the domain is operational and push is not killed on the domain.

6.9 dETH Dispensers Are Not a IsavETHDispenser

Design Low Version 1 Code Corrected

The `savETHGateway` uses `IsavETHDispenser` to communicate with `dETHDestinationDispenser` and `dETHOriginDispenser`. However those contracts do not implement the aforementioned interface. Change of the code can break compliance with the interface, that will not be reported by the compiler.

Code corrected:

`dETHDestinationDispenser` and `dETHOriginDispenser` contracts now implement the `IsavETHDispenser` interface.

6.10 RPBS Endorsement Expiry

Informational Version 1 Code Corrected

Function `verify` in `RPBSVerificationLibrary` fetches the first endorsement and verifies that it is not expired `require(signatureExpiry > block.timestamp, "Signature expired")`. Then, it iterates over the array of endorsements and checks that all of them have the same expiry time as the first endorsement. Though correct, it obliges the endorsers to agree on a common signature expiration. What is the expiration definition procedure and how do endorsers guarantee that this expiration will be the same for all signatures?

Code corrected:

In Version 3 of the code each Endorser need to specify its own expiry period.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Endorser Setting Change

Informational **Version 1** **Acknowledged**

First, the change of the Endorser status from `ACTIVE` to any other status will result in a revert of the pending `push` and `balanceIncrease` transactions if they were endorsed by the deactivated Endorser, while it was still active.

Second, the increase of the `numberOfEndorsementsRequired` threshold can cause similar in-flight message failure.

Acknowledged:

Blockswap responded:

> As long as the user can re-submit the transaction, then there is no issue.

7.2 Gas Optimisation

Informational **Version 1** **Acknowledged**

The codebase has several inefficiencies in terms of gas costs when deploying and executing smart contracts. Here, we report a list of non-exhaustive possible gas optimizations:

1. The modifier `Gateway.whenGatewayAndDomainNotPausedOrKilled` loads information about a specific domain Id to the storage and later reads the underlying information from storage, which is quite inefficient.
2. `ERC20Gateway._assertConsumptionAuthorised` reverts if `msg.sender != _baseParams.ownerGivingConsent`. However, the same check is performed in `_consume` function, before calling to `_assertConsumptionAuthorised`.
3. `Gateway.injectForeignDomainCheckpoint` checks that recovery is enabled for the dispenser and then calls into `Dispenser.injectForeignDomainCheckpoint` which performs the exact same check.
4. `Gateway.batchDeposit` iterates through every deposit in a list and calls to `Gateway.deposit` which solely calls into `Gateway._consume`. By calling `_consume` directly from `batchDeposit` gas can be saved.
5. Once `Gateway.batchDeposit` directly calls to `_consume`, the visibility of `Gateway.deposit` can also be changed to external.
6. `Gateway._dispense`, in case of recovery not being enabled, performs a multitude of checks (e.g. validating consent signature, asserting dispense being authorized, and finally dispensing in the dispenser module). After all of these gas expensive operations, it checks whether UTXO is already spent or not. In case of an attempt to use spent UTXO the revert will happen late in execution. Moving this check earlier would consume less gas in this scenario.

7. `Gateway._dispense`, in case of recovery being enabled, checks that a non-zero recovery Merkle root has been injected. Later, it calls to `dispenseViaRecovery` of the dispenser, which again assures that the recovery Merkle root is injected.
8. `Domain.dispense`, checks isRecoveryEnabled, however, Gateway will only call this function if it is not set.
9. `ConsentVerification.validateConsentSignature` is defined as public but never used internally. Its visibility can be changed to external to save gas.
10. Both functions `_onlyStakeHouseKnotThatHasNotRageQuit` and `_onlyValidStakeHouseKnot` in `savETHRegistryDestinationGateway` have the exact same functionality, only with different naming. It makes bytecode of the contract larger; hence, the deployment costs would be more expensive.
11. `Gateway.deposit` can increase userConsentNonce without the use of the safemath.
12. The fields in the Domain struct from the `IGateway` can be benefit from tight variable packing patterns.
13. The `balanceIncrease` function checks that accumulator of the the domain is not 0. The same check is performed in the `whenGatewayAndDomainNotPausedOrKilled` modifier of the same function.
14. Many functions of the savETHGateway query `saveETHRegistry` from the universe multiple times in the same function.

Acknowledged:

Blockswap will consider these optimizations later and apply changes when needed.

7.3 Hash Function Consideration

Informational **Version 1** **Acknowledged**

The SRG uses `sha256` function in multiple places. For example, in the Accumulator Merkle tree contract and for computation of certain constants like `DEPOSIT_TYPE_HASH`. While the original ETH2.0 Beacon staking contract uses the aforementioned hash function, the main reason for that was an assumption, that node software will be implemented in languages that do not have well-established analog for `keccak256` function.

In EVM `sha256` is a precompiled contract, and calling into it is more expensive than the opcode `keccak256`. `sha256` is twice more expensive than `keccak256` based on gas params. This cost difference also does not include overhead for creating memory layout for the `STATICCALL` to the precompiled contract.

Acknowledged:

Blockswap responded to the issue as:

> We will look into the optimisations and consider them as and when needed.

7.4 Recovery Merkle Tree Considerations

Informational **Version 1** **Acknowledged**

To enable recovery, a domain must be killed first. Once it will be killed, some state extension messages can stuck in-flight, meaning that the deposit tx happened on one domain, but the push for that deposit did not happen on the other domain. Consider the scenario, in which Alice has issued a deposit on mainnet chain to Bob on the Optimism chain. This extension is in-flight and still not pushed on Optimism. If then Optimism will be killed on Mainnet, this transaction must be considered during the Recovery Merkle Tree computation, even if Bob never pushed this on Optimism.

A similar case happens when Bob has issued a deposit from Optimism to Alice on mainnet chain. If then Optimism will be killed on Mainnet, this transaction must be considered during the Recovery Merkle Tree computation, if Alice did not push this on Mainnet before the recovery activation.

Acknowledged:

Blockswap mentioned in the response Doc, that this is by design.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Asset Flow per Time Limit

Note Version 1

The SRG system consists of multiple gateways on different chains. If a certain chain gateway is hacked, e.g. invalid state is inserted into Accumulator, the attacker can use connections between gateways to extend the bad state to the other chains. As a result, a system as a whole depends on the security of any of its components. Limits like "extension of X tokens per day is allowed" can limit the effect of bad state spread and give time for reactive measures.

8.2 Checked Properties

Note Version 1

Certain invariants of Blockswap SRG system were checked during this assessment or explicitly considered:

- GatewayToken needs to be deployed with the same decimal value as the original token.
- ERC20 Gateway is capable of handling simple ERC20 tokens. Any special tokens need to be wrapped. Consider this list: https://github.com/d-xo/weird-erc20
- Tree leaf of the Recovery Merkle tree should never be made of 64 bytes. Intermediate nodes might become claimable due to the collisions with the total length of leaf components matching the length of two concatenated hashes. See: https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3091
- To prevent cross-domain replays any leaf inserted into the accumulator needs to contain origin and destination chain ids, and origin and destination gateway addresses.
- Any kind of leaf that is inserted into Accumulator needs to have a unique prefix like TYPE_HASH to prevent collisions with other leaf types.
- The total supply of ERC20 tokens minted on some destination domain should equal the `totalExtended` value of the origin domain on this destination domain.
- The total supply of ERC20 tokens minted on some destination domain should equal the `totalExtended` value of this destination domain on the origin domain.
- Balances or KNOTs deposited from the origin domain can always be pushed on the destination domain, assuming both domains are not paused/killed.
- In case of ERC20: ERC20OriginDispenser must be the minter of GatewayToken. ERC20DestinationIngestor must be the burner of GatewayToken.
- In the case of ERC20: the sum of leaf amounts in the Recovery Merkle tree should always equal the origin's `totalExtended` value of the recovered domain.
- In the case of dETH: the set of leaf KNOTs in the Recovery Merkle tree should always be the same as KNOTs that belong to the destination index on the origin domain. No leaf should contain the same KNOT twice.

Such invariants need to be considered during future updates.

8.3 Handling ERC20 With Access Control Functionality

Note Version 1

Some ERC20 tokens can ban certain addresses from sending and receiving the tokens, e.g. USDC. Assume a scenario, where Alice deposits USDC tokens from Optimism back to the mainnet. Upon push, the receiver of USDC on the mainnet chain might be blacklisted. In this scenario, Alice's tokens will be locked and the push transaction will revert.

8.4 Restrictive Partially Blind Signatures (RPBS) Do Not Contain Unknown Blind Data

Note Version 1

The blinded message of RPBS contains only the Merkle tree proof. However, the common RPBS info contains depositLeafIndex, gatewayRoot and accumulatorCount. Endorsers even without knowing the actual data in a blinded message, in the current version of the code Endorsers can recompute the proof themself. This can potentially be used by Endorsers to censor the depositor. In addition, RPBS schema does not bring any benefit compared to more simple ECDSA signatures. Effectively no blinding is happening.

Assuming RPBS schema will be used in future versions of the systems, where the blinded data will be used, developers must be careful with what is being blinded.

If the entropy of the blinded data is not great, e.g. the deposit leaf index is blinded, the Endorsers still can randomly guess what data is being blinded. For example with the deposit leaf index - not many indexes can potentially be pending. A source of entropy can be included in the blinded message, which will make guessing impossible.

8.5 knotDETHBalanceInIndex Change Will Revert Pending Transaction

Note Version 1

The `txSummary` of the `savETHGateway` deposit should include the `knotDETHBalanceInIndex`. However, the change in this value can cause pending deposits to revert. Such increases can be caused by `mintDETHReserves` and `balanceIncrease` function calls. Please note that `balanceIncrease` is an unrestricted function. Since the events of balance increase are assumed not to be frequent, the likelihood of this problem is small.