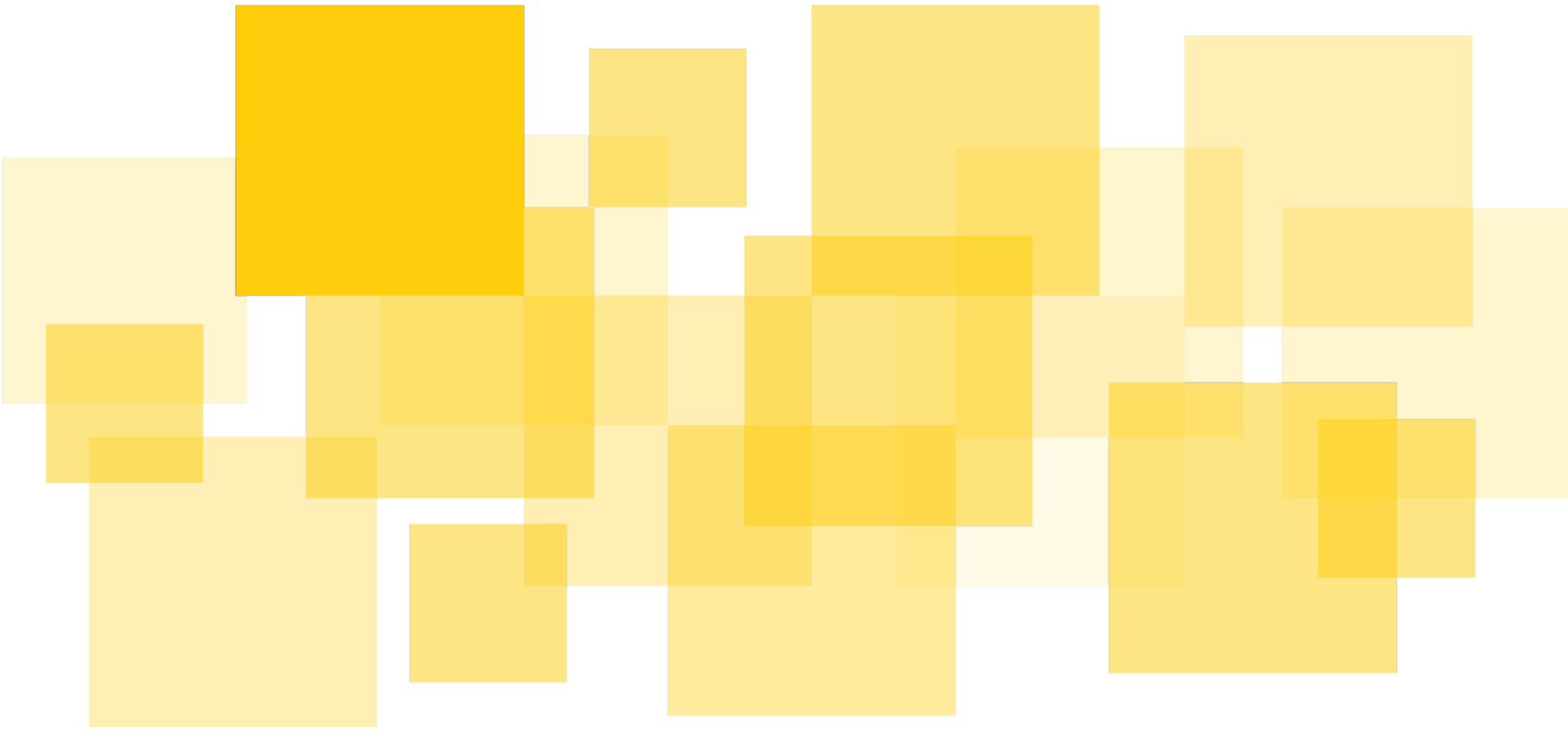
Audit Report

Blockswap Stakehouse

Delivered: 2022-03-27



Prepared for Blockswap by Runtime Verification, Inc.

runtime verification

Summary

Runtime Verification, Inc. has audited the smart contract source code for Blockswap's Stakehouse protocol. The review was conducted from 2022-01-11 to 2022-03-28.

Blockswap engaged Runtime Verification in checking the security of their Stakehouse protocol. Stakehouse is a programmable staking layer built on top of the Ethereum 2.0 Beacon Chain. Unlike standard liquid staking solutions, Stakehouse does not run validators for the users or allow stakes of less than 32 ETH. Instead, it allows a user to register a validator and mint derivative tokens corresponding to the 32 ETH stake, split into 24 dETH and 8 SLOT. dETH is a risk-free asset which can be traded freely and is redeemable 1:1 for ETH, with more dETH being minted as a validator earns inflation rewards. SLOT serves as collateral to protect dETH from slashing, as well as giving rights on the management of the validator. KNOTs, as validators are known within the protocol, are divided into Stakehouses, which serve as communities with an interest in helping their members run their validators effectively.

The issues which have been identified can be found in section Findings. A number of additional suggestions have also been made, and can be found in Informative findings. We have also verified a number of security properties, which can be found under Checked properties.

A protocol of this complexity always carries some amount of risk, but the Stakehouse team has made it clear during the audit that security is one of their priorities. They follow best practices, have provided us with extensive documentation of their design and codebase, and are concerned with understanding different scenarios and edge cases. The team has also been very responsive and open to discussion.

Scope

The audited smart contracts are:

- AccountManager
- BalanceReporter (abstract contract inherited by TransactionManager)
- SignatureValidator (abstract contract inherited by BalanceReporter)
- Streamer (abstract contract inherited by AccountManager)
- TransactionManager
- Banking (abstract contract inherited by StakeHouseUniverse)
- CollateralisedSlotManager (abstract contract inherited by SlotSettlementRegistry)
- SlotSettlementRegistry
- dETH
- sETH
- savETH
- savETHManager

- savETHRegistry
- DKGRegistry (abstract contract inherited by SafeBoxManager)
- SafeBoxManager
- ModuleGuards
- StakeHouseUUPSCoreModule
- UpgradeableBeacon
- StakeHouseAccessControls
- StakeHouseRegistry
- StakeHouseUniverse

The audit has focused on the above smart contracts, and has assumed correctness of the external contracts and libraries they make use of. The libraries are widely used and assumed secure and functionally correct.

The review focused mainly on the `Stakehouse-v2` private code repository, which is a Hardhat project with test scripts. The code was frozen for the initial design review at commit `8a72cd2949d9994fe29df90c0c821f5a907aa55c`. The code review was performed on commit `841bba6f3946e26f4c57199e9906b165e5cad3c7` (corresponding to PR #3), which included fixes for the issues raised during the design review. We document later fixes and code changes in this report, but fully reviewing later versions of the code to ensure that they do not introduce unexpected consequences is outside the scope of this audit.

Blockswap also provided access to the `common-interest-protocol` and `deposit-router` private code repositories, which contain off-chain portions of the protocol. Although reviewing off-chain and client-side code is not in the scope of this engagement, they have been used for reference in order to understand the design of the protocol and assumptions of the on-chain code, as well as to identify potential issues in the high-level design. Details of the implementation of the off-chain code, as well as the security of the cryptography underlying the Common Interest Protocol, are not in scope for this audit.

Assumptions

The audit is based on the following assumptions and trust model.

1. All users that have been assigned a role need to be trusted for as long as they hold that role (see Access Control). The intent is for all roles to be eventually revoked once there is confidence that the protocol is operating as intended and no more upgrades are needed.
2. Whatever source the deposit router fetches is never behind the data from the Beacon Chain.
3. The assumptions regarding the Common Interest Protocol (see Security Model) are satisfied.
4. KNOTs whose collateralized SLOT balance approaches zero are rage-quitted in a timely manner, ensuring that dETH is always fully backed by ETH.

5. dETH has enough liquidity and fluctuations of its value in the open market are not too large.

Note that the assumptions above roughly assume honesty and competence of privileged addresses. However, we will rely less on competence, and point out wherever possible how the contracts could better ensure that unintended mistakes cannot happen.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in Disclaimer, we have used the following approaches to make our audit as thorough as possible. First, we created an abstract high-level model that captures the intended behavior of the system, and based on this model identified the desired properties and invariants of the protocol. Second, we rigorously reasoned about the business logic of the protocol, validating security-critical properties to ensure the absence of loopholes in the business logic. Third, we reviewed the code for inconsistency between the logic and the implementation or vulnerability to known security issues and attack vectors. Fourth, we discussed the most catastrophic outcomes with the team, and reasoned backwards from their places in the code to ensure that they are not reachable in any unintended way. Finally, we regularly participated in meetings with the Stakehouse team to discuss our findings and together come up with mitigations and reasonable trade-offs.

This report describes, in Contract Description and Invariants, the **intended** behavior and invariants of the contracts under review, and then outlines issues we have found in Findings, both in the intended behavior and in the ways the code differs from it. We also point out lesser concerns, deviations from best practice and any other weaknesses we encounter in Informative findings. Finally, we also give an overview of the important security properties we proved during the course of the review in Checked properties.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very possible, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Contract Description and Invariants

This section describes the contracts at a high-level, and which invariants of its state we expect it to always respect at the end of a contract interaction.

The Stakehouse protocol has two types of modules: core modules and adaptors. Core modules provide the main functionality of the protocol, and are only called by adaptors and other core modules. Meanwhile, adaptors are externally-facing modules that can be called by users and external contracts, and provide an interface with the core modules.

savETHRegistry

When a KNOT joins a Stakehouse, 24 ETH out of the initial 32 ETH deposit are minted as dETH (the other 8 ETH are minted as SLOT; see SlotSettlementRegistry). Furthermore, any validator rewards earned by the KNOT in the Beacon Chain and reported via the BalanceReporter are likewise minted as dETH. dETH is a derivative token backed by the KNOT's balance in the Beacon Chain and redeemable 1:1 for ETH (after the Beacon Chain is merged with the Ethereum mainnet). dETH is also designed as a risk-free asset: unlike SLOT, it is not affected by balance reductions.

Invariant: dETH minted for a KNOT = 24 + amount of rewards earned by the KNOT.

Invariant: dETH supply never decreases.

The savETHRegistry contract manages dETH, as well as a related token called savETH, which represents a share that can be redeemed for dETH. When a user first adds their KNOT to a Stakehouse, this KNOT is put in an index owned by the user in the savETHRegistry. The same index can have multiple KNOTs, and entire indices can be transferred between users. While a KNOT is isolated in an index owned by a user, the index owner is entitled to all the rewards earned by the KNOT. However, any dETH minted for the KNOT remains locked in the index.

If an index owner wishes to withdraw the dETH earned by the KNOT, they have to move the KNOT from their index into a special index called the open index. When they do this, the dETH minted for the KNOT (24 + rewards) is added to the open index balance, and the owner is issued savETH corresponding to the amount of dETH added[1], using the current exchange rate. As the name suggests, KNOTs in the open index are owned collectively by the users who own a share of the open index in the form of savETH.

Invariant: savETH supply ≤ dETH in the open index.

[1] This assumes that finding A01 has been addressed.

Invariant: savETH supply is 0 if and only if dETH in the open index is 0.

The exchange rate of savETH to dETH is initially 1. A KNOT that earns rewards while in the open index increases the open index dETH balance, thus increasing the exchange rate. In this way, rewards earned by KNOTs in the open index are distributed proportionally among all savETH holders. savETH can be redeemed at any time for dETH in the open index at the current exchange rate. Circulating dETH can also be deposited in the open index in exchange for savETH, giving the user a share of the open index rewards.

Invariant: The savETH to dETH exchange rate never decreases.

Observations

The savETHRegistry includes an entry-exit rule that constrains the amount of dETH that can be deposited or withdrawn from the contract on a single operation. This amount must be between 0.001 and 960 dETH. The upper limit is an amount corresponding to the initial deposit for 40 KNOTs (24 dETH x 40 KNOTs = 960 dETH). The following functions deposit or withdraw dETH, and therefore revert if the amount of dETH falls outside this range:

- `withdraw` (redeem savETH for dETH)
- `deposit` (deposit dETH)
- `addKnotToOpenPoolAndWithdraw` (move KNOT from a user-owned index to the open index and immediately withdraw the locked dETH)
- `depositAndIsolateKnotIntoIndex` (use dETH to buy a KNOT from the open index, isolating it into a user-owned index)

Index owners can approve other addresses (likely smart contracts) as either KNOT spenders or index spenders.

- Rules for KNOT spenders:
    - A KNOT spender can transfer a KNOT owned by the user to another index (owned by the user or not).
    - KNOT spenders *cannot* transfer KNOTs to the open index (only the owner can do this).
    - Each KNOT can have at most one KNOT spender (besides the owner) at a time.
    - The KNOT spender is cleared when the KNOT is transferred to another index or to the open index, even if this is done by the owner.
    - The KNOT spender is also cleared when the KNOT is rage quit.
- Rules for index spenders:
    - An index spender can transfer one of the user's indices to a different user.
    - Each index can have at most one index spender (besides the owner) at a time.
    - The index spender is cleared when the index ownership is transferred, even if this is done by the owner.

SlotSettlementRegistry[2]

SLOT is the risk-bearing asset of the Stakehouse protocol. When a KNOT joins a Stakehouse, 8 ETH out of the initial 32 ETH deposit are minted as SLOT tokens (the other 24 ETH are minted as dETH; see savETHRegistry), which are managed by the SlotSettlementRegistry contract.

The 8 SLOT minted when a KNOT joins a Stakehouse are split into 4 collateralized SLOT and 4 free-floating SLOT:

- Collateralized SLOT are kept by the contract as collateral in the user's name against slashing events.
- Free-floating SLOT are given to the user in the form of sETH, at a fixed 3:1 exchange rate (12 sETH for 4 SLOT).

Invariant: Every KNOT has at most 4 collateralized SLOT at any given time.

Invariant: The amount of free-floating sETH in circulation for each house is equal to 12 × the number of KNOTs in the house (that have not rage-quit; see TransactionManager/BalanceReporter).

When a KNOT is penalized[3] in the Beacon Chain, the lost ETH can be reported via the BalanceReporter, causing the SlotSettlementRegistry to slash the same amount of collateralized SLOT. A user (either the KNOT's original owner or someone else) can then restore part or all of the slashed SLOT by depositing ETH. The ETH will be used to top up the KNOT's balance in the Beacon Chain, and the SlotSettlementRegistry assigns the same amount in collateralized SLOT within the KNOT's vault to the user.

Invariant: The sum of collateral for a KNOT across all SLOT owners is equal to 4 - the amount of SLOT that has been slashed (and not bought back).

The option of buying back slashed SLOT allows users other than the original owner to acquire a share of the ownership of the KNOT. SLOT owners are hit by slashing in the order that they first acquired SLOT for that KNOT. This means that SLOT owned by the original owner will be slashed first, and if this user's collateral runs out, then the other collateral owners will be slashed in order.

A user that owns a majority of the collateral for a KNOT (more than 2 SLOT) can request the validator signing key via the Common Interest Protocol. This can be used by the KNOT's current

[2] This description assumes that finding A02 has been addressed.
[3] See https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/beacon-chain.md#rewards-and-penalties

operator to recover a lost key, or by a different user to take over the validator if the original operator has not been able to keep it online.

If a KNOT loses its entire 4 SLOT collateral from slashing or is kicked from the Beacon Chain (due to cheating, for example), the KNOT is kicked from the Stakehouse.

Invariant: Every KNOT that has not been kicked has more than 0 SLOT as collateral.

In order to rage-quit the protocol (see TransactionManager/BalanceReporter), a KNOT must first be restored to full health. This means that any slashed SLOT must be bought, so that the KNOT has the full 4 SLOT collateral. A balance report is also required to ensure that the KNOT's balance in the protocol is consistent with the Beacon Chain. Additionally, the KNOT's collateral owners must own enough collateralized SLOT across the house to meet the *redemption threshold*, i.e. the following condition must hold:

$$\text{total collateralized SLOT} \times \text{exchange rate} \geq 4 \times \text{redemption rate}$$

- "total collateralized SLOT" is the total sum of collateralized SLOT owned by the KNOT's collateral owners across *all* KNOTs in the house.
- "exchange rate" is the current sETH exchange rate for the house (see sETH below).
- "redemption rate" is calculated by

  $$\text{dETH minted in the house} / \text{current SLOT balance in the house}$$

  where "current SLOT balance" is the amount of SLOT remaining in the house: (8 × number of KNOTs in the house) - total slashed.

Intuitively, if all KNOTs in the house are healthy, the redemption rate will equal the exchange rate, and therefore the 4 collateralized SLOT for the KNOT are enough to meet the redemption threshold and rage quit. If some other KNOT has been slashed, however, the collateral owners will need to buy some amount of slashed SLOT from one of these KNOTs before they can rage quit their own KNOT. This gives collateral owners an incentive to contribute towards restoring the health of the house.

sETH

sETH is an ERC20 token associated with a specific house, and closely related to SLOT. In practice, all SLOT in circulation is represented by sETH. sETH is always issued at an exchange rate of 3:1 (12 sETH for 4 SLOT) and must be returned at the same exchange rate when rage-quitting.

Separately from this fixed 3:1 exchange rate, a Stakehouse also exports a variable exchange rate given by

dETH minted in the house / SLOT minted in the house

If no rewards have been minted as dETH, this exchange rate equals the base exchange rate of 3:1 (24 dETH per KNOT / 8 SLOT per KNOT). As rewards are minted for KNOTs in the house, this exchange rate might increase above 3 (the exchange rate ignores slashing, therefore the denominator will always equal 8 × the number of KNOTs).

This exchange rate can be used to calculate a user's *active balance* in sETH. For example, a house with an average of 28 dETH minted per KNOT will have an exchange rate of 28 / 8 = 3.5. A user that is issued 12 sETH tokens (4 SLOT) from such a house will have an active balance of 3.5 × 4 = 13 sETH.

Since sETH is issued and redeemed at the base rate of 3:1, the variable exchange rate is mainly designed for external use, as an indicator of the success of the house that can be queried by the market and other systems built on top of the Stakehouse protocol.

AccountManager

The AccountManager contract handles the onboarding process of a KNOT into the protocol. The AccountManager keeps track of each KNOT's lifecycle status, indicating which part of the onboarding process it is in:

0. UNBEGUN: Initial state, indicating a KNOT that has not started the onboarding process.
1. INITIALS_REGISTERED: The owner of the KNOT has registered the KNOT's BLS key and the BLS signature for the initial deposit. This information is stored in an account array in the AccountManager.
2. DEPOSIT_COMPLETED: The initial 32 ETH deposit to the Eth2 Deposit Contract has been completed via the Stakehouse protocol. The block number where this deposit was made is stored in the AccountManager[4].
3. TOKENS_MINTED: The KNOT has been added to a Stakehouse (either an existing or newly-created one), the 24 dETH and 8 SLOT corresponding to the initial deposit have been minted, and the KNOT has been associated with a brand (see Brand Central). The KNOT's active balance in the AccountManager is initialized to 32 ETH.
4. EXITED: The KNOT has rage-quit the protocol.

A KNOT moves through the 5 lifecycle statuses in order (but it's possible to go from status 2 to 4 in one operation, see below). After reaching the TOKENS_MINTED status, the KNOT is considered active in the protocol and can receive balance reports (see BalanceReporter). It will remain at this stage until the owners decide to rage quit, changing the status to EXITED.

Invariant: The lifecycle status of a KNOT never decreases.

Invariant: A KNOT is a member of a Stakehouse if and only if its lifecycle status is TOKENS_MINTED or EXITED.

If the owner of the KNOT decides to rage-quit after making the initial deposit but before minting the derivative tokens, they can move through the TOKENS_MINTED and EXITED status in a single operation. In order to keep the accounting consistent, the KNOT is still added to a Stakehouse and has the derivative tokens minted before being immediately rage-quit. If for some reason the KNOT has been kicked from the Beacon Chain before joining a Stakehouse, they won't be able to mint the derivative tokens, and have to take this option.

All of the AccountManager's external non-view functions can only be called from another core module, specifically the TransactionManager/BalanceReporter. This ensures that the data in the contract is only updated after the appropriate checks have been performed and deposits have been made.

[4] After PR #16, the deposit count of the Eth2 Deposit Contract corresponding to this deposit is also stored in the TransactionManager.

TransactionManager/BalanceReporter

The TransactionManager[5] is the main externally-facing contract (adaptor) in the Stakehouse protocol, through which users can interact with the protocol's core modules. The contract provides external functions for each step of the onboarding process (see AccountManager):

- `registerValidatorInitials`: Submits the BLS key for a new KNOT and the BLS signature for the initial 32 ETH deposit, which are registered to the user's address. Lifecycle status in the AccountManager is set to INITIALS_REGISTERED.
- `registerValidator`: Submits 32 ETH to be deposited to the Beacon Chain via the Eth2 Deposit Contract. Also registers the validator's signing key (encrypted and signed by the Deposit Router) to be used by the Common Interest Protocol to rotate the KNOT. Lifecycle status is set to DEPOSIT_COMPLETED.
- `createStakehouse`: Creates a new Stakehouse (see StakeHouseRegistry) and adds the KNOT to it, minting the initial batch of tokens. Lifecycle status is set to TOKENS_MINTED.
- `joinStakehouse/joinStakehouseAndCreateBrand`: Adds the KNOT to an existing Stakehouse, minting the initial batch of tokens. The former associates the SLOT in the KNOT to an existing brand (see BrandCentral), while the latter creates a new brand. Lifecycle status is set to TOKENS_MINTED.
- `rageQuitPostDeposit`: Used to rage-quit between completing the deposit and minting the tokens (lifecycle status DEPOSIT_COMPLETED). Lifecycle status is set to EXITED.

These functions can be called either by the user themselves, or by an address authorized as a representative. Representatives can be managed using the `authorizeRepresentative` and `removeRepresentative` functions.

The TransactionManager contract also inherits the BalanceReporter abstract contract, which provides functionality for submitting balance reports and interacting with the savETHRegistry and SlotSettlementRegistry contracts:

- `balanceIncrease`: Reports a balance increase[6] of a KNOT in the Beacon Chain and mints corresponding dETH rewards in the Stakehouse.
- `voluntaryWithdrawal`: Reports that a KNOT is in the process of exiting the Beacon Chain and kicks it from the Stakehouse.
- `slash`: Reports a balance decrease of a KNOT in the Beacon Chain and slashes the corresponding collateral in the Stakehouse (potentially kicking the KNOT as well).
- `buySlashedSlot`: Buys SLOT that has been slashed, sending the ETH to the deposit queue (see below) and assigning the SLOT to the buyer.

[5] Renamed TransactionRouter in PR #5.
[6] Caused by inflation rewards, see https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/beacon-chain.md#rewards-and-penalties

- `slashAndBuySlot`: Equivalent to `slash` immediately followed by `buySlashedSlot`.
- `rageQuitKnot`: Rage-quits a KNOT, removing it from the protocol and allowing the owner to withdraw the balance from the Beacon Chain.
- `multiPartyRageQuit`: Same as `rageQuitKnot`, but for the case when multiple different parties own a share of the KNOT (collateralized SLOT, savETH and free-floating sETH).
- `topUpKnot`: Adds ETH to a KNOT's deposit queue (see below), allowing its contents to be flushed to the Beacon Chain.

Most of the TransactionManager/BalanceReporter functions require the user to submit a balance report validated and signed by the Deposit Router. The TransactionManager uses the information in this report to update the state of the Stakehouse, for example by minting rewards, slashing SLOT or kicking a KNOT. The report is also used to check the requirements of some operations; for example, to create or join a Stakehouse a KNOT must have an active balance of at least 32 ETH.

The following functions require a balance report: `createStakehouse`, `joinStakehouse/joinStakehouseAndCreateBrand`, `rageQuitPostDeposit/rageQuitKnot/multiPartyRageQuit`, `balanceIncrease`, `voluntaryWithdrawal` and `slash/slashAndBuySlot`.

Since all deposits to the Beacon Chain are done via the Eth2 Deposit Contract on the execution chain, which requires all deposits to be at least 1 ETH, any deposits made for buying slashed SLOT are accumulated in the KNOT's deposit queue until they add up to at least 1 ETH, at which point they are flushed to the Beacon Chain via the Deposit Contract. Users can also use the `topUpKnot` function to flush the queue by paying at least enough to add up to 1 ETH. These top-ups are ignored in the balance reported by the Deposit Router.

The amount of collateral that can be slashed from a KNOT is capped at 4 SLOT (see SlotSettlementRegistry), even if the KNOT loses more than this amount of ETH in the Beacon Chain. However, a KNOT that loses all 4 SLOT collateral is immediately kicked from the Stakehouse (see StakeHouseRegistry). Because of this, a KNOT that has been kicked might have a lower balance than is reflected in the Stakehouse (see B12), and therefore should be rage-quit to restore the health of the house.

A KNOT is also kicked from the Stakehouse if the validator has been forced to exit the Beacon Chain due to a kickable offense such as double-voting. A KNOT that is kicked for either reason cannot be moved to the open index, but balance reports (either for minting rewards or slashing SLOT) are still possible. This is to allow the KNOT to be rage-quit, which requires the KNOT state in the Stakehouse to be consistent with the Beacon Chain.

Rage-quitting requires burning all tokens minted for a KNOT:

- 24 dETH + rewards
- 4 collateralized SLOT

- 12 free-floating sETH (representing 4 SLOT)

If all of these tokens are owned by the same user, that user can rage-quit by themselves using the `rageQuitKnot` function. If they are split between different users, rage-quitting requires cooperation between all of them (note that while dETH and free-floating sETH can be freely traded, collateralized SLOT cannot). This is done via the `multiPartyRageQuit` function, which requires the signature of all stakeholders of the KNOT:

- The owner of the index where the KNOT is isolated (if the KNOT is in the open index, it must be isolated in a user-owned index to ensure ownership of all dETH minted for the KNOT).
- All owners of the KNOT's 4 collateralized SLOT (any slashed SLOT must be bought back before rage-quitting).
- A user who owns 12 free-floating sETH for the house.

Once all signatures are verified and the tokens are burned, an address designated by the stakeholders is enabled for withdrawing the entire balance of the KNOT from the Beacon Chain[7]. This withdrawal will be able to be performed after the merge.

Invariant: A KNOT is only enabled for withdrawal if its lifecycle status is EXITED.

[7] This assumes that finding A03 has been addressed.

savETHManager

Like the TransactionManager, the savETHManager is an adaptor (externally-facing contract) that can be used by users to interact with the protocol's core modules, in this case the savETHRegistry. The contract provides the following functions:

- `createIndex`: Creates an index owned by the user.
- `transferIndexOwnership`: Transfers the ownership of an index to a different user.
- `approveForIndexOwnershipTransfer`: Approves an external address to transfer ownership of the user's indices.
- `transferKnotToAnotherIndex`: Transfers a KNOT to a different index.
- `approveSpendingOfKnotInIndex`: Approves an external address to transfer the user's KNOT between indices.
- `addKnotToOpenPool`: Moves a KNOT from a user-owned index to the open index, receiving savETH shares corresponding to the amount of dETH associated with the KNOT.
- `isolateKnotFromOpenPool`: Isolates a KNOT in the open index to a user-owned index, after paying an amount of savETH corresponding to the KNOT's worth in dETH.
- `addKnotToOpenPoolAndWithdraw`: Same as `addKnotToOpenPool`, but receiving the KNOT's worth directly in dETH.
- `depositAndIsolateKnotIntoIndex`: Same as `isolateKnotFromOpenPool`, but paying for the KNOT in dETH.
- `withdraw`: Burns savETH shares to withdraw the corresponding amount of dETH from the open index.
- `deposit`: Deposits dETH in the open index in exchange for the corresponding amount of savETH shares.

StakeHouseUniverse/StakeHouseRegistry

The StakeHouseUniverse contract is a core module responsible for handling the creation and management of Stakehouses. It is called by the AccountManager to create a new Stakehouse, add a new member to an existing house, or remove a member from a Stakehouse by rage-quitting. It calls the banking contracts (savETHRegistry and SlotSettlementRegistry) to perform the appropriate operations, suchs as minting tokens when joining a house or burning tokens when rage-quitting.

The StakeHouseRegistry contract represents a Stakehouse, and one instance of this contract is deployed by the StakeHouseUniverse, via a proxy, for each new Stakehouse that is created. The StakeHouseRegistry stores the status of each member KNOT of the Stakehouse: active, kicked or rage-quit. This status is queried by other modules in order to validate operations. When creating a Stakehouse, the StakeHouseUniverse also calls the SlotSettlementRegistry to deploy the house's sETH token and BrandCentral to mint a new brand.

Invariant: If a KNOT has 0 collateralized SLOT, then it is marked as kicked in the StakeHouseRegistry.

Invariant: A KNOT that has been kicked can never become active again.

Invariant: A KNOT is marked as rage-quit if and only if its lifecycle status is EXITED.

The StakeHouseUniverse is also responsible, at initialization, for deploying the other core modules and adaptors, also via proxies: AccountManager, TransactionManager, SlotSettlementRegistry, savETHRegistry, savETHManager, and the Brand Central contracts.

Brand Central

Besides being associated with a Stakehouse, each KNOT is also associated with a *brand*. When a user creates a new Stakehouse, they must also create a new brand to associate with their KNOT, and when a user joins a Stakehouse, they can either join an existing brand or create a new one. Brands are not Stakehouse-exclusive, meaning that KNOTs from different Stakehouses can belong to the same brand.

A brand is identified by a three- to five-letter ticker. When a new brand is created, the creator receives a newly-minted NFT containing this ticker and the ID of the founding KNOT. This NFT represents ownership of the brand, and also allows the owner to set a custom brand description and image URI. When creating a brand, a user also chooses one of 5 building types to be associated with the brand.

When joining an existing brand, instead of a brand NFT the KNOT owner is able to claim a loot bag, which is an NFT formed of three components: a unique index representing a piece of land, the building type associated with the brand and one of 9 characters chosen by the user. The loot bag is a meta-composable NFT, which means that it can be kept “closed” as a single NFT or it can be “opened” to remove its individual components. When a component is removed, it is minted as a separate NFT and can be traded separately by the owner.

Additionally, for every KNOT registered in the protocol, any ETH address can, in a first-come-first-serve basis, claim an item from an open pool of items. This item is pseudo-randomly selected from a list of 8 normal items and 6 rare items (with a lower probability). This item is also minted to the claimant as an NFT.

Access Control

Access control is managed in the contract `StakeHouseAccessControls`. There exist two separate role hierarchies, where the first one is concerned with contract upgrades and managing the set of trusted Deposit Routers.

1. **Admin** can grant and revoke the roles Proxy Admin and Common Interest Manager to any user. Additionally, Admins can replace the entire `StakeHouseAccessControls` contract[8].
2. **Proxy Admin** can upgrade any smart contract of the Stakehouse, as long as the contract has not been *locked*. The locking mechanism is used to make a smart contract immutable once it has been established that it functions properly.
3. **Common Interest Manager** decides which Deposit Routers can be trusted.

The second hierarchy is concerned with managing the set of trusted core modules and adaptors.

1. **Core Module Admin** can grant and revoke the roles Core Module Manager and Core Module.
2. **Core Module Manager** can grant and revoke the role Core Module.
3. **Core Module** can call functions of core modules.

Once all core modules have been locked, the Stakehouse protocol becomes immutable. However, Core Module Managers can still add new adaptors by granting the Core Module, thus making it possible to add new features to the Stakehouse protocol.

Upon deployment, a *super admin* is assigned who has the roles Admin, Common Interest Manager, Core Module Admin, and Core Module Manager. However, besides the super admin, no other user is allowed to hold multiple roles at the same time to encourage role distribution.

Invariant: At any time, a user is assigned at most one role. The exception is the super admin user, who may have the roles Admin, Core Module Admin, Core Module Manager and Common Interest Manager at the same time.

Note that since Admins can replace the `StakeHouseAccessControls` contract, they have power over both role hierarchies[9].

All of the above roles are security critical. A malicious user that is granted any of these roles can cause great damage:

- Proxy Admins can upgrade any unlocked contract and thus radically change the behavior of the protocol.

[8] This feature has been removed from the code after the delivery of this report.
[9] See above.

- Common Interest Managers can decide to trust any Deposit Router, which have the power to sign arbitrary reports. A malicious user could use this to fabricate reports in order to mint dETH.
- The Core Module role allows a contract to call core modules directly instead of via an adaptor. This would allow a malicious user execute certain actions that would be impossible when only using adaptors.

Deposit Router

The description in this section assumes that Finding A20 has been addressed.

The goal of the deposit router is to provide trusted information about the Beacon Chain state of a KNOT to the TransactionManager/BalanceReporter.The deposit router is implemented as an off-chain web service that can fetch information about a KNOT from multiple sources and create a signed report that can then be passed to the TransactionManager, which verifies the signature of the report to ensure it comes from a trusted deposit router instance.

In general, anyone is able to run an instance of the deposit router, but whether a report signed by a specific instance is accepted by the TransactionManager depends on whether that instance has been added to the set of trusted instances. More precisely, the deposit router signs reports according to EIP-712 using the private key associated with an Ethereum address. The TransactionManager then recovers the Ethereum address from the signature and checks whether the address can be trusted. The set of trusted addresses can be managed by users who have the Committee Interest Manager role (see Access Control) by using the function `SignatureValidator::manageCommonInterestCommitteeMember()`.

The deposit router provides two HTTP endpoints:

1. `/bls-authentication`: This endpoint is used when registering a new KNOT and must be called after `TransactionManager::registerValidatorInitials()` and before `TransactionManager::registerValidator()`. It is expected that the following information is provided:
    - The contents of a `deposit_data.json` file, which is generated by the deposit-cli tool and contains the BLS public key of the validator that should be associated with the new KNOT
    - The contents of a `keystore.json` file which uses the EIP-2335 format and contains the encrypted BLS signing key of the validator
    - The password to decrypt the BLS signing key from the provided `keystore.json`
    - A signature `depositorSig` created by the Ethereum account that registered the validator in the Stakehouse

    The deposit router then performs the following actions:

    - Check that no deposits have been made yet to the Eth2 Deposit Contract for the provided BLS public key
    - Recover the Ethereum address of the `depositorSig` and check that it equals either the address that registered the validator in the Stakehouse, or is a valid representative
    - Extract the BLS signature contained in the `deposit_data.json` file. This signature will later be passed to the deposit functions of the Eth2 Deposit Contract. Check

that this BLS signature has been made over the following data[10]:
{
`pubkey:` the given BLS public key,
`withdrawal_credentials:` the address of the AccountManager (formatted according to the spec for withdrawal credentials),
`amount: 32 ETH`
}

- Decrypt the BLS signing key contained in `keystore.json` using the provided password, then encrypt it using a hybrid asymmetric-symmetric encryption scheme. For the asymmetric part, the public key *PK* from the CIP (see Common Interest Protocol) is used for encryption. For the symmetric part using AES, a fresh *AES encryptor key* is generated.

Finally, if all checks were successful, the deposit router returns the following data:

- The re-encrypted BLS signing key
- The AES encryptor key
- A signature over
  - the BLS public key,
  - the re-encrypted BLS signing key,
  - the AES encryptor key, and
  - a nonce, which is always zero,

  signed with the private key of this deposit router instance.

2. `/beacon-chain-authenticator`: This endpoint is used to verify the correctness of Beacon Chain reports. A Beacon Chain report contains information about the current state of the validator on the Beacon Chain that is associated with KNOT. In particular, it consists of the following information:
   - The withdrawal credentials of the validator.
   - Whether the validator has been slashed.
   - The adjusted active balance of the validator, which is computed from the active balance by subtracting all top ups except those that were made to buy slashed SLOT.
   - The effective balance.
   - The exit epoch, activation eligibility epoch, withdrawal epoch, and current checkpoint epoch[11].

[10] See https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/beacon-chain.md#depositmessage
[11] See https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/beacon-chain.md#validator

This endpoint expects that a single Beacon Chain report is provided and then performs the following actions:

- Check that the provided report matches the state of the most recent finalized epoch on the Beacon Chain by consulting three different data sources
- In order to compute the adjusted active balance the deposit router needs to know the sum of all validator top ups that have been made so far. (Here, “top up” refers to any deposit made via the Eth2 Deposit Contract to the validator, except the initial deposit of 32 ETH and any deposit made as part of buying slashed SLOT.) This information is retrieved from an Ethereum node (currently via a subgraph of The Graph).
- Check that the validator’s withdrawal credentials match the address of the AccountManager
- Check that the validator has been registered as a KNOT in the protocol (lifecycle status is either 2 or 3; see AccountManager)

Finally, if all checks were successful, the deposit router returns the following data:

- The verified Beacon Chain report
- A deadline, provided as a Eth1 block number, until which the report is valid
- A signature over
    - the verified Beacon Chain report,
    - the deadline, and
    - the current nonce retrieved from the TransactionManager,

  signed with the private key of this deposit router instance.

A source of complication is that the deposit router needs to combine data from three different sources: (1) Verifying the Beacon Chain report requires querying the Beacon Chain, (2) calculating the adjusted active balance requires querying The Graph, and (3) retrieving the current nonce requires querying the TransactionManager via a node. It is crucial that the data from all three sources is in sync. See findings A14 and A18.

Common Interest Protocol

Relevant contracts: `DKGRegistry`, `SafeBoxManager`. (However, note that most of the functionality is implemented off-chain.)

The description in this section assumes that Findings A04, A05, A20 and B05 have been addressed.

The Common Interest Protocol (CIP) allows anyone who holds more than 2 collateralised SLOT of a KNOT to request the KNOT's signing key. The intention is that if the original owner of a KNOT does not do a good job and the KNOT loses more than 2 collateralised SLOT, then someone else can buy the slashed SLOT, request the KNOT's signing key and then run the KNOT themselves (but see B03) or rage-quit the KNOT.

On a high level, this works as follows:

1. A public-key pair (PK, SK) is created such that PK is known to everyone while knowledge of SK is distributed among the n members of a special CIP committee using a verifiable secret sharing scheme. At least $t + 1$ committee members are needed in order to reconstruct SK, where $t = ceil(\frac{n}{2}) - 1$. This implies that any information encrypted with PK can only be decrypted if at least $t + 1$ committee members work together.
2. Whenever a KNOT is registered in a Stakehouse, its signing key is encrypted using PK and stored on the blockchain. (Encrypting the signing key is done by the Deposit Router, but see A20.)
3. If a user holds more than 2 collateralised SLOT of a KNOT, he can request the KNOT's signing key. Then, at least $t + 1$ committee members need to come together, fetch the encrypted signing key of the KNOT from the blockchain, decrypt it, and send it to the user.

The above functionality is implemented via three protocols:

- *Distributed Key Generation (DKG) Protocol.* Creates the public-key pair (PK, SK), where the shares of SK are distributed among the members of the initial committee such that no single entity knows SK in full. Note that this protocol is executed only a single time, right after deployment.
- *Committee Handover Protocol.* To make it possible for people to enter or leave the committee, the Committee Handover Protocol allows the shares of SK held by the current committee to be transferred to a new committee without revealing SK itself. Note that in order to reconstruct SK, the shares of a single committee have to be used, i.e., the shares from different committees are incompatible with each other and cannot be combined.
- *Threshold Decryption Protocol.* Allows the encrypted signing key of a KNOT to be handed over to a new owner. Note that the committee members partially decrypt the

signing key in such a way that SK does not need to be explicitly reconstructed. At no point is SK known to a single entity (assuming the majority of committee members is honest).

The bulk of these cryptographic protocols is implemented in an off-chain client application. The on-chain code mostly functions as a communication channel for the clients. Note that the audit is focused on on-chain code, which means that we have verified neither the correctness of the cryptographic protocols nor whether the off-chain client application correctly implements them. We did however define a Security Model based on our understanding of the CIP and also examined the on-chain code in this regard. (Note that verification of full functional correctness of the on-chain code is not possible since we do not know all the (implicit) assumptions made by the off-chaincode.)

Details

Let PK and SK denote the CIP key pair, where PK is publicly known while knowledge of SK is distributed among the members of the CIP committee. The members of the committee may change over time, but at any moment, there is only a single committee active. Let $N = 100$ denote the *maximum committee size*, and n_i the size of the ith committee, where $i = 0, 1, \ldots$.

Thus, n_0 denotes the size of the initial committee. Further, let $t_i = ceil(\frac{n_i}{2}) - 1$ denote the *threshold* value. An attacker may control up to t_i committee members without disrupting the correct operation of the CIP. Further, $t_i + 1$ of the shares held by the committee members are needed to reconstruct the shared secret key SK.

DKG Protocol

Upon deployment of the on-chain code, a set of users is chosen as the *initial candidates*. These initial candidates then execute the DKG Protocol in order to generate the shared key pair (PK, SK). Candidates who faithfully follow the DKG Protocol become *committee members*. On the other hand, any candidate deviating from the DKG Protocol is disqualified and will not become a committee member.

Committee Handover Protocol

Anyone who owns at least 1 collateralized SLOT can become a *guardian*. Any guardian can express their interest in becoming a committee member by submitting a *request for candidacy*. Once at least 100 requests for candidacy have been submitted, at most N of them are randomly selected to become *candidates*. Thus, since $N = 100$, the number of candidates is always exactly 100. Guardians who submit requests for candidacy must hold at least 1 collateralised SLOT in a unique address if they want to be able to become a candidate.

Once 100 candidates are selected, they perform the Committee Handover Protocol. Those candidates that are not disqualified during this process become the new committee members. A candidate is disqualified if he deviates from the protocol. Thus, if d denotes the number of disqualified candidates, then the committee size n is given by $n = 100 - d$.

Threshold Decryption Protocol

If a user holds more than 2 collateralized SLOT of a KNOT, he can create a request asking for the KNOT's signing key. Then, $t + 1$ committee members need to work together in order to simultaneously decrypt the signing key and re-encrypt it using the requester's public key.

Security Model

We first define what it means for the CIP to be secure. Then, we give a list of assumptions that, when satisfied, ensure that the CIP satisfies our definition of security. The goal is to make any assumption regarding the security of the CIP explicit so users can make an informed decision about whether to trust the CIP.

Definition: The CIP is *secure* if whenever an entity knows a KNOT's signing key, then (1) the entity is the KNOT's original owner, or (2) the entity holds (or has held in the past) more than 2 of the KNOT's collateralised SLOT.

Assumptions:

A1. A KNOT's signing key is not leaked outside the CIP (e.g., by the KNOT's original owner).
A2. The cryptographic protocols are implemented correctly.
A3. Candidates for the next committee own at least 1 collateralized SLOT when the Committee Handover Protocol starts.
A4. Only users who own more than 2 collateralized SLOT of a KNOT can trigger the Threshold Decryption Protocol for that KNOT.
A5. At least $n_i - t_i$ of the candidates for the ith committee are honest, where $i = 0, 1, 2,....$
A6. If $n_i - t_i$ members of the ith committee are initially honest then they remain so forever, where $i = 0, 1, 2,....$

Claim: If the given assumptions hold, then the CIP is secure.

We can intuitively argue in favor of this claim as follows. Assume Alice knows a KNOT's signing key.

- She cannot have received it from the original owner or anyone else who happens to know it. (Follows from A1)

- She cannot have bribed the current or a previous committee to decrypt the signing key for her. (A5 ensures that when a new committee is convened, the majority of its members are honest. A6 ensures that this remains the case forever)
- She cannot have decrypted the encrypted signing key that is stored on the blockchain herself.
  (A2 ensures that she cannot break the cryptography. A5 and A6 ensure that she cannot bribe the committee to give her the CIP secret key SK)
- She cannot have received the signing key via the Threshold Decryption Protocol without holding 2 collateralised SLOT for the KNOT. (Follows from A4)

Thus, there does not seem a way for Alice to have received the signing key illegitimately. She must either be the KNOT's original owner, or else must have held more than 2 collateralised SLOT in order to trigger the Threshold Decryption Protocol.

Note that the argument did not make use of assumption A3. This is because A3 is not needed to ensure the security of the CIP according to our definition. However, A3 *is* needed to make it more expensive to break assumption A5 by requiring the attacker to invest 1 collateralised SLOT (= 1 ETH) if he wants to control a candidate for the next committee.

We continue with a justification of the assumptions themselves:

- A1 allows us to concentrate on the CIP and ignore all the other ways in which the signing key could be leaked, which – though important – are not relevant in this context.
- A2 and A3 make assumptions about the correctness of off-chain functionality, which is outside the scope of this audit.
- A4 makes an assumption about the correctness of on-chain functionality, which we found to be justified during this audit.
- A5 simply restates a basic assumption made by the underlying cryptographic protocols.
- A6 together with A5 states that no committee is ever malicious. This assumption also applies to old committees that have already handed over their shares to the next committee. The reason this is important is that such old committees still maintain the ability to the reconstruct the shared secret SK using their shares.

Assumptions A5 and A6 cannot in general be guaranteed but are nonetheless crucial. Their importance comes from the fact that once an attacker controls $t + 1$ candidates or members for the DKG or Committee Handover Protocol, he can reconstruct the shared secret SK and then use SK to decrypt the signing key of any validator that is registered in a Stakehouse. While this does not give the attacker any immediate monetary gain (to withdraw a validator's stake he would need its withdrawal credentials, which are distinct from the signing key) it still poses a great risk to the staked funds: Having obtained the signing key of a validator, the attacker could start to sign contradicting messages, causing the validator to get penalized and eventually slashed.

While one has to trust that both A5 and A6 hold, for A5 it is at least possible to calculate concrete probabilities. For this reason we look a bit more closely at A5 in the remainder of this section.

First, note that A5 makes assumptions about both the candidates of the DKG Protocol and about the candidates of any invocation of the Committee Handover Protocol. However, the initial candidates for the DKG Protocol are picked manually before deployment, which is something we cannot formally model. Hence, we do not further analyze this part, and instead focus on the part of the assumption that is about the candidates for the Committee Handover Protocol. In particular, we restrict our analysis to the following assumption:

A5'. At least $n_i - t_i$ of candidates for the ith committee are honest, where $i = 1, 2, 3....$

We are interested in the probability of A5' being violated. This probability depends on two variables:
(1) The number of requests for candidacy submitted by the attacker (denoted by s_a), and (2) the number of requests for candidacy submitted by honest users (denoted by s_h).

The function $g(s_a, n_a, s_h, n_h)$ computes the probability that when s_a requests for candidacy have been submitted by the attacker and s_h by honest users, then exactly n_a of the selected candidates will be controlled by the attacker and n_h of the selected candidates will be honest. (Note that g does not take the maximum committee size N into account.) It is defined by

$$g(s_a, n_a, s_h, n_h) = \frac{\binom{s_a}{n_a} \cdot \binom{s_h}{n_h}}{\binom{s}{n}},$$

where $s = s_a + s_h$ and $n = n_a + n_h$.

The function $f(s_a, s)$ computes the probability that when s_a requests for candidacy have been submitted by the attacker and s requests for candidacy have been submitted in total, then the attacker controls at least $t + 1$ of the selected candidates. (Note that $s_a \leq s$, since s includes requests for candidacy by both the attacker and by honest users.) It is defined by

$$f(s_a, s) = \sum_{n_a=t+1}^{s_a} g(s_a, n_a, s - s_a, n - n_a),$$

where $n = min(s, N)$, $t = ceil(\frac{n}{2}) - 1$, and $N = 100$ is the maximum committee size. Intuitively, given s_a and s, then $f(s_a, s)$ denotes the probability that assumption A5' is violated.

In other words: Given s_a and s, what is the probability that among the n selected candidates, at least $t + 1$ are controlled by the attacker? For example, if the attacker submitted $s_a = 49$ requests for candidacy, and there are $s = 100$ requests for candidacy in total, then $f(49, 100) = 0$, i.e., there is no chance for the attacker to control $t + 1$ candidates. On the other hand, $f(50, 100) = 1$, i.e., i this case the attacker will *always* control at least $t + 1$ candidates.

The Committee Handover Protocol requires a minimum of 100 requests for candidacy, so in the following we assume $s \geq 100$. Further, the number of candidates is always 100, hence the committee size n satisfies $n \leq 100$ and the threshold value $t = ceil(\frac{n}{2}) - 1$ satisfies $t \leq 49$. Thus, in order to compromise the committee, the attacker needs to control at least $t + 1 = 50$ candidates, which is why we get $f(s_a, s) = 0$ for $s_a < 50$. This means the attacker needs to submit at least 50 requests for candidacy, and because each request requires the requester to hold at least 1 collateralized SLOT, this requires the attacker to invest at least 50 ETH.

The following tables show $f(s_a, s)$ for $s_a = 50$ and $s_a = 60$.

s	$f(50, s)$
100	100.000%
105	3.603%
110	0.161%
115	0.009%
120	0.001%
125	0.000%
130	0.000%
135	0.000%
140	0.000%

s	$f(60, s)$
100	100.000%
105	100.000%
110	100.000%
115	93.230%
120	59.647%
125	25.162%
130	8.042%
135	2.180%
140	0.538%

For $s_a = 50$, the probability that the attacker can compromise the committee drops significantly from 100% to 3.603% if the number of total requests for candidacy increases from 100 to 105. Thus, if the attacker has only enough resources to submit 50 requests for candidacy, then even if the total number of requests for candidacy is only slightly above 100, the probability of the attack succeeding is very low. Considering that the attacker may not be able to directly monetize the attack, he might not want to take the significant risk.

On the other hand, for $s_a = 60$, the chances of the attack succeeding are still well above 50% even if the total number of requests for candidacy is 120. If the attacker finds a way to make a

profit from compromising the committee, he might be willing to take the risk even if the number of requests for candidacy is well above the minimum of 100.

DKGRegistry/SafeBoxManager

The on-chain code of the CIP consists of the two contracts `DKGRegistry` and `SafeBoxManager`. Since `SafeBoxManager` derives from `DKGRegistry`, in the following we simply use `SafeBoxManager` to refer to both contracts.

These contracts do not contain a lot of logic, as they are mainly used as a communication channel for the off-chain client. In general, for any message the off-chain client may need to send, `SafeBoxManager` provides a function that can be called by an instance of the client. This function then simply emits an event (potentially performing some checks first) that can be consumed by other instances of the client.

The following functions are related to the DKG Protocol:

- `submitRound1Data()`: Emits an event holding the data of a client for round 1 of the protocol.
- `submitDKGComplaint()`: If a client has received invalid data in round 1 (or in round 3), send a complaint against the sender by emitting one or more events. Note that this function expects an array of complaints, which means a client must send all his complaints at once. See A06.
- `submitNoComplaint()`: Emits a distinct event in case a client does not need to send a complaint for round 1 (or 3). This way, other clients do not needlessly wait for complaints from this client.
- `submitRound3Data()`: Emits an event holding the data of a client for round 1 of the protocol.
- `revealCheatersPrivateKeyPart()`: If a client was caught cheating in round 3, reveal its share of the secret.

The following functions are related to general guardian management:

- `joinGuardians()`: Checks if the sender holds at least 1 collateralised SLOT. If this is the case, emits an event to notify the clients of the new guardian.
- `refuseGuardianDuties()`: Allows a guardian to relinquish his duties, which means he cannot be part of any future committee. Emits an event to notify clients. This action is irreversible and mostly intended to be used by initial committee members. This is because initial committee members are manually picked before deployment by Blockswap, and by relinquishing their guardian duties they want to show that they hand the CIP over to the community.

The following functions are related to the Committee Handover Protocol (reuses some DKG functions):

- `requestHandover()`: Used by guardians to submit a request for candidacy for the next committee. Checks if the sender holds at least 1 collateralised SLOT. If this is the case, emits an event to notify the clients.
- `submitHandoverData()`: Used by members of the old committee to send information for handing over the shares of the CIP secret key SK. Emits an event holding that information.
- `submitHandoverComplaint()`: Used by candidates for the new committee when they have received invalid data from an old committee member. Emits an event holding the necessary information.

The following functions are related to the Threshold Decryption Protocol:

- `applyForDecryption()`: Allows a user to request the signing key of a KNOT. Checks if the user holds more than 2 collateralised SLOT for the KNOT. If this is the case, emits an event to notify the clients of the request. Note that the request also specifies which off-chain communication channel should be used to send the KNOT's signing key to the user (e.g., Telegram, etc).

Findings

A01: Rewards from the open index flow to index owners[12]

[Severity: Medium | Difficulty: Low | Category: Fairness]

Inflation rewards from KNOTs in the open index are distributed pro-rata as dETH to all savETH holders. Since index owners have savETH locked in an index, they also receive a share of the rewards from the open index, in addition to exclusive rights to the rewards from the KNOTs in the index.

Additionally, since index owners get issued more savETH as the KNOTs in the index generate rewards, over time the proportion of shares held by index owners will tend to grow, which means that it's likely that most rewards from the open index will flow to index owners rather than free-floating savETH holders.

Scenario

1. Alice and Bob register a KNOT each, but Alice keeps her KNOT in an index owned by herself, while Bob moves his KNOT to the open index.
2. Initially, Alice has 24 savETH in the index and Bob has 24 savETH in his wallet, with a 1:1 exchange rate for dETH.
3. Alice reports a balance increase of 12 ETH in her KNOT, allowing her to mint 12 dETH as rewards. Since her KNOT is in her own index, Alice is issued an additional 12 savETH. She now has 36 savETH.
4. Bob reports a balance increase of 10 ETH in his KNOT, allowing him to mint 10 dETH as rewards. Since Alice has 36 savETH and Bob has 24 savETH, out of these 10 dETH, 6 dETH will be owned by Alice and 4 dETH by Bob.
5. Since Alice will keep earning savETH, her share of the rewards from Bob's KNOT will increase over time.

Recommendation

Fix design so that index owners do not receive a share of the rewards of the open index, or receive a share that does not increase the more rewards are earned in a user-owned index. Otherwise, document prominently so that users are fully aware of the potential losses involved with moving a KNOT to the open index.

[12] This issue was fixed before the code review, and the description of the savETHRegistry assumes that it has already been fixed.

Status

Addressed in PR #3. Design has been updated so that savETH is no longer issued for KNOTs kept in a user-owned index, only once the KNOT is moved to the open index. When a KNOT from the open index is isolated into a user-owned index, the corresponding amount of savETH is burned. This means that index owners no longer receive any shares of open index rewards (which are now kept track of separately from user-owned index rewards).

A02: Amount of SLOT required to rage quit changes with sETH exchange rate[13]

[Severity: High | Difficulty: Low | Category: Functional Correctness]

It is expected that the owners of a KNOT will burn 8 SLOT in total to rage quit from a Stakehouse (4 free-floating SLOT and 4 collateralised SLOT). However, the SlotSettlementRegistry contract keeps track of collateral in the form of sETH. The exchange rate of sETH to SLOT is initially 3:1 but can increase if KNOTs in the Stakehouse have their collateral slashed (slashing is socialized across the house).

The amount of collateralised SLOT needed for a KNOT to rage quit is calculated on-the-fly from the amount of sETH minted for the KNOT. Therefore, when the exchange rate changes, the amount of SLOT needed to rage quit also changes. This goes against the conceptual model of 8 SLOT corresponding to 8 ETH of the initial 32 ETH deposit.

Scenario

1. Alice and Bob add KNOTs A and B, respectively, to the same Stakehouse (assume initial exchange rate of 3 sETH / 1 SLOT).
    - Total Stakehouse collateral is 8 SLOT (4 from KNOT A, 4 from KNOT B).
    - Alice owns 12 sETH collateral, 12 sETH in her wallet.
    - Bob owns 12 sETH collateral, 12 sETH in his wallet.
    - Exchange rate is 24 sETH / 8 SLOT = 3 sETH / 1 SLOT (no change).
2. KNOT B gets slashed for 2 SLOT.
    - Total Stakehouse collateral is 6 SLOT (4 from KNOT A, 2 from KNOT B).
    - Alice owns 12 sETH collateral, 12 sETH in her wallet.
    - Bob owns 12 sETH collateral, 12 sETH in his wallet.
    - Exchange rate is 24 sETH / 6 SLOT = 4 sETH / 1 SLOT.
3. Alice rage quits KNOT A. The protocol calculates that she has to pay
    - 12 free-floating sETH (which uses a fixed exchange rate of 3:1, corresponding to 4 SLOT).
    - The 12 sETH that has been minted as collateral for KNOT A (which uses the current exchange rate of 4:1, corresponding to 3 SLOT).

Therefore, Alice is able to rage quit by paying only 7 SLOT.

[13] This issue was fixed before the code review, and the description of the SlotSettlementRegistry assumes that it has already been fixed.

Recommendation

Fix code so that the exchange rate is not used to calculate the amount of SLOT needed to rage quit.

Status

Addressed in PR #3. The code has been updated so that a user's collateral is recorded as SLOT rather than sETH. The previous mutable sETH exchange rate is no longer used to calculate the amount of collateral that must be paid in order to rage quit, which is now fixed at 4 SLOT. Instead, a redemption threshold that increases with slashing has been introduced (see SlotSettlementRegistry).

A03: User who submits a multi-party rage quit can unilaterally choose who receives the withdrawn funds

[Severity: High | Difficulty: Low | Category: Security]

When performing a multi-party rage quit, the first address in the array of collateralised SLOT owners is designated as the address enabled for withdrawing the KNOT's balance. The contract has no mechanism for verifying that this is the recipient intended by the other stakeholders of the KNOT; for instance, this information is not included in the signed messages that the stakeholders provide authorizing the rage quit.

As a consequence, the user who calls the `multiPartyRageQuit` function can unilaterally control which collateral owner (for example, themselves) is able to withdraw the funds, by simply choosing who is listed first in the array.

Scenario

1. Alice, Bob and Charlie all own collateralised SLOT for a KNOT, and decide to rage quit. Alice also has the KNOT isolated in an index, and owns 12 free-floating sETH.
2. Alice buys Bob's and Charlie's signatures authorizing the rage quit for an amount proportional to the amount of SLOT that they own, expecting that she will be able to withdraw the entire balance of the KNOT.
3. Charlie front-runs Alice's call of `multiPartyRageQuit`, submitting the same transaction with the order of the collateral owners switched so that he is first.
4. Charlie is enabled for withdrawing the funds instead of Alice.

Recommendation

Fix code. This attack can be prevented by including in the signed messages the recipient address that should be enabled for withdrawal. This also provides flexibility, as an address other than one of the collateral owners can be specified as the recipient (including a multisig wallet or other smart contract).

Status

Addressed in PR#5. The `multipartyRageQuit` function in the BalanceReporter now accepts an additional parameter `_ethRecipient`, which indicates the address that should be enabled to withdraw the balance. A field for the recipient has also been added to the structure of the signed messages, and the signature validation checks that the address in this field matches `_ethRecipient`.

A04: Same sETH address can be passed multiple times to `getCollateralizedSlotAccumulation`

[Severity: High | Difficulty: Low | Category: Functional Correctness, Security]

The `getCollateralizedSlotAccumulation` function in the SlotSettlementRegistry contract calculates, for a given user, how much total collateralized balance they have across multiple houses. The function receives a list of sETH tokens and adds up the collateralized balance that the user has in the corresponding houses.

If the same sETH token appears multiple times in this list, however, the balance for that house will be added multiple times to the result, artificially inflating the user's reported balance.

Since this function is used by the SafeBoxManager to determine if a user has enough SLOT to be a guardian for the common interest protocol, this attack can be used to bypass that threshold. This significantly reduces the investment necessary to become a guardian, which also makes 51% attacks (see B02) against the protocol more feasible.

Other applications built on top of the Stakehouse protocol might likewise use the function to validate users, with the same method being possible for bypassing validation.

Scenario

1. User buys 0.1 SLOT from a KNOT that has been slashed in Stakehouse A.
2. The user calls the SafeBoxManager's `joinGuardians` function, passing a portfolio that consists of 20 copies of the address for Stakehouse A's sETH token.
3. The function will iterate over the list, counting the user's 0.1 SLOT balance 20 times, for a total of 2 SLOT.
4. The function will conclude that the user has 2 SLOT of total collateral and allow them to become a guardian.

Recommendation

Fix code. Add a check to `getCollateralizedSlotAccumulation` that there are no duplicates in the sETH list.

Status

Addressed in PR #5. The `getCollateralizedSlotAccumulation` function now assumes that the sETH list is sorted in ascending order of addresses, and checks for every token in the list that

its address is strictly greater than the previous one. This guarantees that the function will revert if there are any duplicates.

A05: Candidates for CIP committee are selected on a first-come first-served basis

[Severity: High | Difficulty: High | Category: Security]

In the initial version of the CIP, the number of candidates for the Committee Handover Protocol is capped at 50, and in case the number of requests for candidacy exceeds this cap, candidates are chosen on a first-come first-served basis. This means that in order to compromise the committee and be able to reconstruct the shared secret SK, the attacker needs to control at most 25 candidates. (Since n is capped at 50, we get a maximal threshold value of $t = 24$, which means at most 24 dishonest candidates can be tolerated, and 25 shares are needed to reconstruct the secret.) This requires the attacker to control 25 guardians and make them submit requests for candidacy. Then, the only remaining task for the attacker is to ensure that the 25 requests for candidacy submitted by his guardians are among the first 50. We assume that the attacker can accomplish this with relatively high probability, as we do not have enough information about the process to prove otherwise. Once the attacker controls 25 candidates and the Committee Handover Protocol has been executed, the attacker can decrypt the signing key of any KNOT he wishes, potentially causing great damage by intentionally causing slashable offenses.

Scenario

1. The attacker creates 25 Ethereum accounts and buys 1 collateralised SLOT for each of them. The total investment is 25 ETH. Note that buying collateralised SLOT may take some time because validators need to be penalized before you can buy their SLOT.
2. Once the attacker controls 25 accounts that each own 1 collateralised SLOT, he can simply register them as guardians.
3. Next, the attacker makes his 25 guardians submit requests for candidacy, ensuring that these requests will be among the first 50 submitted. Thus, the attacker's 25 guardians are now among the candidates that will execute the Committee Handover Protocol.
4. If none of the attacker's candidates is disqualified during the Committee Handover Protocol, they all become committee members. With 25 committee members under the attacker's control, he can now reconstruct the shared secret SK.
5. Having reconstructed SK, the attacker can now decrypt the signing key of any KNOT he wishes.

Recommendation

Instead of selecting candidates on a first-come first-served basis, candidates should be selected randomly from all requests for candidacy. This removes the potential unfairness and abuse that

comes from the first-come first-served approach, and makes it possible to calculate the probability with which the above attack would succeed. See Security Model.

Status

Addressed. The design has been updated to select candidates randomly from all requests for candidacy. Furthermore, the maximum committee size has been increased to 100, requiring a higher investment from the attacker.

A06: CIP clients cannot inform each other that they have sent all complaints

[Severity: Medium | Difficulty: N/A | Category: Functional Correctness]

The DKG and Committee Handover Protocol are divided into separate communication rounds. In some of these rounds, participating guardians may need to send complaints about other participants that do not follow the protocol. Execution must only proceed to the next round if all participants have sent all of their complaints. Complaints from one participant are communicated to the other participants via on-chain events sent using the `SafeBoxManager` contract, where each complaint is sent using a separate event.

In the initial version of the CIP, there was no explicit way for participants to notify each other once they had sent all their complaints. Thus, participants were forced to wait a predetermined amount of time for complaint events, and had to assume that no more complaints were coming after the time limit. However, this opens the possibility for missed complaints, either due to a temporary network outage or a malicious actor.

Recommendation

Instead of sending complaints individually, transmit all complaints of a participant in a single event, and require that all participants send such an event. In the case where a participant does not have a complaint, he can simply send an empty list. This way, once a participant has received this new event from all other participants, he knows that no more complaints are coming.

Status

Addressed in PR #3. The implementation of `SafeBoxManager` has been changed in such a way that complaints are still sent in individual events, but these events are always sent in a batch such that each batch contains all complaint events from a particular participant. Thus, if participant A sees some complaint events from participant B, then A knows that these are all the complaint events from B. If a participant has no complaints, he can send a distinct no-complaint event.

A07: `DKGRegistry::threshold` used incorrectly

[Severity: Medium | Difficulty: N/A | Category: Functional Correctness]

In contract `DKGRegistry`, the state variable `threshold` stores the threshold value $t = ceil(\frac{n}{2}) - 1$ that is used for the DKG Protocol. However, `threshold` was also erroneously used in the context of the Committee Handover Protocol to perform some checks in the function `DKGRegistry::_submitHandoverReferencePoints()`.

Recommendation

Remove the wrong check involving `threshold`. `DKGRegistry` cannot perform this check since it does not have sufficient information.

Status

Addressed in PR #3.

A08: Isolating a KNOT before reporting balance increase

[Severity: Low | Difficulty: Low | Category: Arbitrage]

If a KNOT in the savETHRegistry's open index has a balance increase, a user can isolate the KNOT into an index before submitting a report to mint rewards. In this case, that user will receive all of the KNOT's rewards, which would otherwise be distributed among all savETH holders. The user can then return the KNOT to the open index.

As long as the profits are higher than the gas fees, a self-interested user has a financial incentive to do this instead of submitting the report while the KNOT is in the open index. If all users do this, however, no rewards are minted in the open index and the value of savETH never increases, which could demotivate users from holding savETH.

Scenario

1. KNOT in the open index has a balance increase from X to X+Y (as yet unreported).
2. User isolates the KNOT for X dETH.
3. User reports the balance increase and mints Y dETH.
4. User returns the KNOT to the open index, receiving X+Y dETH.

If the user had reported the balance increase while the KNOT was still in the open index, they would have received only a fraction of the Y dETH profit. Note that this scenario requires the user to have X dETH to isolate the KNOT, but this amount is returned immediately after. This also opens the possibility for the user to borrow the X dETH via a flash loan.

Recommendation

Fix design or monitor. The following are some options of mitigation:

- Require a balance report before isolating a KNOT, so that this scenario is no longer possible.
- Prevent rewards from being minted immediately after isolating a KNOT. This would not prevent the scenario completely, but would require funds to be locked for longer and prevent the option of a flash loan, thus increasing the difficulty.
- Offer users incentives for keeping KNOTs in the open index.

If the current design is maintained, the system should be monitored to see if users engage in this behavior, and how common it is. If it becomes too prevalent, a design fix can be reconsidered.

Status

Acknowledged. At protocol launch, the client wishes to drive engagement and would prefer to avoid introducing friction by requiring a balance report before isolating a KNOT, but can upgrade the savETHRegistry later if they observe this behavior happening too frequently.

A09: Inconsistent treatment of `isCollateralisedOwner` on rage quit

[Severity: Low | Difficulty: N/A | Category: Functional Correctness]

The `_reduceCollateralisedBalanceForAllHolders` function in the SlotSettlementRegistry contract has an `if` statement separating the cases when there is a single collateral owner or multiple owners. The branch for multiple owners sets `isCollateralisedOwner` to false, but the branch for a single owner does not.

Scenario

If a user rage-quits by themselves, they will still be marked as a collateralised owner of the KNOT. This will not lead to any undesired behavior, since `isCollateralisedOwner` is only checked when rage-quitting or buying SLOT, both of which are not possible after the KNOT has rage-quit. Nevertheless, it would lead to inconsistent state in the SlotSettlementRegistry.

Recommendation

Fix code by setting the flag to false in the single-owner branch as well.

Status

Addressed in PR #5. `_reduceCollateralisedBalanceForAllHolders` now sets `isCollateralisedOwner` to false in the single-owner case as well.

A10: Rage-quit doesn’t check if all collateral owners are current

[Severity: Low | Difficulty: Medium | Category: Input Validation]

`SlotSettlementRegistry::rageQuitKnotOnBehalfOf` doesn't check if each collateralised SLOT owner currently owns SLOT for the KNOT. It does check, in the multi-party case, that `isCollateralisedOwner` is set to true, but this doesn't prevent a user who used to own SLOT in the KNOT but doesn't anymore to be added to the list.

Even if the user does not own collateral for the KNOT, their collateralised SLOT in other KNOTs will be counted towards meeting the redemption threshold. This could allow users to rage-quit without buying back slashed SLOT in other KNOTs, as the redemption threshold is meant to enforce.

Scenario

Suppose that after bringing their KNOT back to full health, the collateral owners of the KNOT are still 0.1 SLOT short of the redemption threshold. Instead of buying slashed SLOT to meet the threshold, they ask another user who used to own collateral for the KNOT in the past to rage quit with them. As long as this user has at least 0.1 collateralized SLOT in other KNOTs, they will be able to meet the redemption threshold without buying any additional SLOT.

Recommendation

Fix code to check if all collateral owners passed to the `rageQuitKnotOnBehalfOf` function currently own collateral in the KNOT.

Status

Addressed in PR #7. In the multiple-owner case, the `_reduceCollateralisedBalanceForAllHolders` function (which is called by `rageQuitKnotOnBehalfOf`) now checks for each owner that their SLOT balance in the KNOT is greater than 0. This is sufficient since the check is not necessary in the single-owner case, as the rage-quit will already revert if the owner has fewer than 4 collateralized SLOT.

A11: Rounding errors in asset conversions

[Severity: Low | Difficulty: High | Category: Arithmetic Precision]

Conversions between savETH and dETH performed in the savETHRegistry are subject to rounding errors that scale with the amount being converted (see C05 for a detailed analysis):

- Converting x dETH to savETH can give the user approximately $x \cdot 10^{-18}$ *more* savETH than they should receive.
- Converting x savETH to dETH can give the user approximately $x \cdot 10^{-18}$ *less* dETH than they should receive.

The bounds on these errors are very small relative to the amount being converted (corresponding to the magnitude of 1 wei in relation to 1 ether), and any potential profits made by a user are likely to be far outweighed by gas fees. Therefore, this is unlikely to be used as an attack. Nevertheless, a visible difference in the amount received from the amount expected can cause confusion and impair the user experience.

Additionally, it is unclear if the arithmetic errors deriving from these conversions might affect other calculations in the protocol or even violate invariants. In particular, the calculation of the error bounds themselves depends on the assumption that the total supply of savETH is always less than the amount of dETH in the open index. If the rounding errors can cause this assumption to be violated, it's possible that the profitability of the attack is increased.

Recommendation

Fix code. Change the calculation of the dETH/savETH conversion so that all multiplications (which don't introduce error terms) are performed first and a single division is performed at the end. This minimizes the error so that the computed result is never greater and is at most 1 wei less than the expected result. This error is constant and independent of the amount being converted.

Status

Addressed in PR #10. The `savETHToDETHExchangeRate` function in the savETHRegistry has been replaced by two functions, `dETHToSavETH` and `savETHTODETH`, which perform their respective conversions using the order of operations that minimizes the error.

A12: Access control: Inconsistent checks for ensuring that users are assigned at most one role

[Severity: Medium | Difficulty: Low | Category: Security]

An invariant of the `StakeHouseAccessControls` contract is that, with the exception of the super admin, a user can only have a single role (see Access Control). However, this invariant is violated by the functions `addAdmin()`, `addProxyAdmin()`, `addCoreModule()`, `addCoreModuleManager()`, and `addCoreModuleAdmin()` from the `StakeHouseAccessControls` contract, which do not check whether the given user already has the Common Interest Manager role.

Scenario

1. Assign the Common Interest Manager role to `user` using `accessControl.addCommonInterestManager(user)`
2. Assign the Admin role to `user` using `accessControl.addAdmin(user)`
3. Now `user` has two roles, Common Interest Manager and Admin, which violates the above mentioned invariant.

Recommendation

Add checks to `addAdmin()`, `addProxyAdmin()`, `addCoreModule()`, `addCoreModuleManager()`, and `addCoreModuleAdmin()` that ensure that the specified user does not already have the Common Interest Manager role.

Status

Addressed in PR #5.

A13: Access control: Checks to ensure that users are assigned at most one role can be circumvented

[Severity: Medium | Difficulty: Low | Category: Security]

An invariant of the `StakeHouseAccessControls` contract is that, with the exception of the super admin, a user can only have a single role (see Access Control). Checks that should uphold this invariant are part of the functions `addAdmin()`, `addProxyAdmin()`, `addCoreModule()`, `addCoreModuleManager()`, `addCoreModuleAdmin()` and `addCommonInterestManager()`, which are responsible for assigning roles to users.

However, because `StakeHouseAccessControls` derives from OpenZeppelin's `AccessControl` contract, it inherits the function `grantRole()` which can also be used to assign roles to users and which does not perform any checks to prevent a user from having multiple roles. Thus, the above mentioned invariant can be easily broken by using `grantRole()`.

Scenario

1. Assign the Proxy Admin role to `user` using `accessControl.grantRole(PROXY_ADMIN_ROLE, user)`
2. Assign the Core Module Admin role to `user` using `accessControl.grantRole(CORE_MODULE_ADMIN_ROLE, user)`
3. Now `user` has two roles, Proxy Admin and Core Module Admin, which violates the above mentioned invariant.

Recommendation

Override the `grantRole()` function in `StakeHouseAccessControls` to either do nothing at all or to perform all the required checks.

Status

Addressed in PR #5. `grantRole()` and `revokeRole()` have been overridden to always revert.

A14: Adjusted active balance may be calculated incorrectly by the deposit router

[Severity: High | Difficulty: Medium | Category: Functional Correctness]

The deposit router calculates the adjusted active balance of a validator by subtracting the sum of all top ups from its active balance. (Here, "top up" refers to any deposit made via the Eth2 Deposit Contract to the validator, except the initial deposit of 32 ETH and any deposit made as part of buying slashed SLOT. See Deposit Router.) Since the top ups are fetched from the Eth1 chain while the active balance is fetched from the Beacon Chain they may temporarily be out of sync, meaning the sum of top ups may include deposits that may not have been processed by the Beacon Chain yet and thus have not been added to the active balance.

This can cause the Stakehouse to mint dETH for a KNOT even though the KNOT has not earned any rewards, which breaks the invariant that dETH is only minted when registering a new KNOT or when earning rewards (see savETHRegistry).

Scenario

The following table demonstrates that under certain circumstances, the adjusted active balance is computed incorrectly by the deposit router. We consider a single, freshly registered KNOT, and show its active balance (stored on the Eth2 Beacon Chain), its total amount of top ups (stored on the Eth1 chain), the amount of slashed SLOT that has been bought for the KNOT (stored on the Eth1 chain), and the KNOT's adjusted active balance, which is computed by subtracting the top ups from the active balance.

	Active balance (Eth2)	**Top ups (Eth1)**	**Bought SLOT (Eth1)**	**Adjusted active balance**	
	32	0	0	32	
1)	*Top up 1 ETH*				
	32	1	0	31	
2)	*Buy 1 slashed SLOT*				
	32	1	1	31	
3)	*Eth2 update*				
	34	1	1	33	←1 dETH minted

We begin with a fresh KNOT that has an active balance of 32 ETH on the Beacon Chain, and no top ups have been made and no slashed SLOT has been bought. Thus, the adjusted active balance is the same as the active balance.

In step 1), we deposit 1 ETH using the Eth2 Deposit Contract. However, we assume that the deposit is not processed by the Beacon Chain until after step 3), which is not unreasonable because the Beacon Chain only considers deposits that are at least 2048 blocks old. For our example this means that when the deposit router calculates the adjusted active balance, it will likely see that a top up has been made, but this top up is not reflected on the Beacon Chain yet, resulting in an adjusted active balance of 31 ETH, even though the correct active balance would be 32 ETH. If we report the reduced active balance to the Stakehouse, it is as if the KNOT had been penalized by 1 ETH, which is equivalent to saying the KNOT has been slashed by 1 SLOT.

In step 2) we use this glitch and buy that slashed SLOT. This results in a deposit to the Eth2 Deposit contract, but since we are buying slashed SLOT, this is not counted towards the total amount of top ups. Again, we assume that the Beacon Chain does not process this deposit until step 3).

In step 3), the Beacon Chain finally catches up and processes both the deposit originating from the top up in step 1) and from buying slashed SLOT in step 2), resulting in an active balance of 34 ETH and an adjusted active balance of 33 ETH. If we report this new adjusted active balance to the Stakehouse, it sees an increase of the highest seen adjusted active balance from previously 32 ETH to now 33 ETH. The Stakehouse will assume this increase came from rewards on the Beacon Chain and will mint 1 dETH. However, in reality, no rewards have been minted, which breaks the invariant that dETH is only minted when registering a new KNOT or when earning rewards.

Recommendation

Fix the deposit router such that it only considers deposits that have been processed by the Beacon Chain. Note that it is not enough to simply wait 2048 blocks, as the Beacon Chain itself may need some time to process a deposit.

Status

Addressed. The proposed fix is to make the deposit router query the *deposit index* of the latest top up to the given KNOT. The deposit index of a deposit is the total number of deposits made to the Eth2 Deposit Contract at the time of the top up. So if 1000 deposits have been made in total to the Deposit Contract before making a top up, then the deposit index of the top up is 1001. Once the deposit router knows the deposit index, it queries the *global deposit counter* from the Beacon Chain, which is a counter that is incremented whenever a deposit is processed by the

Beacon Chain. Then, the deposit router requires that the condition $global\ deposit\ counter \geq deposit\ index$ holds. If this condition holds, then the deposit has been processed by the Beacon Chain (deposits will always be processed in order[14]). If the condition does not hold, the deposit router aborts.

[14] See https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/beacon-chain.md#deposits

A15: Decrease of slashing collateral computed incorrectly

[Severity: High | Difficulty: N/A | Category: Functional Correctness]

When a KNOT gets slashed, its collateralised SLOT balance is decreased. However, the slashing amount must not exceed the amount of collateralised SLOT that the KNOT has left. This means that if a KNOT has 1 collateralised SLOT left and is slashed by 2 SLOT, then the SLOT balance is only decreased by 1 instead of 2.

The following code is supposed to ensure that `decrease` is never larger than `remainingCollateral`:

```
// make sure decrease does not exceed slashing collateral
uint256 currentSlashedAmount =
    universe.slotRegistry().currentSlashedAmountForKnot(_memberId); // wei amount;
uint256 remainingCollateral = 4 ether - currentSlashedAmount; // wei amount
uint256 currentSlashedPlusDecrease = decrease + currentSlashedAmount; // wei amount
if (currentSlashedPlusDecrease >= remainingCollateral) {
        decrease = remainingCollateral;
        isKickRequired = true;
}
```

However, this is wrong. For example, if `decrease = 1` and `remainingCollateral = 2`, then `decrease` is increased to 2, even though it should have remained at 1.

Recommendation

The above code can be fixed as follows:

```
// make sure decrease does not exceed slashing collateral
uint256 currentSlashedAmount =
    universe.slotRegistry().currentSlashedAmountForKnot(_memberId); // wei amount;
uint256 remainingCollateral = 4 ether - currentSlashedAmount; // wei amount
if (decrease >= remainingCollateral) {
        decrease = remainingCollateral;
        isKickRequired = true;
}
```

Status

Addressed in PR #12.

A16: Amount of slashed SLOT in Stakehouse may be less than the amount of incurred penalties on Beacon Chain

[Severity: High | Difficulty: Low | Category: Functional Correctness]

In certain scenarios, the penalties on the Beacon Chain are not correctly reflected in the Stakehouse. In particular, the amount of slashed SLOT of a KNOT in the Stakehouse may be less than the incurred penalties on the Beacon Chain.

For example, it is possible that a KNOT has an active balance of 31 ETH on the Beacon Chain (which means the KNOT has been penalized by 1 ETH) while having zero slashed SLOT in the Stakehouse. Under normal circumstances, one would buy back a KNOT's slashed SLOT to get its active balance back to 32 ETH, but in this case there is no slashed SLOT to buy. Thus, the only thing to do is to wait until the KNOT reaches an active balance of 32 ETH on its own by earning rewards. The rewards earned during this time cannot be reported to the Stakehouse, which means savETH holders will lose out on dETH rewards. Once a KNOT has earned enough rewards to recover any unreportable penalties on the Beacon Chain, the Beacon Chain and the Stakehouse are in sync again.

The same misbehavior may also lead to a KNOT not being kicked from a Stakehouse even though it should.

Scenario

Consider the following scenario which considers a single KNOT.

	Active balance (Beacon Chain)	**SLOT (Stakehouse)**
	32 ETH	8 SLOT
1)	*1 ETH penalty gets reported*	
	31 ETH	7 SLOT
2)	*Another 1 ETH penalty, but it's not reported*	
	30 ETH	7 SLOT
3)	*1 slashed SLOT is bought*	
	31 ETH	8 SLOT

We start with a healthy, freshly registered KNOT that has an active balance of 32 ETH on the Beacon Chain and is fully collateralized, i.e., its SLOT balance is 8. In step 1), a penalty of 1 ETH gets reported, which means 1 SLOT is slashed in the Stakehouse. In step 2) the KNOT gets

another penalty of 1 ETH, but this time it is not reported to the Stakehouse. This means the Stakehouse and the Beacon Chain are temporarily out of sync, but this is expected due to the asynchronous nature of balance reporting. In step 3), 1 slashed SLOT gets bought before reporting the previous penalty. Thus, to the Stakehouse, it now looks like the KNOT is fully collateralized again, even though its active balance on the Beacon Chain still suffers from the unreported penalty. The problem at this point is that the penalty from step 2) can now not be reported anymore. This means savETH holders will not earn dETH for any rewards earned by the KNOT until its active balance reaches 32 ETH again.

Also consider the following, slightly different scenario, in which a KNOT is not kicked from the Stakehouse even though it should.

	Beacon Chain	**Stakehouse**	
	Active balance	**SLOT**	**Last seen balance (before)**
	32 ETH	8 SLOT	32 ETH
1)	*2 ETH penalty gets reported*		
	30 ETH	6 SLOT	32 ETH
2)	*1 slashed SLOT is bought*		
	31 ETH	7 SLOT	30 ETH
3)	*2 ETH penalty gets reported*		
	29 ETH	6 SLOT	30 ETH
4)	*1 slashed SLOT is bought*		
	30 ETH	7 SLOT	29 ETH
5)	*2 ETH penalty gets reported*		
	28 ETH	6 SLOT	29 ETH

Here, we also keep track of the last seen active balance in the Stakehouse, which is used to calculate by how much a KNOT should be slashed. Note that the last seen balance is updated to the value of the Beacon Chain balance whenever a report is made (but not when slashed SLOT is bought), and the table shows its value before this update.

We again start with a freshly registered KNOT. When the 2 ETH penalty is reported in step 1), the difference between the last seen balance in the Stakehouse and the active balance on the Beacon Chain is 2 ETH, hence 2 SLOT will be slashed, giving a final SLOT balance of 6. In step 2), buying 1 slashed SLOT increases both the active balance on the Beacon Chain and the SLOT balance in the Stakehouse by one. Step 3) is where it gets interesting: A 2 ETH penalty decreases the active balance on the Beacon Chain to 29 ETH. When this gets reported, the Stakehouse

computes the slashing amount by subtracting the Beacon Chain balance from its last seen balance, giving 1 ETH. Thus, only 1 SLOT is slashed, even though there was a 2 ETH penalty on the Beacon Chain. This means that on the Beacon Chain, we have an active balance of 29 ETH, indicating that 3 ETH worth of penalties have been incurred, while in the Stakehouse we have a SLOT balance of 6, indicating that only 2 ETH worth of penalties have been incurred. Steps 4) and 5) increase this discrepancy by repeating the previous two steps. In the end, the KNOT lost 4 ETH on the Beacon Chain but only 2 SLOT in the Stakehouse. Losing 4 ETH should lead to the KNOT being kicked from the Stakehouse, but since the Stakehouse is only aware of 2 slashed SLOT, this does not happen.

Recommendation

The code should be fixed such that it is always possible to report penalties incurred on the Beacon Chain.

Status

Addressed in PR #14. The proposed fix is to increase the last active balance in the Stakehouse when buying slashed SLOT. Note that this can make the last active balance surpass the highest-seen balance, in which case it should be possible to submit a balance report to mint the extra balance as dETH. To allow for this, the `balanceIncrease` function in the BalanceReporter also must be modified to allow a balance increase to be reported when the reported balance is equal to the last active balance, and not only when it is greater.

A17: Slashing may be reported again after slashed SLOT has been bought (round 1)

[Severity: High | Difficulty: Low | Category: Functional Correctness]

It is possible for the same penalty to be reported twice: Once when the penalty actually occurred, and then again after the slashed SLOT has been bought back. This is unfair to KNOT owners, as they may be required to buy back the slashed SLOT originating from the same penalty multiple times. Additionally, savETH or index owners can report the second purchase as a balance increase and mint dETH from ETH that did not come from rewards.

Scenario

Assume a freshly registered KNOT.

1. The KNOT gets penalized by 1 ETH, decreasing its active balance on the Beacon Chain from 32 ETH to 31 ETH.
2. The penalty is reported to the Stakehouse, resulting in the slashing of 1 SLOT.
3. Some user, say Alice, gets a signed report from the deposit router certifying that the active balance of the KNOT on the Beacon Chain is 31 ETH.
4. The 1 slashed SLOT is bought back, increasing the active balance of the KNOT to 32 ETH. The KNOT is now at full health again.
5. Alice submits the report she got earlier to the Stakehouse. Since the report states that the KNOT's current active balance is 31 ETH, to the Stakehouse it looks like the KNOT has been penalized again, resulting in another slashing of 1 SLOT, even though there has not been another penalty.
6. After the slashed SLOT is bought back again, the active balance in the Beacon Chain is 33 ETH, and Alice can report a balance increase to mint 1 dETH.

Note that reports from the deposit router are only valid for 500 blocks, which means the time between steps 3 and 5 cannot exceed ca 1.6 hours.

Recommendation

Ensure that reports made before buying slashed SLOT cannot be used afterwards.

Status

Addressed in PR #16. The proposed fix is to invalidate the current nonce (used by the deposit router when signing reports) when buying slashed SLOT. This should ensure that buying slashed SLOT invalidates any report signed before then.

A18: Slashing may be reported again after slashed SLOT has been bought (round 2)

[Severity: High | Difficulty: High | Category: Functional Correctness]

This finding assumes that the proposed fixes from A14 and A16 have been applied.

Even when A17 has been addressed, there is another – though more difficult – way to report the same penalty twice. This crucially depends on the data reported from The Graph being stale, which is unlikely (it usually updates within seconds) but possible.

Scenario

This scenario takes advantage of the fact that the deposit router needs to combine data from different sources that may not be synchronized.

The following table considers a single KNOT. On the Beacon Chain we keep track of the global deposit counter, which is increased whenever a deposit is processed, and of the current active balance of the KNOT. On the Eth1 Chain, we keep track of the KNOT's SLOT balance, the last seen active balance both before and after each step, and the nonce used by the deposit router when signing reports (stored in the TransactionManager; see Deposit Router). Further, the table shows the total number of deposits made to the Eth2 Deposit Contract, which is used by The Graph to compute the deposit index for the KNOT (see proposed fix for A14).

Numbers marked with an asterisk * are stale.

	Beacon Chain		Eth1 chain				
			Stakehouse			Deposit Contract	The Graph
	Global deposit counter	Active balance	Last seen active balance (before/after)	SLOT	nonce	Deposit count	Latest deposit index for KNOT
	1000	32 ETH	32 ETH / 32 ETH	8	0	1000	500
1)	*1 ETH penalty + report*						
	1000	31 ETH	32 ETH / 31 ETH	7	1	1000	500
2)	*1 slashed SLOT is bought*						
	1000	31 ETH*	31 ETH / 32	8	2	1001	500*

			ETH				
3)	*Report is requested from deposit router and submitted to Stakehouse, causing slashing*						
	1000	31 ETH*	32 ETH / 31 ETH	7	3	1001	500*
4)	*The Graph is updated*						
	1000	31 ETH*	31 ETH / 31 ETH	7	3	1001	1001
5)	*Beacon Chain processes remaining deposits*						
	1001	32 ETH	31 ETH / 31 ETH	7	3	1001	1001

We start with a freshly registered KNOT. We assume the Beacon Chain has processed all of the 1000 deposits made to the Deposit Contract. The deposit index of the KNOT is assumed to be 500, which intuitively means that the latest deposit made to the KNOT (in this case the initial deposit, since we are considering a freshly registered KNOT) was the 500th deposit that was ever made to the Deposit Contract. Since the global deposit counter on the Beacon Chain is larger than the deposit index, we know that the deposit has been processed by the Beacon Chain.

In step 1) the KNOT incurs a penalty of 1 ETH on the Beacon Chain which is immediately reported to the Stakehouse. This results in the last seen active balance being decreased by 1, hence 1 SLOT is slashed. Additionally, because a report has been processed, the nonce is increased. In step 2) the slashed SLOT is bought back. The fix for A16 causes the last seen balance to be updated to 32 ETH, and the fix for A17 causes the nonce to be increased. Note that while the deposit count of the Deposit Contract has increased, this is not the case for the global deposit counter on the Beacon Chain, because in this example we assume that this only happens in step 5), hence the active balance is stuck at 31 ETH. Similarly, note that the deposit index maintained by The Graph is not updated, which is assumed to happen in step 4).

Step 3) is where it gets interesting, in particular regarding the deposit router: First, it fetches the global deposit counter from the Beacon Chain (which is 1000) and the deposit index from The Graph (which is 500). Since the global deposit counter is larger than the deposit index, the deposit router assumes all deposits have been processed by the Beacon Chain and continues. (Note that this check succeeding crucially depends on the deposit index reported by The Graph being stale.) Next, the deposit router signs a report containing the current active balance from the Beacon Chain (which is 31 ETH) and the current nonce (which is 3). When this report is submitted to the Stakehouse, the last seen active balance is decreased from 32 ETH to 31 ETH, hence 1 SLOT is slashed. **At this point, we have reported the same slashing twice.** This is unfair to KNOT owners, who must buy back another slashed SLOT in order to restore the KNOT to full health.

Steps 4) and 5) are not strictly part of the scenario anymore. They simply update The Graph and the Beacon Chain to show what their correct values look like. In step 4), the deposit index for the KNOT is updated to 1001, because the latest deposit to the KNOT in step 2) was the 1001th deposit ever made to the Deposit Contract. Step 5) processes the deposit made in step 2), increasing the global deposit counter to 1001 and the active balance to 32 ETH.

Recommendation

The reason the above scenario is possible is because the deposit router fetches data from the Stakehouse (the nonce) that is more recent than the data from The Graph (the deposit index). If both of these data points are stale, or both of them are up-to-date, then the above scenario is not possible. Thus, it is recommended to ensure that the data fetched by the deposit router is always in sync.

Status

Addressed. The proposed fix is to keep track of the latest deposit index of a KNOT not in The Graph but also in the Stakehouse. Then, when the deposit indexes fetched from the Stakehouse and from The Graph are the same, the deposit router knows that the received data is in sync.

A19: Stakehouse can be observed in inconsistent state when a KNOT is registered

[Severity: Low | Difficulty: Low | Category: Security]

During the process of joining a Stakehouse using `TransactionManager::joinStakehouse()`, at some point the function `StakeHouseRegistry::addMember()` is called. There, the user-provided function `gateKeeper.isMemberPermitted()` may be invoked.

Because `StakeHouseRegistry::addMember()` is decorated with the `nonReentrant` modifier, recursive calls to `TransactionManager::joinStakehouse()` are prevented, which ensures that no direct reentrancy attacks are possible.

However, a user may call other Stakehouse functions from `gateKeeper.isMemberPermitted()`. This is potentially problematic, because at this point the Stakehouse is in an inconsistent state. For example, the state of the new KNOT has already been updated to `TOKENS_MINTED`, but neither dETH nor SLOT tokens have been minted yet. Thus, the user can call Stakehouse functions while some invariants are broken, which may have unforeseen consequences.

Recommendation

This issue is classified as low severity as there is no obvious way to use it in an attack. However, to remove any possibility of exploiting temporarily broken invariants, it is recommended to move the call to `gateKeeper.isMemberPermitted()` to the beginning of `TransactionManager::joinStakehouse()`, before any state changes have been made.

Status

Addressed in PR #16. The proposed solution is to update the lifecycle status to `TOKENS_MINTED` only after the KNOT has been added to the Stakehouse and had derivative tokens minted. This applies to all `TransactionManager` functions that update the status to `TOKENS_MINTED`: `createStakehouse`, `joinStakehouse`, `joinStakehouseAndCreateBrand`, and `rageQuitKnot`.

A20: Deposit router does not yet encrypt a KNOT's signing key

[Severity: High | Difficulty: N/A | Category: Functional Correctness]

The ability to encrypt a KNOT's signing key using the CIP public key has not been implemented yet in the deposit router. Instead, the deposit router hashes the signing key of a KNOT with `keccak256()`. This makes it impossible to hand a KNOT's signing key over to another user using the CIP.

Recommendation

In the reposit router, encrypt a KNOT's signing key using the CIP public key.

Status

Addressed.

A21: Signature validation does not confirm identity of free-floating sETH owner

[Severity: High | Difficulty: High | Category: Input Validation]

When performing a multi-party rage quit, the SlotSettlementRegistry does not have a way of confirming that the address recovered from the signature of the free-floating sETH owner is indeed the address intended.

If the signature is for some reason generated incorrectly, the address recovered from it will be incorrect. In the unlikely case that the address returned is a valid address that happens to also own 12 sETH, the rage quit will proceed and the wrong user's tokens will be burned.

Note that the same risk is not present for the collateral owners and index owner, since there cannot be anyone else who owns collateral for the KNOT or has the KNOT isolated in an index.

Recommendation

Fix code. Pass the address of the free-floating sETH owner as a parameter to the BalanceReporter's `multipartyRageQuit` function, and check if the address recovered from the signature matches this.

Status

Addressed by the client after receiving the report.

A22: Slashings may not always be reportable

[Severity: High | Difficulty: N/A | Category: Functional Correctness]

In order to report to the Stakehouse that a KNOT has been kicked from the Beacon Chain one has to call `slash()` of the `BalanceReporter` contract. However, `slash()` can only be called if the new balance of the KNOT is strictly less than the last active balance seen by the Stakehouse, which may not always be the case. This can make it impossible to rage-quit the KNOT in order to withdraw the staked ETH.

Scenario

1. A new KNOT is registered with an initial active balance of 32 ETH, which is also the last seen active balance in the Stakehouse.
2. The KNOT earns rewards until its active balance reaches 34 ETH. However, this is not reported to the Stakehouse.
3. The KNOT performs a slashable offense and must exit the Beacon Chain. After the exit epoch has passed, its active balance has dropped to 33 ETH.
4. A user wants to report the slashing but cannot do so, because the active balance of the KNOT on the Beacon Chain is not strictly smaller than the last seen active balance in the Stakehouse (which is 32 ETH).

Since the KNOT has exited the Beacon Chain in step 3, its active balance cannot change anymore. Thus, the slashing can never be reported. Additionally, rage-quitting the KNOT is impossible, because in order to do so, the exit epoch must be reported to the Stakehouse, but if the KNOT has been slashed then the only way to do that is by reporting the slashing, which is not possible. This means that the KNOT owners cannot withdraw the KNOT's stake.

Recommendation

Fix the code such that `BalanceReporter::slash()` can always be called if a KNOT has been kicked from the Beacon Chain.

Status

Addressed in PR #3.

A23: Signing keys of a kicked KNOT cannot be requested via CIP

[Severity: High | Difficulty: High | Category: Robustness]

The function `SafeBoxManager::applyForDecryption()` that triggers the Threshold Decryption Protocol (see Threshold Decryption Protocol) for a KNOT requires that the KNOT has not been kicked from the Stakehouse. This may make it impossible to rage-quit a KNOT that has been abandoned by the original owner, because in order to rage-quit, the KNOT first needs to exit the Beacon Chain, which requires the KNOT's signing key. Having a constantly leaking KNOT in a Stakehouse lowers the efficiency rating of the house, making it harder to negotiate MEV deals. Additionally, once the KNOT loses more than 4 collateralized SLOT, not all dETH might be backed by ETH anymore.

Scenario

1. A KNOT becomes inactive and starts getting penalized
2. Once the KNOT has lost all its 4 collateralized SLOT, it is kicked from the Stakehouse
3. Alice buys more than 2 of the KNOT's collateralized SLOT
4. Alice requests the KNOT's signing key, which she ought to be able to get because she holds more than 2 collateralized SLOT. However, the request fails because the KNOT has been kicked

Note that step 2 will likely take a long time, and if other people buy enough SLOT in this time they can request the KNOT's signing key before the KNOT is kicked.

Recommendation

Drop the requirement of `SafeBoxManager::applyForDecryption()` that the KNOT has not been kicked from the Stakehouse.

Status

Addressed.

A24: Slashing penalty may be calculated incorrectly

[Severity: High | Difficulty: N/A | Category: Functional Correctness]

In PR #3, the function `BalanceReporter::_slash()` was modified to assume a minimum penalty of 1 ETH in case a KNOT is kicked from the Beacon Chain due to a slashable offense and there has been no balance decrease relative to the last seen active balance:

```
// should there be an unreported slash, it will be at least 1 ETH slash
if (decrease == 0 && isSlashUnreported) {
       // at mainnet slashing kick will be at least 1 ether
       // it should only hit this once per KNOT
       decrease = 1 ether;
}
```

The problem is that the penalty is computed based on the effective balance of the KNOT and may be less than 1 ETH. More precisely, according to the Bellatrix fork, the penalty is computed by $\frac{effective\ balance}{32}$. Thus, if the effective balance is less than 32 ETH, then the penalty on the Beacon Chain will be less than 1 ETH. However, when the kicking is reported to the Stakehouse, it may deduct 1 slashed SLOT even though the actual incurred penalty is less than that. Thus, KNOT owners may need to buy more slashed SLOT than necessary if they want to restore the health of the KNOT in order to rage-quit.

Scenario

1. A freshly registered KNOT is penalized by 2 ETH, resulting in an active balance of 30 ETH on the Beacon Chain. This is reported to the Stakehouse.
2. The KNOT earns 1 ETH in rewards, increasing its active balance to 31 ETH. However, this is not reported to the Stakehouse.
3. The KNOT commits a slashable offense and is kicked from the Beacon Chain. According to the Bellatrix fork, this incurs a penalty of 31 ETH / 32 = 0.96875 ETH, resulting in an active balance of 31 ETH - 0.96875 ETH = 30.03125 ETH on the Beacon Chain.
4. When reporting the kicking to the Stakehouse, the KNOT will be slashed by 1 SLOT as per the above code snippet, which is more than the penalty incurred on the Beacon Chain.

Recommendation

It does not seem possible to compute the exact penalty that was incurred on the Beacon Chain in the Stakehouse, because the effective balance that was used to calculate the penalty is not

known. Thus, the only way to fix this issue seems to be to remove the above code snippet from `BalanceReporter::_slash()`.

Status

Addressed by the client after receiving the report.

Informative findings

B01: Early collateralized SLOT owners might not be able to become majority owners

[Severity: Low | Difficulty: N/A | Category: Fairness]

The SlotSettlementRegistry keeps for each KNOT a list of users that own collateral for that KNOT. Users are added in the order that they first acquired SLOT for this KNOT, and, when SLOT is slashed, earlier users are hit first.

A user that becomes a majority SLOT owner (more than 2 collateralized SLOT) of a KNOT can request the KNOT's signing key via the Common Interest Protocol. However, a user that appears earlier in the list than the current majority owner is unlikely to be able to become a majority owner, as any slashing will affect them first.

Scenario

1. Alice adds a KNOT to a Stakehouse, being assigned 4 SLOT collateral. Current owner list:
    a. Alice (4 SLOT)
2. Alice lets the KNOT leak, getting slashed by 1 SLOT, which is bought back by Bob. Current owner list:
    a. Alice (3 SLOT)
    b. Bob (1 SLOT)
3. Alice gets slashed by another 3 SLOT, which are bought back by Charlie. Current owner list:
    a. Alice (0 SLOT)
    b. Bob (1 SLOT)
    c. Charlie (3 SLOT)
4. Charlie, as the majority owner, uses the common interest protocol to request the KNOT's signing key.
5. Charlie also lets the KNOT leak, causing Bob to get slashed by 1 SLOT. Current owner list:
    a. Alice (0 SLOT)
    b. Bob (0 SLOT)
    c. Charlie (3 SLOT)
6. At this point, if Bob buys back SLOT but the KNOT is still leaking, he will just get slashed again. If Bob wants to recover the signing key, he needs to wait until the KNOT has leaked more than 2 SLOT (1 from Bob and 1 from Charlie) and only then buy back the

SLOT. By then it is likely that another user might come in and buy back the SLOT instead.

Recommendation

From the point of view of the house, the above scenario is not problematic, since other users besides Bob can (and have incentives to) buy back the slashed SLOT and, if necessary, recover the signing key.

It's not expected that the risk of this scenario will discourage users from buying SLOT, as the primary incentive for doing so is to get a share of the revenue associated with the KNOT (such as MEV rewards). Recovering the signing key is seen as a mitigation strategy rather than a goal in itself. In fact, it's not desirable for a KNOT to change hands many times, as this introduces a risk since all previous owners still have access to the signing key. As such, a user who recovers the signing key might prefer to work with the other owners to rage quit the KNOT rather than continue to operate it.

It's expected that any user that buys SLOT is aware of the risks and knows that they might get their collateral slashed at any time. Although this scenario puts early buyers at more risk, the potential profits of owning SLOT are expected to compensate for this risk, keeping SLOT in high demand.

B02: CIP committee may be compromised with an investment of 50 ETH

[Severity: High | Difficulty: High | Category: Security]

If an attacker buys 1 slashed SLOT with 50 different accounts, he can use these accounts to submit 50 requests for candidacy. Assuming that there are 50 other valid requests for candidacy (giving 100 requests in total, which is the minimum), then all submitted requests for candidacy will be selected to become candidates for the next committee. This means the number of candidates is $n = 100$, of which 50 are controlled by the attacker. Further, since $t = ceil(\frac{n}{2}) - 1 = 49$, this means that the attacker controls $t + 1$ candidates, which implies he can reconstruct the shared secret SK (see Common Interest Protocol).

What makes this attack difficult is the high investment of 50 ETH and the fact that the attack does not yield direct monetary gain. Further, buying enough slashed SLOT is likely to take considerable time, and the attacker may need to compete with other users who are also interested in buying slashed SLOT.

Recommendation

None. The attack described in this finding is not possible under our Security Model, but we think it is worth pointing out under what circumstances our assumptions might break.

B03: Old KNOT owner may not cooperate when handing over signing keys

[Severity: High | Difficulty: High | Category: Documentation]

Using the Threshold Decryption Protocol, a user can request the signing key of an arbitrary KNOT in order to become its new owner, provided the user holds more than 2 collateralised SLOT for the KNOT. Once the new owner has obtained the signing key, he can run the validator associated with the KNOT himself and potentially do a better job. However, this requires that the old owner stops his instance of the validator to prevent slashing due to running two instances of the same validator.

Scenario

1. Alice runs a KNOT K but does not do a good job. K loses more than 2 collateralised SLOT.
2. Bob buys more than 2 collateralised SLOT for K and requests its signing key via the Threshold Decryption Protocol.
3. Bob starts a new validator instance using K's signing key while Alice's validator instance with the same signing key is still running.
4. The old and the new instance make different block proposals, causing the validator to be slashed.

Recommendation

The need to cooperate with the old owner of the signing key should be documented prominently to avoid validators from being slashed.

B04: Two different meanings of “active member” in StakeHouseRegistry

[Severity: Low | Difficulty: N/A | Category: Documentation]

When naming functions, the StakeHouseRegistry is inconsistent in its use of the term “active member”. The `isActiveMember` function excludes both members that have rage quit and that have been kicked, but the `numberOfActiveKnots` function excludes only those that have rage quit, and not those that have been kicked. This can lead to confusion for users about what these functions are supposed to return.

Recommendation

Rename one of the functions so that the difference is clear.

Status

Addressed in PR #16. The `numberOfActiveKnots` function has been renamed `numberOfActiveKnotsThatHaveNotRageQuit`.

B05: Gas optimization for SafeBoxManager

[Severity: Informative | Difficulty: N/A | Category: Optimization]

The `SafeBoxManager` contract is mainly used by the off-chain CIP clients to exchange messages. A client sends a message by calling a corresponding function of the `SafeBoxManager` and passing the required data, which is then stored in the contract's storage. However, messages are only read by other off-chain clients, which means they can be stored using events rather than in storage, which is more gas efficient.

Recommendation

Emit events that carry the necessary information instead of storing the information in the contract's storage.

Status

Addressed in PR #3.

B06: CIP committee members need to be trusted forever

[Severity: Informative | Difficulty: N/A | Category: Security]

One has to trust that the majority of the members of the current committee is honest, meaning that they do not misuse their power and decrypt the signing keys of arbitrary KNOTs. What might be less obvious is the fact that one also has to trust that the majority of the members of *previous* committees remains honest forever. This is because when a committee passes its knowledge of the shared secret *SK* on to the next committee, the members of the old committee may still use their shares to reconstruct SK, giving them the power to cause great damage.

Scenario

1. Members of the current committee i and candidates of the next committee $i + 1$ run the Committee Handover Protocol.
2. Now, both the members of committee i and of committee $i + 1$ have the power to decrypt a KNOT's signing key. They will keep this power forever.

Recommendation

None. The issue described in this finding is not possible under our Security Model, but we think it is worth pointing out under what circumstances our assumptions might break.

B07: Invalid signature passed to DepositContract

[Severity: Informative | Difficulty: N/A | Category: Documentation]

The function `BalanceReporter::_addTopUpToQueue()` passes a potentially invalid signature to the `deposit()` function provided by the Eth2 Deposit Contract. The passed signature may be invalid because it assumes a deposit amount of 32 ETH, but the actual amount may be different.

However, the fact that an invalid signature is used is currently not a problem because the Eth2 Deposit Contract does not verify the signature, and the Beacon Chain only verifies the signature when doing the initial deposit. Since `BalanceReporter::_addTopUpToQueue()` can only be called for a KNOT after the initial deposit, the Beacon Chain does not actually verify the signature either in this case.

In summary, everything is working as expected at the moment, but since this behavior of the Beacon Chain is not explicitly documented and the funds of users may get lost if the current behavior changes, it may be sensible to check forks for breaking changes.

Recommendation

Monitor changes to the Beacon Chain specification that are concerned with processing validator deposits. Additionally, document the call to `deposit()` to make it clear that it is okay to pass an invalid signature.

B08: Incorrect EIP-712 encoding

[Severity: Informative | Difficulty: N/A | Category: Functional Correctness]

The contract `SignatureValidator` encodes several data structures using the EIP-712 encoding. However, this encoding is not always performed correctly.

In function `_isReportSignatureValid()`, the type of the data structure that is to be encoded is given as `KeyEncryptionPacket(bytes blsPublicKey,bytes encryptedSigningKey)`, but the actual type is `KeyEncryptionPacket(bytes blsPublicKey,bytes encryptedSigningKey,uint256 deadline,uint256 nonce)`.

In function `_getStructHash()`, instead of encoding the value of `_blsPublicKey` directly, the value of `keccak256(_blsPublicKey)` should be used. This is because `_blsPublicKey` is of type `bytes`, which EIP-712 requires to be hashed.

Recommendation

Fix code to follow EIP-712.

Status

Addressed in PR #7.

B09: abi.encodePacked() is used to combine multiple variable-length values

[Severity: Informative | Difficulty: N/A | Category: Best Practice]

In contract `SignatureValidator`, the functions `_isReportSignatureValid()` and `_isReportSignatureValid()` use `abi.encodePacked()` to concatenate multiple variable-length values before hashing the result. Since using `abi.encodePacked()` this way may lead to hash collisions, it is generally recommended to use `abi.encode()`.

In the specific case of the `SignatureValidator`, collisions are prevented by including a nonce, which means there is no inherent danger in using `abi.encodePacked()`.

Recommendation

Use `abi.encode()` instead of `abi.encodePacked()` when possible.

B10: Ownership of a second healthy KNOT allows user to always meet redemption threshold

[Severity: Medium | Difficulty: Low | Category: User Behavior]

Note that, regardless of the redemption threshold (see SlotSettlementRegistry), 8 SLOT in collateral is enough to rage-quit any healthy KNOT:

$$\text{total collateralized SLOT} \times \text{exchange rate} \geq 4 \times \text{redemption rate}$$
$$8 \times \text{exchange rate} \geq 4 \times \text{redemption rate}$$
$$8 \times (\text{dETH minted} / \text{SLOT minted}) \geq 4 \times (\text{dETH minted} / \text{current SLOT})$$
$$8 \geq 4 \times (\text{SLOT minted} / \text{current SLOT})$$
$$2 \geq \text{SLOT minted} / \text{current SLOT}$$

Since only half of the SLOT in the house can be slashed, the above is always true.

This means that if a user is the single collateral owner of a healthy KNOT, that user will always meet the redemption threshold of any other KNOT that they help rage-quit. It would be easy and costless for such a user to help other users rage-quit their KNOTs while ignoring the redemption threshold.

Scenario

1. Alice owns a KNOT that has been slashed 1 SLOT.
2. A different KNOT in the same house has also been slashed, therefore even after topping up her own KNOT Alice will need to buy additional SLOT to pass the redemption threshold, say 0.1 SLOT.
3. Bob owns a third, healthy KNOT in the same house. He proposes the following to Alice:
    1. Instead of buying the 1 SLOT herself, Alice pays Bob 1.01 ETH to do it for her.
    2. Now Bob is a collateral owner for Alice's KNOT. Since together Bob and Alice own 8 SLOT of collateral (Bob owns 5 between his KNOT and Alice's, and Alice owns 3 from her own KNOT), Alice can rage quit her KNOT and withdraw the balance.
    3. Bob received 1.01 ETH from Alice and had to pay 1 ETH to buy the 1 SLOT, giving him a profit of 0.01 ETH. Alice paid Bob 1.01 ETH, but recovered 1 ETH when the balance was withdrawn. She had to spend 0.01 ETH, but this is 1/10 of what she would have to pay to buy the extra 0.1 SLOT to meet the redemption threshold.

Whether this is worth it for Alice depends on how much she values the 0.1 SLOT that she would have otherwise. If she would rather have 0.09 ETH than 0.1 SLOT, accepting Bob's offer is a better option for her.

Furthermore, participating in this process costs nothing for Bob, so if Alice is not willing to pay 0.01 ETH he can arbitrarily lower the price and still make a profit. Especially since she is rage

quitting, it's possible that Alice values ETH more than SLOT, in which case she has an incentive to hire Bob instead of buying slashed SLOT to rage quit.

Recommendation

Monitor user behavior to detect if it becomes common for users to rage quit without buying slashed SLOT from other KNOTs. If so, consider upgrading the protocol to give users additional incentives to buy SLOT before rage-quitting.

From the point of view of the Stakehouse, this scenario does not violate any invariants. The redemption threshold is not essential for accounting, as long as 8 SLOT and all minted dETH for the KNOT is burned at rage quit. Furthermore, there are other incentives for users to buy slashed SLOT, so even if this scenario does happen occasionally it might not be a problem as long as the house still recovers from slashing in other ways.

It is also possible that owners of healthy KNOTs (Bob in the above example) might not want to engage in this behavior, since if the redemption rate does not recover that will make it harder for them to rage quit in the future.

B11: Slashed SLOT might temporarily block part of the rewards from being minted

[Severity: Low | Difficulty: N/A | Category: Protocol Invariants]

A user of the Stakehouse protocol might expect that, immediately following a balance report, the Stakehouse and Beacon Chain would be in equivalent states; that is, the sum of dETH and SLOT for a KNOT is equal to its active balance in the Beacon Chain (ignoring unknown top-ups).

However, when a KNOT earns rewards between SLOT being slashed and being bought back, part of the Beacon Chain balance will not be reflected in the Stakehouse until all the slashed SLOT is bought.

Scenario

1. A KNOT starts with 32 ETH in the Beacon Chain and 24 dETH and 8 SLOT minted.
2. The KNOT leaks to 31 ETH, and SLOT gets slashed to 7 SLOT.
3. The KNOT is turned back on and earns 3 ETH in inflation rewards, bringing the balance up to 34 ETH.
4. The balance increase is reported and 2 dETH is minted.
5. The slashed SLOT is bought back, bringing the balance up to 35 ETH.
6. The balance increase is reported and 1 dETH is minted.

After step 4, the balance in the Beacon Chain is 34 ETH, but the Stakehouse balance is 26 dETH + 7 SLOT = 33. This is because 1 ETH of the rewards earned in step 3 went into recovering from the leak rather than minting dETH. This only happened because rewards were earned before the slashed SLOT was bought back.

This state is temporary, since once the slashed SLOT is bought back in step 5, the same amount becomes available for minting, bringing the Stakehouse in sync with the Beacon Chain again.

Recommendation

Document prominently, so that users and other systems that interact with the protocol are aware that minting of rewards might be delayed until slashed SLOT is bought.

B12: Kicked KNOT can be undercollateralized even after being brought back to health

[Severity: N/A | Difficulty: N/A | Category: Protocol Invariants]

A KNOT that loses all 4 of its collateralized SLOT is kicked from the Stakehouse. Any additional leakage of the Beacon Chain balance is not reflected by slashing SLOT. This means that the kicked KNOT's active balance in the Beacon Chain might be lower than the sum of its dETH and SLOT balances in the Stakehouse, even if all slashed SLOT is bought back and the queue is empty.

In this case, the amount of tokens burned to rage-quit the KNOT will be different from the amount of ETH withdrawn from the Beacon Chain after exit.

Scenario

1. A KNOT starts with 32 ETH in the Beacon Chain and 24 dETH and 8 SLOT minted.
2. The KNOT leaks 5 ETH, causing its balance to drop to 27 ETH. This is reported to the Stakehouse, causing 4 SLOT to be slashed and the KNOT to be kicked.
3. The KNOT's owner buys back the 4 slashed SLOT, which are deposited to the Beacon Chain and bring the balance up to 31 ETH.
4. The KNOT's owner pays 24 dETH to isolate the KNOT into an index and rage quits.
5. The owner withdraws the Beacon Chain balance, which adds up to 31 ETH, even though they burned the equivalent of 32 ETH in derivatives.

Recommendation

Document prominently. Barring black-swan events, it should take very long for a KNOT to lose all of its 4 SLOT collateral, and it's expected that the Stakehouse as a group will contribute to bring a KNOT back to health, by buying slashed SLOT and, in an extreme case, rotating the signing key via the CIP.

Users and other systems built on top of the Stakehouse protocol should be aware of the risks associated with kicked KNOTs in a house, which should give additional incentive to prevent kicking by undercollateralization. If it does happen, users involved in rage-quitting the kicked KNOT should be aware of its current balance when deciding how to distribute the withdrawn funds among the stakeholders.

Checked properties

There is more to an audit than finding bugs. We also want to report what errors we have ruled out, and what properties we have verified.

C01: dETH calculated from the exchange rate will never be more than dETH in the open index

The `withdraw` function of the savETHRegistry calculates the dETH to be withdrawn by burning `_amount` of savETH as

```
uint128 dETHFromExchangeRate = uint128((uint256(_amount) *
    saveETHToDETHExchangeRate()) / EXCHANGE_RATE_SCALE);
```

where `EXCHANGE_RATE_SCALE = 1e18` and `saveETHToDETHExchangeRate()` is computed as

```
function saveETHToDETHExchangeRate() public override view returns (uint256)
{
      uint256 totalSavETHSupply = saveETHToken.totalSupply();
      if (totalSavETHSupply == 0) {
            // avoid division by zero when no SaveETH minted
            return EXCHANGE_RATE_SCALE;
      }

      return uint256(dETHMetadata.dETHUnderManagementInOpenPool)
             .sDivision(totalSavETHSupply);
}
```

where `x.sDivision(y)` is calculated as `x * 1e18 / y`.

This calculation is followed by the following if statement, meant to prevent calculation imprecisions calculating more dETH than is available to withdraw:

```
if (dETHFromExchangeRate > dETHMetadata.dETHUnderManagementInOpenPool) {
    dETHFromExchangeRate = dETHMetadata.dETHUnderManagementInOpenPool;
}
```

However, this if statement is unnecessary, as the condition can never be true. This is because the `withdraw` function is protected by the following requires clause:

```
require(saveETHToken.balanceOf(_savETHOwner) >= _amount);
```

which implies that `_amount` is less than or equal to `saveETHToken.totalSupply()`. Therefore, the maximum value that `dETHFromExchangeRate` can take is

```
saveETHToken.totalSupply() * saveETHToDETHExchangeRate() / 1e18
```

Following the definition of `saveETHToDETHExchangeRate()`, this value simplifies to either `0` if `saveETHToken.totalSupply()` is `0`, or otherwise to `dETHMetadata.dETHUnderManagementInOpenPool`. Therefore, `dETHFromExchangeRate` can never be greater than `dETHMetadata.dETHUnderManagementInOpenPool`.

Note that this analysis ignores imprecisions in fixed-point arithmetic. However, as can be seen in the fixed-point precision analysis, arithmetic imprecision can only make `dETHFromExchangeRate` smaller than it should be, not larger. Therefore, even when accounting for imprecisions, the if condition can never be satisfied.

We have verified this fact by running the Solidity SMTChecker on a simplified version of the contract containing only the `withdraw` function and the functions required to calculate `dETHFromExchangeRate` (see Appendix 1). The property being verified was that `dETHFromExchangeRate <= dETHMetadata.dETHUnderManagementInOpenPool`. The verification passed, confirming that this property is true independently of the state of the contract when the function is called.

Note: Although the SMTChecker was run on a version of the code that used the conversion logic prior to the fixes for finding A11, the same reasoning above also holds for the new conversion logic.

C02: Balance reporting griefing attack rebuttal

A question was raised on whether it would be possible, by exploiting the top-up queue, to mint dETH from a balance increase caused not by inflation rewards, but by buying slashed SLOT. This would violate the property that dETH is only minted from inflation rewards, and would allow an attacker to essentially double-mint part of the balance as both SLOT and dETH.

The table below shows the proposed scenario in the first column ("With queue"). At step 7, dETH is minted from a balance increase caused by flushing the queue, which seems to violate the desired property.

However, by comparing with the second column ("Without queue"), which describes how the scenario would play out if any funds from buying slashed SLOT could be immediately deposited to the Beacon Chain and no queue was necessary, we see that the end states of both are equivalent. This means that no additional tokens are minted by exploiting the queue beyond what should be minted anyway.

The reason is that, because the ETH used to buy back SLOT was not deposited immediately (going into the queue instead), 0.9 ETH in inflation rewards that should have been minted as dETH instead went into paying for the leak (in steps 3-4). So when the queue is flushed in step 6, 0.9 ETH should go into making up for those rewards that were never minted. Note that this is similar to the scenario described in finding B11.

In contrast, if the balance report in step 7 was blocked from being performed, the Stakehouse would end in an inconsistent state with the Beacon Chain, as the final state would be as follows:

- 33.9 ETH in beacon chain
- 25 dETH
- 8 SLOT
- 0 ETH in queue

Note that there is 0.9 ETH in the beacon chain that is not reflected in the protocol. This is the same amount earned in total from inflation rewards in steps 3-4.

With queue	**Without queue**
0. Start - 33 ETH in beacon chain - 25 dETH - 8 SLOT - 0 ETH in queue	0. Start - 33 ETH in beacon chain - 25 dETH - 8 SLOT
1. Turn off validator, leak 0.1 ETH, report	1. Turn off validator, leak 0.1 ETH, report

slashing - 32.9 ETH in beacon chain - 25 dETH - 7.9 SLOT - 0 ETH in queue	slashing - 32.9 ETH in beacon chain - 25 dETH - 7.9 SLOT
2. Buy back 0.1 SLOT - 32.9 ETH in beacon chain - 25 dETH - 8 SLOT - 0.1 ETH in queue	2. Buy back 0.1 SLOT - 33 ETH in beacon chain - 25 dETH - 8 SLOT
3. Turn validator back on, inflation rewards earn 0.1 ETH - 33 ETH in beacon chain - 25 dETH - 8 SLOT - 0.1 ETH in queue	3. Turn validator back on, inflation rewards earn 0.1 ETH - 33.1 ETH in beacon chain - 25 dETH - 8 SLOT
4. Repeat steps 1-3 8 more times - 33 ETH in beacon chain - 25 dETH - 8 SLOT - 0.9 ETH in queue	4. Repeat steps 1-3 8 more times - 33.9 ETH in beacon chain - 25 dETH - 8 SLOT
5. Repeat step 1 - 32.9 ETH in beacon chain - 25 dETH - 7.9 SLOT - 0.9 ETH in queue	5. Repeat step 1 - 33.8 ETH in beacon chain - 25 dETH - 7.9 SLOT
6. Repeat step 2, queue is flushed - 33.9 ETH in beacon chain - 25 dETH - 8 SLOT - 0 ETH in queue	6. Repeat step 2 - 33.9 ETH in beacon chain - 25 dETH - 8 SLOT
7. Report balance increase - 33.9 ETH in beacon chain - 25.9 dETH - 8 SLOT - 0 ETH in queue	7. Report balance increase - 33.9 ETH in beacon chain - 25.9 dETH - 8 SLOT

C03: Withdrawal credentials protection

A known security vulnerability for delegated staking applications is the possibility of users front-running the initial deposit and registering their own withdrawal credentials (see https://research.lido.fi/t/mitigations-for-deposit-front-running-vulnerability/1239). In the Stakehouse protocol, this would allow a user to add a KNOT to a Stakehouse, mint the derivative tokens, and then withdraw from the Beacon Chain, thus keeping both their 32 ETH stake and the minted tokens.

The Stakehouse protocol protects against this by requiring a balance report signed by the Deposit Router before minting the tokens corresponding to the initial deposit (step TOKENS_MINTED in the AccountManager). This report includes the validator's withdrawal credentials, which the TransactionManager checks to make sure that they match the protocol's. If not, the user is blocked from joining a Stakehouse and minting tokens, leaving them only the option to rage-quit. This way, if a user front-runs the initial deposit in the previous step, they will be unable to proceed with the registration process.

The Deposit Router is also used to check at the REGISTER_INITIALS stage of the registration process that the BLS key being registered has not received a deposit in the past. If it has, the registration process cannot be started. This does not prevent the aforementioned front-running attack, but it does protect a user from accidentally registering a key with the wrong withdrawal credentials and being prevented from proceeding after performing the initial deposit.

C04: Relation between Beacon Chain balance and Stakehouse state

This analysis assumes that the last active balance is updated after a user buys slashed SLOT (see A16) and that the decrease in collateral is calculated correctly (see A15).

The invariants below captures the relation between the balance of a KNOT in the Beacon Chain and the Stakehouse:

Invariant: Assuming the KNOT has not been kicked, $dETH + 8 - slashed \leq balance + queue$.

- *dETH* is the total amount of dETH minted for the KNOT (24 dETH for the initial deposit + minted rewards).
- *8* is the total amount of SLOT minted for the KNOT (before slashing).
- *slashed* is the amount of slashed SLOT.
- *balance* is the last active balance of the Beacon Chain registered in the Stakehouse (adjusted to ignore unknown top-ups; see Deposit Router).
- *queue* is the amount stored in the KNOT's top-up queue.

Note: The more general invariant $dETH + 8 - slashed = balance + queue$ does NOT hold (see B11).

The invariant holds when the KNOT first joins the Stakehouse:

- $dETH = 24$
- $slashed = 0$
- $balance = 32$
- $queue = 0$

We now show that the invariant is preserved after every operation that modifies the variables that it depends on.

Balance increase: A balance increase can be reported when the new active balance b is greater than or equal to the last active balance, which is then updated to b. If the new active balance is also greater than $dETH + 8$ (the highest-seen balance before slashing), the difference is minted as dETH, giving us (using x' to denote the new value of variable x after the update):

- if $b > dETH + 8$:
  - $dETH' = dETH + b - (dETH + 8) = b - 8$
- if $b \leq dETH + 8$:
  - $dETH' = dETH$
- $slashed' = slashed$
- $balance' = b$

- *queue' = queue*

$$dETH' + 8 - slashed' = b - 8 + 8 - slashed = b - slashed \text{ (if } b > dETH + 8)$$

$$dETH' + 8 - slashed' = dETH + 8 - slashed \text{ (if } b \leq dETH + 8)$$

$$balance' + queue' = b + queue$$

If $b > dETH + 8$, then since both *slashed* and *queue* are nonnegative, we have that $dETH' + 8 - slashed' = b - slashed \leq b + queue = balance' + queue'$.

If $b \leq dETH + 8$, then assuming the invariant holds before the balance increase, we have that $dETH' + 8 - slashed' = dETH + 8 - slashed \leq balance + queue$. Since $balance \leq b$, we can conclude that $balance + queue \leq b + queue = balance' + queue'$. Therefore $dETH' + 8 - slashed' \leq balance' + queue'$.

Slashing: Since we are assuming the KNOT is not kicked, slashing can be reported only when the new active balance *b* is less than *balance*, and the difference must be less than $4 - slashed$ (otherwise the KNOT would be kicked for having all 4 SLOT collateral slashed). The difference is converted into slashed SLOT and the last active balance is updated:

- *dETH' = dETH*
- *slashed' = slashed + (balance - b)*
- *balance' = b*
- *queue' = queue*

$$dETH' + 8 - slashed' = dETH + 8 - slashed - (balance - b)$$

$$balance' + queue' = b + queue$$

Assuming the invariant holds before the slashing was reported, we can conclude that it still holds afterwards:

$$dETH + 8 - slashed \leq balance + queue$$
$$dETH + 8 - slashed - (balance - b) \leq balance + queue - (balance - b)$$
$$dETH + 8 - slashed - (balance - b) \leq b + queue$$
$$dETH' + 8 - slashed' \leq balance' + queue'$$

Buying slashed SLOT: When a user buys slashed SLOT, the amount *a* bought is sent to the KNOT's queue. If the contents of the queue surpass 1 ETH, then the queue is flushed and the amount is deposited in the Beacon Chain. In this case, the last active balance is updated to reflect the deposit:

- *dETH' = dETH*

- *slashed'* = *slashed* - *a* (note that *a* ≤ *slashed* or the operation reverts)
- if *queue* + *b* < *1*:
  - *balance'* = *balance*
  - *queue'* = *queue* + *a*
- if *queue* + *a* ≥ *1*:
  - *balance'* = *balance* + *queue* + *a*
  - *queue'* = *0*

$$dETH' + 8 - slashed' = dETH + 8 - slashed + a$$

$$balance' + queue' = balance + queue + a$$

Note that *balance'* + *queue'* is the same regardless of whether the queue was flushed or not. Then, assuming that the invariant holds before the purchase, we can conclude that it still holds after:

$$dETH + 8 - slashed \leq balance + queue$$
$$dETH + 8 - slashed + a \leq balance + queue + a$$
$$dETH' + 8 - slashed' \leq balance' + queue'$$

Queue top-up: When a user tops up the KNOT's queue, they must provide an amount *a* equal to at least *1* - *queue*. The queue is then flushed, depositing *queue* + *a* to the Beacon Chain and updating the active balance. Since *a* did not come from buying back SLOT, it's considered an unknown top-up and will not be counted in future balance reports. Therefore, only the amount already in the queue is registered to the balance:

- *dETH'* = *dETH*
- *slashed'* = *slashed*
- *balance'* = *balance* + *queue*
- *queue'* = *0*

$$dETH' + 8 - slashed' = dETH + 8 - slashed$$

$$balance' + queue' = balance + queue + 0$$

Therefore, assuming *dETH* + *8* - *slashed* ≤ *balance* + *queue* held before, *dETH'* + *8* - *slashed'* ≤ *balance'* + *queue'* holds after.

C05: Fixed-point precision analysis

This analysis calculates the error bounds on conversions between tokens in the savETHRegistry. See finding A11 for a summary and further discussion.

Converting dETH to savETH

We use d to denote the amount of dETH in the open index and s to denote the total supply of savETH in circulation. Both d and s (as well as a below) are assumed to be unsigned integers measured in wei (10^{-18} ETH), and we assume no overflow.

To convert a dETH to the corresponding amount x in savETH, the savETHRegistry uses the formula

$$x = \frac{a \cdot 10^{18}}{r}$$

where r is the exchange rate between dETH and savETH, calculated as

$$r = \frac{d \cdot 10^{18}}{s}$$

unless $s = 0$, in which case $r = 10^{18}$ (note that the 10^{18} factors cancel out when performing the conversion).

Solidity only has integer division, which can introduce rounding errors. Division of unsigned integers is always rounded down to the next smaller integer. Therefore, if $\frac{x}{y}$ is the ideal result of a division (a real number), then the result of the division in Solidity (an unsigned integer) is $\left\lfloor \frac{x}{y} \right\rfloor = \frac{x}{y} - \epsilon$, where $0 \leq \epsilon < 1$.

Therefore, the actual exchange rate calculated by the contract is

$$\tilde{r} = \left\lfloor \frac{d \cdot 10^{18}}{s} \right\rfloor$$

$$= \left(\frac{d \cdot 10^{18}}{s} \right) - \epsilon_1$$

$$= r - \epsilon_1$$

and the converted savETH amount calculated is (approximately)

$$\tilde{x} = \left\lfloor \frac{a \cdot 10^{18}}{\tilde{r}} \right\rfloor$$

$$= \left\lfloor \frac{a \cdot 10^{18}}{r - \epsilon_1} \right\rfloor \qquad (0 \le \epsilon_1 < 1)$$

$$= \left(\frac{a \cdot 10^{18}}{r - \epsilon_1} \right) - \epsilon_2 \qquad (0 \le \epsilon_2 < 1)$$

$$\approx \left(\frac{(r + \epsilon_1) \cdot a \cdot 10^{18}}{r^2} \right) - \epsilon_2 \qquad (*)$$

$$= \left(\frac{a \cdot 10^{18}}{r} \right) + \left(\frac{\epsilon_1 \cdot a \cdot 10^{18}}{r^2} \right) - \epsilon_2$$

$$= x + \left(\frac{\epsilon_1 \cdot a \cdot 10^{18}}{r^2} \right) - \epsilon_2$$

$$= x + \left(\frac{s^2 \cdot \epsilon_1 \cdot a \cdot 10^{18}}{d^2 \cdot 10^{36}} \right) - \epsilon_2$$

$$= x + \left(\frac{s^2 \cdot \epsilon_1 \cdot a}{d^2 \cdot 10^{18}} \right) - \epsilon_2$$

The step marked $(*)$ above uses the binomial approximation, which states that $(1 + x)^n$ can be approximated by $1 + nx$ when $|x| < 1$ and $|nx| \ll 1$. This can be used to derive the following approximation of $\frac{1}{a + b} = (a + b)^{-1}$, if $\left|\frac{b}{a}\right| \ll 1$:

$$(a + b)^{-1} = \left(a \left(1 + \frac{b}{a} \right) \right)^{-1}$$

$$= a^{-1} \left(1 + \frac{b}{a} \right)^{-1}$$

$$\approx a^{-1}\left(1-\frac{b}{a}\right)$$

$$=\frac{a-b}{a^2}$$

This gives the approximation $\frac{1}{r-\epsilon_1} \approx \frac{r+\epsilon_1}{r^2}$ used in step $(*)$. Since $0 \leq \epsilon_1 < 1$ and r should be in the order of 10^{18}, the requirement $\left|-\frac{\epsilon_1}{r}\right| \ll 1$ should hold.

Therefore, we have that $\tilde{x} = x - \epsilon$, where

$$\epsilon \approx \epsilon_2 - \epsilon_1 \cdot a\frac{s^2}{d^2 \cdot 10^{18}}$$

Note that if the term $\epsilon_1 \cdot a\frac{s^2}{d^2 \cdot 10^{18}}$ is larger than ϵ_2, then ϵ will be negative, and therefore $\tilde{x} > x$. In that case, the user will receive more savETH than they should. Therefore, it's in our interest to bound the size of ϵ. To do this we consider the following worst-case bounds:

- $0 \leq \epsilon_2$
- $\epsilon_1 < 1$
- $\frac{s^2}{d^2} \leq 1$ (under the assumption that $s \leq d$)

We can use these bounds to bound the value of ϵ relative to a:

$$\epsilon \approx \epsilon_2 - \epsilon_1 \cdot a\frac{s^2}{d^2 \cdot 10^{18}}$$

$$> 0 - 1 \cdot a\frac{1}{10^{18}}$$

$$= -\frac{a}{10^{18}}$$

Therefore, a user taking advantage of the rounding errors to make a profit can get an amount of additional savETH of at most $\frac{1}{10^{18}}$ of the amount of dETH they spend.

Note that the assumption in the last bullet point above that $s \leq d$ is important. If it doesn't hold, then it adds a multiplicative factor to the magnitude of ϵ, making this attack potentially much more profitable. It is unclear whether the rounding errors themselves can make s surpass d, allowing the attack to be used to increase its own profitability. Because of this, it is recommended to change the conversion calculation so that the error is minimal and independent of a (see finding A11).

Converting savETH to dETH

To convert x savETH to the corresponding amount a in dETH, the savETHRegistry uses the formula

$$a = \frac{x \cdot r}{10^{18}}$$

where r is calculated as previously (and again subject to rounding errors). We can therefore calculate the actual converted amount of dETH calculated by the contract:

$$\begin{aligned}
\tilde{a} &= \left\lfloor \frac{x \cdot \tilde{r}}{10^{18}} \right\rfloor \\
&= \left\lfloor \frac{x \cdot (r - \epsilon_1)}{10^{18}} \right\rfloor && (0 \leq \epsilon_1 < 1) \\
&= \left(\frac{x \cdot (r - \epsilon_1)}{10^{18}} \right) - \epsilon_2 && (0 \leq \epsilon_2 < 1) \\
&= \left(\frac{x \cdot r}{10^{18}} \right) - \left(\frac{x \cdot \epsilon_1}{10^{18}} \right) - \epsilon_2 \\
&= a - \left(\frac{x \cdot \epsilon_1}{10^{18}} \right) - \epsilon_2
\end{aligned}$$

Therefore, the total calculation error is $\epsilon = x\frac{\epsilon_1}{10^{18}} + \epsilon_2$. Note that, unlike the opposite direction, in this case the error is always positive. This means that a user can never get more dETH than they should, only less. Therefore, the most profitable case for the user is when $\epsilon = 0$. On the other hand, the amount that the user can lose is bounded by

$$\epsilon = \left(\frac{x \cdot \epsilon_1}{10^{18}} \right) + \epsilon_2$$

$$< \left(\frac{x \cdot 1}{10^{18}}\right) + 1$$

$$= \left(\frac{x}{10^{18}}\right) + 1$$

C06: savETHRegistry invariants

We have used the Solidity SMTChecker to verify the following two invariants in a simplified version of the savETHRegistry contract:

Invariant: savETH supply is 0 if and only if dETH in the open index is 0.

Invariant: The savETH to dETH exchange rate never decreases.

The simplified contract is an executable version of our high-level model of the savETHRegistry, focusing on capturing the balance adjustments of each operation. To make the contract self-contained and not require external calls, the contract keeps track of dETH and savETH balances internally, rather than using external ERC20 token contracts.

Appendix 1: Simplified savETHRegistry

The following is the simplified version of the savETHRegistry contract used for checking the property described in C01:

```
pragma solidity 0.8.11;

contract savETHRegistry {
  uint256 public constant EXCHANGE_RATE_SCALE = 1e18;
  uint128 dETHUnderManagementInOpenPool;
  uint256 totalSavETHSupply; // Type consistent with ERC20 totalSupply()

  function setDETHUnderManagementInOpenPool(uint128 _amount) external {
    dETHUnderManagementInOpenPool = _amount;
  }

  function setTotalSavETHSupply(uint256 _amount) external {
    totalSavETHSupply = _amount;
  }

  function withdraw(uint128 _amount) external override {
    // Calculate how much dETH is owed to the user
    uint128 dETHFromExchangeRate = uint128((uint256(_amount) *
      saveETHToDETHExchangeRate()) / EXCHANGE_RATE_SCALE);

    assert(dETHFromExchangeRate <= dETHUnderManagementInOpenPool);

    // If the above assert is correct, the body of this if statement
    // will never execute

    if (dETHFromExchangeRate > dETHUnderManagementInOpenPool) {
      dETHFromExchangeRate = dETHUnderManagementInOpenPool;
    }

    // _assert_dETHEntryExitRule(dETHFromExchangeRate); // Not relevant

    // Likewise, if the assert is correct, this should never overflow
    dETHUnderManagementInOpenPool -= dETHFromExchangeRate;

    // Abstract away burning savETH and minting dETH
    totalSavETHSupply -= uint256(_amount);
```

```
    emit dETHWithdrawnFromOpenPool(dETHFromExchangeRate);
  }

  function saveETHToDETHExchangeRate()
      public override view returns (uint256) {
    if (totalSavETHSupply == 0) {
      return EXCHANGE_RATE_SCALE;
    }

    return sDivision(uint256(dETHUnderManagementInOpenPool),
                     totalSavETHSupply);
  }

  function sDivision(uint256 _numerator, uint256 _denominator)
      internal pure returns (uint256) {
    uint256 numeratorScaled = _numerator * 1e18;

    return numeratorScaled / _denominator;
  }
}
```