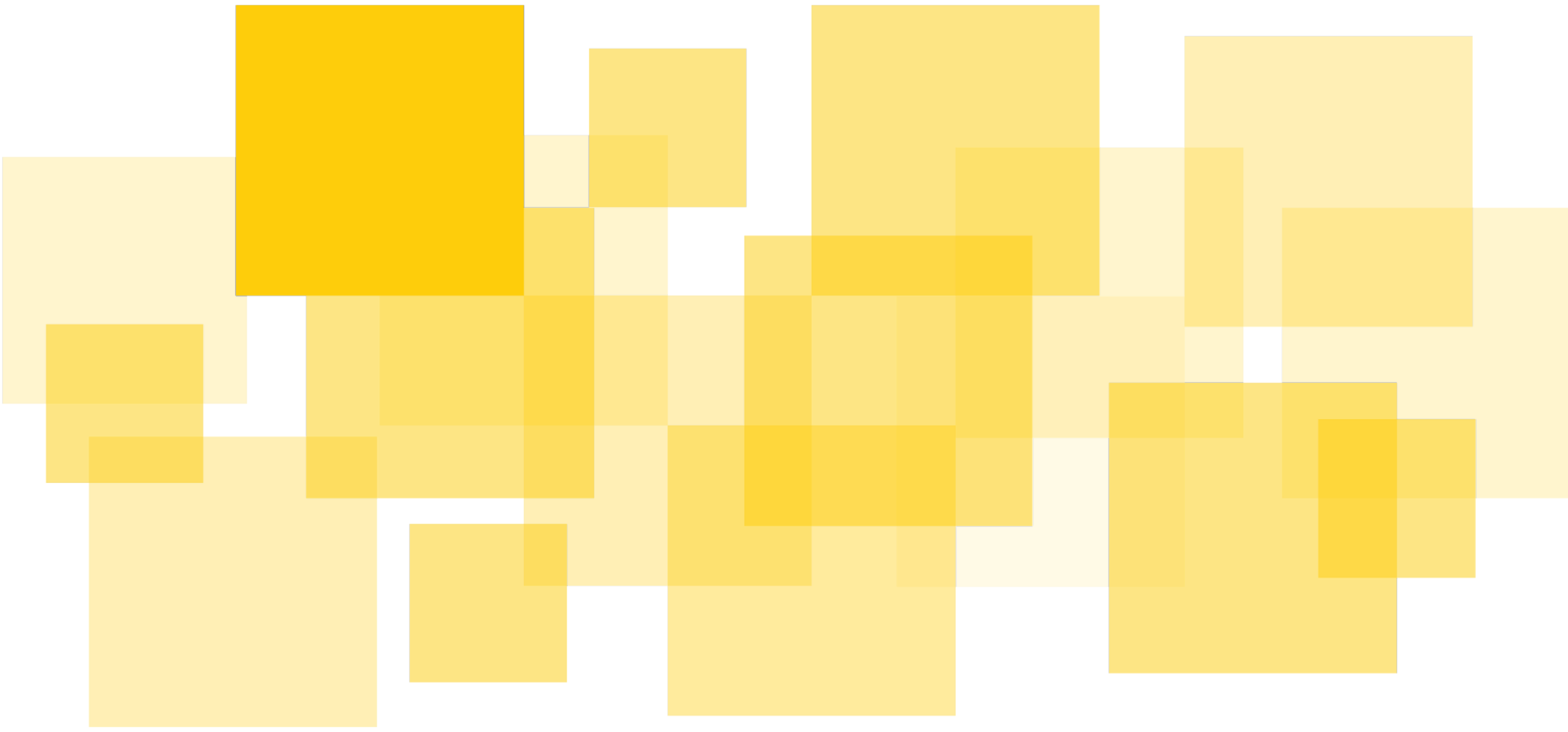
Audit Report

Blockswap Stakehouse Withdrawals

Delivered: 2023-06-29



Prepared for Blockswap by Runtime Verification, Inc.

runtime verification

Summary

Runtime Verification, Inc. has audited the source code of Blockswap's smart contracts that handle post-Shanghai withdrawals. The review was conducted from 2023-04-24 to 2023-05-12 and from 2023-05-29 to 2023-06-02, a total of 4 weeks.

Blockswap engaged Runtime Verification in checking the security of their withdrawal-related contracts. The newly added withdrawal mechanism in the Stakehouse protocol will allow users to claim their corresponding ETH sent from the consensus layer to the protocol.

The issues which have been identified can be found in the Findings section.

Scope

The audited smart contracts are:

- `withdrawals/ExitedUnknownSweeps.sol`
- `withdrawals/FullWithdrawals.sol`
- `withdrawals/PartialWithdrawals.sol`
- `withdrawals/QueueFlushETHClaim.sol`
- `withdrawals/ShanghaiSweepReporting.sol`
- `withdrawals/WithdrawalsDataStructures.sol`
- `accounts/BalanceReporter.sol`
- `accounts/AccountManager.sol (withdrawETH function)`
- `banking/savETHRegistry.sol (burnDETHRewards function)`

Although the audit has focused on the above smart contracts, possible issues within the rest of the protocol that may affect or be affected by the withdrawals are also investigated. Audit assumed the correctness of the libraries and external contracts that were made use of. The libraries are widely used and assumed to be secure and functionally correct.

The review encompassed `bswap-eng/Stakehouse-V2` private code repository. The code was frozen for review at commit `3e68bb7cecf256cfc7efd43bad4dfb339b270fd8`.

Blockswap also provided access to the `deposit-router` private code repository, which contains 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.

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. 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 the first audit report) are satisfied.
4. KNOTs who are penalized by more than 8 ETH 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.

For more details on these assumptions, see the previous audit reports.

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

This is the third time Runtime Verification has audited the Stakehouse protocol.[1]

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in the Disclaimer, we have used the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. Second, we carefully checked if the code was vulnerable to known security issues and attack vectors. Finally, we regularly met with the Blockswap team to provide feedback and suggested development practices and design improvements.

Additionally, we developed an invariant test suite for the dETH balance accounted by the protocol using the Foundry framework (see Appendix A - Withdrawals Invariant Testing). This invariant test suite allowed us to better conceptualize the code and aid in the search for unwanted behaviors. The test suite provided can be reused to include additional invariants than the ones already tested and be run on posterior iterations of the codebase.

[1] See the first and second audits for more information.

This report describes the **intended** behavior of the contracts under review and then outlines issues we have found, both in the intended behavior and in the ways the code differs from it. We also point out lesser concerns, deviations from best practices, and any other weaknesses we encounter.

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 real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Stakehouse Withdrawals Mechanism

This section describes the audited extension to the Stakehouse Protocol, the withdrawals mechanism. For a detailed description of the overall protocol, see the first and second audit reports.

The protocol extension mirrors the Shanghai and Chapella updates to the Ethereum execution and consensus layer, respectively, which enables ETH withdrawals from the consensus layer to the execution layer.
Since the execution layer recipient for ETH withdrawals of the Stakehouse validators is one of the protocol's smart contracts (the `AccountManager`), this extension seeks to distribute the received ETH correctly to their intended recipients.

What follows is a high-level description of the above-mentioned extension of the protocol, highlighting some of its important features and properties.

Consensus Layer Background

To better understand the audited extension, some background information on how the consensus layer articulates returning ETH to the execution layer is necessary.

Withdrawal Credentials

The withdrawal credential of a validator is a public key, which can either be a BLS public key or an ECDSA address. These are respectively referred to as 0x00 and 0x01 credentials due to the prefix they are stored with.
Credentials are set up in the first deposit to the deposit contract. Once an ECDSA address is set as a withdrawal credential, it cannot be changed. If a BLS public key is set as a credential, it can be changed to an ECDSA address later on, but only once. To change it, a signed message from the current BLS credential is required.

The withdrawal credentials for all validators in the Stakehouse protocol must be set to the address of the Account Manager contract.

Partial and Full Withdrawals

Following the Chapella upgrade, the consensus layer can now transfer ETH to the execution layer. Such transferences are called withdrawals and can be partial or full.
Partial withdrawals are the most common, they transfer all the validator's balance above 32ETH to the designated ECDSA credential. In contrast, full withdrawals transfer all of the validator's

balance. Final withdrawals will only occur when a validator has exited the consensus layer and is marked as withdrawable.

Withdrawal scheduling

The consensus layer withdrawals are executed in a push scheme. In each slot, in an increasing order based on the validators' indexes, 16 validators are checked for withdrawal eligibility. If they are eligible, the corresponding amount of ETH (partial or full) is sent to their corresponding ECDSA address.

For a validator to be eligible for withdrawal, it must have an ECDSA address as the withdrawal credential and satisfy one of the following:

- (Partial withdrawal): Be in active status and have an active balance higher than 32ETH
- (Full withdrawal): Have exited and become withdrawable, and have an active balance of more than 0ETH

Note that this push schedule implies that the time between two consecutive withdrawals for any given validator depends on the number of other withdrawal-qualified validators present in the network.

Stakehouse Withdrawals

With the above concepts about the consensus layer, we can better understand the protocol's mechanism for distributing the withdrawals.

All withdrawals are sent to the `AccountManager` contract. The following contracts provide the necessary logic to account for or withdraw such received ETH.

Sweep Reporting

The `ShanghaiSweepReporting` contract is the entry point for reporting sweeps (withdrawals). After a (partial) sweep has occurred, anyone can report it through this contract.
To report a withdrawal (or multiple), it must be verified by an off-chain component, the Deposit Router. The Deposit Router will take data provided by the users, query different nodes for cross-validation, and sign the provided data so that it can be submitted to the protocol.

There are two possible sources of income for a validator's balance: deposits and inflation rewards. Both are withdrawn without any label of what kind of income they represent. That is, a single sweep can indistinguishably contain ETH from both deposits to a validator and inflation rewards.
At the same time, the only source of validator income that should count towards dETH minting (dETH is a derivative token minted for the 24 of the staked 32 ETH by a validator, see

Blockswap Stakehouse first audit report) is inflation rewards. As an example, assume a validator's balance is 33.1 ETH with 1 ETH deposit + 0.1 ETH rewards of excess ETH. Then, the validator will be swept for an amount of 1.1 ETH. However, when reporting that sweep, no more than 0.1 dETH should be minted.
To avoid minting dETH from deposits to the validator, the protocol keeps track of all deposits made, and only mints dETH when all deposited ETH (above 32) has been reported via sweeps. In the above scenario, assume the 1.1 excess ETH was transferred in two sweeps, one of 1 ETH and another sweep of 0.1 ETH. If the 0.1 ETH sweep were reported before the 1 ETH deposit sweep, it shouldn't mint any dETH since the protocol would know that there is a 1 ETH unreported deposit sweep. The 0.1 dETH would be minted once the remaining sweep of 1 ETH is reported.

To report sweeps through the `ShanghaiSweepReporting` contract, the KNOT has to be in Active state[2]. Once the KNOT has set an exit epoch, all unreported sweeps will have to be reported through the `FullWithdrawals` contract.

Partial Withdrawals

The `PartialWithdrawals` contract allows users to withdraw the ETH that has been reported as inflation rewards. Note that swept ETH corresponding to deposits won't be withdrawable through this contract. Also, even if the `AccountManager` received a sweep with inflation rewards, until said inflation rewards are reported, it will not be possible to withdraw them as ETH.

Full Withdrawals

The `FullWithdrawals` contract is for distributing the remaining ETH corresponding to a validator that has already been withdrawn from the consensus layer.
Once a validator has reached the withdrawable epoch, its next sweep will send its active balance over to the `AccountManager`. To obtain all remaining ETH sent to the `AccountManager` from the withdrawn validator, the designated address upon ragequit must perform the claim. Such a claim must contain all unreported sweeps. These include all sweeps that were not reported when the validator was active and all sweeps (including the final one) that occurred after the exit epoch was set.
The amount claimed also includes any dETH unclaimed rewards and all deposits made for that validator.

After addressing A03: FullWithdrawals::reportFinalSweepAndWithdraw sets withdrawalAmount too high, the formula for computing the validator's final worth is

[2] After this audit, sweeps can be reported when a validator has exited the network. Reporting sweeps in this state will not mint any dETH.

`withdrawalAmount = finalSweepAmountWei + reportedNotWithdrawn + unreported + totalTopUps`.

Postmortem Withdrawals

Even if a validator has been withdrawn from the consensus layer and has had the final sweep in the protocol, it can still receive deposits. If a deposit occurs, setting the balance of a withdrawn validator from 0 ETH to (say) 1 ETH, a withdrawal will eventually occur. Such withdrawals will send that 1 ETH deposit to the `AccountManager`.
In cases like these, a designated DAO can reclaim the ETH via the `ExitedUnknownSweeps` contract and decide how to give it back to the sender. Such claims can only be made once the KNOT has been withdrawn from the protocol.

This contract could also be used to claim any unreported sweeps once the KNOT has been withdrawn.

New Slashing Mechanism

The slashing mechanism has been modified to accommodate withdrawals. Previously, any balance decrease could be slashed since there was only one way to decrease the balance of a validator. That is, by means of a penalty. However, now there can be lawful balance decreases by means of a withdrawal.
In light of this, the protocol now only allows reporting balance decreases when the resulting decreased balance is below 32 ETH.

Excess ETH / Flushed ETH

The protocol allows users to buy back (top-up) slashed SLOT. Said SLOT can be sent to the deposit contract. However, the Deposit Contract requires a minimum of 1 ETH per deposit. If the SLOT that has been bought back is less than 1 ETH, users can send the additional ETH to allow the protocol to make a deposit with the bought SLOT. This additional ETH, called excess ETH or ETH to flush the queue, can be claimed once it's back in the protocol.

However, as a result of addressing finding A05, the protocol doesn't have this behavior anymore. Rather, it is the node runner's responsibility to keep the effective balance at 32ETH by means of direct deposits, external to the protocol's bookkeeping.
Buying back slashed SLOT will have multiple benefits for SLOT holders (e.g., a share of MEV rewards and fees), and is still required to have 4 SLOT to ragequit a KNOT.

Findings

A01: `savETHRegistry::burnDETHRewards` does not decrease `knotDETHBalanceInIndex`

[Severity: High | Difficulty: Low | Category: Implementation Flaw]

In `savETHRegistry::burnDETHRewards()`, if the KNOT is in a private index, the code executed is

```
if (associatedSavETHIndex != 0) {
      require(indexIdToOwner[associatedSavETHIndex] == _owner,
                                              "Only index owner");
} else { ...
```

The value `knotDETHBalanceInIndex[index][validator]` should be decreased by the function input `_amount` in that `if` branch.

Since `burnDETHRewards()` doesn't decrease `knotDETHBalanceInIndex[index][validator]`, the dETH accounted for when moving the KNOT to the open index is bigger than it should be if the owner of the private index previously called `partialWithdrawals::unwrapDETH`. This is because the variable to know how much dETH is being moved into the open index is `knotDETHBalanceInIndex[index][validator]`, which is not updated when claiming rewards.

This wouldn't give excess ETH upon KNOT ragequitting. But it would cause imbalances in the ragequitting process since `savETHRegistry::_rageQuitKnotInIndex` performs the following computations:

```
totalDETHMintedWithinHouse[_stakeHouse] -=
                          knotDETHBalanceInIndex[indexIdForKnot][_memberId];
dETHMetadata.dETHInCirculation -=
                uint128(knotDETHBalanceInIndex[indexIdForKnot][_memberId]);
```

Recommendation

Decrease `knotDETHBalanceInIndex[index][validator]` by the function input `_amount` in the above-mentioned `if` branch.

Status

Addressed by the client.

A02: `BalanceReporter::_addTopUpToQueue` can try to deposit less than 1 ETH

[Severity: Low | Difficulty: Low | Category: Input validation]

When SLOT is being purchased, the transaction can have a `msg.value` higher than the `_amountOfSLOTPurchased`, intended to flush the queue. If the excess of ETH sent when purchasing SLOT is not enough to flush the queue (i.e., the amount of total SLOT bought plus the excess ETH is less than 1 ETH), the `_addTopUpToQueue` should not attempt to make a deposit.

However, `_addTopUpToQueue` is double-counting `_amountOfSLOTPurchased` when checking that enough excess ETH has been sent, thus possibly attempting to make a deposit with insufficient ETH.

```
require(msg.value >= _amountOfSLOTPurchased, "Few ETH");

stakeHouseMemberQueue[_blsPublicKey] += _amountOfSLOTPurchased;

uint256 queueAmount = stakeHouseMemberQueue[_blsPublicKey];

bool sendFundsToDepositContract;
uint256 excessETH;
uint256 specialExitFeeForKnot = specialExitFee[_blsPublicKey];
if (msg.value > _amountOfSLOTPurchased) {
      require(queueAmount + msg.value >= 1 ether, "Invalid ETH");
```

Scenario

1. Assume there's 0 SLOT in the queue (`stakeHouseMemberQueue[_blsPublicKey] = 0`) and there's 0.5 SLOT available for purchase
2. Alice calls `BalanceReporter::topUpSlashedSlot(stakeHouse, knot, address(alice), 0.5 ether)` with `msg.value = 0.5 ether + 1 gwei`
3. The amount of ETH in the queue is increased from 0 to 0.5: `stakeHouseMemberQueue[knot] = 0.5`
4. The local variable `queueAmount` is set to the current queue amount: `queueAmount = 0.5`
5. The `if` branch of line 392 is executed since `msg.value > _amountOfSLOTPurchased`
6. The `requires` clause of line 396 is satisfied since `queueAmount + msg.value = 0.5 ether + 0.5 ether + 1 gwei >= 1 ether`
7. In line 428, a deposit of `0.5 ether + 1 gwei` will fail since the deposit contract only allows transfers of at least one ETH

Recommendation

Require `queueAmount + excessETH >= 1 ether` instead of `queueAmount + msg.value >= 1 ether`, using `excessETH = msg.value - _amountOfSLOTPurchased` as set in the line below the requires clause.

Status

The mechanism to make deposits from the protocol has been removed. This implies the removal of the above-mentioned logic in the `_addTopUpToQueue` function. For more details on the simplifications to that function, see https://github.com/bswap-eng/Stakehouse-V2/blob/post-withdrawal-audit/contracts/accounts/BalanceReporter.sol#L274.

A03: `FullWithdrawals::reportFinalSweepAndWithdraw` sets `withdrawalAmount` too high

[Severity: High | Difficulty: Low | Category: Implementation Flaw]

The final amount of ETH sent to a user is computed as

```
uint256 withdrawalAmount = _finalSweep.sweep.amount * 1 gwei +
totalUnknownTopUpsNow + reported + unreported;
```

We have that `totalUnknownTopUpsNow = (_totalUnknownTopUps * 1 gwei)` where `_totalUnknownTopUps` is the sum of all deposits made for the validator in gwei.

Since all deposits made to the validator are accounted for in all final and non-final unreported sweeps, adding `totalUnknownTopUpsNow` will double-spend the deposits made for a validator.

Status

The formula has been updated to https://github.com/bswap-eng/Stakehouse-V2/blob/post-withdrawal-audit/contracts/withdrawals/FullWithdrawals.sol#L147.

A04: `BalanceReporter::multipartyRageQuit` deadlock if SLOT is owned by a contract

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

Currently, if a contract that is not the original SLOT owner is used to buy slashed SLOT, the `multipartyRageQuit` function cannot be used since a signature is required from SLOT owners different from the original one.

Status

An additional multiparty rage quit that will allow collateralized owners to claim ETH post ragequit has been introduced:
https://github.com/bswap-eng/Stakehouse-V2/blob/post-withdrawal-audit/contracts/accounts/BalanceReporter.sol#L165.

A05: Queue flush ETH (or, potentially, rewards) can be unclaimable

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

It is not guaranteed that a call to `QueueFlushETHClaim::claim` will always work when there is some unclaimed excess ETH.

If a validator is slashed down to 31.8 ETH and a user calls `BalanceReporter::topUpSlashedSlot` with `msg.value` = 1ETH and `_amount` = 0.1 ETH, we have that `TransactionRouter::totalExcessETHSentForSlotTopUpsForUser[user][validator]` = 0.9 ETH. However, since the active balance of the validator will be 32.8 ETH, only 0.8 ETH will make it back to the protocol.

Thus, a call to `QueueFlushETHClaim::claim(validator)` would either revert or take funds from other sources, such as rewards.

Note that as per B02: Not every penalty can be reported, node runners could be incentivized to try to leave some SLOT unbought, making it more probable to only partially buy slashed SLOT.

Scenario

Since penalties that occur with an adjusted balance of 32ETH cannot be reported, this could also impact the excess ETH accounting:

1. There's 0.1 slashed SLOT
2. 0.1 slashed SLOT bought with a 1 ETH deposit, 0.9 excess ETH
3. Before the sweep comes in, a leak of 0.2 ETH occurs
4. A sweep of 0.7 occurs
5. The protocol will try to give 0.9 ETH to the corresponding user

Here, the situation can be worse than not giving away rewards if there was also leakage. In the case of the rewards, they wouldn't be acknowledged by the protocol in the first place. Whereas here, the unreportable penalties can be silently transferred to other KNOTs or rewards.

Status

The mechanism to make deposits from the protocol has been removed. This implies the removal of the queue flush ETH return mechanism.

A06: `QueueFlushETHClaim::claim` insufficient wait period check

[Severity: Medium | Difficulty: Low | Category: Implementation Flaw]

The `QueueFlushETHClaim::claim` function enforces a delay between claims to ensure enough time has passed for the flushed ETH to arrive from the consensus layer to the protocol. However, the waiting period is computed as

```
require(lastClaim[msg.sender][_blsPublicKey]
                              + flushClaimDelay < block.timestamp, "Wait");
```

with `flushClaimDelay` = 16 days and `lastClaim` updated to the last timestamp in which `claim` was called.

This check may not prevent a user from claiming flushed ETH immediately after flushing the queue since the only requirement is that enough time has passed since the last claim of flushed ETH, not since the last time the queue was flushed. This means that a user could claim ETH that has not yet been transferred from the consensus layer to the protocol.

Status

The mechanism to make deposits from the protocol has been removed. This implies the removal of the queue flush ETH return mechanism.

A07: `TransactionManager::totalExcessETHSentForSlotTopUpsForUser` can be withdrawn twice

[Severity: High | Difficulty: Low | Category: Implementation Flaw]

The amount corresponding to `TransactionManager::totalExcessETHSentForSlotTopUpsForUser` can be withdrawn twice from the protocol due to the following reasons:

1. It is not deducted from the `_totalUnknownTopUps` in the `FullWithdrawals::reportFinalSweepAndWithdraw` function
2. If for some `user` and `blsKey` we have `TransactionManager::totalExcessETHSentForSlotTopUpsForUser[user][blsKey]` > 0, the function `QueueFlushETHClaim::claim(blsKey)` can be called by `user` at any point, even when the final withdrawal has occurred

Hence, a user can claim flushed ETH after the KNOT has made the final withdrawal, which already includes the flushed ETH. Or can claim the flushed ETH and later perform a final withdrawal, which would include the flushed ETH again.

Note that in light of B03: TransactionRouter::totalExcessETHSentForSlotTopUps and TransactionRouter::totalExcessETHSentForSlotTopUpsForUser can be arbitrarily increased, this could be exploited to withdraw virtually any possible amount of ETH.

Status

The mechanism to make deposits from the protocol has been removed. This implies the removal of the queue flush ETH return mechanism.

A08: Protocol deposits are double-deducted from the amount of rewards reported

[Severity: High | Difficulty: Low | Category: Implementation Flaw]

The function `shanghaiSweepReporting::_adjustSumOfSweepsAgainstTopUps` adjusts the reported sweep amount to account for direct deposits to the validator and deposits made by the protocol. However, if there are any deposits made by the protocol, said function double-subtracts these deposits to the reported sweep amount that counts towards inflation rewards.

The user-provided argument `_totalUnknownTopUps` is validated in the Deposit Router to be the sum of all deposits made for a validator minus the initial deposit.

Scenario

Assume `_totalUnknownTopUps` only contains deposits for the validator not made by the protocol. After subtracting protocol-unrelated deposits in lines 387-392, the `try-catch` block accounts for deposits made by the protocol. Notice that `transactionManager::getSumOfETHTopUpsAndExcessETHForBlsPublicKey` (called in line 399) contains the ETH used to flush the queue and the SLOT purchased. However, SLOT should never be contained in any sweep. Thus, by subtracting purchased SLOT from the swept amount, the protocol is canceling out past (or possibly future) rewards. An example:

1. A validator leaks 0.5 ETH (Active Balance 31,5 ETH)
2. Before purchasing any SLOT, gains 0.5 ETH in rewards (AB 32 ETH)
3. 0.5 slashed SLOT is purchased and 0.5 ETH used to flush the queue (AB 33 ETH)
4. A sweep of 1 ETH is performed. However, when reporting the sweep, it is adjusted twice:
    a. First, the 1 ETH sweep gets subtracted the difference between `_totalUnknownTopUps` and `totalSweepsReportedAgainstUnknownTopUpsForBlsPublicKey[_blsPublicKey]`. This renders the 0.5 ETH reward null
    b. Second, since the protocol made the deposit, a total of 1 ETH (`totalExcessETHSentForSlotTopUps[_blsPublicKey] + totalSlotTopUps[_blsPublicKey]`) worth of future rewards will be required to make up for the double accounting of the deposit made by the protocol

Recommendation

Only subtract the appropriate amount sent to the Deposit Contract that wasn't top-up SLOT.

Status

ETH from topping up slot is not sent to deposit contract anymore. For reference, see https://github.com/bswap-eng/Stakehouse-V2/blob/post-withdrawal-audit/contracts/withdrawals/ShanghaiSweepReporting.sol#L490.

A09: Ragequitting can be adversely deadlocked

[Severity: High | Difficulty: Medium | Category: Design]

Currently, the only way to ragequit for a KNOT with multiple SLOT owners is through the `BalanceReporter::multipartyRageQuit` function. The function requires the signature of all subsequent SLOT owners to perform the ragequit. Should a SLOT owner not sign for the KNOT ragequit, the KNOT would effectively be in a deadlock.
Moreover, SLOT owner signatures are required regardless of how small the owned SLOT amount is. Thus, a malicious actor could buy tiny leaks of SLOT and then ask for a ransom in exchange for the needed signature.

Status

An additional multiparty rage quit that will allow collateralized owners to claim ETH post ragequit will be introduced. For reference, see https://github.com/bswap-eng/Stakehouse-V2/blob/post-withdrawal-audit/contracts/accounts/BalanceReporter.sol#L165.

Informative findings

B01: `FullWithdrawals::reportFinalSweepAndWithdraw` check to identify final sweep

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

The function `reportFinalSweepAndWithdraw` doesn't currently make a check to have a higher assurance that the reported final sweep is indeed the sweep that contains the initial 32ETH deposit.

To aid in ruling out mistakes when reporting the final sweep, a requires clause in the lines of `_finalSweep.sweep.amount * 1 gwei >= 15ETH` could be added if the associated validator has not been slashed.

Status

A check has been introduced here https://github.com/bswap-eng/Stakehouse-V2/blob/post-withdrawal-audit/contracts/withdrawals/FullWithdrawals.sol#L129.

B02: Not every penalty can be reported

[Severity: - | Difficulty: - | Category: Design]

There are two states that a validator can be in for which, if it incurred in a leakage penalty, it would not be slashable.

1. `activeBalance > 32 ETH`: If a validator has more than 32 ETH and incurs a penalty that doesn't decrease its balance below 32, it won't be possible to report such a penalty. This is due to the adjusted active balance being capped at 32 ETH. If a validator leaks from 32.5 to 32.3, it won't be possible to slash 0.2 SLOT since, from the protocol perspective, the validator's adjusted balance is 32 ETH for both states.

   Moreover, if a validator leaks from 32.2 to 31.8, only 0.2 instead of 0.4 SLOT will be slashable.

2. `activeBalance < 32 ETH` acknowledged by the protocol: if a validator has an adjusted active balance of less than 32 ETH, there is no mechanism that allows an update for rewards to increase the adjusted balance between the current slashed value and 32 ETH (excluded). So if rewards are accrued and then leaked, that penalty won't be reportable.

   To exemplify it, assume a validator is slashed 0.5 ETH with an active balance of 32 ETH, so the current adjusted active balance is 31.5. If no SLOT is bought back and the queue flushed, the only possible update of the adjusted active balance is to go below 31.5 granularly or to jump directly to 32 ETH. This means that if a penalty of 0.2 ETH occurs when the consensus active balance is at 31.7, the adjusted active balance will remain at 31.5, rendering the penalty unreportable.

Status

A mechanism to increase the active balance, even if below 32 ETH, has been added here https://github.com/bswap-eng/Stakehouse-V2/blob/post-withdrawal-audit/contracts/accounts/BalanceReporter.sol#L42.

B03: `TransactionRouter::totalExcessETHSentForSlotTopUps` and `TransactionRouter::totalExcessETHSentForSlotTopUpsForUser` can be arbitrarily increased

[Severity: - | Difficulty: - | Category: Design]

A user can arbitrarily increase `TransactionRouter::totalExcessETHSentForSlotTopUps` and `TransactionRouter::totalExcessETHSentForSlotTopUpsForUser` by calling `BalanceReporter::topUpKNOT` at any point. There are two scenarios in which a user can top up a KNOT that serve no purpose to the protocol and would be safer to restrict (especially in light of A07: TransactionManager::totalExcessETHSentForSlotTopUpsForUser can be withdrawn twice):

1. Empty `stakeHouseMemberQueue[blsKey]`. If there is no SLOT bought back waiting in the queue to be flushed, topping up the KNOT `blsKey` would only increase the above-mention variables but would serve no real purpose (that is, sending the SLOT to the consensus layer).
2. It is also possible to flush the queue even when the KNOT is not in the `TOKENS_MINTED` state. If a KNOT has rage quitted or is withdrawn, it should not be possible to flash the queue either. Note that this case could be covered by addressing the first one.

Recommendation

Allow only to add enough excess ETH to arrive at the 1 ETH minimum deposit amount when there is ETH in the queue.

Status

The client will be removing `BalanceReporter::topUpKNOT` in line with stopping the ability to flush the queue. Additionally, `TransactionRouter::totalExcessETHSentForSlotTopUps` and `TransactionRouter::totalExcessETHSentForSlotTopUpsForUser` variables were only valid when we were flushing ETH to the deposit contract. Without the flush, it is not necessary to track these values.

B04: `FullWithdrawals::reportFinalSweepAndWithdraw` duplicated `userWithdrawn` check

[Severity: - | Difficulty: - | Category: Gas Optimization]

During `FullWithdrawals::reportFinalSweepAndWithdraw`, the check `universe.slotRegistry().userWithdrawn(msg.sender, _blsPublicKey)` is checked twice. First, just before the `universe.accountManager().markKnotAsWithdrawn(msg.sender, _blsPublicKey)` call, and second, during said call.

Recommendation

Remove the first check since said check is performed during the subsequent call.

Status

The check has been removed. For reference, see https://github.com/bswap-eng/Stakehouse-V2/blob/post-withdrawal-audit/contracts/withdrawals/FullWithdrawals.sol#L132.

Appendix A - Withdrawals Invariant Testing

An invariant test suite has been developed to test if the dETH balance accounting in the `savETHRegistry` contract has been affected negatively by the latest modifications to the protocol. The test suite is developed using the Foundry's invariant testing capability.

The developed invariant test basically tests if the savETH and dETH in circulation are accounted for properly. As mentioned earlier, dETH and savETH are derivative tokens that account for 24 of the 32 ETH staked by the participants.

dETH tokens are used to represent the 24 ETH of the staked amount as the KNOT associated with the staker moves between the open index and isolated index. dETH is accounted for (not actually minted) as reserves for isolated KNOTs and converted to savETH tokens once the KNOT moves to an open index. Additionally, dETH can also be accounted for as rewards received by a KNOT in the open index. For this case, received rewards are also accounted for as reserves in a common pool kept for KNOTs in the open index. dETH reserves are minted as dETH tokens during withdrawals after burning a corresponding amount of savETH tokens. Another option to withdraw is to directly move a KNOT to the open index and mint dETH, skipping any savETH minting/burning. savETH tokens are minted only when a KNOT is in an open index, and burned either during KNOT isolation or dETH withdrawal.

During the development of the invariant tests, all the movements of the dETH reserves and minted tokens, as summarized above, have been followed using different variables (including ghost variables). From a wider perspective, the following invariant is tested during the invariant tests:

```
INCOMING_DETH == DETH_IN_CIRCULATION + OUTGOING_DETH
```

An important assumption made during the invariant tests is that there exists a 1:1 exchange rate between dETH and ETH. On the other hand, the exchange rate between savETH and dETH tokens may vary.

Here, `INCOMING_DETH` represents the total amount of ETH deposited to the `savETHRegistry` contract as dETH reserves. This value keeps track of the reserve increase caused by the invocation of either `mintSaveETHBatchAndDETHReserves` or `mintDETHReserves` functions. These are the only two functions that cause an increase in dETH reserves.

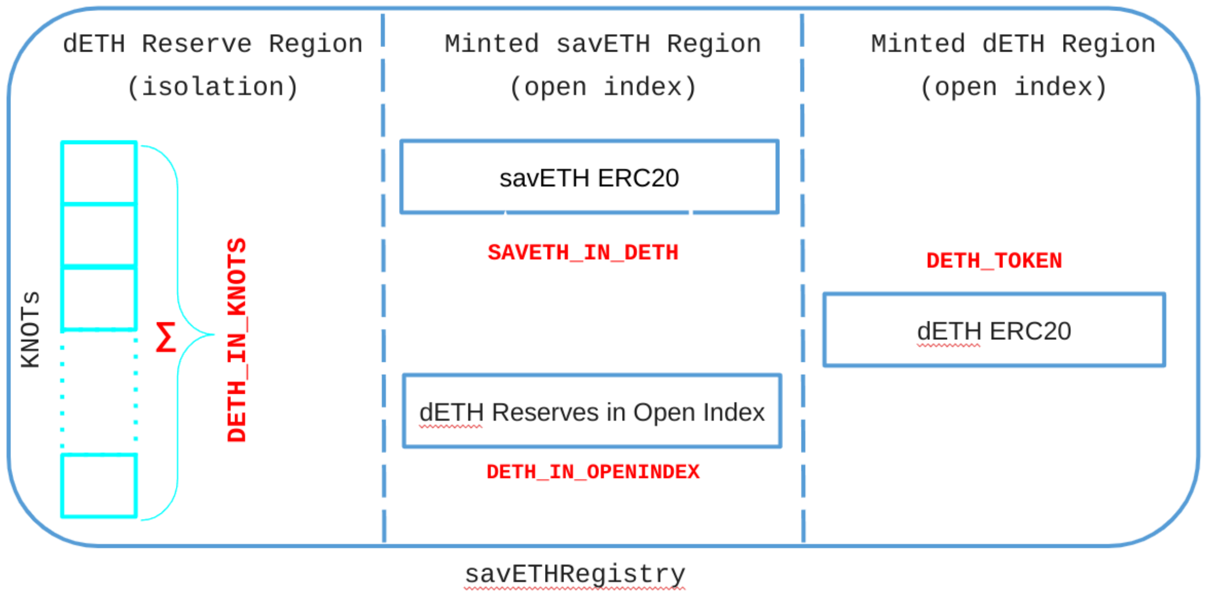
As the name suggests, `OUTGOING_DETH` represents the total amount of dETH withdrawn from the `savETHRegistry` contract. Withdrawals always burn dETH tokens that have previously been minted from the reserve pool. This value keeps track of the reserve increase caused by the invocation of either `_rageQuitKnotInIndex` or `burnDETHRewards` functions.

Finally, `DETH_IN_CIRCULATION` represents the total amount of dETH in circulation via various different forms, such as dETH reserves and minted dETH or savETH tokens. This value can be further broken down into sub-components as follows:

```
DETH_IN_CIRCULATION == DETH_IN_KNOTS + DETH_TOKEN + SAVETH_IN_DETH
                                                  + DETH_IN_OPENINDEX
```

Here, `DETH_IN_KNOTS` represents the total amount of dETH residing as reserves in the isolated KNOTs. `DETH_TOKEN` represents the total supply of minted dETH tokens. This value, for the sake of simplicity, is kept track of by the `token.totalSupply()` function in the tests. However, it may also be kept track of via a ghost variable increased/decreased by each mint/burn in case it may be possible to mint/burn/exchange dETH externally. The next variable is `SAVETH_IN_DETH` which keeps track of the total amount of savETH in circulation *in dETH units*. This is kept track of via a ghost variable which is incremented/decremented via each savETH mint/burn. The reason for this situation is the variable exchange rate between dETH and savETH over time. Finally, `DETH_IN_OPENINDEX` keeps track of the dETH reserves in the open index. It is possible to deposit to and withdraw from the KNOTs in the open index, so the reserves in the open index need to be accounted for as well.

The following figure represents the above-summarized concepts.



During the development of the invariant tests, Foundry's invariant testing capability have been used. Following the recommended practices by Foundry a handler class is implemented as a wrapper around the functionality offered by the `savETHRegistry` contract. The handler contract initializes a simplistic configuration for the steakhouse protocol with a single steakhouse and a varying number of stakers (which can be configured during initialization). In the handler, some housekeeping operations are performed, such as the selection of appropriate users via fuzzed variables, preparing basic sweeping reports, and staker states used while withdrawing and rage quitting operations, etc. Additionally, a mock signature validator contract has also been prepared and used during the tests that skip the signature verification and directly approve all the reports.

One important notice is the `check_*_constraints_and_prepare()` functions that basically check the constraints of the protocol's function-to-be-called and filter out the calls that would result in reverting of the protocol. Even though these constraints filter out quite a large amount of the function calls (sometimes up to 50%), they ensure that any expected reverts do not occur so that any reverts during the invariant tests indicate potential errors that are unaccounted for.

Developed invariant tests have been applied to the protocol under three different sets of operations. These sets are indicated as `DETH_ONLY`, `SAVETH_ONLY`, and `ALL in` the invariant testing contract. `DETH_ONLY` set only contains operations that change the overall dETH reserves in the contract and checks if the incoming ETH is equal to the outgoing ETH and the total amount of dETH being kept in the indexes. `SAVETH_ONLY` set does not perform any withdrawal and sweeps but contains movement between the isolation and open index and savETH operations as well. Finally, as the name suggests, `ALL` contains the combination of both sets. During the application of invariant sets, the appropriate sets of function selectors should be presented to the handler contract in order to properly check the invariant. For instance, for the `DETH_ONLY` set, any savETH exchange would not be accounted for and would not update the appropriate ghost variables resulting in improperly breaking the invariant.