hexens × PERSISTENCE

DEC.23

SECURITY REVIEW REPORT FOR PERSISTENCE

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered the new liquid stake module for pStake, part of the Persistence One blockchain, as well as the Superfluid LP smart contract. The liquid stake module is a new Go module for the Cosmos-based chain, which allows users to stake their XPRT. The Superfluid LP smart contract is deployed on the same chain using CosmWasm and interacts with the module to allow for Dexter liquidity providing.

Our security assessment was a full review of the liquid stake module and the Superfluid LP smart contract, spanning a total of 4 weeks.

During our audit, we have identified 1 critical severity vulnerability. The vulnerability would allow direct theft of user assets from the Superfluid LP smart contract.

We have also identified 2 high severity vulnerabilities.

Finally, all of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

SCOPE

The analyzed resources are located on:

https://github.com/persistenceOne/pstake-native/commit/384e05381584a80c05cea8e42a1a92af2e5b9fc3

https://github.com/dexter-zone/dexter_core/commit/555c31c39b7f2111e65b4f2bfe56dd86675fa8a5

The issues described in this report were fixed in the following commit:

https://github.com/persistenceOne/pstake-native/commit/f05ded769667bd5ee43f1e80c8d7d3487739df39

AUDITING DETAILS

STARTED
19.02.2023

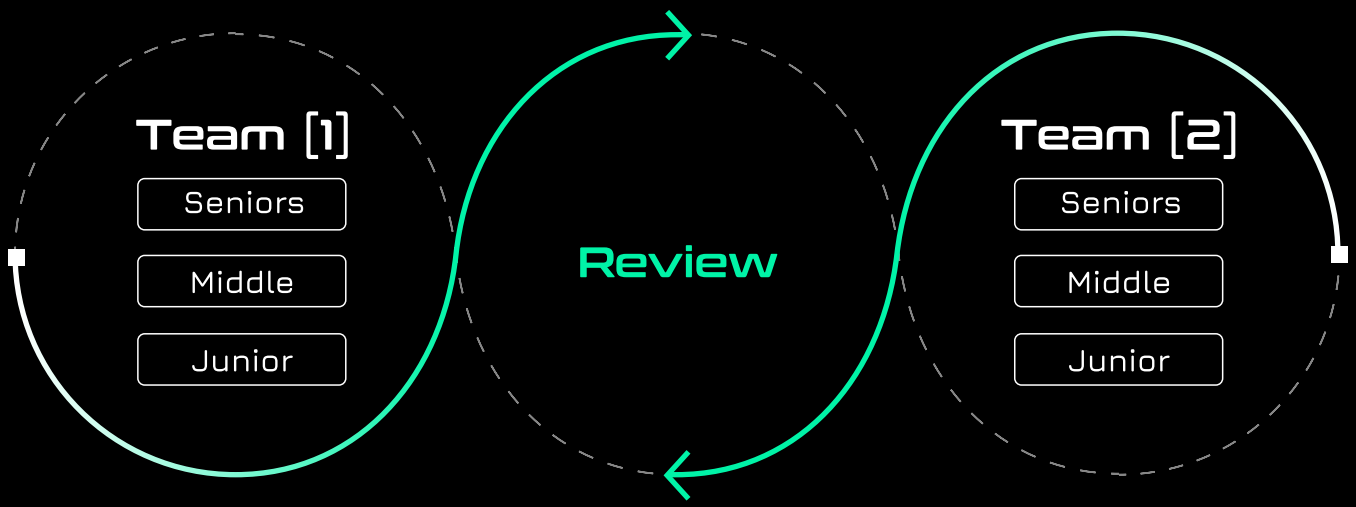
DELIVERED
12.01.2024

Review Led by

KASPER ZWIJSEN
Head of Smart Contract Audits | Hexens

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

Impact	Probability			
	rare	unlikely	likely	very likely
Low/Info	Low/Info	Low/Info	Medium	Medium
Medium	Low/Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

High

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

Medium

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

Low

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

Informational

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

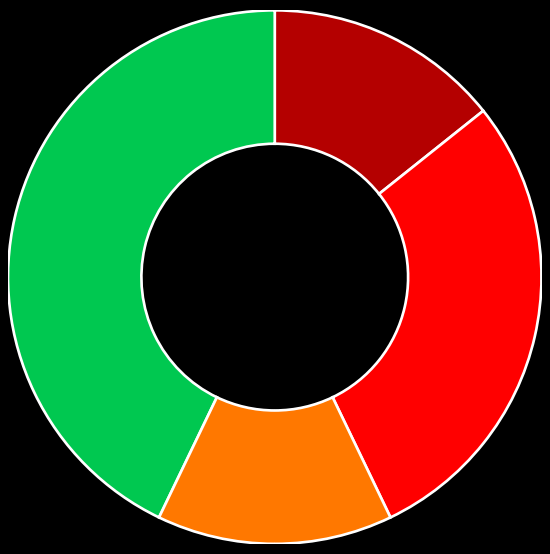
ISSUE SYMBOLIC CODES

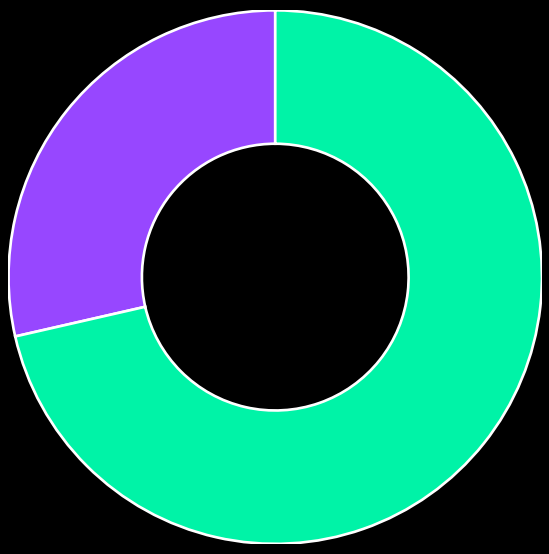
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	1
High	2
Medium	1
Low	3
Informational	0

Total: 7





WEAKNESSES

This section contains the list of discovered weaknesses.

PRST-5

USER CAN LOCK AND STAKE ARBITRARY AMOUNT OF TOKENS WITHOUT PAYING

SEVERITY: Critical

PATH:

contracts/superfluid_lp/src/contract.rs

REMEDIATION:

Add a check which enforces that the user sends the required token with the transaction.

STATUS: Fixed

DESCRIPTION:

In the **superfluid_lp/src/contract.rs** contract the user has the ability to lock and stake native tokens. While staking the user has the following methods of paying for the staking:

- Send the actual tokens with the transaction when staking
- Use their locked up tokens as payment, which will be transferred from the contract

The locking function has a faulty amount check **superfluid_lp/src/contract.rs:L95-109**:

```
match &asset.info {
   AssetInfo::NativeToken { denom } => {
       for coin in info.funds.iter() {
           if coin.denom == *denom {
               // validate that the amount sent is exactly equal to the amount
that is expected to be locked.
               if coin.amount != asset.amount {
                   return Err(ContractError::InvalidAmount);
               }
           }
       }
   }
   AssetInfo::Token { contract_addr: _ } => {
       return Err(ContractError::UnsupportedAssetType);
   }
}
```

The issue lies in the for loop part. If the user doesn't supply any tokens while calling the function, the **coin.amount** check will never happen and because the function doesn't check if the user has sent some tokens, the user locked amount will be incremented by the value that was supplied to the lock function. Even in the case where there was a requirement which enforced that the user has sent some tokens, the user could supply another native token and once again bypass the check.

Because of this the user can lockup any amount of tokens and later stake using other user's tokens without actually paying them which can lead to the following scenario:

- Alice locks up 100 native tokens inside of the contract to be later used in staking.
- Bob seeing as Alice has locked up native tokens without staking decides to abuse the invalid check and locks up 100 native tokens without actually paying those.
- Bob immediately after falsely locking the tokens, instantly joins the pool using his fake locked tokens and the contract transfers the locked tokens of Alice but from the name of Bob.

```
ExecuteMsg::LockLstAsset { asset} => {

            let user = info.sender.clone();

            // validate that the asset is allowed to be locked.
            let config = CONFIG.load(deps.storage)?;
            let mut allowed = false;
            for allowed_asset in config.allowed_lockable_tokens {
                if allowed_asset == asset.info {
                    allowed = true;
                    break;
                }
            }

            if !allowed {
                return Err(ContractError::AssetNotAllowedToBeLocked);
            }

            let mut locked_amount: Uint128 = LOCK_AMOUNT
                .may_load(deps.storage, (&user, &asset.info.to_string()))?
                .unwrap_or_default();

            // confirm that this asset was sent along with the message. We only
support native assets.
            match &asset.info {
                AssetInfo::NativeToken { denom } => {
                    for coin in info.funds.iter() {
                        if coin.denom == *denom {
                            // validate that the amount sent is exactly equal to
the amount that is expected to be locked.
                            if coin.amount != asset.amount {
                                return Err(ContractError::InvalidAmount);
                            }
                        }
                    }
                }
                AssetInfo::Token { contract_addr: _ } => {
                    return Err(ContractError::UnsupportedAssetType);
                }
            }
```

```
            // add the amount to the locked amount
            locked_amount = locked_amount + asset.amount;

            // update locked amount
            LOCK_AMOUNT.save(deps.storage, (&user, &asset.info.to_string()),
&locked_amount)?;
            Ok(Response::default())
        }
```

PRST-1

DIVISION BY ZERO IN SHARE CONVERSION LEADS TO PANIC AND DENIAL-OF-SERVICE

SEVERITY: High

PATH:

x/liquidstake/types/liquidstake.go:NativeTokenToStkXPRT, StkXPRTToNativeToken:L167-175

REMEDIATION:

Both functions should correctly handle the case where the denominator is zero in the same was that MintRate is calculated in GetNetAmountState.

STATUS: Fixed

DESCRIPTION:

Both functions **NativeTokenToStkXPRT** and **StkXPRTToNativeToken** are used to convert **xprt** to **stkxprt** and vice versa. The conversion is based on the total net amount of assets and the stkXPRT total supply.

However, neither function checks whether the denominator is zero, nor is this check done when either function is called.

This becomes a problem in the **keeper.LiquidStake** function when the user deposits **xprt** for **stkxprt**. Here **NativeTokenToStkXPRT** is called to calculate the amount of **stkxprt**.

If the protocol is in a state where there are some stkXPRT shares but no net assets, then a division by zero will happen. This will cause the message to revert with a Go run-time panic. This would make staking not possible.

```
// NativeTokenToStkXPRT returns StkxprtTotalSupply * nativeTokenAmount /
netAmount
func NativeTokenToStkXPRT(nativeTokenAmount, stkXPRTTotalSupplyAmount
math.Int, netAmount math.LegacyDec) (stkXPRTAmount math.Int) {
   return
math.LegacyNewDecFromInt(stkXPRTTotalSupplyAmount).MulTruncate(math.LegacyNe
wDecFromInt(nativeTokenAmount)).QuoTruncate(netAmount.TruncateDec()).Truncat
eInt()
}

// StkXPRTToNativeToken returns stkXPRTAmount * netAmount /
StkxprtTotalSupply with truncations
func StkXPRTToNativeToken(stkXPRTAmount, stkXPRTTotalSupplyAmount math.Int,
netAmount math.LegacyDec) (nativeTokenAmount math.LegacyDec) {
   return
math.LegacyNewDecFromInt(stkXPRTAmount).MulTruncate(netAmount).Quo(math.Lega
cyNewDecFromInt(stkXPRTTotalSupplyAmount)).TruncateDec()
}
```

PRST-6

FIRST DEPOSITOR CAN STEAL ASSETS DUE TO MISSING SLIPPAGE PROTECTION

SEVERITY: High

PATH:

x/liquidstake/keeper/liquidstake.go:LiquidStake

REMEDIATION:

We would recommend to also add a parameter containing the minimum amount of shares expected to be minted by the user. The amount would be calculated at the moment of signing and as a result, the transaction would fail if any manipulation would have happened before the moment of execution.

STATUS: Acknowledged, see commentary

DESCRIPTION:

The function to mint **stkXPRT** does not have any slippage protection. It only uses the input amount in **xprt**, but the output amount of minted shares is not checked, such as against a minimum amount out parameter. The output amount is only checked against zero, which is insufficient.

When a user deposits their **xprt** for **stkXPRT** the amount of minted shares are calculated from the exchange rate. The exchange rate is calculated from the total net asset amount and the total supply. The total net asset amount includes a **balanceOf** of **xprt**, which can be increased by an attacker to manipulate the share rate and cause rounding issues.

As a result, the first depositor can lose part of their funds to an attacker that front-runs their deposit message with the well-known deposit/donation attack.

Proof-of-concept:

```
func (s *KeeperTestSuite) TestFirstDepositAttack() {
    _, valOpers, _ := s.CreateValidators([]int64{1000000})
    params := s.keeper.GetParams(s.ctx)
    params.MinLiquidStakeAmount = math.NewInt(50000)
    s.keeper.SetParams(s.ctx, params)
    s.keeper.UpdateLiquidValidatorSet(s.ctx)

    params.WhitelistedValidators = []types.WhitelistedValidator{
        {ValidatorAddress: valOpers[0].String(), TargetWeight:
math.NewInt(1)},
    }
    s.keeper.SetParams(s.ctx, params)
    s.keeper.UpdateLiquidValidatorSet(s.ctx)

    s.advanceHeight(1, true)

    _, _, _ = s.keeper.LiquidStake(
        s.ctx, types.LiquidStakeProxyAcc, s.delAddrs[0],
        sdk.NewCoin(sdk.DefaultBondDenom, math.NewInt(50000)),
    )

    _, unbondingAmt, _, unbondedAmt, _ := s.keeper.LiquidUnstake(
        s.ctx, types.LiquidStakeProxyAcc, s.delAddrs[0],
        sdk.NewCoin(s.keeper.LiquidBondDenom(s.ctx), math.NewInt(49998)),
    )

    s.app.BankKeeper.SendCoins(
        s.ctx, s.delAddrs[0], types.LiquidStakeProxyAcc,
        []sdk.Coin{sdk.NewCoin(sdk.DefaultBondDenom,
math.NewInt(500000001))},
    )

    _, _, _ = s.keeper.LiquidStake(
        s.ctx, types.LiquidStakeProxyAcc, s.delAddrs[1],
        sdk.NewCoin(sdk.DefaultBondDenom, math.NewInt(500000000)),
    )

    s.advanceHeight(1, true)
```

```
    _, unbondingAmt, _, unbondedAmt, _ = s.keeper.LiquidUnstake(
        s.ctx, types.LiquidStakeProxyAcc, s.delAddrs[0],
        sdk.NewCoin(s.keeper.LiquidBondDenom(s.ctx), math.NewInt(2)),
    )
    fmt.Println("Attacker unbondingAmt", unbondingAmt)
    fmt.Println("Attacker unbondedAmt", unbondedAmt)

    _, unbondingAmt, _, unbondedAmt, _ = s.keeper.LiquidUnstake(
        s.ctx, types.LiquidStakeProxyAcc, s.delAddrs[1],
        sdk.NewCoin(s.keeper.LiquidBondDenom(s.ctx), math.NewInt(1)),
    )
    fmt.Println("Victim unbondingAmt", unbondingAmt)
    fmt.Println("Victim unbondedAmt", unbondedAmt)
}
```

Code Snippet:

```
func (k Keeper) LiquidStake(
   ctx sdk.Context, proxyAcc, liquidStaker sdk.AccAddress, stakingCoin
sdk.Coin) (newShares math.LegacyDec, stkXPRTMintAmount math.Int, err error) {
   [..]
   nas := k.GetNetAmountState(ctx)
   [..]
   // mint stkxprt, MintAmount = TotalSupply * StakeAmount/NetAmount
   liquidBondDenom := k.LiquidBondDenom(ctx)
   stkXPRTMintAmount = stakingCoin.Amount
   if nas.StkxprtTotalSupply.IsPositive() {
       stkXPRTMintAmount = types.NativeTokenToStkXPRT(stakingCoin.Amount,
nas.StkxprtTotalSupply, nas.NetAmount)
   }
   [..]
}

func (k Keeper) GetNetAmountState(ctx sdk.Context) (nas types.NetAmountState)
{
   totalRemainingRewards, totalDelShares, totalLiquidTokens :=
k.CheckDelegationStates(ctx, types.LiquidStakeProxyAcc)

   totalUnbondingBalance := sdk.ZeroInt()
   ubds := k.stakingKeeper.GetAllUnbondingDelegations(ctx,
types.LiquidStakeProxyAcc)
   for _, ubd := range ubds {
       for _, entry := range ubd.Entries {
           // use Balance(slashing applied) not InitialBalance(without
slashing)
           totalUnbondingBalance = totalUnbondingBalance.Add(entry.Balance)
       }
   }

   nas = types.NetAmountState{
       StkxprtTotalSupply:    k.bankKeeper.GetSupply(ctx,
k.LiquidBondDenom(ctx)).Amount,
       TotalDelShares:        totalDelShares,
       TotalLiquidTokens:     totalLiquidTokens,
       TotalRemainingRewards: totalRemainingRewards,
       TotalUnbondingBalance: totalUnbondingBalance,
       ProxyAccBalance:       k.GetProxyAccBalance(ctx,
types.LiquidStakeProxyAcc).Amount,
   }
```

```
    nas.NetAmount = nas.CalcNetAmount()
    nas.MintRate = nas.CalcMintRate()
    return
}

func NativeTokenToStkXPRT(nativeTokenAmount, stkXPRTTotalSupplyAmount
math.Int, netAmount math.LegacyDec) (stkXPRTAmount math.Int) {
    return
math.LegacyNewDecFromInt(stkXPRTTotalSupplyAmount).MulTruncate(math.LegacyNe
wDecFromInt(nativeTokenAmount)).QuoTruncate(netAmount.TruncateDec()).Truncat
eInt()
}
```

Commentary from the client:

“ - A parameter containing the minimum amount of shares expected to be minted by the user will be added in the next iterations, fixing the issue in the future.”

PRST-4

UNBONDING OF VALIDATORS DOES NOT GIVE PRIORITY TO INACTIVE VALIDATORS

SEVERITY: Medium

PATH:

x/liquidstake/keeper/liquidstake.go:LiquidUnstake:L344-459

REMEDIATION:

The function DivideByCurrentWeight could take the active status of each validator into account and use that to calculate the sum of all liquid tokens of inactive validators first and return the full amounts in the outputs. The leftover could then be taken from active validators pro rata.

STATUS: Fixed

DESCRIPTION:

When a user wants to withdraw their **stkXPRT** for **xprt**, they will call **LiquidUnstake**. In the function, the module will back out delegations for each validator according to their weight for a total of the unbonding amount. The module takes the whole set of validators and does not check their active status.

By not giving priority to unbonding inactive validators first, it will further lower the APY of staked XPRT.

```
func (k Keeper) LiquidUnstake(
 ctx sdk.Context, proxyAcc, liquidStaker sdk.AccAddress, unstakingStkXPRT
sdk.Coin,
) (time.Time, math.Int, []stakingtypes.UnbondingDelegation, math.Int, error)
{

 // check bond denomination
 params := k.GetParams(ctx)
 liquidBondDenom := k.LiquidBondDenom(ctx)
 if unstakingStkXPRT.Denom != liquidBondDenom {
  return time.Time{}, sdk.ZeroInt(), []stakingtypes.UnbondingDelegation{},
sdk.ZeroInt(), errors.Wrapf(
   types.ErrInvalidLiquidBondDenom, "invalid coin denomination: got %s,
expected %s", unstakingStkXPRT.Denom, liquidBondDenom,
  )
 }

 // Get NetAmount states
 nas := k.GetNetAmountState(ctx)

 if unstakingStkXPRT.Amount.GT(nas.StkxprtTotalSupply) {
  return time.Time{}, sdk.ZeroInt(), []stakingtypes.UnbondingDelegation{},
sdk.ZeroInt(), types.ErrInvalidStkXPRTSupply
 }

 // UnstakeAmount = NetAmount * StkXPRTAmount/TotalSupply * (1-
UnstakeFeeRate)
 unbondingAmount := types.StkXPRTToNativeToken(unstakingStkXPRT.Amount,
nas.StkxprtTotalSupply, nas.NetAmount)
 unbondingAmount = types.DeductFeeRate(unbondingAmount,
params.UnstakeFeeRate)
 unbondingAmountInt := unbondingAmount.TruncateInt()

 if !unbondingAmountInt.IsPositive() {
  return time.Time{}, sdk.ZeroInt(), []stakingtypes.UnbondingDelegation{},
sdk.ZeroInt(), types.ErrTooSmallLiquidUnstakingAmount
 }
```

```
 // burn stkxprt
 err := k.bankKeeper.SendCoinsFromAccountToModule(ctx, liquidStaker,
types.ModuleName, sdk.NewCoins(unstakingStkXPRT))
 if err != nil {
  return time.Time{}, sdk.ZeroInt(), []stakingtypes.UnbondingDelegation{},
sdk.ZeroInt(), err
 }
 err = k.bankKeeper.BurnCoins(ctx, types.ModuleName,
sdk.NewCoins(sdk.NewCoin(liquidBondDenom, unstakingStkXPRT.Amount)))
 if err != nil {
  return time.Time{}, sdk.ZeroInt(), []stakingtypes.UnbondingDelegation{},
sdk.ZeroInt(), err
 }

 liquidVals := k.GetAllLiquidValidators(ctx)
 totalLiquidTokens, liquidTokenMap := liquidVals.TotalLiquidTokens(ctx,
k.stakingKeeper, false)

 // if no totalLiquidTokens, withdraw directly from balance of proxy acc
 if !totalLiquidTokens.IsPositive() {
  if nas.ProxyAccBalance.GTE(unbondingAmountInt) {
   err = k.bankKeeper.SendCoins(
    ctx,
    types.LiquidStakeProxyAcc,
    liquidStaker,
    sdk.NewCoins(sdk.NewCoin(
     k.stakingKeeper.BondDenom(ctx),
     unbondingAmountInt,
    )),
   )
   if err != nil {
    return time.Time{}, sdk.ZeroInt(), []stakingtypes.UnbondingDelegation{},
sdk.ZeroInt(), err
   }

   return time.Time{}, sdk.ZeroInt(), []stakingtypes.UnbondingDelegation{},
unbondingAmountInt, nil
  }
```

```
  // error case where there is a quantity that are unbonding balance or
remaining rewards that is not re-stake or withdrawn in netAmount.
  return time.Time{}, sdk.ZeroInt(), []stakingtypes.UnbondingDelegation{},
sdk.ZeroInt(), types.ErrInsufficientProxyAccBalance
 }

 // fail when no liquid validators to unbond
 if liquidVals.Len() == 0 {
  return time.Time{}, sdk.ZeroInt(), []stakingtypes.UnbondingDelegation{},
sdk.ZeroInt(), types.ErrLiquidValidatorsNotExists
 }

 // crumb may occur due to a decimal error in dividing the unstaking stkXPRT
into the weight of liquid validators, it will remain in the NetAmount
 unbondingAmounts, crumb := types.DivideByCurrentWeight(liquidVals,
unbondingAmount, totalLiquidTokens, liquidTokenMap)
 if !unbondingAmount.Sub(crumb).IsPositive() {
  return time.Time{}, sdk.ZeroInt(), []stakingtypes.UnbondingDelegation{},
sdk.ZeroInt(), types.ErrTooSmallLiquidUnstakingAmount
 }

 totalReturnAmount := sdk.ZeroInt()

 var ubdTime time.Time
 ubds := make([]stakingtypes.UnbondingDelegation, 0, len(liquidVals))
 for i, val := range liquidVals {
  // skip zero weight liquid validator
  if !unbondingAmounts[i].IsPositive() {
   continue
  }

  var ubd stakingtypes.UnbondingDelegation
  var returnAmount math.Int
  var weightedShare math.LegacyDec
```

```
 // calculate delShares from tokens with validation
  weightedShare, err = k.stakingKeeper.ValidateUnbondAmount(ctx, proxyAcc,
val.GetOperator(), unbondingAmounts[i].TruncateInt())
  if err != nil {
   return time.Time{}, sdk.ZeroInt(), []stakingtypes.UnbondingDelegation{},
sdk.ZeroInt(), err
  }

  if !weightedShare.IsPositive() {
   continue
  }

  // unbond with weightedShare
  ubdTime, returnAmount, ubd, err = k.LiquidUnbond(ctx, proxyAcc,
liquidStaker, val.GetOperator(), weightedShare, true)
  if err != nil {
   return time.Time{}, sdk.ZeroInt(), []stakingtypes.UnbondingDelegation{},
sdk.ZeroInt(), err
  }

  ubds = append(ubds, ubd)
  totalReturnAmount = totalReturnAmount.Add(returnAmount)
 }

 return ubdTime, totalReturnAmount, ubds, sdk.ZeroInt(), nil
}
```

PRST-2

WHITELISTED VALIDATOR TARGET WEIGHT IS UNCAPPED AND UNITLESS

SEVERITY: Low

PATH:
x/liquidstake/types/params.go:validateWhitelistedValidators:
L101-126

REMEDIATION:

Implement some kind of cap and unit for the target weight. For example, 1 Dec would mean 1x normal stake for a validator and there could be a cap of 10x or 100x as desired (or configurable by Governance).

STATUS: Fixed

DESCRIPTION:

The function **validateWhitelistedValidators** will validate a new set of whitelisted validators as proposed by Governance through the UpdateParams message.

However, this function only checks whether the target weight is positive, the actual value is not checked or scaled against anything.

This makes it so that a validator's target weight by itself is meaningless, you have to compare it to the total target weight of all validators. Furthermore, the target weight is uncapped, so one validator with an enormous target weight would disable all other validators and take all power for themselves.

```
func validateWhitelistedValidators(i interface{}) error {
    wvs, ok := i.([]WhitelistedValidator)
    if !ok {
        return fmt.Errorf("invalid parameter type: %T", i)
    }
    valsMap := map[string]struct{}{}
    for _, wv := range wvs {
        _, valErr := sdk.ValAddressFromBech32(wv.ValidatorAddress)
        if valErr != nil {
            return valErr
        }

        if wv.TargetWeight.IsNil() {
            return fmt.Errorf("liquidstake validator target weight must not
be nil")
        }
        if !wv.TargetWeight.IsPositive() {
            return fmt.Errorf("liquidstake validator target weight must be
positive: %s", wv.TargetWeight)
        }

        if _, ok := valsMap[wv.ValidatorAddress]; ok {
            return fmt.Errorf("liquidstake validator cannot be duplicated:
%s", wv.ValidatorAddress)
        }
        valsMap[wv.ValidatorAddress] = struct{}{}
    }
    return nil
}
```

PRST-3

WHITELISTED VALIDATORS CANNOT BE INACTIVATED

SEVERITY: Low

PATH:
x/liquidstake/types/params.go:validateWhitelistedValidators:
L101-126

REMEDIATION:

Either the documentation should be amended to correctly state that inactive validators would not be included in the set, or the validation code should also allow a target weight of zero.

STATUS: Fixed

DESCRIPTION:

According to the documentation of a **WhitelistedValidator**, a validator is deemed inactive if their target weight is set to zero. This can be seen in x/**liquidstake/types/liquidstake.pb.go** on lines 122-125:

```
// WhitelistedValidator consists of the validator operator address and the
// target weight, which is a value for calculating the real weight to be derived
// according to the active status. In the case of inactive, it is calculated as
// zero.
```

However, this is not possible when updating the whitelisted validator set through **UpdateParams**. When it is validating the new set in **validateWhitelistedValidators** on lines 116-118:

```
if !wv.TargetWeight.IsPositive() {
   return fmt.Errorf("liquidstake validator target weight must be positive:
%s", wv.TargetWeight)
}
```

The check will return an error if the target weight is <= 0, isPositive only returns true if the value is greater than zero.

```
func validateWhitelistedValidators(i interface{}) error {
   wvs, ok := i.([]WhitelistedValidator)
   if !ok {
       return fmt.Errorf("invalid parameter type: %T", i)
   }
   valsMap := map[string]struct{}{}
   for _, wv := range wvs {
       _, valErr := sdk.ValAddressFromBech32(wv.ValidatorAddress)
       if valErr != nil {
           return valErr
       }

       if wv.TargetWeight.IsNil() {
           return fmt.Errorf("liquidstake validator target weight must not
be nil")
       }
       if !wv.TargetWeight.IsPositive() {
           return fmt.Errorf("liquidstake validator target weight must be
positive: %s", wv.TargetWeight)
       }

       if _, ok := valsMap[wv.ValidatorAddress]; ok {
           return fmt.Errorf("liquidstake validator cannot be duplicated:
%s", wv.ValidatorAddress)
       }
       valsMap[wv.ValidatorAddress] = struct{}{}
   }
   return nil
}
```

PRST-7

TOTAL UNBONDING AMOUNTS CAN BE GREATER THAN TOTAL LIQUID TOKENS IN UNSTAKE

SEVERITY: Low

PATH:

x/liquidstake/keeper/liquidstake.go:LiquidUnstake

REMEDIATION:

The LiquidUnstake function has a branch where the user receives the xprt from the module's balance, but this only happens if the totalLiquidTokens is 0 and there is enough balance. We would recommend to have the function first use any existing balance to covert the xprt amount and use the remaining value to unbond from validators.

STATUS: Acknowledged, see commentary

DESCRIPTION:

The function **LiquidUnstake** allows a user to unstake by burning their shares and receiving unbondings for **xprt**.

The amount of **xprt** is calculated using the **StkXPRTToNativeToken** function, which takes the net asset value from **GetNetAmountState**. Afterwards, the amount is divided among validators based on their liquid token amounts and the total liquid token amount using **DivideByCurrentWeight**.

However, the net asset value calculation uses the module's **xprt** balance, as well as unclaimed rewards in the total net amount, which is used to convert the user's shares to the **xprt** amount.

As a result, the **unbondingAmount** could be greater than **totalLiquidTokens** in the call to **DivideByCurrentWeight**.

In this case, too much would be unnecessarily unbonded from the validators or the execution would fail.

```
func (k Keeper) LiquidUnstake(
 ctx sdk.Context, proxyAcc, liquidStaker sdk.AccAddress, unstakingStkXPRT
sdk.Coin,
) (time.Time, math.Int, []stakingtypes.UnbondingDelegation, math.Int, error) {
 [..]
 // Get NetAmount states
 nas := k.GetNetAmountState(ctx)

 if unstakingStkXPRT.Amount.GT(nas.StkxprtTotalSupply) {
  return time.Time{}, sdk.ZeroInt(), []stakingtypes.UnbondingDelegation{},
sdk.ZeroInt(), types.ErrInvalidStkXPRTSupply
 }

 // UnstakeAmount = NetAmount * StkXPRTAmount/TotalSupply * (1-UnstakeFeeRate)
 unbondingAmount := types.StkXPRTToNativeToken(unstakingStkXPRT.Amount,
nas.StkxprtTotalSupply, nas.NetAmount)
 unbondingAmount = types.DeductFeeRate(unbondingAmount, params.UnstakeFeeRate)
 unbondingAmountInt := unbondingAmount.TruncateInt()
 [..]
 liquidVals := k.GetAllLiquidValidators(ctx)
 totalLiquidTokens, liquidTokenMap := liquidVals.TotalLiquidTokens(ctx,
k.stakingKeeper, false)
 [..]
 unbondingAmounts, crumb := types.DivideByCurrentWeight(liquidVals,
unbondingAmount, totalLiquidTokens, liquidTokenMap)
 if !unbondingAmount.Sub(crumb).IsPositive() {
  return time.Time{}, sdk.ZeroInt(), []stakingtypes.UnbondingDelegation{},
sdk.ZeroInt(), types.ErrTooSmallLiquidUnstakingAmount
 }

 totalReturnAmount := sdk.ZeroInt()

 var ubdTime time.Time
 ubds := make([]stakingtypes.UnbondingDelegation, 0, len(liquidVals))
 for i, val := range liquidVals {
  // skip zero weight liquid validator
  if !unbondingAmounts[i].IsPositive() {
   continue
  }
```

```
  var ubd stakingtypes.UnbondingDelegation
  var returnAmount math.Int
  var weightedShare math.LegacyDec

  // calculate delShares from tokens with validation
  weightedShare, err = k.stakingKeeper.ValidateUnbondAmount(ctx, proxyAcc,
val.GetOperator(), unbondingAmounts[i].TruncateInt())
  if err != nil {
   return time.Time{}, sdk.ZeroInt(), []stakingtypes.UnbondingDelegation{},
sdk.ZeroInt(), err
  }

  if !weightedShare.IsPositive() {
   continue
  }

  // unbond with weightedShare
  ubdTime, returnAmount, ubd, err = k.LiquidUnbond(ctx, proxyAcc, liquidStaker,
val.GetOperator(), weightedShare, true)
  if err != nil {
   return time.Time{}, sdk.ZeroInt(), []stakingtypes.UnbondingDelegation{},
sdk.ZeroInt(), err
  }

  ubds = append(ubds, ubd)
  totalReturnAmount = totalReturnAmount.Add(returnAmount)
 }

 return ubdTime, totalReturnAmount, ubds, sdk.ZeroInt(), nil
}

func DivideByCurrentWeight(lvs LiquidValidators, input math.LegacyDec,
totalLiquidTokens math.Int, liquidTokenMap map[string]math.Int) (outputs
[]math.LegacyDec, crumb math.LegacyDec) {
 if !totalLiquidTokens.IsPositive() {
  return []math.LegacyDec{}, sdk.ZeroDec()
 }

 totalOutput := sdk.ZeroDec()
 unitInput := input.QuoTruncate(math.LegacyNewDecFromInt(totalLiquidTokens))
 for _, val := range lvs {
  output :=
unitInput.MulTruncate(math.LegacyNewDecFromInt(liquidTokenMap[val.OperatorAddress
])).TruncateDec()
```

```
  totalOutput = totalOutput.Add(output)
  outputs = append(outputs, output)
 }

 return outputs, input.Sub(totalOutput)
}

func StkXPRTToNativeToken(stkXPRTAmount, stkXPRTTotalSupplyAmount math.Int,
netAmount math.LegacyDec) (nativeTokenAmount math.LegacyDec) {
 return
math.LegacyNewDecFromInt(stkXPRTAmount).MulTruncate(netAmount).Quo(math.LegacyNew
DecFromInt(stkXPRTTotalSupplyAmount)).TruncateDec()
}

func (k Keeper) GetNetAmountState(ctx sdk.Context) (nas types.NetAmountState) {
 totalRemainingRewards, totalDelShares, totalLiquidTokens :=
k.CheckDelegationStates(ctx, types.LiquidStakeProxyAcc)

 totalUnbondingBalance := sdk.ZeroInt()
 ubds := k.stakingKeeper.GetAllUnbondingDelegations(ctx,
types.LiquidStakeProxyAcc)
 for _, ubd := range ubds {
  for _, entry := range ubd.Entries {
   // use Balance(slashing applied) not InitialBalance(without slashing)
   totalUnbondingBalance = totalUnbondingBalance.Add(entry.Balance)
  }
 }

 nas = types.NetAmountState{
  StkxprtTotalSupply:    k.bankKeeper.GetSupply(ctx,
k.LiquidBondDenom(ctx)).Amount,
  TotalDelShares:        totalDelShares,
  TotalLiquidTokens:     totalLiquidTokens,
  TotalRemainingRewards: totalRemainingRewards,
  TotalUnbondingBalance: totalUnbondingBalance,
  ProxyAccBalance:       k.GetProxyAccBalance(ctx,
types.LiquidStakeProxyAcc).Amount,
 }

 nas.NetAmount = nas.CalcNetAmount()
 nas.MintRate = nas.CalcMintRate()
 return
}
```

Commentary from the client:

“ - We need to test if this edge case can ever be achieved, as the remediation will significantly complicate the logic of the rest of the module. So, we can mark as won’t fix for now but we will keep this in mind.”


hexens
×
PERSISTENCE