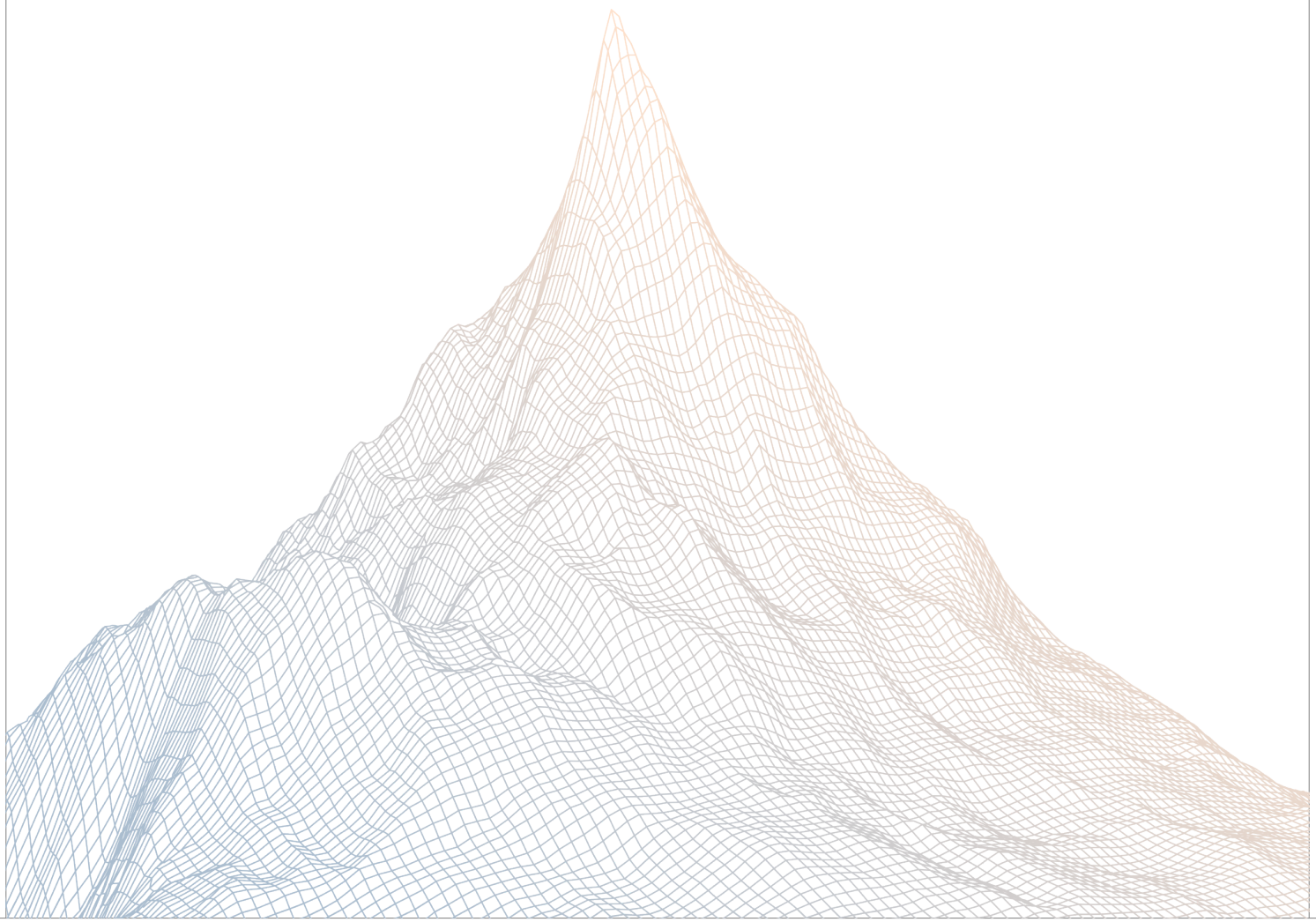
Zenith

Infrared

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: June 27th to July 7th, 2025

AUDITED BY: cccz
ether_sky

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Infrared

Infrared is focused on building infrastructure around the Proof of Liquidity (PoL) mechanism pioneered by Berachain. The protocol aims to maximize value capture by providing easy-to-use liquid staking solutions for BGT and BERA, node infrastructure, and vaults. Through building solutions around Proof of Liquidity (PoL), Infrared is dedicated to enhancing the user experience and driving the growth of the Berachain ecosystem.

2.2 Scope

The engagement involved a review of the following targets:

Target	infrared-contracts
Repository	https://github.com/infrared-dao/infrared-contracts
Commit Hash	`c4df171e000aae85f0352e5fad54d398e4230371`
Files	`Changes in PR #603`

2.3 Audit Timeline

June 27, 2025	Audit start
July 7, 2025	Audit end
July 5, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	2
Low Risk	9
Informational	4
Total Issues	**15**

3

Findings Summary

ID	Description	Status
M-1	InfraredBERAV2.burn() may fail due to insufficient confirmed deposit	Resolved
M-2	Storage conflict between InfraredBERAWithdrawor and InfraredBERAWithdraworLite	Resolved
L-1	In the registerViaProofs function, any increased stakes should be added to the deposits	Resolved
L-2	The minActivationDeposit check might be bypassed during the second deposit	Resolved
L-3	InfraredBERAWithdrawor.execute() may withdraw too much funds	Resolved
L-4	registerViaProofs() may block sweepForcedExit()	Resolved
L-5	InfraredBERAV2.maxWithdraw() does not consider burnFee	Resolved
L-6	InfraredBERARateProvider.getRate() always returns 0	Resolved
L-7	sweepUnaccountedForFunds() may not work	Resolved
L-8	The sweepForcedExit function is unnecessary in InfraredBERAWithdrawor	Acknowledged
L-9	The registerViaProofs function does not validate the provided validator address	Resolved
I-1	A zero address check should be added in the claimExitFees function	Resolved
I-2	Some proof length required are missing in BeaconRootsVerify	Acknowledged
I-3	sweepForcedExit() may be incompatible with Timelock Governor	Resolved
I-4	The burnFee must not be lower than MINIMUM_WITHDRAW_FEE	Acknowledged

4

Findings

4.1 Medium Risk

A total of 2 medium risk findings were identified.

[M-1] `InfraredBERAV2.burn()` may fail due to insufficient confirmed deposit

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- InfraredBERAWithdrawor.sol#L158-L160

Description:

When a user calls `InfraredBERAV2.burn()` to redeem iBera for withdrawal, `InfraredBERAWithdrawor.queue()` will be called to queue the user's withdrawal request.

```
function _withdraw(address receiver, uint256 amount)
    private
    returns (uint256 nonce)
{
    if (!_initialized) revert Errors.NotInitialized();

    // request to withdrawor contract to eventually forward to precompile
    nonce = IInfraredBERAWithdrawor(withdrawor).queue(receiver, amount);
    // update tracked deposits with validators *after* queue given used by
    withdrawor via confirmed
    deposits -= amount;
}
```

`InfraredBERAWithdrawor.queue()` requires the `amount` to be less than the confirmed deposit, which ensures that subsequent withdrawals from the validator meet the user's withdrawal needs.

```
if (amount == 0 || amount > IInfraredBERAV2(InfraredBERA).confirmed()) {
    revert Errors.InvalidAmount();
}
```

However, this check may DOS iBera redemption.

This check is used to ensure that the confirmed deposit must be able to cover the user's withdrawal request, but it needs to be taken into account that when the user's withdrawal request is more than the confirmed deposit, the user cannot withdraw directly from the Depositor's reserve and it requires the Keeper to deposit the Depositor's reserve to the validator first, and then make a withdraw from the validator. Since there are limits on the deposit(10,000 Bera initial deposit / 500,000 Bera second deposit), this may DOS iBera redemption.

Consider users with total deposits of 19,000 Bera and Keeper makes an initial deposit of 10,000 Bera to a validator.

The user calls `InfraredBERAV2.burn()` to initiate a withdrawal request of 11,000 Bera, the transaction will always fail, and the`InfraredBERAV2.burn()` call succeeds only when the deposit reaches 20,000 Bera and Keeper makes an initial 10,000 Bera deposit to a new validator.

Recommendations:

It is recommended to remove this check so that `InfraredBERAV2.burn()` always succeeds and the Keeper can then make deposits/withdrawals to satisfy the user's withdrawal request.

```
if (amount == 0 || amount > IInfraredBERAV2(InfraredBERA).confirmed()) {
    revert Errors.InvalidAmount();
}
```

Infrared: Resolved with PR-610.

Zenith: Verified.

[M-2] Storage conflict between InfraredBERAWithdrawor and InfraredBERAWithdraworLite

SEVERITY: Medium	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- InfraredBERAWithdrawor.sol

Description:

As illustrated in the comments, `InfraredBERAWithdrawor` is the upgraded version of `InfraredBERAWithdraworLite`.

- InfraredBERAWithdrawor.sol#L45

```
contract InfraredBERAWithdrawor is Upgradeable, IInfraredBERAWithdrawor {
    /// @dev unused storage vars from withdraworLite
    uint256 public nonceSubmit;
    uint256 public nonceProcess;
}
```

This can also be verified in the `UpgradeInfraredBERAWithdrawor.s` file.

- UpgradeInfraredBERAWithdrawor.s.sol#L28-L32

```
contract UpgradeInfraredBERAWithdrawor is Script {
    InfraredBERAWithdraworLite public withdraworLite;
    InfraredBERAWithdrawor public withdrawor;

    function run(address _withdraworLite, address _withdrawalPrecompile)
        external
    {
        withdraworLite
    = InfraredBERAWithdraworLite(payable(_withdraworLite));

        vm.startBroadcast();
        withdrawor = new InfraredBERAWithdrawor();

        // perform upgrade
```

```
        (bool success,) = address(withdraworLite).call(
            abi.encodeWithSignature(
                "upgradeToAndCall(address,bytes)", address(withdrawor), ""
            )
        );
        require(success, "Upgrade failed");

        // point at proxy
        withdrawor
    = InfraredBERAWithdrawor(payable(address(withdraworLite)));
        // initialize
        withdrawor.initializeV2(_withdrawalPrecompile);

        vm.stopBroadcast();
    }
}
```

Below is the `storage layout` of `InfraredBERAWithdraworLite`.

- InfraredBERAWithdraworLite.sol#L24-L31

```
contract InfraredBERAWithdraworLite is
    Upgradeable,
    IInfraredBERAWithdraworLite
{
    uint8 public constant WITHDRAW_REQUEST_TYPE = 0x01;
    address public WITHDRAW_PRECOMPILE; // @dev: EIP7002
    address public InfraredBERA;
    address public claimor;
    struct Request {
        address receiver;
        uint96 timestamp;
        uint256 fee;
        uint256 amountSubmit;
        uint256 amountProcess;
    }
    mapping(uint256 => Request) public requests;
    uint256 public fees;
    uint256 public rebalancing;
    uint256 public nonceRequest;
    uint256 public nonceSubmit;
    uint256 public nonceProcess;
    uint256[40] private __gap;
}
```

And below is the `storage layout` of `InfraredBERAWithdrawor`.

- InfraredBERAWithdrawor.sol#L24-L31

```
contract InfraredBERAWithdrawor is Upgradeable, IInfraredBERAWithdrawor {
    address public WITHDRAW_PRECOMPILE; // @dev: EIP7002
    address public InfraredBERA;
    mapping(uint256 ⇒ WithdrawalRequest) public requests;
    uint256 public requestLength;
    uint256 public requestsFinalisedUntil;
    uint256 public totalClaimable;
    uint256 public minActivationBalance;
    uint256 public nonceSubmit;
    uint256 public nonceProcess;
    uint256[40] private __gap;
}
```

There is a clear `storage collision` between the two.

I have checked the current deployed contract: https://berascan.com/address/0x8c0E122960dc2E97dc0059c07d6901Dce72818E1#readProxyC It seems that no values were set in `InfraredBERAWithdraworLite`. However, this upgrade introduces a `storage layout violation` between `implementation contracts`. As seen in the changes to `UpgradeInfraredBERAWithdrawor.sol`, the old `InfraredBERAWithdrawor` shares the same `storage layout` as `InfraredBERAWithdraworLite`. This inconsistency poses a long-term maintenance risk, as it will be difficult to manage and ensure correct storage alignment in future upgrades.

Recommendations:

Switch the order of the `requestLength` variable and the `requests` mapping.

```
contract InfraredBERAWithdrawor is Upgradeable, IInfraredBERAWithdrawor {
    address public WITHDRAW_PRECOMPILE; // @dev: EIP7002
    address public InfraredBERA;
   uint256 public requestLength;
    mapping(uint256 ⇒ WithdrawalRequest) public requests;
    uint256 public requestLength;
    uint256 public requestsFinalisedUntil;
    uint256 public totalClaimable;
    uint256 public minActivationBalance;
    uint256 public nonceSubmit;
    uint256 public nonceProcess;
    uint256[40] private __gap;
```

Infrared: Resolved with @f6d82fbd9d...

Zenith: Verified.

4.2 Low Risk

A total of 9 low risk findings were identified.

[L-1] In the registerViaProofs function, any increased stakes should be added to the deposits

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- InfraredBERAV2.sol

Description:

The `registerViaProofs` function allows the internal accounting of a `validator's stake` to be updated. In this function, the `validator's stake` is updated to a new `balance`, but the `deposits` value is not adjusted accordingly.

- InfraredBERAV2.sol#L540

```
function registerViaProofs(
    BeaconRootsVerify.BeaconBlockHeader calldata header,
    BeaconRootsVerify.Validator calldata _validator,
    bytes32[] calldata validatorMerkleWitness,
    bytes32[] calldata balanceMerkleWitness,
    uint256 validatorIndex,
    bytes32 balanceLeaf,
    uint256 nextBlockTimestamp
) external onlyKeeper {

    // set internal accounting balance to correct CL balance
    _stakes[_pubkeyHash] = _balance;

}
```

For example, suppose a total of `1,000,000 BERA` was deposited through `InfraredBERAV2`, and `200,000 BERA` was actually deposited to the `CL` for `validator V`. In this case:

- The reserves of `InfraredBERADepositorV2` would be `800,000`.
- The `confirmed()` function would return 200,000, which is correct.

InfraredBERAV2.sol#L397

```
function confirmed() external view returns (uint256) {
    uint256 _pending = pending();
    // If pending is greater than deposits, return 0 instead of underflowing
    return _pending > deposits ? 0 : deposits - _pending;
}
```

Now, assume that additional `stakes` for `validator V` (outside of the protocol) increase the `validator's total stake` to `450,000`. While this scenario is unlikely, it is theoretically possible. A keeper then updates the `validator's stake` using the `registerViaProofs` function.

Later, if the `validator` is exited and the full `450,000 stake` is transferred to `InfraredBERADepositorV2` via the `sweepForcedExit` function, an `confirmed()` function returns 0.

- InfraredBERAWithdrawor.sol#L529-L531

```
function sweepForcedExit(
    BeaconRootsVerify.BeaconBlockHeader calldata header,
    BeaconRootsVerify.Validator calldata validator,
    uint256 validatorIndex,
    bytes32[] calldata validatorMerkleWitness,
    uint256 nextBlockTimestamp
) external onlyGovernor {
    // register new validator delta
    IInfraredBERAV2(InfraredBERA).register(
        validator.pubkey, -int256(amount)
    );

    // re-stake amount back to ibera depositor

    InfraredBERADepositorV2(IInfraredBERAV2(InfraredBERA).depositor()).queue{
        value: amount
    }();
}
```

This happens because `deposits` remains at `1,000,000`, while the actual `reserves` become `1,250,000`. This inconsistency prevents `withdrawals` from the contract.

To prevent this, the increased stake should be reflected in the `deposits` value, and the additional stake should also be shared with depositors.

Recommendations:

```
function registerViaProofs(
    BeaconRootsVerify.BeaconBlockHeader calldata header,
    BeaconRootsVerify.Validator calldata _validator,
    bytes32[] calldata validatorMerkleWitness,
    bytes32[] calldata balanceMerkleWitness,
    uint256 validatorIndex,
    bytes32 balanceLeaf,
    uint256 nextBlockTimestamp
) external onlyKeeper {
    uint256 stake = _stakes[_pubkeyHash];
    // CL balance for validator (given in gwei)
    // CL balances are packed, 4 per bytes32 chunk. Offsets are index % 4.
    uint256 _balance = uint256(
        BeaconRootsVerify.extractBalance(balanceLeaf, validatorIndex % 4)
    ) * 1 gwei;

    // check internal balance versus cl balance
    if (stake == _balance) return;


    if (_balance > stake) {
        deposits += _balance - stake;
    }

    // set internal accounting balance to correct CL balance
    _stakes[_pubkeyHash] = _balance;
}
```

Infrared: Resolved with PR-611.

Zenith: Verified.

[L-2] The minActivationDeposit check might be bypassed during the second deposit

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- InfraredBERADepositorV2.sol

Description:

The `initial deposit` is made using the `executeInitialDeposit` function, with an amount of 10,000. The `minActivationDeposit` is set to 500,000 and is used to validate the amount of the `second deposit`.

- InfraredBERADepositorV2.sol#L56

```
function initializeV2() external onlyGovernor {
    // needs to be enough to guarentee activation (250k) + inclusion in
    active set (depends on competition)
    minActivationDeposit = 500_000 ether;
}
```

In the `execute` function, if the current staked amount is 10,000, the `deposit` is treated as a `second deposit`, and the `deposited amount` must exceed 500,000.

- InfraredBERADepositorV2.sol#L245-L249

```
function execute(
    BeaconRootsVerify.BeaconBlockHeader calldata header,
    BeaconRootsVerify.Validator calldata validator,
    bytes32[] calldata validatorMerkleWitness,
    bytes32[] calldata balanceMerkleWitness,
    uint256 validatorIndex,
    bytes32 balanceLeaf,
    uint256 amount,
    uint256 nextBlockTimestamp
) external onlyKeeper {
    // ensure second deposit is enough to meet active validator set
```

```
    if (stake == InfraredBERAConstants.INITIAL_DEPOSIT) {
        if (amount < minActivationDeposit) {
            revert Errors.DepositMustBeGreaterThanMinActivationBalance();
        }
    }
}
```

However, this check can be bypassed. For example, if an `external deposit` (outside of the protocol) increases the `validator's balance` from 10,000 to 20,000, this change won't be recognized unless the keeper updates the balance using the `registerViaProofs` function before calling `execute`.

If the balance is updated via `registerViaProofs` first, the internal state no longer shows 10,000, and the `minActivationDeposit` check is effectively bypassed, allowing the `execute` function to proceed without enforcing the intended minimum deposit constraint.

Recommendations:

The main goal is to ensure that the sum of the first two deposits exceeds the `minActivationDeposit`.

```
function execute(
    BeaconRootsVerify.BeaconBlockHeader calldata header,
    BeaconRootsVerify.Validator calldata validator,
    bytes32[] calldata validatorMerkleWitness,
    bytes32[] calldata balanceMerkleWitness,
    uint256 validatorIndex,
    bytes32 balanceLeaf,
    uint256 amount,
    uint256 nextBlockTimestamp
) external onlyKeeper {
    // ensure second deposit is enough
    // to meet active validator set
    if (stake == InfraredBERAConstants.INITIAL_DEPOSIT) {
        if (amount < minActivationDeposit) {
    if (stake < minActivationDeposit) {
      if (amount < minActivationDeposit - stake) {
       revert Errors.DepositMustBeGreaterThanMinActivationBalance();
        }
    }
}

function setMinActivationDeposit(uint256 _minActivationDeposit)
    external
```

```
    onlyGovernor
{
    if (
        _minActivationDeposit
            >= InfraredBERAConstants.MAX_EFFECTIVE_BALANCE
                - InfraredBERAConstants.INITIAL_DEPOSIT
    ) revert Errors.ExceedsMaxEffectiveBalance();
    minActivationDeposit = _minActivationDeposit;
    emit MinActivationDepositUpdated(_minActivationDeposit);
}
```

Infrared: Resolved with PR-617.

Zenith: Verified.

[L-3] `InfraredBERAWithdrawor.execute()` may withdraw too much funds

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- InfraredBERAWithdrawor.sol#L277-L279

Description:

When Keeper calls `InfraredBERAWithdrawor.execute()` to withdraw funds, the protocol limits the amount withdrawn to be less than the user's withdrawal needs.

```
if (_amount > (queuedAmount - _reserves + 1 gwei)) {
    revert Errors.InvalidAmount();
}
```

However, according to the documentation, the withdrawal will arrive 256 epochs (about 27 hours) after the withdrawal request is made, the Keeper can continue to call `InfraredBERAWithdrawor.execute()` to make withdrawal requests during this time, making the original restriction not work.

> Withdrawals occur at the end of the 256th epoch (~27 hours) after the epoch in which you perform the withdrawal.

Recommendations:

One option is to limit the calling rate of `InfraredBERAWithdrawor.execute()` to once every 256 epochs. Or since Keeper is trusted, just document the issue.

Infrared: Resolved with PR-613.

Zenith: Verified.

[L-4] `registerViaProofs()` may block `sweepForcedExit()`

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- InfraredBERAV2.sol#L548-L552

Description:

`sweepForcedExit()` can be called by the Governor to sweep the validators that are forced to exit and queue the refunded funds to the depositor. To prevent duplicate sweeps, it requires `hasExited(validator)` to be false.

```
function sweepForcedExit(
    BeaconRootsVerify.BeaconBlockHeader calldata header,
    BeaconRootsVerify.Validator calldata validator,
    uint256 validatorIndex,
    bytes32[] calldata validatorMerkleWitness,
    uint256 nextBlockTimestamp
) external onlyGovernor {
    ...
    // Check if validator has already exited
    if (IInfraredBERAV2(InfraredBERA).hasExited(validator.pubkey)) {
        revert Errors.ValidatorForceExited();
    ...

    InfraredBERADepositorV2(IInfraredBERAV2(InfraredBERA).depositor()).queue{
        value: amount
    }();
```

However, if the keeper calls `registerViaProofs()` on the forced exit validator to update the validator's state, it will make `hasExited(validator)` true because the balance is 0. This will prevent Governor from calling `sweepForcedExit()` to queue the refunded funds.

```
if (_balance == 0) {
    _staked[_pubkeyHash] = false;
    _exited[_pubkeyHash] = true;
}
```

Recommendations:

Since `registerViaProofs()` is only used to update the status of deposit bypassed validators, it is recommended to require that balance is not 0 in `registerViaProofs()`.

Infrared: Resolved with PR-613.

Zenith: Verified.

[L-5] `InfraredBERAV2.maxWithdraw()` does not consider burnFee

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- InfraredBERAV2.sol#L609-L617

Description:

`InfraredBERAV2.convertToAssets()` is used to convert shares to assets and is used in `maxWithdraw()` to calculate the maximum withdrawal amount for users.

```
function maxWithdraw(address owner) external view returns (uint256) {
    return withdrawalsEnabled ? convertToAssets(balanceOf(owner)) : 0;
}
...
function convertToAssets(uint256 shares)
    public
    view
    virtual
    returns (uint256)
{
    uint256 supply = totalSupply();
    return supply == 0 ? shares : shares * totalAssets() / supply;
}
```

However, since it does not consider `burnFee`, the user withdrawal amount is different from the return value.

Recommendations:

It is recommended to consider `burnFee` in `InfraredBERAV2.maxWithdraw()`

Infrared: Resolved with PR-611.

Zenith: Verified.

[L-6] `InfraredBERARateProvider.getRate()` always returns 0

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- InfraredBERARateProvider.sol#L22-L25

Description:

`InfraredBERARateProvider.getRate()` calculates the iBERA to BERA rate by calling `ibera.previewBurn(1 ether)`.

```
function getRate() external view override returns (uint256) {
    uint256 amount = ibera.previewBurn(1 ether);
    return amount;  }
```

In `InfraredBERAV2.previewBurn()`, the protocol always charges a fixed `burnFee` of 1 ether, which causes `InfraredBERARateProvider.getRate()` to always return 0.

```
} else {
    beraAmount = (depositsAfterCompound * (shareAmount - fee)) / ts;  }
```

Also, the burnFee for iBERA is fixed and not proportional, which makes the calculated rate inaccurate.

Recommendations:

Option 1: Make `previewBurn()` return `beraAmount` without `burnFee` and fixed `burnFee` respectively, and make `InfraredBERARateProvider.getRate()` return them as well.

Option 2: Charge a proportional burnFee.

Option 3: Use `convertToAssets()` to get the rate.

Infrared: Resolved with PR-609.

Zenith: Verified.

[L-7] `sweepUnaccountedForFunds()` may not work

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- InfraredBERAWithdrawor.sol#L536-L560

Description:

`sweepUnaccountedForFunds()` is used to handle excess stake that was refunded from a validator due to non-IBERA (bypass) deposits exceeding `MAX_EFFECTIVE_BALANCE`.

When implemented, it requires that the amount withdrawn must exceed the queued withdrawals, and since the keeper cannot make a withdraw from the validator that exceed the queued withdrawals, once new withdrawals are queued, these funds will be used as user withdrawals and cannot be withdrawn to receivor.

```
function sweepUnaccountedForFunds(uint256 amount) external onlyGovernor {
    // revert if amount exceeds balance available
    {
        uint256 _queue = getQueuedAmount();
        uint256 _reserves = reserves();
        if (_queue > _reserves || amount > _reserves - _queue) {
            revert Errors.InvalidAmount();
        }
    }

    address receivor = IInfraredBERAV2(InfraredBERA).receivor();
    // transfer amount to ibera receivor
    SafeTransferLib.safeTransferETH(receivor, amount);

    emit Sweep(receivor, amount);
}
```

Considering that the current queued withdrawals are 9000, 10000 bypassed deposits are refunded to withdrawor, Keeper cannot make any withdrawals at this time. 9000 of these 10000 will be used for user withdrawals, and only 1000 can be withdrawn by `sweepUnaccountedForFunds()`, and once a new 1000 withdrawal is queued, these 1000 cannot be withdrawn by `sweepUnaccountedForFunds()`.

Recommendations:

One option is to allow the Governor to withdraw an amount up to `reserves()` from the Withdrawor to the Depositor instead of the Receivor, on the one hand the Governor is trusted, and on the other hand withdrawing funds to the Depositor will not break the accounting in InfraredBERAV2.

Infrared: Resolved with @d678d471d6...

Zenith: Verified.

[L-8] The sweepForcedExit function is unnecessary in InfraredBERAWithdrawor

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- InfraredBERAWithdrawor.sol

Description:

`Validators` can be `forcibly withdrawn` to the `InfraredBERAWithdrawor`, and the `sweepForcedExit` function is responsible for sending these funds to the `InfraredBERADepositorV2`.

- InfraredBERAWithdrawor.sol#L529-L531

```
function sweepForcedExit(
    BeaconRootsVerify.BeaconBlockHeader calldata header,
    BeaconRootsVerify.Validator calldata validator,
    uint256 validatorIndex,
    bytes32[] calldata validatorMerkleWitness,
    uint256 nextBlockTimestamp
) external onlyGovernor {
    // re-stake amount back to ibera depositor

    InfraredBERADepositorV2(IInfraredBERAV2(InfraredBERA).depositor()).queue{
        value: amount
    }();

    emit Sweep(InfraredBERA, amount);
}
```

However, this function is unnecessary and may interfere with the execution of other functions.

Let's consider the following scenario:

- Users have submitted `withdrawal requests`, and the current `getQueuedAmount()` is `10,000`.

- The current `reserve` is `5,000`.
- A `validator` with `20,000` was `forcibly withdrawn` some time ago, and those funds will be arrived in the future.

Since the exact timing of that `withdrawal` is in the future, it's difficult for the `governor` to call `sweepForcedExit` at the right moment. Suppose those `withdrawn funds` eventually arrive in the `InfraredBERAWithdrawor`, increasing the `reserve` to `25,000`. At this point, keepers may try to call the `execute` function to fulfill `pending withdrawal requests`. However, this transaction will revert due to the following check:

- InfraredBERAWithdrawor.sol#L268-L270

```
function execute(
    BeaconRootsVerify.BeaconBlockHeader calldata header,
    BeaconRootsVerify.Validator calldata validator,
    bytes32[] calldata validatorMerkleWitness,
    bytes32[] calldata balanceMerkleWitness,
    uint256 validatorIndex,
    bytes32 balanceLeaf,
    uint256 amount,
    uint256 nextBlockTimestamp
) external payable onlyKeeper whenNotPaused {
    {
        uint256 queuedAmount = getQueuedAmount();
        uint256 _reserves = reserves() - msg.value;
        if (queuedAmount < _reserves) {
            revert Errors.ProcessReserves();
        }
    }
}
```

There are two problematic scenarios that can occur:

1. **Keepers call the *process* function using current reserves**, which reduces the `reserves` to `15,000`.

   Later, when the `governor` attempts to call `sweepForcedExit`, the call reverts due to insufficient `reserves`.

2. **The governor calls *sweepForcedExit* first**, which transfers funds and reduces the `reserves` to `5,000`.

   `Keepers` attempt to call the `process` function, assuming there are enough `reserves`—especially since a previous `execute` call reverted.

   However, now the `process` function itself also `reverts`.

It becomes very difficult to track which portion of the funds came from normal `execute` operations versus `forced withdrawals`. Also, the `keepers` can `rebalance` these `forced`

`withdrawal` amounts at any time using the `queue` function. Additionally, the `registerViaProofs` function of `InfraredBERAV2` correctly updates the `internal account balances`. This means there is **no risk** in removing the `sweepForcedExit` function entirely. Finally, as detailed in Issue #8, the `sweepForcedExit` and `process` functions may **block each other**, further complicating the system.

Recommendations:

The `sweepForcedExit` function could be removed.

Infrared: Acknowledged. We will leave the function in and make sure our keeper is closely monitoring forced exits before any other calls are made.

[L-9] The registerViaProofs function does not validate the provided validator address

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- InfraredBERAV2.sol

Description:

The `registerViaProofs` function does not verify whether the `validator` is an `Infrared validator` and the `initial deposit` was made through `InfraredBERADepositorV2`.

- InfraredBERAV2.sol#L513-L524

```
function registerViaProofs(
    BeaconRootsVerify.BeaconBlockHeader calldata header,
    BeaconRootsVerify.Validator calldata _validator,
    bytes32[] calldata validatorMerkleWitness,
    bytes32[] calldata balanceMerkleWitness,
    uint256 validatorIndex,
    bytes32 balanceLeaf,
    uint256 nextBlockTimestamp
) external onlyKeeper {
    // cache pubkey ref
    bytes32 _pubkeyHash = keccak256(_validator.pubkey);
    // internal accounting balance for validator
    uint256 stake = _stakes[_pubkeyHash];
    // CL balance for validator (given in gwei)
    // CL balances are packed, 4 per bytes32 chunk.  Offsets are index % 4.
    uint256 _balance = uint256(
        BeaconRootsVerify.extractBalance(balanceLeaf, validatorIndex % 4)
    ) * 1 gwei;

    // check internal balance versus cl balance
    if (stake == _balance) return;

    // check proof data is not stale
    if (block.timestamp > nextBlockTimestamp + proofTimestampBuffer) {
```

```
            revert Errors.StaleProof();
        }

        // verify stake amount againt CL via beacon roots proof
        if (
            !BeaconRootsVerify.verifyValidatorBalance(
                header,
                balanceMerkleWitness,
                validatorIndex,
                _balance,
                balanceLeaf,
                nextBlockTimestamp
            )
        ) {
            revert Errors.BalanceMissmatch();
        }

        // verify validator againt CL via beacon roots proof
        if (
            // note: beaconroots call above, so we can now internally verify
        against state root
            !BeaconRootsVerify.verifyValidator(
                header.stateRoot,
                _validator,
                validatorMerkleWitness,
                validatorIndex
            )
        ) {
            revert Errors.InvalidValidator();
        }

        // set internal accounting balance to correct CL balance
        _stakes[_pubkeyHash] = _balance;

        // update whether have staked to validator before
        if (_balance > 0 && !_staked[_pubkeyHash]) {
            _staked[_pubkeyHash] = true;
        }

        // only 0 if validator was force exited
        if (_balance == 0) {
            _staked[_pubkeyHash] = false;
            _exited[_pubkeyHash] = true;
        }

        emit RegisterViaProof(_validator.pubkey, _balance, stake);
    }
```

It only checks the `validator's balance` and existence in the `CL`. As a result, **any validator** can potentially be added using this function.

Recommendations:

The following check would be sufficient to prevent this:

```
function registerViaProofs(
    BeaconRootsVerify.BeaconBlockHeader calldata header,
    BeaconRootsVerify.Validator calldata _validator,
    bytes32[] calldata validatorMerkleWitness,
    bytes32[] calldata balanceMerkleWitness,
    uint256 validatorIndex,
    bytes32 balanceLeaf,
    uint256 nextBlockTimestamp
) external onlyKeeper {
    // cache pubkey ref
    bytes32 _pubkeyHash = keccak256(_validator.pubkey);
    // internal accounting balance for validator
    uint256 stake = _stakes[_pubkeyHash];

    if (stake == 0) revert();
}
```

A non-zero `stake` indicates that the `validator` made its `initial deposit` through `InfraredBERADepositorV2` and is therefore a `valid Infrared validator`.

Infrared: Resolved with @c53a2bb0a7f...

Zenith: Verified.

4.3 Informational

A total of 4 informational findings were identified.

[I-1] A zero address check should be added in the claimExitFees function

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- InfraredBERAV2.sol

Description:

The `claimExitFees` function does not include a zero address check.
InfraredBERAV2.sol#L438

```
function claimExitFees(address to) external onlyGovernor {
    uint256 amount = exitFeesToCollect;
    delete exitFeesToCollect;
    _transfer(address(this), address(to), amount);
    emit ExitFeesCollected(amount, to);
}
```

Recommendations:

```
function claimExitFees(address to) external onlyGovernor {
    if (to == address(0)) revert Errors.ZeroAddress();

    uint256 amount = exitFeesToCollect;
    delete exitFeesToCollect;
    _transfer(address(this), address(to), amount);
    emit ExitFeesCollected(amount, to);
}
```

Infrared: Resolved with PR-612.

Zenith: Verified.

[I-2] Some proof length required are missing in BeaconRootsVerify

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- BeaconRootsVerify.sol#L219-L224
- BeaconRootsVerify.sol#L249-L254
- BeaconRootsVerify.sol#L293-L298

Description:

When the protocol calculates `balancesListRoot` and `validatorListRoot`, the maximum length of `Validators` and `Balances` in the `BeaconState` structure is clearly specified, so the height of the Merkle tree and the length of proofs required are determined.

```
uint256 public constant VALIDATOR_PROOF_DEPTH = 41;

/**
 * @notice Beacon state balance proof depth in list container
 */
uint256 public constant BALANCE_PROOF_DEPTH = 39;
```

Similarly, when calculating `header.stateRoot` through `balancesListRoot`/`validatorListRoot` and `proofs`, since the length of the `BeaconState` structure is fixed (17, see state.go#L34-L69)), the height of their Merkle tree will be 5, that is, the length of the proof should be 5.

Recommendations:

It is recommended to make the proof of `validatorListRoot` and `balancesListRoot` to `header.stateRoot` have a length of 5.

Infrared: Acknowledged.

[I-3] `sweepForcedExit()` may be incompatible with Timelock Governor

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- InfraredBERAWithdrawor.sol#L477-L489

Description:

`sweepForcedExit()` requires the Governor to provide a time-sensitive proof to call it. When the Governor is a time-locked contract, `sweepForcedExit()` will fail to call due to expired proof.

```
function sweepForcedExit(
    BeaconRootsVerify.BeaconBlockHeader calldata header,
    BeaconRootsVerify.Validator calldata validator,
    uint256 validatorIndex,
    bytes32[] calldata validatorMerkleWitness,
    uint256 nextBlockTimestamp
) external onlyGovernor {
    // check proof data is not stale
    if (
        block.timestamp
            > nextBlockTimestamp
                + IInfraredBERAV2(InfraredBERA).proofTimestampBuffer()
    ) revert Errors.StaleProof();
```

Recommendations:

It is recommended to remove the time check from `sweepForcedExit()`.

Infrared: Resolved with PR-614.

Zenith: Verified.

[I-4] The burnFee must not be lower than MINIMUM_WITHDRAW_FEE

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- InfraredBERAV2.sol

Description:

The initial `burnFee` is set to `MINIMUM_WITHDRAW_FEE` in the `initializeV2` function.

- InfraredBERAV2.sol#L123

```
function initializeV2() external onlyGovernor {
    withdrawalsEnabled = true;
    burnFee = InfraredBERAConstants.MINIMUM_WITHDRAW_FEE;
    proofTimestampBuffer = 10 minutes;
}
```

As the name suggests, `MINIMUM_WITHDRAW_FEE` represents the `minimum allowable burn fee`. However, the `updateBurnFee` function does not enforce this constraint—there is no check to prevent setting a `burnFee` below the minimum.

- InfraredBERAV2.sol#L167-L170

```
function updateBurnFee(uint256 _fee) external onlyGovernor {
    burnFee = _fee;
    emit BurnFeeUpdated(_fee);
}
```

Recommendations:

```
function updateBurnFee(uint256 _fee) external onlyGovernor {
    if (burnFee < InfraredBERAConstants.MINIMUM_WITHDRAW_FEE) revert();
    burnFee = _fee;
```

```
        emit BurnFeeUpdated(_fee);
    }
```

Infrared: Acknowledged.