Blast Contracts Security Review

Auditors

Christoph Michel, Lead Security Researcher

Desmond Ho, Lead Security Researcher

Milo Truck, Security Researcher

Csanuragjain, Security Researcher

Report prepared by: Lucas Goiriz

January 29, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Blast is the only Ethereum L2 with native yield for ETH and stablecoins. Blast yield comes from ETH staking and RWA protocols. The yield from these decentralized protocols is passed back to Blast users automatically. The default interest rate on other L2s is 0%. On Blast, it's 4% for ETH and 5% for stablecoins.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of blast-contracts according to the specific commit. Any modifications to the code will require a new security review.

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

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 10 days in total, Blast engaged with Spearbit to review the blast-contracts protocol. In this period of time a total of **77** issues were found.

Summary

Project Name	Blast
Repository	blast-contracts
Commit	be5187...601c32
Type of Project	L2, Yield
Audit Timeline	Jan 1 to Jan 11

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	8	0	0
High Risk	6	0	0
Medium Risk	11	0	0
Low Risk	14	0	0
Gas Optimizations	7	0	0
Informational	31	0	0
Total	**77**	**0**	**0**

5 Findings

5.1 Critical Risk

5.1.1 `msg.sender` has to be un-aliased in `L2BlastBridge.finalizeBridgeETHDirect()`

Severity: Critical Risk

Context: L2BlastBridge.sol#L48

Description: `L2BlastBridge.finalizeBridgeETHDirect()` checks that `msg.sender` is `L1BlastBridge`:

```
require(msg.sender == address(OTHER_BRIDGE), "StandardBridge: function can only be called from the
↪  other bridge");
```

However, since the function is called directly as a L1 → L2 transaction initiated by `L1BlastBridge`, which is a contract, `msg.sender` will actually be the aliased address of `L1BlastBridge`. `OTHER_BRIDGE` is the un-aliased address of `L1BlastBridge` as other functions in `L2BlastBridge` use it, such as `_initiateBridgeETH()`.

As such, this check will always revert, thus users who deposit stETH in `L1BlastBridge` will not receive ETH on L2.

Recommendation: Un-alias `msg.sender` in the check as such:

```
- require(msg.sender == address(OTHER_BRIDGE), "StandardBridge: function can only be called from the
↪  other bridge");
+ require(AddressAliasHelper.undoL1ToL2Alias(msg.sender) == address(OTHER_BRIDGE), "StandardBridge:
↪  function can only be called from the other bridge");
```

5.1.2 `L1BlastBridge` uses wrong token order when bridging USD yield tokens

Severity: Critical Risk

Context: L1BlastBridge.sol#L196-L197, StandardBridge.sol#L265-L267

Description: The `_initiateBridgeERC20` function encodes the USD bridge message as:

```
messenger.sendMessage(
    Predeploys.L2_BLAST_BRIDGE,
    abi.encodeWithSelector(
        StandardBridge.finalizeBridgeERC20.selector,
        address(USDConversions.DAI),
        Predeploys.USDB,
        _from,
        _to,
        amountToMintWad,
        _extraData
    ),
    _minGasLimit
);
```

The `DAI` and `USDB` tokens need to be swapped. Currently, the L2 `L2BlastBridge.finalizeBridgeERC20` will decode `_localToken=DAI`, `_remoteToken=USDB`. It will then try to call transfer on the DAI address on L2 using `safeTransfer` which will fail as no contract is deployed.

Bridging USD yield tokens currently fails.

Recommendation: Swap the local and remote tokens in `L1BlastBridge._initiateBridgeERC20`.

5.1.3 `_delegatecall_uint256_arr_arg_returns_uint256` wrong calldata encoding

Severity: Critical Risk

Context: DelegateCalls.sol#L26-L84

Description: The `Delegatecalls` library is used by the `YieldManager` to delegatecall into a `YieldProvider`. The calldata encoding for most of the `_delegatecall_*` functions is excessive (uses more calldata bytes than needed) or incorrect:

- The `abi.encodePacked` function is used to encode the calldata, using a 256-bit selector argument when 4 bytes are enough.
- The `_delegatecall_uint256_arg_returns_uint256` can write into unallocated memory in `returndatacopy(ptr, 0, size)` if `size > 0x24`.
- The `_delegatecall_uint256_arr_arg_returns_uint256` encodes the calldata as `abi.encodePacked(selector, uint256[] arg)` which leads to an incorrect calldata encoding:

```
0x00-0x20: selector
0x20-0x40: arg[0]
0x40-0x60: arg[1]
// ...
```

But this should be abi encoded, i.e., `abi.encodeWithSelector(bytes4(bytes32(selector)), arg)` should give you the correct result:

```
aabbccdd // selector
0000000000000000000000000000000000000000000000000000000000000020 // arr offset
0000000000000000000000000000000000000000000000000000000000000002 // arr length
0000000000000000000000000000000000000000000000000000000000001337 // arr[0]
0000000000000000000000000000000000000000000000000000000000001338 // arr[1]
```

Currently, the `YieldProvider.claim(uint256[] calldata requestIds)` calls are all performed with an empty `requestedIds` no matter what `requestedIds` the calling `YieldManager` defined. The `YieldManager.claimPending(uint256 idx, address providerAddress, uint256[] requestIds)` calling `LidoYieldProvider.claim(requestIds)` will not actually perform a withdrawal from Lido. It's impossible to claim the unstaked funds from Lido, preventing L2 to L1 withdrawals.

Recommendation: Considerl cleaning up the functions in the `DelegateCalls` library and using `abi.encodeWithSelector(selector, args)` everywhere instead of hand-crafting the calldata:

```
  // always use encodeWithSelector with an optional argument
- abi.encodePacked(selector, arg)
+ abi.encodeWithSelector(bytes4(bytes32(selector)), arg)
```

Alternatively, directly define the interface for the desired functions:

```
  // pseudo code

- function _delegatecall_uint256_arr_arg_returns_uint256(address provider, uint256 selector, uint256[]
↪  memory arg) internal returns (uint256) {
-     (bool success, bytes memory res) = provider.delegatecall(abi.encodePacked(selector, arg));
-     require(success, "delegatecall failed");
-     return abi.decode(res, (uint256));
- }


+ interface IDelegateCalls {
+     function claim(uint256[] calldata requestIds) external returns (uint256 claimed);
+     // ...
+ }

+ function _delegatecall_claim(address provider, uint256[] memory arg) internal returns (uint256) {
+     (bool success, bytes memory res) = provider.delegatecall(abi.encodeCall(IDelegateCalls.claim,
↪  (arg)));
+     require(success, "delegatecall failed");
+     return abi.decode(res, (uint256));
+ }
```

Consider adding mainnet fork integration tests with the actual protocols. The `YieldManager.t.sol:test_claimPending_Lido_succeeds` currently passes because the `MockLidoWithdrawalQueue.claimWithdrawals` ignores the array parameters.

5.1.4 Changing yield from `Claimable` cause fund loss

Severity: Critical Risk

Context: ERC20Rebasing.sol#L348

Description: If `Claimable` yield is changed to any other mode, then users will lose all unclaimed yield. This happens because changing from `Claimable` to any other mode, only takes the fixed part of balance and ignores the claimable part.

Recommendation: While configuring from `Claimable` to any other mode, the overall balance (fixed plus claimable) should be considered as new balance for the new yield mode (while calling `_setBalance`).

5.1.5 Calling `findCheckpointHints()` with `_firstIndex` as `0` will always revert

Severity: Critical Risk

Context: LidoYieldProvider.sol#L93-L97

Description: `LidoYieldProvider.claim()` calls `findCheckpointHints()` with `_firstIndex` as `0`:

```
uint256[] memory hintIds = WITHDRAWAL_QUEUE.findCheckpointHints(
    requestIds,
    0,
    WITHDRAWAL_QUEUE.getLastCheckpointIndex()
);
```

However, `_firstIndex` cannot be `0` since Lido's checkpoint list is 1-indexed, as stated in the documentation here:

> `_firstIndex` must be greater than 0, because checkpoint list is 1-based array

`findCheckpointHints()` will revert in this check:

```
if (_start == 0 || _end > lastCheckpointIndex) revert InvalidRequestIdRange(_start, _end);
```

This will cause all withdrawals from Lido to be unclaimable.

Recommendation: Calls `findCheckpointHints()` with `_firstIndex` as `1`:

```
  uint256[] memory hintIds = WITHDRAWAL_QUEUE.findCheckpointHints(
      requestIds,
-     0,
+     1,
      WITHDRAWAL_QUEUE.getLastCheckpointIndex()
  );
```

5.1.6 Withdrawing discounted ETH from L2 always fails

Severity: Critical Risk

Context: OptimismPortal.sol#L387, CrossDomainMessenger.sol#L240

Description: In case of negative yield events, withdrawing ETH from L2 will result in receiving fewer ETH on L1. The `OptimismPortal` will use the discounted ETH value `txValueWithDiscount` instead of the amount specified in the withdrawal transaction on the `L2ToL1MessagePasser`.

As ETH withdrawals are initiated through the `L2BlastBridge / L2StandardBridge` → `L2CrossDomainMessenger` → `L2ToL1MessagePasser`, the finalization flow on L1 goes through `OptimismPortal` → `L1CrossDomainMessenger` → `L1BlastBridge / L1StandardBridge`.

The `L2CrossDomainMessenger` calls `relayMessage()` with `msg.value`:

```
abi.encodeWithSelector(
    this.relayMessage.selector, messageNonce(), msg.sender, _target, msg.value, _minGasLimit, _message
)
```

However, both `L1CrossDomainMessenger` and `L1BlastBridge / L1StandardBridge` verify that the received `msg.value` matches the amount specified on L2:

```
if (_isOtherMessenger()) {
    // These properties should always hold when the message is first submitted (as
    // opposed to being replayed).
    assert(msg.value == _value);
    assert(!failedMessages[versionedHash]);
}
```

All discounted ETH withdrawals will fail.

Recommendation: The fundamental issue is that the L2 does not know about the discount on L1 (it cannot know the final discount at the time of withdrawal) and always schedules a withdrawal of the full amount. In Optimism, discounts are impossible and the contracts are incompatible with the Blast discount feature. Careful changes to the `L1CrossDomainMessenger.relayMessage` and `L1StandardBridge/L1BlastBridge.finalize*` Optimism contracts are required. Consider running a testnet and adding more end-to-end tests.

5.1.7 Fund duplication via ERC20 self-transfer

Severity: Critical Risk

Context: ERC20Rebasing.sol#L241-L247

Description: The `from` and `to` balances are fetched and cached, then updated via `_setBalance()`. Should a user do an asset self-transfer such that `from == to` with a specified `amount`, there would be fund duplication where his balance would increase by `amount`.

Recommendation: Either shift the balance retrieval of `to` to after the update of `from`:

```
- uint256 toBalance = balanceOf(to);
  _setBalance(from, fromBalance - amount, currentSharePrice, false);
+ uint256 toBalance = balanceOf(to);
  _setBalance(to, toBalance + amount, currentSharePrice, false);
```

or use the `_withdraw()` and `_deposit()` methods:

```
function _transfer(
    address from,
    address to,
    uint256 amount
) internal virtual {
    if (from == address(0)) revert TransferFromZeroAddress();
    if (to == address(0)) revert TransferToZeroAddress();

    _withdraw(from, amount);
    _deposit(to, amount);

    emit Transfer(from, to, amount);
}
```

5.1.8 Message can be passed through `OptimismPortal` to maliciously call `ethYieldManager`

Severity: Critical Risk

Context: OptimismPortal.sol#L387

Description: Through `L2ToL1MessagePasser.initiateWithdrawal()`, one can set `ethYieldManager` as the `_tx.target` to invoke its permissioned `requestWithdrawal()` and `claimWithdrawal()` methods. The consequences are:

1. Brick existing finalised LIDO withdrawal requests, but yet to be finalised on the bridge via `finalizeWithdrawalTransaction()`, since they will revert with `RequestAlreadyClaimed(_requestId);`.
2. Brick the withdrawal queue by requesting a large enough amount such that the cumulative amount is close to `type(uint128).max`, causing subsequent `proveWithdrawalTransaction()` to revert when it tries to increment the cumulative amount.

Recommendation: Prevent `tx.target` from being set to `yieldManager`:

```
if (_tx.target == address(yieldManager)) revert Unauthorized()
```

5.2 High Risk

5.2.1 Inflated `_sharePrice()` from inclusion of `lockedAmount` funds

Severity: High Risk

Context: YieldManager.sol#L348-L349

Description: The value used for calculation of `_sharePrice()` is `totalValue()`, which consists of `lockedValue()` and `totalProviderValue()`. `lockedValue()`, according to the natspec, is meant to return `the amount of the withdrawal token that is held by the yield manager.`

This means that the inherited `WithdrawalQueue`'s `lockedAmount` is also included, which shouldn't be so. For the `ETHYieldManager` specifically, once requests are finalised, `accumulatedNegativeYields` is decremented if non-zero, but `totalValue()` remains unchanged, so `sharePrice()` would be inflated for subsequent finalisations should `accumulatedNegativeYields` be non-zero.

Finalized ETH is considered to have been burnt on L2 and out of the system on L1 and (part of the) potential negative yield recovered; so `lockedAmount` funds should not influence future share prices anymore.

Recommendation: Subtract the locked amount in `value`.

```
- uint256 value = totalValue();
+ uint256 value = totalValue() - getLockedAmount();
```

5.2.2 `commitYieldReport()` will revert when withdrawing insurance to cover negative yield

Severity: High Risk

Context: YieldManager.sol#L264-L265, DSRYieldProvider.sol#L74-L76

Description: `commitYieldReport()` calls `commitYield()` on the provider to determine how much yield was gained since the last call:

```
// Commit the yield for the provider
int256 committedYield = YieldProvider(_providers.at(i)).commitYield();
```

This returns `yield()`, which is calculated as `stakedValue() - stakedBalance`, as seen below:

```
function yield() public view override returns (int256) {
    return int256(stakedValue()) - int256(stakedBalance);
}
```

However, `committedYield` will still be negative for `DSRYieldProvider` after funds are withdrawn from insurance to cover losses.

The issue is that `DSRYieldProvider` unstakes and holds DAI in the insurance contract. As such, when `withdrawFromInsurance()` is called to cover the loss, it transfers DAI to the `YieldManager`.

For `DSRYieldProvider`, `stakedValue()` won't increase after `withdrawFromInsurance()` as the withdrawn DAI remains unstaked, so `committedYield` won't change.

This will cause `commitYieldReport()` to revert in the sanity check below.

Recommendation: In `withdrawFromInsurance()`, consider staking the withdrawn DAI using `DSR_MANAGER.join()`.

Additionally, add a `//@dev` comment in `withdrawFromInsurance()` that it should ensure that withdrawn funds increase `stakedValue()`, ie. they should be staked.

5.2.3 `WETHRebasing` share price precision issue breaks ERC20 invariants

Severity: High Risk

Context: WETHRebasing.sol#L99

Description: The `WETHRebasing` share price is dynamically defined as:

```
(address(this).balance - _totalVoidAndRemainders) / _totalShares
```

In certain cases, when a share is split into two remainder balances, the share price can increase. This happens due to precision issues in this calculation. The `_totalShares` decreases by 1 but the `_totalVoidAndRemainders` might only decrease by the *rounded-down* share price amount, resulting in the share price increasing by 1. The increase in share price can lead to balances and `totalAssets` unexpectedly increasing and breaking core ERC20 invariants that users/contracts rely on:

- post-transfer: `balanceOf(from) -= amount; balanceOf(to) += amount` as well as `totalSupply() += amount`.
- post-claim: `balanceOf(recipient) += claimedAmount;`.
- `deposit`, `withdraw`, `transfer`, `claim`, `configure` should not change the `sharePrice`.

Recommendation: The share price should not change in any of the `deposit`, `withdraw`, `transfer`, `claim`, `configure` functions. It should only change when new ETH yield is distributed (or when any other ETH donation to the contract happens). Consider keeping track of the share price explicitly and only increasing it when new yield comes in, similar to how USDB and rebasing ETH work.

Example:

```
address(this).balance = 109
_totalVoidAndRemainders = 0
_totalShares = 10
sharePrice = floor(109 / 10) = floor(10.9) = 10
```


`A` transfers 5 to `B`, `A`'s 1 share (valued at a share price of 10) is split up into 2 remainders of 5. New state:

```
address(this).balance = 109
_totalVoidAndRemainders = 10
_totalShares = 9
sharePrice = floor((109 - 10) / 9) = floor(11) = 11
```

Proof of Concept:

Logs:

```
=== totalSupply === 60004
=== sharePrice === 15001
=== totalSupply === 60007
=== sharePrice === 15002
Error: !bob
Error: a == b not satisfied [uint]
      Left: 15004
      Right: 15003
```

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.15;

// Testing utilities
import { Test, StdUtils } from "forge-std/Test.sol";
import { WETHRebasing } from "src/L2/WETHRebasing.sol";
import { Shares } from "src/L2/Shares.sol";
import { Blast } from "src/L2/Blast.sol";
import { Gas } from "src/L2/Gas.sol";
```

```
import { Predeploys } from "src/libraries/Predeploys.sol";

// WETHRebasing.t.sol
import { YieldMode } from "src/L2/ERC20Rebasing.sol";
import { MockYield } from "test/CommonTest.t.sol";
import { console2 } from "forge-std/console2.sol";

contract SpearbitTest is Test {
    address constant alice = address(0x1337);
    address constant bob = address(0x1338);
    Shares SHARES;
    WETHRebasing internal WETH;

    function setUp() public virtual {
        vm.label(alice, "alice");
        vm.label(bob, "bob");

        vm.etch(Predeploys.GAS, address(new Gas(address(this), Predeploys.BLAST, address(0), 0, 1, 1,
↪  2, 2)).code);
        MockYield mockYield = new MockYield();
        vm.etch(Predeploys.BLAST, address(new Blast(Predeploys.GAS, address(mockYield))).code);

        Shares shares = new Shares({ _price: 1e4, _reporter: address(0) });
        vm.etch(Predeploys.SHARES, address(shares).code);
        SHARES = Shares(Predeploys.SHARES);
        SHARES.initialize(1e4);
        vm.label(address(SHARES), "SHARES");

        vm.deal(address(this), 100 ether);
        WETH = new WETHRebasing();
        vm.etch(Predeploys.WETH_REBASING, address(WETH).code);
        WETH = WETHRebasing(payable(Predeploys.WETH_REBASING));
        WETH.initialize{ value: shares.price() }();
        vm.label(address(WETH), "WETH");
    }

    function addBalance(address account, uint256 amount) internal {
        vm.deal(account, account.balance + amount);
    }

    function test_Claim_invariants() public {
        claim_invariants(30000, 20007);
    }

    function claim_invariants(uint256 balance, uint256 claim) internal {
        addBalance(alice, balance);
        vm.startPrank(alice);
        WETH.configure(YieldMode.CLAIMABLE);
        WETH.deposit{value: balance}();
        vm.stopPrank();

        addBalance(address(WETH), claim);

        vm.startPrank(alice);
        uint256 claimable = WETH.getClaimableAmount(alice);
        // 4*15,001+0 == 60,003
        console2.log("=== totalSupply ===", WETH.totalSupply());
        console2.log("=== sharePrice ===", WETH.sharePrice());
        WETH.claim(bob, claimable);
        // 3*15,002+14,999+2 == 60,007
        console2.log("=== totalSupply ===", WETH.totalSupply());
        console2.log("=== sharePrice ===", WETH.sharePrice());
```

```
        vm.stopPrank();

        assertEq(WETH.balanceOf(alice), balance, "!alice");
        assertEq(WETH.balanceOf(bob), claimable, "!bob");
    }
}
```

Also works with a simple transfer:

```
function test_transfer_invariants() public {
    uint248 balance = 30_000;
    uint248 yield = 20_007;
    uint248 toTransfer = yield;
    vm.assume(balance > 0);

    addBalance(alice, balance);
    vm.startPrank(alice);
    WETH.configure(YieldMode.CLAIMABLE);
    WETH.deposit{value: balance}();
    vm.stopPrank();

    addBalance(address(WETH), yield);

    vm.startPrank(alice);
    WETH.transfer(bob, toTransfer);
    vm.stopPrank();

    assertEq(WETH.balanceOf(alice), balance - toTransfer, "!alice");
    assertEq(WETH.balanceOf(bob), toTransfer, "!bob");
}
```

Blast: I think we can take some inspiration from how we handle this problem for the native ether on the L2 to solve this precision issue. In op-geth, the share price for ether only changes when new yield comes in, and it can never change due to user actions like deposits/withdrawals/transfers/shares converting to remainder.

5.2.4 Unset `governor` allows to steal both yield and gas refund

Severity: High Risk

Context: Blast.sol#L59

Description: `CrossDomainMessenger` predeploy does not have its `governor` configured in `Blast.sol`. This means that if an attacker sends a message to L2 like shown below, then it would become the `governor` of the `CrossDomainMessenger` contract:

```
// `relayMessage` called with:

_target = Blast contract
selector = configure
_yieldMode = Automatic
Gasmode = CLAIMABLE
governor = Attacker
```

This allows them to claim:

1. Yield from the collected ETH in `L2CrossDomainMessenger`, if the `relayMessage` (temporarily) failed.
2. Gas for the `L2CrossDomainMessenger`.

Note: This issue affects any contract on L2 that allows arbitrary calls.

Recommendation: Add `Blast` contract to `_isUnsafeTarget` or initialize the predeploy with itself as the governor. Also, for all L2 contracts allowing arbitrary calls, consider setting some governor to prevent an attacker from taking

control.

5.2.5 Unsafe ERC-20 transfer breaks USDT bridging in `L1BlastBridge`

Severity: High Risk

Context: L1BlastBridge.sol#L177

Description: In `_initiateBridgeERC20()`, USDT is transferred from the user to `L1BlastBridge` using `transferFrom()`:

```
IERC20(_localToken).transferFrom(_from, address(this), _amount);
```

This will revert for USDT (which is an approved USD yield token) since its `transferFrom()` function does not return a `bool`, but the `IERC20` interface expects one to be returned. As such, users will not be able to bridge USDT for USDB.

Recommendation: Consider handling all ERC-20 token operations with OpenZeppelin's `SafeERC20` library, which is what Optimism's `StandardBridge` does.

It's also best to use `safeTransfer()`/`safeTransferFrom()` for the following instances of `transfer()`/`transferFrom()` since they might not work for future yield tokens:

- Insurance.sol#L52
- WithdrawalQueue.sol#L402
- L1BlastBridge.sol#L177
- L1BlastBridge.sol#L211

5.2.6 ETH yield token bridge transactions use fixed gas and are not replayable

Severity: High Risk

Context: L1BlastBridge.sol#L219-L224

Description: When using the `L1BlastBridge` to bridge an ETH yield token, the `OptimismPortal` is called directly instead of routing through the `L1CrossDomainMessenger`, and a fixed gas amount of `RECEIVE_DEFAULT_GAS_LIMIT = 100_000` gas is always used:

```
portal.depositTransaction(
    Predeploys.L2_BLAST_BRIDGE,
    _amount,
    RECEIVE_DEFAULT_GAS_LIMIT,
    false,
    abi.encodeWithSelector(/* ... */)
);
```

As no `CrossDomainMessenger` intermediate contract is used, the `finalizeBridgeETHDirect` bridge transactions are not replayable. They will be executed once and if the transaction fails, the bridged ETH is lost (remains in the aliased `L1BlastBridge` address on L2). This call might fail if the `to` address consumes more than the allocated `RECEIVE_DEFAULT_GAS_LIMIT` gas in its `receive` function. The `_minGasLimit` parameter of the `bridgeERC20*` functions is currently ignored for ETH yield tokens which can be very misleading for users and lead to losses.

Recommendation: Consider using the `_minGasLimit` parameter in `_initiateBridgeERC20` not only for the USD yield tokens but also for the ETH yield tokens instead of the hardcoded `RECEIVE_DEFAULT_GAS_LIMIT` gas limit.

5.3 Medium Risk

5.3.1 `YieldManager.finalize` can underflow for `accumulatedNegativeYields`

Severity: Medium Risk

Context: YieldManager.sol#L312

Description: The `YieldManager.finalize` function reduces the `accumulatedNegativeYields` by:

```
// sharePrice = totalValue() * E27_PRECISION_BASE / (totalValue() + accumulatedNegativeYields)
// realAmount = (nominalAmount * sharePrice) / E27_PRECISION_BASE;
if (accumulatedNegativeYields > 0) {
    accumulatedNegativeYields -= (nominalAmount - realAmount);
}
```

The `nominalAmount - realAmount` term can underflow with `nominalAmount - realAmount > accumulatedNegativeYields` because of the limited share price precision and the fact that when `realAmount` rounds down then `nominalAmount - realAmount` rounds up.

This is a DoS on the withdrawal finalizations.

Recommendation: One way to address this could be to simply cap `accumulatedNegativeYields` at 0 in case it would turn negative. The `if (accumulatedNegativeYields > 0)` could also better be rephrased to `if (nominalAmount > realAmount)` as this is the real indicator of when `accumulatedNegativeYields` should change.

```
if (nominalAmount > realAmount) {
    accumulatedNegativeYields = _subClamped(accumulatedNegativeYields, nominalAmount - realAmount);
}

function _subClamped(uint256 x, uint256 y) internal pure returns (uint256 z) {
    unchecked {
        z = x > y ? x - y : 0;
    }
}
```

Proof of Concept:

```
nominalAmount = 34632200351743
totalValue = nominalAmount
negYields = 1

# realAmount will be computed as
sharePrice = 34632200351743 * 1e27 / 34632200351744 = 999999999999971125138170735
realAmount = (nominalAmount * sharePrice) / E27_PRECISION_BASE = 34632200351741
nominalAmount - realAmount = 2 > 1 = negYields
```

And the corresponding test:

```
uint256 internal constant RAY = 1e27;

function test_math(uint120 _nominalAmount, uint120 _totalValue, uint120 _negYields) public {
    vm.assume(_negYields > 0);
    uint256 negYields = _negYields;
    uint256 totalValue = _totalValue;
    uint256 nominalAmount = bound(_nominalAmount, 0, totalValue);
    uint256 sharePrice = totalValue * RAY / (totalValue + negYields);
    uint256 realAmount = nominalAmount * sharePrice / RAY;
    // uint256 realAmount = (nominalAmount * totalValue) / (totalValue + negYields);
    assertGe(negYields, nominalAmount - realAmount);
}
```

5.3.2 `L1BlastBridge._initiateBridgeERC20()` directly sends `_amount` of ETH without converting to 18 decimals

Severity: Medium Risk

Context: L1BlastBridge.sol#L221-L233, L2BlastBridge.sol#L49

Description: `L1BlastBridge._initiateBridgeERC20()` initiates a L1 → L2 deposit transaction as shown:

```
portal.depositTransaction(
    Predeploys.L2_BLAST_BRIDGE,
    _amount,
    RECEIVE_DEFAULT_GAS_LIMIT,
    false,
    abi.encodeWithSelector(
        L2BlastBridge.finalizeBridgeETHDirect.selector,
        _from,
        _to,
        USDConversions._convertDecimals(_amount, ethYieldToken.decimals, USDConversions.WAD_DECIMALS),
        _extraData
    )
);
```

It calls `L2BlastBridge.finalizeBridgeETHDirect()` with `USDConversions._convertDecimals(_amount, ethYieldToken.decimals, USDConversions.WAD_DECIMALS)`, but sends `_amount` of ETH through `OptimismPortal`.

If a future `ethYieldToken` has more/less than 18 decimals, this will sending the wrong amount of ETH and the following check in `finalizeBridgeETHDirect()` will fail:

```
require(msg.value == _amount, "StandardBridge: amount sent does not match amount required");
```

Recommendation: Consider sending `_amount` with 18 decimals of ETH through `OptimismPortal` as well:

```
+ uint256 ethAmount = USDConversions._convertDecimals(_amount, ethYieldToken.decimals,
↪  USDConversions.WAD_DECIMALS);
  portal.depositTransaction(
      Predeploys.L2_BLAST_BRIDGE,
-     _amount,
+     ethAmount,
      RECEIVE_DEFAULT_GAS_LIMIT,
      false,
      abi.encodeWithSelector(
          L2BlastBridge.finalizeBridgeETHDirect.selector,
          _from,
          _to,
-         USDConversions._convertDecimals(_amount, ethYieldToken.decimals, USDConversions.WAD_DECIMALS),
+         ethAmount,
          _extraData
      )
  );
```

5.3.3 `YieldManager` can claim fewer unstaked tokens than expected resulting in insolvency

Severity: Medium Risk

Context: YieldManager.sol#L156

Description: When unstaking from Lido the funds are pending in Lido's withdrawal queue and need to be claimed later. The claimed amount can at the time of claiming be less than the requested amount (due to negative rebases), see `WithdrawalQueueBase._calculateClaimableEther`:

```
if (batchShareRate > checkpoint.maxShareRate) {
    eth = shares * checkpoint.maxShareRate / E27_PRECISION_BASE;
}
```

The `YieldManager` will first increase the `pendingBalance` by `amount` in `unstake` and then decrease the amount by `claimed` in `claimPending`. In case `claimed < amount`, the accounting is wrong as the contract still thinks it would receive a `pendingBalance` of `amount - claimed`. The pending balance for this withdrawal needs to be cleared such that `totalValue` is correctly tracked and not overestimated. In addition, the loss needs to be accounted for in the `unstake` function to not be insolvent for L2 withdrawals (because `totalValue()` decreased).

Recommendation: The pending balance for this claim needs to be cleared and the loss of `requestedUnstake - claimedUnstake` must be booked. Ideally, the entire insurance fund logic of `commitYieldReport` would run as well for this loss.

```
// pseudo code, should be verified with tests

// in YM.claimPending, we don't track all of these vars yet
// reduce pending balance by initial request amount
YieldProvider(providerAddress).recordClaimed(unstakeRequest.requestedAmount);
// book difference as a loss
// ideally this would run the insurance code again, and cover losses if called with enableInsurance
accumulatedNegativeYields += (unstakeRequest.requestedAmount - unstakeRequest.claimedAmount);
```


5.3.4 `USDConversions` can swap locked funds

Severity: Medium Risk

Context: USDConversions.sol#L71-L73, USDYieldManager.sol#L413

Description: DAI tokens in the yield manager that are locked for finalized withdrawals can be swapped to other USD yield tokens. User withdrawals can fail after finalization because of this.

Recommendation: Consider reverting if locked funds are swapped. It should be enough to check this in the `USDYieldManager.convert` call as other calls to `_convert` always perform a transfer from the user before (the `USDConversions` library does not have access to the locked amount).

5.3.5 `YieldManager` can stake locked funds

Severity: Medium Risk

Context: DSRYieldProvider.sol#L86, LidoYieldProvider.sol#L75

Description: When L2 bridge withdrawals are finalized in the `WithdrawalQueue` the funds are considered locked as the user can now withdraw these funds from the queue. However, both `YieldProvider`'s `stake` functions allow staking this locked amount.

Recommendation: Consider reverting if locked funds are staked:

```
  // DSRYieldProvider
  /// @inheritdoc YieldProvider
  function stake(uint256 amount) external override onlyDelegateCall {
-     uint256 daiBalance = DAI.balanceOf(address(YIELD_MANAGER));
+     uint256 daiBalance = YIELD_MANAGER.getTokenBalance();
      if (amount > daiBalance) {
          revert InsufficientStakableFunds();
      }
      if (amount > 0) {
          DSR_MANAGER.join(address(YIELD_MANAGER), amount);
      }
  }

  // LidoYieldProvider
  /// @inheritdoc YieldProvider
  function stake(uint256 amount) external override onlyDelegateCall {
-     if (amount > YIELD_MANAGER.lockedValue()) {
+     if (amount > YIELD_MANAGER.getTokenBalance()) {
          revert InsufficientStakableFunds();
      }
      LIDO.submit{value: amount}(address(0));
  }
```

Additionally, consider renaming `lockedValue` to `totalBalance` and `getTokenBalance` to `getAvailableBalance` or similar names as the current names are ambiguous.

5.3.6 Fraud recovery logic is missing

Severity: Medium Risk

Context: OptimismPortal.sol#L287

Description: As per documentation, a partial mitigation for fraud is mentioned as:

> We can partially mitigate this as the admin by not finalizing withdrawals prior to the fraud period ending.

But if the `Admin` doesn't finalize the withdrawal, then the withdrawal request will remain stuck in the queue along with other requests, causing DOS on every user's withdrawal request.

In the end, the `Admin` will be forced to `finalize` the fraud request (since withdrawal requests in the queue can't be skipped). This means that the ETH for said request will be stuck in `lockedAmount` forever (an attacker also cannot claim it due to the intervention of the Challenger).

Recommendation: Consider adding a new function in `OptimismPortal` which allows the `Admin` claiming funds linked to fraudulent requests and sending funds back to yield manager.

5.3.7 Initial depositor can inflate share to siphon yield of smaller deposits

Severity: Medium Risk

Context: WETHRebasing.sol#L53-L54

Description: The initial `_totalShares` of 1 is insufficient to guard against a share inflation attack that affects yield accrual.

Using the intended config of `price() = 1e9` (1 gwei), an initial depositor can:

- Deposit 1 share of 1 gwei.
- Create a contract that `selfdestruct()` to forcibly send ETH to the contract to inflate `_sharePrice()`, e.g. 1 ETH.
- `_sharePrice()` becomes `(1e18 + 2e9) / 2` = `0.500000001` ETH.

Because yield doesn't get distributed to remainders, this will prevent small deposits and remainders of <= 0.5 ETH from receiving yield, which is distributed amongst those with shares (>= 0.500000001 ETH), although in the case above, the attacker loses 0.5 ETH as a trade-off.

Recommendation: Increase the initial `_totalShares` to a larger number like `1000`, which is what UniswapV2 uses.

5.3.8 Reinitialization causes metering parameter to be reset

Severity: Medium Risk

Context: OptimismPortal.sol#L146

Description: Reinitialization (with version on `Constants.INITIALIZER` changed) will cause `ResourceParams` for Metering to be re-initialized, thus all `params` will be set to their default values (impacting gas price calculation) on `metered` modifier.

```
function __ResourceMetering_init() internal onlyInitializing {
    params = ResourceParams({ prevBaseFee: 1 gwei, prevBoughtGas: 0, prevBlockNum: uint64(block.number)
↪   });
}
```

Recommendation: This should only be updated in the case of a fresh initialization:

```
if (params.prevBlockNum == 0) {
    params = ResourceParams({ prevBaseFee: 1 gwei, prevBoughtGas: 0, prevBlockNum: uint64(block.number)
↪   });
}
```

See this reference for more context.

5.3.9 `admin` in the `Insurance` contract can never be set

Severity: Medium Risk

Context: Insurance.sol#L41-L45

Description: The `initialize()` function in `Insurance.sol` does not set `admin`. As such, it is impossible for the `admin` to be set as `setAdmin()` can only be called by the contract's admin.

This makes it impossible for the admin to call `coverLoss()` should the need arise.

Recommendation: Consider setting `admin` in `initialize()`.

5.3.10 `donateETH` funds are stuck in `OptimismPortal`

Severity: Medium Risk

Context: OptimismPortal.sol#L207

Description: The Blast `OptimismPortal` inherits the `donateETH` function from Optimism. It's not needed in Blast as it was used for the migration to bedrock. The donated funds will be stuck in the contract. When withdrawing, the withdrawal transaction's ETH is claimed from the yield manager.

Recommendation: Consider removing this function if it is not expected to be needed or forward the donated funds to the yield manager.

5.3.11 Actual claim rate may be below `minClaimRateBips`

Severity: Medium Risk

Context: Gas.sol#L148-L169

Description: Assuming `minClaimRateBips <= ceilClaimRate`, it's possible for `minClaimRateBips` to not be respected. Consider the following configuration:

- `zeroClaimRate` = 2500 (25%)
- `baseClaimRate` = 5000 (50%)
- `ceilClaimRate` = 8000 (80%)
- `baseGasSeconds` = 60 (60s)
- `ceilGasSeconds` = 100 (100s)

Suppose the user has vested 1 ETH over 80s: `etherSeconds` = 1e18 * 80 = 80e18, `etherBalance` = 1e18. Assume the user wants to claim at a `minClaimRateBips` of 6000 (60%). Plugging in the values into the helper contract below,

```
contract Test {
    uint256 public zeroClaimRate = 2500;
    uint256 public baseClaimRate = 5000;
    uint256 public ceilClaimRate = 8000;
    uint256 public baseGasSeconds = 60;
    uint256 public ceilGasSeconds = 100;

    function claimGasAtMinClaimRate(uint256 etherBalance, uint256 secondsStaked, uint256
↪   minClaimRateBips) external view returns (uint256, uint256) {
        uint256 etherSeconds = etherBalance * secondsStaked;
        uint256 bipsDiff = minClaimRateBips - baseClaimRate;
        uint256 secondsDiff = ceilGasSeconds - baseGasSeconds;
        uint256 rateDiff = ceilClaimRate - baseClaimRate;
        uint256 minSecondsStaked = baseGasSeconds + (bipsDiff * secondsDiff / rateDiff);
        uint256 maxEtherClaimable = etherSeconds / minSecondsStaked;
        if (maxEtherClaimable > etherBalance)  {
            maxEtherClaimable = etherBalance;
        }
        uint256 secondsToConsume = maxEtherClaimable * minSecondsStaked;
        return getClaimRateBps(secondsToConsume, maxEtherClaimable);
    }

    function getClaimRateBps(uint256 gasSecondsToConsume, uint256 gasToClaim) public view returns
↪   (uint256, uint256) {
        uint256 secondsStaked = gasSecondsToConsume / gasToClaim;
        if (secondsStaked < baseGasSeconds) {
            return (zeroClaimRate, 0);
        }
        if (secondsStaked > ceilGasSeconds) {
            uint256 gasToConsumeNormalized = gasToClaim * ceilGasSeconds;
            return (ceilClaimRate, gasToConsumeNormalized);
        }

        uint256 rateDiff = ceilClaimRate - baseClaimRate;
        uint256 secondsDiff = ceilGasSeconds - baseGasSeconds;
        uint256 secondsStakedDiff = secondsStaked - baseGasSeconds;
        uint256 additionalClaimRate = rateDiff * secondsStakedDiff / secondsDiff;
        uint256 claimRate = baseClaimRate + additionalClaimRate;
        return (claimRate, gasSecondsToConsume);
    }
}
```

the resultant claim rate is 5975, which is less than the expected minimum claim rate of 6000.

Recommendation: Consider using OpenZeppelin Math's `ceilDiv` for the calculation of `minSecondsStaked` to consume more etherSeconds to fulfil the minimum requested claim rate.

```
- uint256 minSecondsStaked = baseGasSeconds + (bipsDiff * secondsDiff / rateDiff);
+ uint256 minSecondsStaked = baseGasSeconds + ceilDiv(bipsDiff * secondsDiff, rateDiff);
```

5.4 Low Risk

5.4.1 `Admin` should not be allowed to revoke its role

Severity: Low Risk

Context: Insurance.sol#L44

Description: `Admin` involvement is required to call `coverLoss` in the event the yield report is not properly allocating funds (if negative yield is high and `Insurance` funds are lagging by small amounts, then `commitYieldReport` will fail to use `Insurance` funds). Thus, `Admin` should not be allowed to revoke itself.

Recommendation: Consider adding the check below:

```
require(_admin != address(0), "Cannot revoke Admin role");
```

5.4.2 `Blast.claimYield()` should revert when claiming more than the available amount

Severity: Low Risk

Context: Blast.sol#L200-L203, contracts.go#L1252-L1256, contracts.go#L1243-L1246, ERC20Rebasing.sol#L201-L204

Description: In `Blast.sol`, `claimYield()` doesn't check if `amount` exceeds the contract's maximum claimable amount. The precompile at `YIELD_CONTRACT` handles this by transferring the claimable amount when `amount` is larger:

```
if claimableAmount.Cmp(desiredAmount) < 0 {
    amount = claimableAmount
} else {
    amount = desiredAmount
}
```

However this might be a footgun for contracts. Some contracts might expect its ETH balance to increase by `amount` after calling `claimYield()`, but it only increases by a smaller amount.

In contrast, `ERC20Rebasing.claim()` reverts when a user attempts to claim more than his claimable amount.

Additionally, if `recipientOfYield` is `address(0)`, the yield gets redirected to `contractAddress` instead:

```
// assign recipient to contract if nil
if recipient.Cmp(common.Address{}) == 0 {
   recipient = contract
}
```

Recommendation: In `claimYield()`, consider reverting if `amount > IYield(YIELD_CONTRACT).getClaimableAmount(contractAddress)`. Additionally, document that yield is redirected to `contractAddress` when `address(0)` is specified as the recipient.

5.4.3 `DSRYieldProvider.sol.isStakingEnabled()` does not check liveness of Maker's protocol

Severity: Low Risk

Context: DSRYieldProvider.sol#L50-L52

Description: In `DSRYieldProvider.sol`, `isStakingEnabled()` does not check `daiJoin.live()` or `pot.live()`:

```
function isStakingEnabled(address token) public pure override returns (bool) {
    return token == address(DAI);
}
```

In the event of an emergency shutdown, maker will call the `cage()` functions in their contracts, and if that happens DAI staking should be disabled.

This is because `unstake()` will also revert due to this check in `daiJoin.exit()`, which would make it impossible for any deposited DAI to be withdrawn.

Recommendation: Consider if `isStakingEnabled()` should check `daiJoin.live()` or `pot.live()` as well, and add them if appropriate.

5.4.4 Non-zero Maker's PSM `buyGem()` fee will cause DAI → USDC swaps to fail

Severity: Low Risk

Context: USDConversions.sol#L78

Description: Maker's PSM has a fee mechanism (currently zero both ways). The amount specified for `PSM.buyGem()` is the USDC receivable, but the fee charged `tout` is in DAI. Hence, the amount of DAI pulled will be greater than `inputAmount` if `tout` is non-zero. This could post an issue for `YieldManager.convert()` that might perform DAI → USDC swaps, where the requested USDC amount will be too high since it assumes a 1:1 conversion with zero fee.

Recommendation: Swap to using `minOutputAmount` instead of `inputAmount`. The drawback with this method is that there could be remaining `inputAmount` since one is specifying exact output instead of expending the entire input.

```
- PSM.buyGem(address(this), _wadToUSD(inputAmount));
+ PSM.buyGem(address(this), minOutputAmount);
```

5.4.5 `LidoYieldProvider.isStakingEnabled` is incorrect

Severity: Low Risk

Context: LidoYieldProvider.sol#L55

Description: The `LidoYieldProvider. isStakingEnabled` returns `true` only if Lido is paused.

Recommendation: It should return `false` when Lido is paused:

```
  function isStakingEnabled(address token) public view override returns (bool) {
-     return token == address(LIDO) && LIDO.isStakingPaused();
+     return token == address(LIDO) && !LIDO.isStakingPaused();
  }
```

5.4.6 Missing `onlyEOA` modifier

Severity: Low Risk

Context: L1BlastBridge.sol#L107

Description: As per the comment on the `receive` function, `onlyEOA` should be allowed. But the corresponding restriction is missing, allowing contracts to bridge ETH by sending directly to the `L1BlastBridge`.

Recommendation: Add the `onlyEOA` modifier in the `receive` function:

```
- receive() external payable override {
+ receive() external payable override onlyEOA {
```

5.4.7 `WETHRebasing` virtual share earns yield

Severity: Low Risk

Context: WETHRebasing.sol#L54

Description: The `WETHRebasing` contract starts with a `totalShares` of 1. This share corresponds to `initialSharePrice` ETH tokens initially but it earns yield and its share will grow. This yield is not accessible to anyone as nobody owns the virtual share. The yield is essentially lost.

Recommendation: As it's only a single share the lost yield should not have a big impact. (When moving to an ETHRebasing-/USDB-type `price` approach for `WETHRebasing` this initial share can be removed).

5.4.8 Gas claim rate is non-continuous

Severity: Low Risk

Context: Gas.sol#L230

Description: The `getClaimRateBps` function is non-continuous in both `(claimRate, gasSecondsToConsume)` return parameters. It jumps at `baseGasSeconds` from `(zeroClaimRate, 0)` to `(baseClaimRate, gasSecondsToConsume)`. This can lead to strange user claim incentives:

Depending on the `baseGasSeconds` and `baseClaimRate` values, it might be beneficial to be on either side of `baseGasSeconds`. If `baseClaimRate` is close to `zeroClaimRate`, it might not make sense to burn `etherSeconds` for the little extra claim rate. If `baseGasSeconds` is low but the claim rate difference is large, it might make sense to always wait until enough `etherSeconds` are accumulated to hit it.

Recommendation: Choose appropriate `baseGasSeconds`, `baseClaimRate` and `zeroClaimRate` parameters to balance these incentives.

5.4.9 Standard `ERC20Permit` allows different `name` initialisation in constructor and initialiser

Severity: Low Risk

Context: ERC20Rebasing.sol#L23, ERC20Rebasing.sol#L78-L90, draft-EIP712.sol#L74

Description: `ERC20Rebasing` is meant to be upgradeable as it inherits `Initializable`, but it inherits the non-upgradeable `ERC20Permit` which has a constructor that takes in `_name` as an input to calculate `_HASHED_NAME` for the EIP712 domain separator. Hence, it is possible for contracts inheriting `ERC20Rebasing` to set a different name in the constructor and initializer (perhaps due to upgrades where a token name change is desired), which causes difficulties in verifying signed permits if `_HASHED_NAME` = `keccak256(bytes(name))` != initialized `name`.

Recommendation: Use `ERC20PermitUpgradeable` instead of `ERC20Permit`.

5.4.10 Claiming `gas` can run out of gas in `transfer`

Severity: Low Risk

Context: Gas.sol#L218

Description: The `claim` function performs a `payable(recipientOfGas).transfer(userEther)` call to send the gas rebate. This `transfer` is restricted to a low gas amount. If the recipient is a contract implementing a gas-intense `receive()` function this call might fail.

Recommendation: Consider using a low-level call to transfer the `userEther` which does not have a fixed gas limit.

5.4.11 `claimGasAtMinClaimRate` uses all `etherSeconds` when `minClaimRateBips <= zeroClaimRate`

Severity: Low Risk

Context: Gas.sol#L151

Description: The `claimGasAtMinClaimRate` function allows a user to claim the maximum gas at a minimum claim rate. It's unclear whether the function should 1) further try to maximize the claim rate if possible or 2) claim at the minimum claim rate to save the `etherSeconds`.

The default case does not further maximize the claim rate if the entire `etherBalance` can be claimed because of this code:

```
if (maxEtherClaimable > etherBalance)  {
    maxEtherClaimable = etherBalance;
}
// will claim maxEtherClaimable at ~minClaimRate
uint256 secondsToConsume = maxEtherClaimable * minSecondsStaked;
```

However, the `minClaimRateBips <= zeroClaimRate` case will maximize the claim rate by using `claimAll`.

Recommendation: Clarify the behavior of this function. The function should not have two different behaviors depending on the `minClaimRate` parameter. We believe the function should *not* optimize the claim rate beyond `minClaimRate`. Consider adjusting the `minClaimRateBips <= zeroClaimRate` to claim with 0 `etherSeconds`:

```
  if (minClaimRateBips <= zeroClaimRate) {
-     return claimAll(contractAddress, recipientOfGas);
+     return claim(contractAddress, recipientOfGas, etherBalance, 0);
  }
```

In addition, consider documenting that `claimAll` uses all available `etherSeconds` to maximize the claim rate.

5.4.12 `etherSeconds` can be saved up to be used on vesting subsequent gas claims

Severity: Low Risk

Context: Gas.sol#L216

Description: `etherSeconds` is the integral of unclaimed ether over time (ether * seconds vested). There is no limit to its accumulation, so `etherSeconds` continues to grow while gas remains unclaimed. This allows the accumulated gas to be "saved up" and be used for subsequent gas claims to be claimed at the maximum ceiling rate immediately.

Recommendation: Consider imposing an upper bound to the accumulation of `etherSeconds` at `etherBalance * ceilGasSeconds`.

5.4.13 USDC to DAI conversion can fail once debt limits are exceeded

Severity: Low Risk

Context: USDConversions.sol#L76, PSM

Description: The USD yield manager converts between stablecoins. For the USDC to DAI path, it always uses the PSM USDC contract's `sellGem` function. The PSM takes on debt to mint DAI through the `vat.frob(ilk, ..., int256(gemAmt18), int256(gemAmt18))` call. This call can fail if the ilk (the PSM's USDC collateral) hits its line (its debt limit), or the Line (total overall debt limit) is exceeded.

```
require(either(/*...*/, both(_mul(ilk.Art, ilk.rate) <= ilk.line, debt <= Line)),
↪  "Vat/ceiling-exceeded");
```

Direct USDC deposits could be disabled in this case. The current debt limit (as of writing) is set to `789_651_294` USD and `242_649_337`debt is used.

Recommendation: Consider checking if enough USDC can be sold into the PSM (`inputWad <= vat.ilks(psm.ilk()).line / RAY - vat.ilks(psm.ilk()).Art`), otherwise, use a different conversion path. Alternatively, add an option for the user to define what conversion path should be used. Furthermore, consider keeping the conversion system upgradeable in case the current USDC PSM is deprecated.

5.4.14 Unsafe type casts

Severity: Low Risk

Context: WithdrawalQueue.sol#L357, DSRYieldProvider.sol#L75, LidoYieldProvider.sol#L65

Description: When type-casting from a type with a larger range to a type with a smaller range, Solidity truncates any bits that exceed the new range instead of reverting. This silent truncation can lead to unexpected errors. The following code performs truncated type-casts:

- WithdrawalQueue.sol#L357
- DSRYieldProvider.sol#L75
- LidoYieldProvider.sol#L65

Recommendation: Consider using a `SafeCast` library as the default way to perform type casts.

5.5 Gas Optimization

5.5.1 Shift non-zero `insurance` address check outside loop

Severity: Gas Optimization

Context: YieldManager.sol#L236-L245

Description: `insurance != address(0)` is checked every iteration in `commitYieldReport()`. It only needs to be checked once, so this check can be shifted outside the loop.

Recommendation: Shift the non-zero `insurance` address check outside the `for` loop.

5.5.2 Return parameter and increment can be combined

Severity: Gas Optimization

Context: WithdrawalQueue.sol#L377-L378

Description: The `lastCheckpointId` increment and its return can be combined into a single line for gas savings.

Recommendation:

```
- lastCheckpointId += 1;
- return lastCheckpointId;
+ return ++lastCheckpointId;
```

5.5.3 Redundant balance check in `USDB.burn`

Severity: Gas Optimization

Context: USDB.sol#L126-L129, ERC20Rebasing.sol#L319-L322

Description: The `USDB.burn` and the `ERC20._withdraw` function that it calls perform the same balance check:

```
// USDB.burn
uint256 accountBalance = balanceOf(_from);
if (_amount > accountBalance) {
    revert InsufficientBalance();
}

// ERC20._withdraw
uint256 balance = balanceOf(account);
if (amount > balance) {
    revert InsufficientBalance();
}
```


Recommendation: Consider removing the balance check from `USDB.burn` as it is performed in `_withdraw` already.

5.5.4 `msg.sender` check in `claimWithdrawal()` can be performed earlier

Severity: Gas Optimization

Context: WithdrawalQueue.sol#L394-L397

Description: `claimWithdrawal()` calculates the amount claimable by a recipient and updates storage before checking if `msg.sender` is the recipient, which is gas inefficient if the check fails.

Recommendation: Perform the `msg.sender` check earlier:

- WithdrawalQueue.sol#L387-L397:

```
  if (request.claimed) revert RequestAlreadyClaimed(_requestId);

+ address recipient = request.recipient;
+ if (msg.sender != recipient) {
+     revert CallerIsNotRecipient();
+ }

  request.claimed = true;

  uint256 realAmount = _calculateClaimableEther(_requestId, _hintId);
  lockedAmount -= realAmount;

- address recipient = request.recipient;
- if (msg.sender != recipient) {
-     revert CallerIsNotRecipient();
- }
```

5.5.5 `WithdrawalQueue` checkpoint creation could be optimized

Severity: Gas Optimization

Context: WithdrawalQueue.sol#L237

Description: The `WithdrawalQueue` creates checkpoints to record the discounted withdrawal price (called share price) starting from a request id. A new checkpoint is created whenever finalizing any requests. However, most of the time the share price should not change as negative yield events are expected to be rare.

Recommendation: Consider reusing the previous checkpoint if the share price has not changed.

```
  function _createCheckpoint(uint256 firstRequestIdToFinalize, uint256 sharePrice) internal returns
↪   (uint256) {
+     if (checkpoints[lastCheckpointId].sharePrice == sharePrice) return lastCheckpointId;
      checkpoints.push(Checkpoint(firstRequestIdToFinalize, sharePrice));
      lastCheckpointId += 1;
      return lastCheckpointId;
  }
```

5.5.6 Boundary equality `ceilGasSeconds == ceilGasSeconds` case can be short circuited

Severity: Gas Optimization

Context: Gas.sol#L235

Description: The equality case where `secondsStaked == ceilGasSeconds` will also be the ceiling rate and `gasToConsumeNormalized` as `secondsStaked > ceilGasSeconds`.

Recommendation: Consider applying the following change:

```
- if (secondsStaked > ceilGasSeconds) {
+ if (secondsStaked >= ceilGasSeconds) {
```

5.5.7 Redundant `blastBridge` null address check

Severity: Gas Optimization

Context: OptimismPortal.sol#L456, ETHYieldManager.sol#L51)

Description: The `yieldManager.blastBridge() / blastBridge == address(0)` is redundant because it will pass the first conditional check of `msg.sender != blastBridge`, unless someone sends the transaction from the null address, which is highly unlikely.

Recommendation: The check can be removed:

```
- if (msg.sender != yieldManager.blastBridge() || yieldManager.blastBridge() == address(0)) {
+ if (msg.sender != yieldManager.blastBridge()) {

- if (msg.sender != blastBridge || blastBridge == address(0)) {
+ if (msg.sender != blastBridge) {
```

5.6 Informational

5.6.1 UX change to aid bridged ERC20 withdrawal

Severity: Informational

Context: L1BlastBridge.sol#L143

Description: `finalizeBridgeERC20` calls `_requestWithdrawal` without storing its return value. Since this function does not provide the resulting `_requestId`, it would be difficult for a user to find their `_requestId` from the event log.

So, the UI is expected to check the event log in order to provide the `_requestId` to the user who can later claim using the same id.

Recommendation: UX change need to be made in order to store `_requestId` for a user so they can check their corresponding request id for subsequent withdrawal.

Blast: Need to make a UX adjustment here, both in returning the request id and possibly recording request ids by address so it they can be queried later

5.6.2 Ambiguous bridge event emitted when bridging ERC20 yield token to ETH

Severity: Informational

Context: L1BlastBridge.sol#L237

Description: When bridging an ETH yield token (like `StEth`) to ETH using the `L1BlastBridge` an ERC20 bridge event is emitted via `_emitERC20BridgeInitiated`. On L1 an ERC20 token is used but on L2 the native ETH token is received. It's unclear if this should be considered an ERC20 or an ETH bridge.

Recommendation: It's up to interpretation if this should be an ETH or ERC20 bridge event. Double-check if `_emitERC20BridgeInitiated` or `_emitETHBridgeInitiated` should be used in this case. Our interpretation tends more towards `_emitETHBridgeInitiated`.

5.6.3 Changing `Constants.INITIALIZER` also requires unrequired initializations

Severity: Informational

Context: OptimismPortal.sol#L138

Description: There are multiple contracts in protocol, for which the version for reinitializer is coming from the Constants contract.

Changes in any one of such upgradable contracts will update the `Constants.INITIALIZER`, which means the version for all linked upgradable contracts will be updated. All of them now, need to be reinitialized even though they have no updates. Also, this all needs to happen in single transaction to avoid any third party from initializing any of these contracts maliciously.

Recommendation: Maintaining a local version for each contract may help, so that updating one contract does not impact others.

5.6.4 Event missing if `Admin` calls `coverLoss`

Severity: Informational

Context: Insurance.sol#L47-L53

Description: `coverLoss` can be called either by the `Admin` or the `YieldManager`. In normal cases, `YieldManager` will be calling this to cover losses by certain yield provider, which then generates an event like the one below showing how many funds were taken from the `Insurance` contract:

```
emit YieldReport(totalYield, totalInsurancePremiumPaid, totalInsuranceWithdrawal);
```

But if the same function was called by the `Admin`, then the event generation is missing.

Recommendation: For accounting purposes, it would be good to have an event generated for insurance withdrawal in the case `coverLoss` is called by `Admin`.

5.6.5 Withdrawal can bypass slashing

Severity: Informational

Context: YieldManager.sol#L306

Description: The withdrawal amount depends on `accumulatedNegativeYields` updated through `commitYieldReport` function.

In case `Admin`, by mistakes mistakenly, directly `finalize`s the withdrawal request without calling `commitYieldReport`, then the withdrawal will not get discounted even though it was meant to (since `finalize` has no check to ensure that `commitYieldReport` has been called in past `X` hours).

Recommendation: A check can be added to `finalize` to ensure that `commitYieldReport` has been called in the past `X` hours. Else, the call will fail. Also, a `force` can be added which can bypass the above check. This could be used in the case the slashing is too large and `Admin` wants to accommodate it later so that withdrawals are not heavily impacted.

Blast: We're planning on having daily updates for staking and yield reports. We don't intend on manipulating the frequency to favor some withdrawals, but we do expect some level of randomness if there is significant slashing.

5.6.6 Setting the `governor` address to `address(0)` gives governor permissions back to the contract

Severity: Informational

Context: Blast.sol#L41-L43, Blast.sol#L49-L51

Description: `governorNotSet()` returns `true` if the governor address is set to `address(0)`:

```
function governorNotSet(address contractAddress) internal view returns (bool) {
    return governorMap[contractAddress] == address(0);
}
```

As such, the `isAuthorized` modifier will still return `true` when the governor for a contract is configured to `address(0)`.

This might be misleading for developers since the documentation suggests that setting `governor` to the zero address revokes all governor functionality permanently:

> If a smart contract has never adjusted its yield/gas mode, then it can do so by sending a `Blast.configure` transaction to the main Blast deploy:
>
> After this initial call however, only the configured `governor` can adjust the contract's yield mode/gas mode/governor. The governor can be `address(this)`, a multisig, or an EOA.
>
> After setting a non-self-address governor, a contract won't be able to configure itself. This is intentional so that contracts that make untrusted delegate calls (for whatever reason) can prevent unauthorized access to their config and yield/gas claims.

Recommendation: Consider documenting that setting `governor` to `address(0)` gives governor permissions back to the contract itself.

5.6.7 Document that `YieldMode` is `AUTOMATIC` by default for `ERC20Rebasing` tokens

Severity: Informational

Context: Blast.sol#L5-L9, ERC20Rebasing.sol#L47-L48

Description: For `ERC20Rebasing` tokens (e.g. USDB, WETH), the default yield mode for all addresses, including contracts, is `AUTOMATIC`:

```
enum YieldMode {
    AUTOMATIC,
    VOID,
    CLAIMABLE
}
```

This is different from native ETH on L2 where the default for contracts is `VOID`, which might be a footgun for developers.

Recommendation: Consider documenting that the default yield mode for `ERC20Rebasing` tokens is `AUTOMATIC`.

5.6.8 Required approvals are commented out

Severity: Informational

Context: DSRYieldProvider.sol#L41-L43, LidoYieldProvider.sol#L43-L46, 'USDConversions.sol#L49-L55

Description: Approvals are commented out in the following functions:

- `DSRYieldProvider.initialize()`
- `LidoYieldProvider.initialize()`
- `USDConversions._init()`

However, these approvals are needed for their respective contracts/libraries to work.

For example, depositing DAI with `DSR_MANAGER.join()` requires allowance and `stake()` doesn't call `approve()` before depositing. `YieldManager` and other related contracts that delegatecall into the providers don't grant approval as well.

Recommendation: Remember to uncomment these lines before deployment. Alternatively, instead of granting an infinite allowance upon initialization, consider granting allowance only when required.

5.6.9 Centralization Risks

Severity: Informational

Context: Gobal scope

Description: Besides inheriting the centralization risks of Optimism, Blast introduces an `admin` that has the following powers:

- L2 $\rightarrow$ L1 withdrawals need to be finalized by the admin to be processed and paid out.
- Funds can be staked and are routed through admin-whitelisted contracts (providers) for yield:
  - Providers are `delegatecall`ed into from the `YieldManager`, resulting in a provider essentially controlling the `YieldManager`. The `L1BlastBridge` itself is the USD yield manager.
  - Funds can naturally be lost by the yield providers (for example, slashing events in Lido).
- USD-equivalent funds can be converted via `USDYieldManager.convert` which can incur liquidity and fee costs.

Recommendation: User should be aware of the risks. The `admin` needs to be thoroughly safeguarded from compromise.

5.6.10 Negative yield events can affect withdrawals that happened before

Severity: Informational

Context: YieldManager.sol#L306

Description: Withdrawals can be discounted in case a negative yield event happens. The discounted price is determined by the `accumulatedNegativeYields` variable which is only increased in `commitYieldReport`.

An L2 ETH withdrawal may be requested, the bridge's finalization period has passed but the admin has not finalized the request yet. Then a negative yield event happens and the admin calls `commitYieldReport` and `finalize` and the withdrawal is discounted, the user receives fewer ETH even though their withdrawal was practically finalized before the negative yield event.

Recommendation: In the current system there is no easy way to fix this as all steps involve manual admin interaction. In general, we recommend the admin to often commit yield and finalize withdrawals.

5.6.11 `enableInsurance` flag controls two distinct features

Severity: Informational

Context: YieldManager.sol#L229

Description: The `YieldManager.commitYieldReport` takes an `enableInsurance` parameter. This parameter controls two distinct features:

1. If insurance is taken on positive yield.
2. If insurance funds should be paid out on negative yield.

In addition, it's a single flag for all the providers. Meaning, it could be that only a single provider's insurance cannot pay out and the flag needs to be set to `false`. But then the positive yield of all other providers doesn't contribute to their respective insurances.

Recommendation: Think about if these should be two different flags. The first one of whether to pay insurance premiums could be defined on a provider basis (could repurpose `YieldProvider(_providers.at(i)).supportsInsurancePayment()`).

5.6.12 OpenZeppelin's v5 `Ownable2StepUpgradeable` does not initialize `owner`

Severity: Informational

Context: YieldManager.sol#L6

Description: The imported OpenZeppelin Upgradeable version is 4.7.3, which lacks `Ownable2StepUpgradeable` as it was introduced in a later version. Should the latest version be used, it is important to note that `Ownable2StepUpgradeable` does not call `__Ownable_init_unchained`, leaving the `owner` uninitialized.

Recommendation: Ensure that `__Ownable_init_unchained` is called in the inherited contracts, regardless of version imported.

5.6.13 Allow DAI ↔ USDC swaps via Curve's 3Pool

Severity: Informational

Context: USDConversions.sol#L24-L25

Description: DAI ↔ USDC swaps are restricted to Maker's PSM. There could be some scenarios where using Curve's 3Pool could be better. For instance, when there is a de-peg of either coin and there is positive slippage.

Recommendation: Allow DAI ↔ USDC swaps to use Curve's 3Pool.

5.6.14 USD conversion slippage parameter is bridged as `extraData`

Severity: Informational

Context: L1BlastBridge.sol#L183

Description: Users can directly bridge USDC or USDT (or other enabled USD yield tokens). These tokens are swapped to DAI before the actual DAI bridging happens. The users can define a slippage parameter for the swap. Because of technical reasons, this slippage parameter is passed as the `_extraData` parameter to the L2. There's no need to pass this data to L2. Furthermore, no other extra data can be defined as the code checks that the size is exactly the size of the slippage parameter.

Recommendation: Consider adjusting the code to not require the slippage parameter as part of the `_extraData`.

5.6.15 DAI withdrawals need to be manually claimed

Severity: Informational

Context: L1BlastBridge.sol#L143

Description: Contrary to ETH, when bridging back USDB the DAI is not transferred upon executing the message via `OptimismPortal.finalizeWithdrawalTransaction`. After this step, the admin still needs to finalize withdrawals and then the user needs to manually call `claimWithdrawal` to receive the DAI.

- Users need to wait for the bridge finalization period and only then the waiting period for the admin to finalize withdrawals begins.
- Some contracts might not have the functionality to claim the withdrawals.

Recommendation: Users and contract developers need to be aware that DAI withdrawals might take more time than ETH withdrawals and that they need to manually claim it.

Blast: This is correct, unfortunately, there is not a convenient way to initiate the ERC20 withdrawal request during the `OptimismPortal` prove withdrawal step as there is for ETH where we can just request `tx.value`. Without decoding the calldata from the withdrawal, the ERC20 withdrawal value is opaque until it calls the Blast bridge. Considering contracts that do not have this capability may require adding an authority to the withdrawal request that can call `claimWithdrawal` on the contracts behalf.

5.6.16 `L1BlastBridge.finalizeBridgeERC20` natspec clarification

Severity: Informational

Context: L1BlastBridge.sol#L112

Description: The natspec for `L1BlastBridge.finalizeBridgeERC20` states that it can only be called by "the other StandardBridge". However, it can only be called by the other `L2BlastBridge`.

Recommendation: As both `StandardBridge`s and `BlastBridge`s are deployed, the distinction becomes important. Consider changing the natspec:

```
  /// @notice Finalizes an ERC20 bridge on this chain. Can only be triggered by the other
- ///         StandardBridge contract on the remote chain.
+ ///         BlastBridge contract on the remote chain.
```

5.6.17 Additional `setYieldToken` checks

Severity: Informational

Context: L1BlastBridge.sol#L81-L104

Description: A token should never be a yield token for both ETH and USD. The contract bridging logic would break in this case.

Recommendation: Consider adding checks to `setUSDYieldToken` and `setETHYieldToken` : The token should not already be a yield token for the opposite yield token (USD $\leftrightarrow$ ETH).

5.6.18 `ILido.stake` could use full signature with return values

Severity: Informational

Context: LidoYieldProvider.sol#L11

Description: The signature for Lido's `stake` function is:

```
function submit(address _referral) external payable returns (uint256)
```

The `ILido` interface that is defined does not define the return value.

Recommendation: Consider using the full signature with the return value in the interface even if the return value is not needed.

5.6.19 `L2BlastBridge` error prefixes

Severity: Informational

Context: L2BlastBridge.sol#L48-L51

Description: The `L2BlastBridge.finalizeBridgeETHDirect` function currently uses `StandardBridge` error prefixes.

Recommendation: Consider changing the error prefix to `L2BlastBridge`.

5.6.20 Yield token price peg assumptions

Severity: Informational

Context: L1BlastBridge.sol#L230

Description: The USDB on L2 is backed by DAI and therefore pegged to the DAI price. There's currently also an implicit assumption that all ETH yield tokens are priced 1-to-1 with ETH.

Recommendation: Users and contract developers should be aware of the tokens backing the bridged L2 versions and that this price might deviate and not be observable, or only be observed delayed, on L2.

5.6.21 Yield distribution may become unfair

Severity: Informational

Context: OptimismPortal.sol#L413

Description: Lets say `YieldProvider` (used by `Blast` protocol) stops any new staking request for an indefinite time and is giving high yield to existing stakers. Users can bypass this by depositing funds to L2.

Although `Admin` will not be able to stake these new deposits, yield from existing stakers will get distributed to these new stakers. This might not be expected by existing users.

Note: This will eventually stop since more of such new user deposits will eventually decrease the effective yield distributed to all users.

Recommendation: Documenting such scenario can make users aware of yield splitting.

5.6.22 Clarity on Bridge selection

Severity: Informational

Context: L1BlastBridge.sol#L108, L1StandardBridge.sol#L87

Description: Currently both `L1StandardBridge/L2StandardBridge and L1BlastBridge/L2BlastBridge` can be used for bridging ETH from L1 to L2 and vice-versa. This could confuse users on which bridge to select for bridging ETH.

Recommendation: Consider documenting that both bridges can be used equally for bridging ETH without any loss for the user.

Blast: Allowing both bridges to deposit ETH isn't necessary, but given the way ETH deposits worked, it resulted in less modification to the `StandardBridge` base contract to allow both bridges to handle ETH bridging. It also shouldn't introduce any issues beyond possible confusion for users as you mentioned.

5.6.23 Claim on self address should be allowed

Severity: Informational

Context: ERC20Rebasing.sol#L191

Description: Claiming requires a user to provide a separate address. It would be very difficult for a user to maintain multiple address just for this purpose, while being more convenient if a user could claim to their own address instead.

Recommendation: Allow a user to claim on their own address. This can be done by modifying the current `claim` function (reversing `_deposit` and `_updateBalance`) as:

```
- if (account == recipient) {
-     revert CannotClaimToSameAccount();
- }
  // ...
- _deposit(recipient, amount);
- _updateBalance(account, newShares, newRemainder, _fixed[account]);
+ _updateBalance(account, newShares, newRemainder, _fixed[account]);
+ _deposit(recipient, amount);
```

5.6.24 Directly use `L1Bridge` for consistency with `_remoteToken`

Severity: Informational

Context: USDB.sol#L48-L51

Description: The natspec says that the `L1 USD Bridge` is to be used, but the actual bridge used is `L2BlastBridge`, as seen in the test initialization:

```
USDB usdb = new USDB({
   _bridge: address(l2BlastBridge),
   _remoteToken: address(DAI)
});
```

Recommendation: For consistency with `_remoteToken`, the `L1BlastBridge` should be used instead.

```
- SharesBase(address(StandardBridge(payable(_bridge)).OTHER_BRIDGE()))
+ SharesBase(_bridge)
```

5.6.25 Initialize all proxied logic contracts

Severity: Informational

Context: WETHRebasing.sol#L42, ETHYieldManager.sol#L14

Description: It's best practice to initialize all proxied logic contracts such that no malicious party can initialize them and potentially create issues. (Mostly through delegatecalling into a `selfdestruct`).

Recommendation: Consider initializing the proxied contracts mentioned above in the `constructor()` or by using the `_disableInitializers` function.

5.6.26 `claim()` recipient can be null

Severity: Informational

Context: ERC20Rebasing.sol#L187

Description: `claim()` doesn't sanity check that the recipient isn't the null address, unlike `_transfer()` which performs these checks on both `from` and `to`. It is thus possible for someone to send yield to the null address.

Recommendation: Consider checking that the recipient isn't the null address:

```
if (recipient == address(0)) revert TransferToZeroAddress();
```

5.6.27 `getClaimableEther` name is misleading

Severity: Informational

Context: WithdrawalQueue.sol#L148

Description: The `WithdrawalQueue` is used by both the ETH and USDB yield managers. The function to get the claimable funds for withdrawal requests is named `getClaimableEther`. In the case of the USDB yield manager, it returns the claimable DAI funds.

Recommendation: Consider renaming the `getClaimableEther`, `_getClaimableEther`, `_calculateClaimableEther` functions (and any other references to "Ether") to a more general name like `getClaimableFunds` or `getClaimableAmount`.

5.6.28 Code Redundancies

Severity: Informational

Context: WithdrawalQueue.sol#L6, WithdrawalQueue.sol#L17, WithdrawalQueue.sol#L99-L123, WithdrawalQueue.sol#L137-L203, WithdrawalQueue.sol#L30-L35, DSRYieldProvider.sol#L21, YieldManager.sol#L103, OptimismPortal.sol#L15

Description: There are various code redundancies (unused imports, duplicate imports, errors, internal getters) in `WithdrawalQueue`, `DsrYieldManager` and `YieldManager`.

Furthermore, some storage variables in `WithdrawalQueue` are missing visibility specifiers and default to internal. It's unclear why the internal getters for the internal storage variables are needed.

Recommendation: Consider adding explicit visibility specifiers to the storage variables and removing unused code.

5.6.29 Clarify behaviour when `minClaimRateBips` > `ceilClaimRate`

Severity: Informational

Context: Gas.sol#L159-L168

Description: The current implementation implicitly claims at the ceiling rate `ceilClaimRate` if `minClaimRateBips` > `ceilClaimRate`, which might seem as unexpected behaviour to some users.

Recommendation: For greater clarity, document this in the natspec, or have the function revert.

5.6.30 Sanity check that `_ceilClaimRate` does not exceed 100%

Severity: Informational

Context: Gas.sol#L61-L62, Gas.sol#L106-L107

Description: In addition to these checks, consider checking that `_ceilClaimRate` doesn't exceed 100% (`10_000`).

Recommendation: Add the following sanity check:

```
require(_ceilClaimRate <= 10_000, "_ceilClaimRate cannot exceed 100%");
```

5.6.31 Spelling / Grammar Improvements

Severity: Informational

Context: OptimismPortal.sol#L27, OptimismPortal.sol#L46, OptimismPortal.sol#L465, ERC20Rebasing.sol#L394, USDB.sol#L19, Insurance.sol#L11, L1BlastBridge.sol#L21, L1BlastBridge.sol#L30, L1BlastBridge.sol#L210, L2BlastBridge.sol#L13, USDConversions.sol#L57, USDConversions.sol#L143, LidoYieldProvider.sol#L34, YieldProvider.sol#L147, WETHRebasing.sol#L26, ERC20Rebasing.sol#L363, YieldManager.sol#L196, Blast.sol#L78, OptimismPortal.sol#L89, L1BlastBridge.sol#L114. L1BlastBridge.sol#L210

Description: The referenced lines have comments / spelling errors that can be improved for clarity.

Recommendation:

```
- Timestamp at whcih the withdrawal was proven.
+ Timestamp at which the withdrawal was proven.

- If the of this variable is the default L2 sender address
+ If address of this variable is the default L2 sender address

- recieved
+ received

- paramters
+ parameters

- interfactions
+ interactions

- Insurace
+ Insurance

- transfering
+ transferring

- uin256
+ uint256

- can only be bridge to ETH
+ can only be bridged to ETH
```

```
- stablcoin
+ stablecoin

- repsentation
+ representation

- WitdrawalQueue
+ WithdrawalQueue

- overriden
+ overridden

- it’s own share price that’s computed based on it’s current balance
+ its own share price that’s computed based on its current balance

- according to it's yield mode
+ according to its yield mode

- /// @param amount Amount to stake (wad).
+ /// @param amount Amount to unstake (wad).

-- // only allows contract or governor to configure contract
++ // only allow governor, or if no governor is set, the contract itself to configure

-- event WithdrawalProven(bytes32 indexed withdrawalHash, address indexed from, address indexed to,
↪  uint256 requestId);
++ /// @param requestId             Id of the withdrawal request
++ event WithdrawalProven(bytes32 indexed withdrawalHash, address indexed from, address indexed to,
↪  uint256 requestId);

- /// @param _remoteToken Address of the ERC20 on this chain.
+ /// @param _remoteToken Address of the corresponding token on the remote chain.

- require(_remoteToken == address(0), "L1BlastBridge: this token can only be bridge to ETH");
+ require(_remoteToken == address(0), "L1BlastBridge: this token can only be bridged to ETH");
```