Blast Node Security Review

Auditors

Dtheo, Lead Security Researcher

Mattsse, Lead Security Researcher

The-lichking, Lead Security Researcher

Blockdev, Security Researcher

Sujith Somraaj, Associate 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-node 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-node protocol. In this period of time a total of **22** issues were found.

Summary

Project Name	Blast
Repository	blast-node
Commit	be5187...601c32
Type of Project	L2, Infrastructure
Audit Timeline	Jan 5th to Jan 19th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	0	0
Medium Risk	1	0	0
Low Risk	13	0	0
Gas Optimizations	0	0	0
Informational	7	0	0
Total	**22**	**0**	**0**

5 Findings

5.1 High Risk

5.1.1 Gas tracking introduces resource consumption related DOS

Severity: High Risk

Context: op-geth/core/vm/gas_tracker.go#L35, op-geth/core/vm/runtime/runtime.go#L205

Description: Blast introduces gas tracking so that developers have new ways to capture revenue. There are two forms of DOS that this might introduce. The first is related to gas consumption and is mentioned in the notion doc provided by the blast team. The second is actual node resource consumption (eg. how long it takes to process a transaction and complete a state transition). This issue is referring to the later.

In the current implementation every `vmenv.Call()` must initialize a new gas tracker that will keep track of the total gas used by each contract during the processing of a transaction. For each contract used during a transaction's execution overhead will also be incurred when `SetState()` commits the the state to disk. There are also some places with redundant hashing in the gas allocation code (ref. AllocateDevGas() contains redundant hashing issue). This increases resource overhead during the processing (example) as well as when the gas accounting is updated on state transitions.

Blast aims to use the default optimism config which supports an upper bounds of 6 blocks per mainnet block (2 seconds per blast block). If extra resources consumed in every state transition do not carry adequate gas costs, then a malicious user could possibly DOS the sequencer by push the processing time outside of this 2 second bound.

One example way to stress the node in this manner would be to deploy 10K contracts that are just nested calls into each other. Respecting the max call depth would require separating this into multiple call chains. Then a user can make ~30 calls that would call into all 10K contacts and force the gas tracker to track and update 10K storage slots in the state transition. The initial deploying of the 10K malicious contracts is a one time cost then abusing the sequencer with 10K storage updates each subsequent block would be relatively cheap.

Recommendation: Consider:

- Optimizations of the gas tracking code.
- Limiting EVM call stack depth to prevent the 10K nested call attack mentioned above.
- Reducing the config to a safe number of blast blocks to target per mainnet block.
- Updating gas costs for all call related opcodes to account for the extra resource consumption required to track gas.
- Exploring other ways to prevent small calls from abusing this (limiting gas redemption for calls with at least X gas, making all call related opcodes cost increase if more than Y in a transactions, etc...).

5.2 Medium Risk

5.2.1 `MemoryStateDB` contains data race in `DeleteState()`

Severity: Medium Risk

Context: op-chain-ops/state/memory_db.go

Description: Upstream Optimism has two functions in `op-chain-ops/state/memory_db.go` that do not use the `MemoryStateDB` `sync.RWMutex` correctly. This was discovered when getting some background knowledge to aid in reviewing [DIFF] optimism/op-chain-ops. While this is outside of the scope of the Optimism diff it is worth mentioning here to make sure that it does not get left out of Blast for whatever reason. This could not be abused by an attacker but could cause instability when using the chain-ops utilities to modify the chain. Here is a PR with a fix.

Recommendation: Either merge in the upstream change, merge in the single commit with the fix from the upstream PR via `git cherry-pick`, or make the same change to blast manually.

5.3 Low Risk

5.3.1 `(i ImmutableConfig) Check()` is missing validation checks for new Blast fields

Severity: Low Risk

Context: optimism/op-chain-ops/immutables/immutables.go#L31, PR 29

Description: `(i ImmutableConfig) Check()` contains checks for nearly every field in `Immutable`. Checks have been added for the following additional fields added by blast: `i["Shares"]["price"]`, `i["Gas"]["baseClaimRate"]`, and `i["L2BlastBridge"]["otherBridge"]`. However, the following news blast fields still do not have a check:

- `immutable["Shares"]["reporter"]`
- `immutable["Gas"]["admin"]`
- `immutable["Gas"]["zeroClaimRate"]`
- `immutable["Gas"]["baseGasSeconds"]`
- `immutable["Gas"]["ceilGasSeconds"]`
- `immutable["Gas"]["ceilClaimRate"]`
- `immutable["Blast"]["yieldContract"]`
- `immutable["USDB"]["bridge"]`
- `immutable["USDB"]["remoteToken"]`

Adding these checks will help the `BuildOptimism()` functionality error gracefully if these fields are not included or malformed in the target `json` file. This will prevent unexpected behavior.

Recommendation: Add checks for the new fields that do not have any.

5.3.2 USDB predeployment is skipped

Severity: Low Risk

Context: optimism/op-chain-ops/immutables/immutables.go#L198-L206

Description: USDB predeployment is skipped as highlighted above. Blast team says the USDB related code was commented as there was an issue with USDB's constructor and this has since been fixed outside of review scope.

Recommendation: Fix the USDB deployment.

Note: This appears to be an artifact of the development process that the developers are aware of so the severity is set to low. If this is not addressed before launch the impact will critical. It is strongly advised that this issue be addressed with care. The code as it stands at the review commit hash could prompt a large loss of funds.

5.3.3 `op-geth/core/vm/contracts.go` change makes multiple methods less efficient

Severity: Low Risk

Context: PR 27

Description: op-geth/core/vm/contracts.go contains multiple `Run()` methods for various hashing (and other related) algorithms that appear to have been changed to include extra arguments (`caller`, `db`, and `readOnly`). These extra references are not needed and may slow down the use of these methods. This can increase the possibility of resource consumption related attacks on the node (see `AllocateDevGas()` contains redundant hashing and Gas tracking introduces resource consumption related DOS).

Recommendation: Remove the unneeded arguments.

5.3.4 Invariant panics risk node operation

Severity: Low Risk

Context: op-geth/core/vm/gas_tracker.go#L66, op-geth/core/state_transition.go#L505, op-geth/core/vm/gas_tracker.go#L28

Description: There are multiple places where invariants are tracked with `panic`. The use of panics in critical block processing code should be carefully considered before the code is used in production. If any of these trigger the node will experience a liveness failure.

```
if totalGasAccount.Cmp(remainingGas) > 0 {
    panic("gas accounting inflation")
}
```

```
if amount > gtm.gasUsed || amount > gtm.allocations[address] {
    panic("impossible")
}
```

```
if !isGasAccountingCorrect {
    // TODO(blast): remove panic?
    panic(fmt.Sprintf("Gas used mismatch: st.gasUsed() = %d, gasTracker.GetGasUsed() = %d",
↪   st.gasUsed(), gasTracker.GetGasUsed()))
}
```

Note: This audit has not identified a way to trigger these invariants so this issue is classified as a low severity. However, if an unknown issue or future regression triggers any of these conditionals the L2 will experience a liveness failure.

Recommendation: Consider using a logging system that can surface these invariants as they arise. If panics are desirable to surface invariants while fuzzing then consider wrapping them in a compiler directive that does not compile them into production code or using `recover()` to catch them.

5.3.5 `AllocateDevGas()` contains redundant hashing

Severity: Low Risk

Context: op-geth/core/vm/gas_tracker.go#L134

Description: `AllocateDevGas()` contains redundant calls to `getHash()` which performs the computationally expensive `Keccak256Hash` algorithm. This is done twice for every contract that is used during a transaction. Having redundant computation can expose the node to resource consumption related DOS, exacerbating the attack mentioned here. Gas tracking should be as implemented as efficient as possible in order to charge the lowest possible gas cost for call related opcodes as well as to prevent resource costs from exceeding appropriate gas costs.

The redundant calls are made through the following call chains:

- `AllocateDevGas()` → `UpdateGasPredeploy()` → `UpdateGasParameters()` → `getStorageSlots()` → `getHash()`
- `AllocateDevGas()` → `getGasMode()` → `getHash()`

Call chains entry points denoted in source by "`AUDIT COMMENT`" below (as seen in op-geth/core/vm/gas_tracker.go#L35):

```
func (gtm *GasTracker) AllocateDevGas(gasPrice *big.Int, refund uint64, state StateDB, timestamp
↪  uint64) {
    remainingGas := new(big.Int).SetUint64(gtm.gasUsed - refund)
    netGas := new(big.Int).SetUint64(gtm.gasUsed)
    accumulatedGas := new(big.Int)
    totalGasAccount := new(big.Int)
    blockTimestamp := new(big.Int).SetUint64(timestamp)
    for addr, rawAmount := range gtm.allocations {
        // find scaled gas units
        parsedRawAmount := new(big.Int).SetUint64(rawAmount)
        scaledGasUnits := new(big.Int).Div(new(big.Int).Mul(remainingGas, parsedRawAmount), netGas)
        totalGasAccount.Add(totalGasAccount, scaledGasUnits)

        // skip allocation of gas to contracts that dont accumulate
        // AUDIT COMMENT: getHash() call chain
        gasMode := getGasMode(state, addr)
        if !gasMode {
            continue
        }

        accumulatedGas.Add(accumulatedGas, scaledGasUnits)

        // calculate gas in wei terms
        fee := new(big.Int).Mul(scaledGasUnits, gasPrice)

        // update gas predeploy
        if fee.Cmp(common.Big0) > 0 {
            // AUDIT COMMENT: getHash() call chain
            updateGasPredeploy(state, addr, fee, blockTimestamp)
        }
    }
```

Recommendation: Either calculate the hash once per contract and pass the hash into the required calls or implement an efficient cache to store recently calculated hashes.

Note: This could be used to abuse the issue "`Gas tracking introduces resource consumption related DOS`" which would make this issue significantly more severe. While we have identified it as a low it it highly recommend that it be addressed.

5.3.6 `(b *blast) Run()` `caller` authorization conditionals should be placed before input deserialization

Severity: Low Risk

Context: op-geth/core/vm/contracts.go#L1239, op-geth/core/vm/contracts.go#L1278

Description: `(b *blast) Run()` contains two code blocks performing byte deserialization that require an authorized `caller`. The first is the `claimSelector` handler and the second is the `configureSelector` handler. Neither of the conditionals (1, 2) that check the authorization of the `caller` have a dependency on `solidityInput`, which is a byte buffer originating from an untrusted source. Any user making a transaction calling these handlers can populate this buffer with whatever they want.

Due to this there are currently multiple byte deserialization routines that are reachable by untrusted callers (1, 2). This is attack surface that does not need to be exposed. Moving these conditionals to the start of their respective handler routines will not only be more efficient in handling malformed calls but it will also reduce the code that the node exposes to untrusted inputs.

Recommendation: Make the following modifications:

- op-geth/core/vm/contracts.go#L1216:

```
  if bytes.Equal(selector, claimSelector) {
+     // authorize contract
+     if caller != params.BlastAccountConfigurationAddress {
+         return nil, ErrExecutionReverted
+     }
      contract, err := solidityInput.readAddress()
      if err != nil {
          return nil, err
      }
      recipient, err := solidityInput.readAddress()
      if err != nil {
          return nil, err
      }
      desiredAmount, err := solidityInput.readU256()
      if err != nil {
          return nil, err
      }
      if readOnly {
          return nil, ErrExecutionReverted
      }

      // validate that desired amount is > 0
      if desiredAmount.Sign() < 0 {
          return nil, ErrExecutionReverted
      }

-     // authorize contract
-     if caller != params.BlastAccountConfigurationAddress {
-         return nil, ErrExecutionReverted
-     }
```

- op-geth/core/vm/contracts.go#L1264:

```
  } else if bytes.Equal(selector, configureSelector) {
+     // authorize contract
+     if caller != params.BlastAccountConfigurationAddress {
+         return nil, ErrExecutionReverted
+     }
      contract, err := solidityInput.readAddress()
      if err != nil {
          return nil, err
      }
      flags, err := solidityInput.readUint8()
      if err != nil {
          return nil, err
      }
      if readOnly || flags > 2 {
          return nil, ErrExecutionReverted
      }

-     // authorize contract
-     if caller != params.BlastAccountConfigurationAddress {
-         return nil, ErrExecutionReverted
-     }
```

5.3.7 `AllocateDevGas()` divide-by-zero can cause denial of service

Severity: Low Risk

Context: op-geth/core/vm/gas_tracker.go#L44

Description: The following line in `AllocateDevGas()` will `panic("division by zero")` if `netGas` is 0:

```
scaledGasUnits := new(big.Int).Div(new(big.Int).Mul(remainingGas, parsedRawAmount), netGas)
```

If this is triggered it will bring the node down.

`NewGasTracker()` sets this field as 0 initially. There are a few ways that this field may continue to be 0 upon reaching this `Div()`.

One is through the `stateTransition()->TransitionDB()->innerTransitionDB()->AllocateDevGas()` call chain in testing or offline state modifications.

The other more serious possibility would be down-stream of the `ApplyMessage()` call chain that can originate from 16 possible functions including `callContract()`, `Apply()`, `precacheTransaction()`, `applyTransaction()`, and `doCall()`. It might also be possible for an attacker to abuse other gas tracking functionality like exempt precompiles or conditions triggering `RefundGas()` to control the value of `netGas` to trigger this panic. A malicious sequencer would have even more control.

Recommendation: Check that `gtm.gasUsed` is greater than zero upon entering `AllocateDevGas()` and return immediately if it is not. This will avoid the panic and `AllocateDevGas()`'s functionality will not be needed if `gtm.gasUsed` is 0.

Note: During this review there was no call chain identified that can trigger this bug outside of a testing environment so it has been labeled as a low. If an unknown issue can be abused to trigger this than the impact would be significant as the sequencer would stop running. It is highly recommended that this issue be addressed despite its low severity categorization.

5.3.8 No `nil` check on `ZeroClaimRate`

Severity: Low Risk

Context: optimism/op-chain-ops/genesis/config.go#L230

Description: Blast defines some new parameters:

```
GasAdmin       common.Address `json:"gasAdmin"`
ZeroClaimRate  *hexutil.Big   `json:"zeroClaimRate"`
BaseClaimRate  *hexutil.Big   `json:"baseClaimRate"`
CeilClaimRate  *hexutil.Big   `json:"ceilClaimRate"`
BaseGasSeconds *hexutil.Big   `json:"baseGasSeconds"`
CeilGasSeconds *hexutil.Big   `json:"ceilGasSeconds"`
```

It then does a zero or `nil` check on them expect for `ZeroClaimRate`:

```
if d.BaseGasSeconds == nil {
    return fmt.Errorf("%w: Base Gas Seconds cannot be nil", ErrInvalidDeployConfig)
}
if d.BaseClaimRate == nil {
    return fmt.Errorf("%w: Base Claim Rate cannot be nil", ErrInvalidDeployConfig)
}
if d.CeilGasSeconds == nil {
    return fmt.Errorf("%w: Ceil Gas Seconds cannot be nil", ErrInvalidDeployConfig)
}
if d.CeilClaimRate == nil {
    return fmt.Errorf("%w: Ceil Claim Rate cannot be nil", ErrInvalidDeployConfig)
}
```

Recommendation: Add a `nil` check for `ZeroClaimRate`.

5.3.9 To implement TODOs found in the code risking node ops

Severity: Low Risk

Context: account_proof.go#L101, account_proof.go#L107

Description: This contains the TODO's mentions in the code which point to missing code implementation. This acts as more of a tracker/reminder that these need to be implemented.

Recommendation: The todo's mentioned to be implemented in the correct way as mentioned or edited for changes through review.

5.3.10 `SelfDestruct` permanently deletes all unclaimed yield

Severity: Low Risk

Context: op-geth/core/vm/instructions.go#L826

Description: Contracts using `yieldClaimable` will have all unclaimed yield deleted when selfdestructing.

Consider a contract that configures claimable yield like the `ContractWithAutomaticYield` example on the blast website devs page.

If the contract calls `SelfDestruct` only its `fixed` balance will be sent to the `beneficiary`. Its `shares` and `remainder` will be deleted, even if their value exceeds `fixed`, which is the case when before is claimed.

The accounting logic for self destruct can be seen in `opSelfdestruct()` which calls `GetBalance()`. `GetBalance()` calls `Balance()` which handles the automatic yield contracts' balances correctly. The `YieldClaimable` case is treated the same as the `YieldFixed` case, disregarding any yield that has accrued. Since both the automatic and fixed yield cases are handled correctly when selfdestructing (eg. sending the contracts entire yield-included value to the beneficiary), the claimable yield case can catch developers off guard, leading to lost funds.

Recommendation: To adhere to the principle of least surprise, make SelfDestruct flows handle claimable yield contracts similar to automatic and fixed, cashing out the contract for its full value to the `beneficiary`, unclaimed yield included. If it is important to keep this functionality to ensure EVM compatibility then consider providing a method for all `yieldClaimable` contracts to use that will allow them to claim all yield and gas refunds atomically in the same function that they call `SelfDestruct`. At minimum it is recommended to include a warning about this in the documentation explaining `yieldClaimable` contract types.

Note: Changing the functionality of the `SelfDestruct` opcode can change EVM equivalence so the severity of the issue has been downgraded as the tradeoff of the fix must be considered. Documentation of the nuances of `yieldClaimable` contracts and their behavior when executing `SelfDestruct` is recommended.

5.3.11 Configuring a `YieldClaimable` account to `YieldClaimable` resets the claimable balance

Severity: Low Risk

Context: op-geth/core/state/state_object.go#L511-L513

Description: Setting a flag of an account in `YieldClaimable` mode to again in `YieldClaimable` mode updates its `fixed` field to include the previous claimable balance, and the claimable amount goes to 0:

```
case types.YieldClaimable:
    fixed = value
    shares, remainder = computeSharesAndRemainder(sharePrice, value)
```

This is a special case where the yield mode isn't changed. Note that the claimable balance goes to 0 whenever the yield mode is changed but that may be expected if the new mode isn't `YieldClaimable`.

Recommendation: No code change needed if this is to be expected. A note to developers in Blast documentation will be useful to avoid any surprises.

5.3.12 `SubClaimableAmount()` can claim more than the maximum claimable balance

Severity: Low Risk

Context: op-geth/core/state/state_object.go#L545

Description: `SubClaimableAmount(amount *big.Int)` doesn't check that `amount` doesn't exceed the maximum claimable. The current call path checks this isn't the case before calling this function. However, a new invocation may not do that which could lead to issues with the miscalculated shares and remainder, affecting the system overall.

Recommendation: Check that `amount` doesn't exceed the maximum claimable amount, and if so reduce the amount to be claimed to the maximum like done at op-geth/core/vm/contracts.go#L1248-L1259.

5.3.13 Share remainder becomes increasingly inefficient over time

Severity: Low Risk

Context: op-geth/core/state/state_object.go#L430

Description: Share price can never decrease. This introduces an inefficiency that will continue to get worse over time when the number of shares are calculated in `computeSharesAndRemainder()`. Since the remainder can be as large as the share price without converting to a full share the size of the inefficient remainder will continue to grow and could become very large in the future. This will get worse if the price of Ethereum grows over time as well.

Recommendation: Support some functionality to reduce share price overtime. This could be the the form of something similar to a stock split in tradfi, possibly another duty of the Blast admin.

5.4 Informational

5.4.1 Op-geth and optimism contain multiple failing tests and code without tests

Severity: Informational

Context: PR 39, PR 40, PR 51

Description: There are multiple tests that are failing or have been commented out. Some of these are Blast specific tests that are not inherited from upstream Optimism code. There are also multiple large portions of code that do not have any testing coverage for which tests should be created.

Recommendation: Tests should be fixed to prevent regression as development continues. New tests should be created for new code without test coverage.

5.4.2 Referencing enums in their integer notation makes code less readable

Severity: Informational

Context: optimism/op-chain-ops/genesis/layer_two.go#L82-L83, op-geth/core/vm/gas_tracker.go#L49, op-geth/core/vm/gas_tracker.go#L94

Description: There are multiple places where `Flags` and `gasMode` are referenced or compared with integers. This makes the code less readable and more difficult to maintain.

Recommendation: Reference enums by name (eg. `types.yieldAutomatic` instead of `1`).

5.4.3 `UseGasWithOp()` contains an unneeded conditional and call

Severity: Informational

Context: op-geth/core/vm/contract.go#L205-L209

Description: `UseGasWithOp()` contains an unneeded conditional and call to `op.IsContractCreate()`:

```
if op.IsContractCall() {
    c.gasTracker.UseGas(c.Address(), evm.baseGasTemp+evm.coldCostTemp)
} else if op.IsContractCreate() {
    c.gasTracker.UseGas(c.Address(), gas)
} else {
    c.gasTracker.UseGas(c.Address(), gas)
}
```

The `else if op.IsContractCreate()` conditional branch shown above can be removed.

Recommendation: Remove the `else if op.IsContractCreate()` conditional.

```
  // UseGas attempts the use gas and subtracts it and returns true on success
  func (c *Contract) UseGasWithOp(gas uint64, op OpCode, evm *EVM) (ok bool) {
      if c.Gas < gas {
          return false
      }
      c.Gas -= gas

      if op.IsContractCall() {
          c.gasTracker.UseGas(c.Address(), evm.baseGasTemp+evm.coldCostTemp)
-     } else if op.IsContractCreate() {
-         c.gasTracker.UseGas(c.Address(), gas)
      } else {
          c.gasTracker.UseGas(c.Address(), gas)
      }

      return true
  }
```

5.4.4 Typos

Severity: Informational

Context: state_object.go#L483, statedb.go#L427, state_object.go#L659, state_object.go#L78, journal.go#L112, statedb.go#L371, state_account.go#L39

Description: There are a few typos in the code and comments:

- **statedb.go**

```
- func (s *StateDB) GetFlags(addr common.Address) uint8
+ func (s *StateDB) GetFlag(addr common.Address) uint8
```

```
  type BalanceValues struct {
- Flags     uint8

  type BalanceValues struct {
+ Flag     uint8
```

- **state_object.go**:

```
- func (s *stateObject) SetFlags(flags uint8)
+ func (s *stateObject) SetFlag(flag uint8)
```

```
- func (s *stateObject) Flags() uint8
+ func (s *stateObject) Flag() uint8
```

```
- // Cache flags.
+ // Cache flag.
```

- **journal.go**

```
- balanceValuesChange struct {
  // ...
- prevFlags     uint8

+ balanceValueChange struct {
  // ...
+ prevFlag    uint8
```

- **state_account.go**

```
- Flags uint8 // valid values: 0-2
+ Flag uint8  // valid values: 0-2
```

Recommendation: Consider fixing the above mentioned typos and related typos, i.e. `flags` → `flag`.

5.4.5 Code diff can be explained in comments

Severity: Informational

Context: layer_two.go#L72-L76

Description: Code diff from Optimism sometimes is not obvious for example at optimism/op-chain-ops/genesis/layer_two.go#L72-L76:

```
} else if db.Exist(addr) {
    db.DeleteState(addr, AdminSlot)
} else {
    // TODO(p): this is kinda weird. maybe we can just make a custom namespace so this behaves correctly
    db.CreateAccount(addr)
```

Recommendation: Consider adding comments to explain why the code was updated.

5.4.6 `UseGasNatively()` and `UseGasForConstantCost()` are duplicates

Severity: Informational

Context: op-geth/core/vm/contract.go#L175-L194

Description: `UseGasNatively()` and `UseGasForConstantCost()` have the same function body. One of them can be removed.

Recommendation: Delete one of these functions.

5.4.7 Incorrect comments

Severity: Informational

Context: op-geth/core/vm/contracts.go#L1233-L1234, op-geth/core/vm/contract.go#L196-L197

Description:

- `desiredAmount.Sign() < 0` ensures that `desiredAmount` amount is ≥ 0. Hence, it doesn't follow the comment.

```
// validate that desired amount is > 0
if desiredAmount.Sign() < 0 {
    return nil, ErrExecutionReverted
}
```

- This comment for `UseGasWithOp()` is copied from `UseGas()`:

```
// UseGas attempts the use gas and subtracts it and returns true on success
func (c *Contract) UseGasWithOp(gas uint64, op OpCode, evm *EVM) (ok bool) {
```

Recommendation:

- Determine if `desiredAmount == 0` is to be allowed, and update the code or the comment.
- Update the comment for `UseGasWithOp()`.