Overprotocol Security Review

Auditors

Dtheo, Lead Security Researcher

Jtraglia, Lead Security Researcher

Shotes, Security Researcher

Report prepared by: Lucas Goiriz

August 19, 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

OverProtocol is a brand new layer 1 with lightweight nodes empowering personal computers, enabling anyone to run a node on their PCs and become a validator. This is made possible by OverProtocol's layered architecture through Ethanos, which significantly decrease the resources required for block validation.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of kairos 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, Overprotocol engaged with Spearbit to review the kairos protocol. In this period of time a total of **15** issues were found.

Summary

Project Name	Overprotocol
Repository	kairos
Commit	826a05...16a752
Type of Project	Infrastructure, Node
Audit Timeline	Jun 6 to Jun 20
Two week fix period	Jul 1 - Jul 9

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	1	1	0
High Risk	1	1	0
Medium Risk	0	0	0
Low Risk	4	4	0
Gas Optimizations	0	0	0
Informational	9	0	0
Total	**15**	**6**	**0**

5 Findings

5.1 Critical Risk

5.1.1 A malicious entity can grieve Restoration Servers for their funds

Severity: Critical Risk

Context: core/vm/evm.go#L813-L862

Description: Due to the location of the `CanTransfer()` check in `(evm *EVM) Restore()` it is possible for a malicious entity to grieve honest restoration servers for significant amount of gas. This can become a more serious issue if a large portion of restoration servers run out of gas and their accounts expire before owners realize. A competing restoration server may want to do this to harm its competition.

An example attack could be a malicious actor creating a large number of accounts (100+) and seeding them all with 10 OVER. It would then need to let some time pass and the accounts go dormant (the longer the accounts are dormant the larger the restoration proof and the more effective this attack will be). The entity then could create its own restoration proofs with the fees costing the entire 10 OVER for each account. This would effectively collect all of the account balances into its own account. It could wait until it either controls a validator that is the proposer for the next block or it can use increased priority fees to make sure that its transactions are placed early in the next block. It can send all of its own `restorationTX`'s at the same time that it makes restoration requests to all of other known restoration servers in the network.

The remaining honest restoration servers will query their local nodes that will not have their state updated yet with the malicious entity's self created `restorationTX`'s. All checks will pass including `checkFeePayable()` which will see 10 OVER in each account. The honest restoration servers will all create and submit `restorationTX`'s for the 100+ accounts. By the time they are processed by the EVM the malicious entities `restorationTX`'s will have already processed and the victim restoration servers' `restorationTX`'s will fail at the requestors balance check in `(evm *EVM) Restore()`. When this happens all of the honest restoration servers' gas will be consumed:

```
// The sender of the restore data has to have enough balance to send the restoration fee
if restoreData.FeeRecipient != nil && restoreData.Fee.Sign() != 0 {
    if !evm.Context.CanTransfer(evm.StateDB, sender, restoreData.Fee) {
        err = ErrInsufficientBalance
    } else {
        evm.Context.Transfer(evm.StateDB, sender, *restoreData.FeeRecipient, restoreData.Fee)
    }
}
if err != nil {
    evm.StateDB.RevertToSnapshot(snapshot)
    if err != ErrExecutionReverted {
        gas = 0
    }
}
```

The result of this attack is that the restoration servers will have their gas funds burned without being able to recoup the fees require to cover their loss. It its current form the max gas that can be grieved is only limited by the base fee and and the size of the proof (length of time the account has been dormant). Using hundreds or thousands of accounts requiring large proofs can enable an attacker to trick honest restoration servers from burn all of their balances in gas.

Recommendation: Move the following check up to before the gas consumption portion of `(evm *EVM) Restore()`

```
if restoreData.SourceEpoch != epochCoverage {
    return nil, gas, ErrInvalidSourceEpoch
}
```

Overprotocol: Fixed in commit b4625c76.

Spearbit: The Spearbit Review team confirmed that this issue has been mitigated by moving the `CanTransfer()`

and `restoreData.SourceEpoch != epochCoverage` checks up to before the gas proof gas is consumed in `Restore()`. These changes were introduced in commit b4625c76.

5.2 High Risk

5.2.1 Precisely timed malicious restoration requests can grieve restoration servers

Severity: High Risk

Context: core/vm/evm.go#L821-L826

Description: In order to execute a restoration transaction, the `restoreDataSigner.Sender` account that signed and sent the `restoreData` request must have sufficient balance to cover the fee. However, there is no check to make sure that the `restoreDataSigner.Sender` account actually exists in the current state. If that account has been expired from the state, the transaction will fail upon checking `CanTransfer`, and the gas paid by the honest restoration server will be burned without any repercussions to the `restoreDataSigner.Sender` account. Therefore it is possible to execute a timing attack where the `restoreDataSigner.Sender` account exists when the restoration server verifies a request but then expires from the state before the restoration transaction gets executed in the EVM.

For example, I have an account we'll call "malicious account" that has been inactive for `2*SweepEpoch - 1` blocks. This account will expire on the next epoch. In the last slot of that epoch before expiration, the "malicious account" will sign and send a valid `restoreData` request for any expired account to an honest restoration server. The restoration server will generate the valid proof and send a valid restoration transaction for this request. By the time this restoration transaction gets executed in the EVM (which would be in the next slot), the "malicious account" will have expired. The "malicious account" will not have to pay any fee while costing the restoration server all of the gas of the transaction which could be made to be very large.

If this got scaled up similar to a scenario described in the issue "A malicious entity can grieve OVER network Restoration Servers for their funds", then this could cause some problems. Though this scenario is similar, the source and trigger of the vulnerability is unique and requires its own solution. It is also a more constrained attack scenario, requiring correct timing on the epoch boundary. But each slot is 12 seconds, and I believe that is plenty of time to execute this with reasonable expectations of success.

Recommendation: If the `restoreDataSigner.Sender` account is not the same as the `restoreData.Target` account, then implement a check to make sure the restoreDataSigner.Sender account exists in the current state.

```
  sender, err := restoreDataSigner.Sender(restoreData)
  if err != nil {
      return nil, 0, err
  }
  // Retrieve and get the current state of the target account
  target := restoreData.Target
+
+ // Check sender exists in current state if it's not the restoration target.
+ if sender != target {
+     if !evm.StateDB.Exist(sender) {
+         return nil, gas, ErrNonexistentSender
+     }
+ }
```

Though unnecessary, you could replace the `!evm.StateDB.Exist(sender)` with `!evm.Context.CanTransfer(evm.StateDB, sender, restoreData.Fee)` for the additional assurance that it can do the transfer.

Overprotocol: Fixed in commit b4625c76.

Spearbit: The Spearbit Review team confirmed that this issue is mitigated by checking if the sender can pay the restoration fee before performing the transaction. This check is at core/vm/evm.go#L806-L810 and was introduced in commit b4625c76.

5.3 Low Risk

5.3.1 Ignored returned gas and nonce update in `Create*WithUiHash`

Severity: Low Risk

Context: core/vm/evm.go#L620-L622

Description: If a call to precompiled contracts `createContractWithUiHash` or `create2ContractWithUiHash` fails before attempting to deploy the new contract code, the remaining gas returned will all be burned due to checks in `evm.Call` which that consume all gas on error.

There are 6 locations where an error could happen and the call attempts to return gas:

- core/vm/evm.go#L621
- core/vm/evm.go#L624
- core/vm/evm.go#L727
- core/vm/evm.go#L734
- core/vm/evm.go#L750
- core/vm/evm.go#L757

In each of these locations, an error happens and it attempts to return the gas. However, the gas returned here will get consumed anyways due to the fact that that it is a precompile. We can see this gas consumption behavior in the function `evm.Call` at core/vm/evm.go#L259-L263 (as well as in the other call functions `evm.CallCode`/`evm.DelegateCall`/`evm.StaticCall`).

This means that any failures in the call to the precompiled contracts will always consume all gas if the failure is not a REVERT and it fails before attempting to deploy the code. This differs from the `CREATE`/`CREATE2` functionality which would return any leftover gas.

In addition to this issue, there is another issue with the snapshotting and reverting happening twice when a REVERT occurs, once in `evm.Call` at core/vm/evm.go#L260 and once in `evm.createWithUi` at core/vm/evm.go#L693. This is an issue for two reasons - it doubles the amount of work when reverting, and the `evm.Call` revert also reverts the updated Nonce at core/vm/evm.go#L737 and the updated access list at core/vm/evm.go#L629.

Recommendation: Match the behavior of `CREATE`/`CREATE2` by creating a gas exception for the two precompiles. Since the two `CreateWithUI` precompiles revert themselves on error, we do not need to revert in the `evm.Call` when we are executing the two precompiles. This can be done by checking if the called contract is a "Creation Precompile" using `common.IsCreationPrecompiled()`. This bends the rules of calling contracts specificly for the precompiles which should not be done lightly. This can be done by adding the following addition in each "Call" function core/vm/evm.go#L260-L263, core/vm/evm.go#L313-L317, core/vm/evm.go#L357-L360, core/vm/evm.go#L413-L416:

```
-   if err != nil {
+   if err != nil && !common.IsCreationPrecompiled(addr){
        evm.StateDB.RevertToSnapshot(snapshot)
        if err != ErrExecutionReverted {
            gas = 0
        }
    }
    return ret, gas, err
```

In performing this check, we are fixing both issues at the same time - allowing the return of gas from those two "Creation Precompiles", only reverting once when an error happens in the "Creation Precompiles", and keeping the updated values of the account nonce and access list.

Overprotocol: Fixed in commit 8e9e8f93.

Spearbit: The Spearbit Review team confirmed that this issue is mitigated by the suggested fixes being introduced in commit 8e9e8f93.

5.3.2 `REVERT` will not return remaining gas in `Create*WithUiHash`

Severity: Low Risk

Context: core/vm/contracts.go#L695-L697, core/vm/contracts.go#L727-L729

Description: If a `REVERT` opcode were executed during contract deployment in a `createContractWithUiHash` or `create2ContractWithUiHash` precompile call, the expectation is that the rest of the not-yet-consumed gas will be returned. There is even a special check in `evm.createWithUiHash` (see core/vm/evm.go#L693-L695) that ensures that the `REVERT` opcode has the privilege of getting gas back. This is also the behavior of `evm.create` and `evm.Call`.

However, the `createContractWithUiHash` and `create2ContractWithUiHash` precompiles both ignore the returned gas value if there is any error in the `evm.CreateWithUiHash` and `evm.Create2WithUiHash` operations. On error, they instead return 0 gas, burning any leftover gas. You can see this behavior at core/vm/contracts.go#L695-L697 and at core/vm/contracts.go#L727-L729.

It's not super common to execute a `REVERT` inside of contract deployment code, but it does happen. Burning all of the gas on `REVERT` in contract deployment code is unexpected behavior.

Recommendation: Change `return nil, 0, suberr` to `return nil, returnGas, suberr` on lines core/vm/contracts.go#L728 and core/vm/contracts.go#L696.

Overprotocol: Fixed in commit c1210994.

Spearbit: The Spearbit Review team confirmed that this issue is fixed by changing the return value at core/vm/contracts.go#L697 and core/vm/contracts.go#L736 and was introduced in commit c1210994.

5.3.3 `Database#Recoverable` does not use `ckptRoot` parameter

Severity: Low Risk

Context: trie/triedb/pathdb/database.go#L443

Description: The `ckptRoot` parameter is not used in this function. I'm a little concerned that we might be missing a check here.

Recommendation: Either use this parameter or remove it.

Overprotocol: Fixed in commit c1210994.

Spearbit: The Spearbit Audit Team has verified that this issue has been fixed in c1210994. The `ckptRoot` parameter is used now.

5.3.4 Check for empty restoration proof only does nil check

Severity: Low Risk

Context: core/state_transition.go#L436

Description: In `StateTransition#TransitionDb`, we ensure there's a restoration proof. The error, `ErrEmptyRestorationProof`, indicates that `msg.Data` should not be empty, but the check only ensures the field is not nil. This will allow an empty byte slice, `[]byte{}`, though. I think we should disallow this too.

Recommendation: Doing a length check instead will catch both situations:

```
- if msg.Data == nil {
+ if len(msg.Data) == 0 {
```

Overprotocol: Fixed in commit c1210994.

Spearbit: The Spearbit Audit Team has verified that this issue has been fixed in c1210994. The recommended change was made.

5.4 Informational

5.4.1 Restoration clients with unknown `SourceEpoch` cannot restore

Severity: Informational

Context: cmd/restoration/handler.go#L108-L110, core/vm/evm.go#L908-L910

Description: Clients that do now know their correct `SourceEpoch` will be unable to generate a `ResotrationProof` and will have to revert to 3rd prty data sources to learn what their correct `SourceEpoch` should be. This is due to the fact that there is no way for them to request their current `EpochCoverage` from a restoration server and the error returned when they use the incorrect `SourceEpoch` does not tell them what the correct epoch is. This goes against the assumption that Over Protocol clients should not need to keep the entire state in order to participate in the network.

Recommendation: Change this error returned by the restoration server to specify the correct Epoch.

-->

5.4.2 Performance optimization in `RestorationTX` verification

Severity: Informational

Context: (trie/node.go#L134-L141)

Description: In `trie.VerifyProof`, there is a call to `decodeNode`. The comments indicate that it isn't fully performant:

```
// decodeNode parses the RLP encoding of a trie node. It will deep-copy the passed
// byte slice for decoding, so it's safe to modify the byte slice afterwards. The-
// decode performance of this function is not optimal, but it is suitable for most
// scenarios with low performance requirements and hard to determine whether the
// byte slice be modified or not.
```

Since this `VerifyProof` call is in the evm, I think it qualifies for needing to be performant code. The current usage of `VerifyProof` does not ever modify the data in the returned slice. It may be worth creating a different `VerifyProofUnsafe` function that does not use the deepcopy and would instead call `decodeNodeUnsafe`. This would mean that the input slice and the output slice will end up sharing memory, so modifying one will modify the other. Therefore there would be an expectation that the input slice and the output slice are not modified and are treated as read-only.

According the the benchmarks performed in this comment, this will increase the performance of calls to the `VerifyProof` by ~6.6% on average.

Recommendation: Create a new function `VerifyProofUnsafe` that calls `decodeNodeUnsafe` instead of `decodeNode`. The proofDb input will be sharing memory with the returned byte slice, so both pieces of data should be treated as read-only.

```
// VerifyProofUnsafe checks merkle proofs. The given proof must contain the value for
// key in a trie with the given root hash. VerifyProofUnsafe returns an error if the
// proof contains invalid trie nodes or the wrong value. Values read from the proofDb
// will not be deep-copied, so the input will share memory with the output slice. The
// data in proofDb MUST not be changed after.
func VerifyProofUnsafe(rootHash common.Hash, key []byte, proofDb ethdb.KeyValueReader) (value []byte,
↪  err error) {
    key = keybytesToHex(key)
    wantHash := rootHash
    for i := 0; ; i++ {
        buf, _ := proofDb.Get(wantHash[:])
        if buf == nil {
            return nil, fmt.Errorf("proof node %d (hash %064x) missing", i, wantHash)
        }
        n, err := decodeNodeUnsafe(wantHash[:], buf)
        if err != nil {
            return nil, fmt.Errorf("bad proof node %d: %v", i, err)
        }
        keyrest, cld := get(n, key, true)
        switch cld := cld.(type) {
        case nil:
            // The trie doesn't contain the key.
            return nil, nil
        case hashNode:
            key = keyrest
            copy(wantHash[:], cld)
        case valueNode:
            return cld, nil
        }
    }
}
```

5.4.3 Outdated docstring for `BlockChainAPI#GetEpochByNumber`

Severity: Informational

Context: internal/ethapi/api.go#L831-L834

Description: This function's docstring is for `GetHeaderByNumber` and `GetHeaderByNumber` is missing a docstring.

Recommendation: Fix the docstrings for `GetEpochByNumber` and `GetHeaderByNumber`.

5.4.4 Inconsistent field ordering for Trie ID structure

Severity: Informational

Context: trie/trie_id.go#L21-L27

Description: The order of these fields are inconsistent. For example:

```
// StorageTrieID constructs an identifier for storage trie which ...
// state and contract specified by the stateRoot and owner.
func StorageTrieID(stateRoot common.Hash, epoch uint32, owner common.Owner, root common.Root) *ID {
    return &ID{
        StateRoot: stateRoot,
        Epoch:     epoch,
        Owner:     owner,
        Root:      root,
    }
}
```

and

```
// TrieID constructs an identifier for a standard trie (not a secondary ...
// with provided root. It's msotly used in tests and some other ...
func TrieID(root common.Hash) *ID {
    return &ID{
        StateRoot: root,
        Owner:     common.Hash{},
        Root:      root,
        Epoch:     0,
    }
}
```

Recommendation: In the structure definition & declarations, put `Epoch` between `StateRoot` and `Owner`.

5.4.5 Unnecessary read/write locks in `BlobPool` accessors

Severity: Informational

Context: core/txpool/blobpool/blobpool.go#L1526-L1527, core/txpool/blobpool/blobpool.go#L1539-L1540, core/txpool/blobpool/blobpool.go#L1552-L1553

Description: In `EpochCoverage`, `Nonce`, and `Stats` we use a read/write lock when we could use a read lock.

Recommendation: Replace `p.lock.Lock()` with `p.lock.RLock()` and `p.lock.Unlock()` with `p.lock.RUnlock()`. This is kind of an issue upstream, but it might be worth fixing anyway.

5.4.6 Misleading comment about base fee being burned

Severity: Informational

Context: core/state_transition.go#L481-L485

Description: It appears that the base fee is sent to the OverProtocol treasury instead of being burned. The comment says that the base fee is burned, but this isn't technically true. In my opinion, this should be clearly advertised somewhere too.

Recommendation: Remove the word "burn" from the comment.

5.4.7 Unhandled errors when using `abi.NewType`

Severity: Informational

Context: core/vm/contracts.go#L698-L699, core/vm/contracts.go#L730-L731

Description: In `createContractWithUiHash#Run` and `create2ContractWithUiHash#Run`, we do not handle the potential errors when using `abi.NewType`. I wouldn't actually expect these to fail, but I suggest always handling errors.

Recommendation: Handle these errors.

5.4.8 New `ErrInsufficientFee` error is unused

Severity: Informational

Context: core/vm/errors.go#L44

Description: There's a new error, `ErrInsufficientFee`, which was used before but isn't anymore.

Recommendation: One of the two:

- Return `ErrInsufficientFee` instead of `ErrInsufficentBalance` in `Restore()`.
- Delete `ErrInsufficientFee` (I think this would be my preference).

5.4.9 Missing nil check for header

Severity: Informational

Context: core/vm/evm.go#L879

Description: In `verifyRestorationProof`, when getting the block root hash, we do not check if the header is nil. If for some reason this is nil, it will panic when accessing the `Root` field. `GetHeaderByNumber` could theoretically return nil, but I'm pretty sure prior checks in `verifyRestorationProof` will prevent this. Better to be safe though.

Recommendation: Consider applying the following change:

```
- rootHash := evm.Context.GetHeaderByNumber(lastCkptBn).Root
- leafNode, err := trie.VerifyProof(rootHash, targetKey, proofDB)
+ header := evm.Context.GetHeaderByNumber(lastCkptBn)
+ if header == nil {
+     return 0, 0, nil, ErrHeaderIsNil
+ }
+ leafNode, err := trie.VerifyProof(header.Root, targetKey, proofDB)
```