CANTINA

Ethereum Pectra: Go Ethereum

Competition

July 21, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	**Description**
High	*Must* fix as soon as possible (if already deployed) and can be triggered by any user without significant constraints, generating outsized returns to the exploiter. For example: loss of user funds (significant amount of funds being stolen or lost) or breaking core functionality (failure in fundamental protocol operations).
Medium	Global losses <10% or losses to only a subset of users, requiring significant constraints (capital, planning, other users...) to be exploited. For example: temporary disruption or denial of service (DoS), minor fund loss or exposure or breaking non-core functionality
Low	Losses will be annoying but easily recoverable, requiring unusual scenarios or admin actions to be exploited.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above matrix. High severity findings represent the most critical issues that must be addressed immediately, as they either have high impact and high likelihood of occurrence, or medium impact with high likelihood.

Medium severity findings represent issues that, while not immediately critical, still pose significant risks and should be addressed promptly. These typically involve scenarios with medium impact and medium likelihood, or high impact with low likelihood.

Low severity findings represent issues that, while not posing immediate threats, could potentially cause problems in specific scenarios. These typically involve medium impact with low likelihood, or low impact with medium likelihood.

Lastly, some findings might represent improvements that don't directly impact security but could enhance the codebase's quality, readability, or efficiency (Gas and Informational findings).

2 Security Review Summary

Ethereum is a worldwide system, an open-source platform to write computer code that stores and automates digital databases using smart contracts, without relying upon a central intermediary, solving trust with cryptographic techniques.

From Feb 21st to Mar 27th Cantina hosted a competition based on the Ethereum Pectra upgrade. The present report focuses in the go-ethereum implementation. The participants identified a total of **4** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 0
- Low Risk: 1
- Gas Optimizations: 0
- Informational: 3

3 Findings

3.1 Low Risk

3.1.1 EIP-7702: Deadlock in the mempool when a transaction has an out-of-order nonce

Submitted by Haxatron

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: When a EOA is a delegated account, there can only be at most one in-flight transaction (legacypool.go#L608-L645):

```go
// validateAuth verifies that the transaction complies with code authorization
// restrictions brought by SetCode transaction type.
func (pool *LegacyPool) validateAuth(tx *types.Transaction) error {
    from, _ := types.Sender(pool.signer, tx) // validated

    // Allow at most one in-flight tx for delegated accounts or those with a
    // pending authorization.
    if pool.currentState.GetCodeHash(from) != types.EmptyCodeHash || len(pool.all.auths[from]) != 0 {
        var (
            count  int
            exists bool
        )
        pending := pool.pending[from]
        if pending != nil {
            count += pending.Len()
            exists = pending.Contains(tx.Nonce())
        }
        queue := pool.queue[from]
        if queue != nil {
            count += queue.Len()
            exists = exists || queue.Contains(tx.Nonce())
        }
        // Replace the existing in-flight transaction for delegated accounts
        // are still supported
        if count >= 1 && !exists {
            return ErrInflightTxLimitReached
        }
    }
    ...
}
```

Now, when a transaction is of the correct nonce, it will be added to `pending`, but if a transaction has a nonce that is out-of-order, it will be added to `queue`. To remove a transaction, it must be replaced with a transaction of the same nonce that has a higher gas price. This means that if an EOA account with delegated code has an out-of-order nonce, it will be in added to `queue`. To replace this transaction, the transaction with the same out-of-order nonce must be added.

Effectively, this means that an EOA with delegated code that submits a transaction with out-of-order nonce will be stuck in `queue` forever which will result in a deadlock scenario. This occurs because sending a transaction with the correct nonce will result in `count == 1` and `exists == false` causing `ErrInflightTxLimitReached` error to trigger.

Recommendation: Allow at most one pending in-flight transaction (only transactions with correct nonces) and do not allow any queued in-flight transaction to prevent deadlock. This match the behaviour of Nethermind - DelegatedAccountFilter.cs#L45.

3.2 Informational

3.2.1 EIP-7002 can be DoS via `prohibitive fee attack` and potentially cause `fund loss` to validator by front-run

Submitted by dontonka

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Summary: This report will expose how `EIP-7002` specification and `geth` implementation (well probably all clients) is vulnerable to **DoS risk** and how it can cause **fund loss** to validator.

Finding Description: EIP-7002 and his analysis seems inacurate in terms of DoS analysis (attack via prohibitive fee section), and this report will expose this in details. With the current implementation, the whole EIP can be "almost" permanently DoS by an attacker at a cost of 0.02284 ETH / hour (see the proof of concept but most importantly, this comment in the competition page) which seems very cheap for such grief. Additionally, there is a front-run scenario which could cause harm to validator (by paying much more fees, see Impact below) when they use this new functionality.

The idea is already partially described in the analysis, so an attacker would fill up withdrawal requests such that it doesn't cost him too much todo so, but while the next one in line will have to pay an exorbitant fee, most likely deceiving anyone to use the feature as too expensive. Then once this cool off a bit, the attacker can do this attack again to maintain such high cost.

Impact Explanation: High:

- Temporary DoS, but can be maintained for not that much investement (see PoC), so it's borderline a permanent DoS, but validator can still exit throught the legacy mechanism I assume.
- If this is executed from a smart contract the fee paid could be **much higher** then what the validator might have anticipated. Let's consider the following function (as suggested in the specification) being called by an EOA and the smart contract have a balance of 5 ETH. An attacker could front-run this transaction, filling up the request (see PoC) and here when this transaction would be executed, the fee could be super high (let say 4 ETH), and the transaction would be successfull (since 4 < 5 ETH), which would translate in a **huge loss** for the validator trying to withdraw. No profit for the attacker, but griefed the validator. This is a clear concern as even suggested in the specification. Of course, if the smart contract doesn't hold funds and funds are sent by the EOA when sending the request, that would revert and works fine, but such defensive behavior is not garanteed to be used by all validators.

```
function addWithdrawal(bytes memory pubkey, uint64 amount) private {
    assert(pubkey.length == 48);

    // Read current fee from the contract.
    (bool readOK, bytes memory feeData) = WithdrawalsContract.staticcall('');
    if (!readOK) {
        revert('reading fee failed');
    }
    uint256 fee = uint256(bytes32(feeData));

    // Add the request.
    bytes memory callData = abi.encodePacked(pubkey, amount);
    (bool writeOK,) = WithdrawalsContract.call{value: fee}(callData);
    if (!writeOK) {
        revert('adding request failed');
    }
}
```

Likelihood Explanation: Medium: Anyone can execute this attack without much funds required. It could be also triggered when there is an increase in demand to exit due to an external event (panic in the market in ETH) or simply by front-running them and causing validator to pay excessive fees or having their transaction revert due to insuficient funds.

Proof of Concept: Define a genesis (in a file called `prestate.json`) which will be used as a base, mainly require to have some funds to send which is required to pay the fees for adding a withdrawal request. Here I'm funding both `0xfffffffffffffffffffffffffffffffffffffffe` and `0xfffffffffffffffffffffffffffffffffffffffa`:

```
{
  "alloc": {
    "0xfffffffffffffffffffffffffffffffffffffffe": {
      "balance": "0x56BC75E2D63100000",
      "code": "",
      "nonce": "0x0"
    },
    "0xfffffffffffffffffffffffffffffffffffffffa": {
      "balance": "0x56BC75E2D63100000",
      "code": "",
      "nonce": "0x0"
    }
  },

  "gasLimit": "0x1000000",
  "difficulty": "0x1",
  "coinbase": "0x0000000000000000000000000000000000000000",
  "number": "0x0",
  "timestamp": "0x0",
  "config": {
    "chainId": 1,
    "homesteadBlock": 0,
    "eip150Block": 0,
    "eip155Block": 0,
    "eip158Block": 0,
    "byzantiumBlock": 0,
    "constantinopleBlock": 0,
    "petersburgBlock": 0,
    "istanbulBlock": 0,
    "berlinBlock": 0,
    "londonBlock": 0,
    "shanghaiTime": 0,
    "terminalTotalDifficulty": 0
  }

}
```

Run the following command, which execute the `add_withdrawal_request` code path, basically adding a single withdrawal request and paying 1 wei for the fee. This is using the input from blockchain_test.go#L4139. The command as follow is not complete to not make this report too wide, but the code to be pasted is protocol_params.go#L209:

```
./build/bin/evm run --prestate prestate.json --sender 0xfffffffffffffffffffffffffffffffffffffffa --value 1
↪   --input 0xb917cfdc0d25b72d55cf94db328e1629b7f4fde2c30cdacf873b664416f76a0c7f7cc50c9f72a3cb84be88144cde9125 ⌋
↪   0000000000000d80 --statdump --dump 0x3373fffffffffffffffffffffffffffffffffffffffe1460c...
```

We can see this consume `113945 gas` without the intrisic gas overhead. So for a ETH gas limit at 30M gas, that translate into a maximum of 263 requests per block, much lower then what the analysis mention at 2000.

./img/2000_req_max.png

```
INFO [02-20|12:09:54.590] Persisted trie from memory database      nodes=3 size=411.00B time="606.883s"
↪  gcnodes=0 gcsize=0.00B gctime=0s livenodes=0 livesize=0.00B
INFO [02-20|12:09:54.594] Trie dumping started                     root=823347..7e728a
INFO [02-20|12:09:54.594] Trie dumping complete                    accounts=3 elapsed="58.807µs"
{
    "root": "823347a63b790d140c64bcd362cb07b4e33a0436837d401ae2c78cdafa7e728a",
    "accounts": {
        "0x0000000000000000000000007265636569766572": {
            "balance": "1",
            "nonce": 0,
            "root": "0x4f381414026ffa507a216cc2f372951a863ff9ac2e5b637693e29262f9f2140b",
            "codeHash": "0x0345a365d2f4c5975b9f1599abe0a2ee76b7a3a731bc68781bd04c84e4858f50",
            "code": "0x3373fffffffffffffffffffffffffffffffffffffffe1460c...",
            "storage": {
                "0x0000000000000000000000000000000000000000000000000000000000000001": "01",
                "0x0000000000000000000000000000000000000000000000000000000000000003": "01",
                "0x0000000000000000000000000000000000000000000000000000000000000004":
                ↪  "fffffffffffffffffffffffffffffffffffffffa",
                "0x0000000000000000000000000000000000000000000000000000000000000005":
                ↪  "b917cfdc0d25b72d55cf94db328e1629b7f4fde2c30cdacf873b664416f76a0c",
                "0x0000000000000000000000000000000000000000000000000000000000000006":
                ↪  "7f7cc50c9f72a3cb84be88144cde912500000000000000d80000000000000000"
            },
            "address": "0x0000000000000000000000007265636569766572",
            "key": "0x30d7a0694cb29af31b982480e11d7ebb003a3fca4026939149071f014689b142"
        },
        "0xfFFffFFFfFfFFfFffFFfFFfFffffFFFFFFFffA": {
            "balance": "99999999999999999999",
            "nonce": 0,
            "root": "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
            "codeHash": "0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470",
            "address": "0xfffffffffffffffffffffffffffffffffffffffa",
            "key": "0x31c6791d973269820648fdf6017851482256c1e5e64c7a8e3e782de9915e2328"
        },
        "0xffffFFFfFFffffffffffffffFfFFFffffFFFfFFfE": {
            "balance": "100000000000000000000",
            "nonce": 0,
            "root": "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
            "codeHash": "0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470",
            "address": "0xfffffffffffffffffffffffffffffffffffffffe",
            "key": "0xfcbd49b3a106f7e49c6e147b76ca4682aefd4fe6d07f4368f542751aaf85d596"
        }
    }
}
EVM gas used:     113945
execution time:   464.822µs
allocations:      179
allocated bytes:  22320
```

Considering `EIP-1559`, the attacker could fill requests to not reach 50% of block gas limit (so use less than 15M gas, also considering others transaction in the block, so probably aiming for 10-12M), to ensure the gas fee doesn't increase on the next attack, which translate into approx. 100 requests per block.

At ETH valued at 2750 USD and gas price at 0.5 gwei (which is not abnormal to see), as follow are the costs for this attack, which is composed of two contributor.

- The gas fee for the transaction itself.
- The fee introduced here specially to protect against griefing attack.

An advanced attacker could combine sending only 100 requests (to ensure no gas increase) on the `first 5 chunks` but on the final blow the full amount, so 263 requests, since after that, he doesn't care about the gas price increase, it's actually something that he wants, to make honest validator pay even more.

Gas fee:

- Assuming a constant gas price for simplification sake on the first chunks, 100 requests will spend 11.5M gas ((113945 * 100) + 21k), translating into 15 USD per attack.
- Plus the final blow of 263 requests `(113945 * 263 + 21k)` will translate into 41 USD.
- So a total of `((15 USD * 5) + 41 USD) = 116 USD`.

Griefing fee:

- The first 5 attacks as we can see below will pretty much cost nothing (about 2721 gwei in total, so almost 1 cent).
- The final one will not be much more expensive at 0.000866 ETH (2.4 USD). This result in 753 request in excess, which increase the fee for the next request to 17 ETH! (47k USD).

```
Block1:    0-100 requests : fee --> 1 (x 100 --> 100 wei - 0 USD)
Block2:   98-198 requests : fee --> 316 (x 100 --> 31600 wei - 0 USD)
Block3:  196-296 requests : fee --> 101550 (x 100 --> 0.01 gwei - 0 USD)
Block4:  294-394 requests : fee --> 32409127 (x 100 --> 3.24 gwei - 0 USD)
Block5:  392-492 requests : fee --> 10333911150 (x 100 --> 2717.82 gwei - 0.01 USD)
Block6:  490-753 requests : fee --> 3294931309780 (x 263 --> 0.000866 ETH - 2.4 USD)
Block7:  753-X   requests : fee --> 17245321207026308417 (17 ETH per request! - 47k USD)
```

At 300 blocks per hours (that is a block every 12 sec.), and a TARGET of 2 (that's hardcoded in the contract), it will require 132 blocks to go from 753 to 490 excess request, which mean about 26 min, let's round it up to 30 min. At that point the attacker can again send a final blow to maintain the attack.

- So to ramp up the attack, it cost 116 USD in gas fee + 2.5 USD in griefing fee: 118.5 USD (or 0.043 ETH).
- Afterward, to maintain the attack it cost 41 + 2.4: 43 USD (or 0.0158 ETH).

So overall, there is a one time cost of approximatly 0.043 ETH and then fix attack cost of 0.0316 ETH / hour to DoS and maintain this attack, which doesn't seems that expensive depending on the attacker budget and desire. This will make the fee to submit a withdrawal request range from 0.000866 ETH (low bound at 490 request in excess) to 17 ETH (high bound at 753), which will be much higher then what the specification aim to have which is `1 gwei per request`.

```
Assuming 1 Gwei to be a reasonable fee per request ...
```

Recommendation: I think the current specification is currently vulnerable to DoS attack but also validator could be tricked in paying much more fees.

- DoS: One option could be to increase the fee upon request submission instead of request processing, that would eliminate the main edge the attacker is exploiting here for this attack. I would probably also increase the base fee and maybe also change the formula such that the fee increase faster. Here we can see from 0 to 500 excess request the fee is very low, almost nothing.

  We can see that the main mitigation here is the normal gas fee for the transasction cost, which is an "ok" mitigation, but with gas price being low on mainnet, and going to remain that way probably in the future (or even be lower), that suggest the mitigation will loose strenght overtime, and I consider it already kind of weak.

- Front-run: You could introduce a parameter (similar to slippage) which is the maximum fee a validator is willing to pay otherwise transaction would revert, that would protects against this attack vector.

Appendix: This is a little script which use the specification code to calculate the fee according to 3 parameters. This is how I calculated the griefing fee listed above according to specific excess request count:

- Factor: `MIN_WITHDRAWAL_REQUEST_FEE` $\rightarrow$ 1 (hardcoded).
- Numerator: the excess request count.
- Denominator: `WITHDRAWAL_REQUEST_FEE_UPDATE_FRACTION` $\rightarrow$ 17 (hardcoded).

```python
def fake_exponential(factor: int, numerator: int, denominator: int, debug: bool = False) -> int:
    """
    Calculate a fake exponential series approximation.

    Args:
        factor (int): Initial factor to multiply with denominator
        numerator (int): Value for the numerator in series calculation
        denominator (int): Value for the denominator in series calculation
        debug (bool): If True, prints debug information for each iteration

    Returns:
        int: The calculated result after dividing by denominator
    """
    i = 1
    output = 0
    numerator_accum = factor * denominator
    iterations = 0

    while numerator_accum > 0:
        output += numerator_accum
        iterations += 1

        if debug:
            print(f"Iteration {iterations}:")
            print(f"  numerator_accum = {numerator_accum}")
            print(f"  output = {output}")

        numerator_accum = (numerator_accum * numerator) // (denominator * i)
        i += 1

    if debug:
        print(f"\nTook {iterations} iterations")
        print(f"Final sum: {output}")
        print(f"Final result (sum // {denominator}): {output // denominator}")

    return output // denominator

# Example usage
if __name__ == "__main__":
    result = fake_exponential(factor=1, numerator=0, denominator=17, debug=False)
    print(f"\nResult for x=0: {result}")

    result = fake_exponential(factor=1, numerator=98, denominator=17, debug=False)
    print(f"\nResult for x=98: {result}")

    result = fake_exponential(factor=1, numerator=196, denominator=17, debug=False)
    print(f"\nResult for x=196: {result}")

    result = fake_exponential(factor=1, numerator=294, denominator=17, debug=False)
    print(f"\nResult for x=294: {result}")

    result = fake_exponential(factor=1, numerator=392, denominator=17, debug=False)
    print(f"\nResult for x=392: {result}")

    result = fake_exponential(factor=1, numerator=440, denominator=17, debug=False)
    print(f"\nResult for x=440: {result}")

    result = fake_exponential(factor=1, numerator=490, denominator=17, debug=False)
    print(f"\nResult for x=490: {result}")

    result = fake_exponential(factor=1, numerator=540, denominator=17, debug=False)
    print(f"\nResult for x=540: {result}")

    result = fake_exponential(factor=1, numerator=588, denominator=17, debug=False)
    print(f"\nResult for x=588: {result}")

    result = fake_exponential(factor=1, numerator=620, denominator=17, debug=False)
    print(f"\nResult for x=620: {result}")

    result = fake_exponential(factor=1, numerator=650, denominator=17, debug=False)
    print(f"\nResult for x=650: {result}")

    result = fake_exponential(factor=1, numerator=686, denominator=17, debug=False)
    print(f"\nResult for x=686: {result}")

    result = fake_exponential(factor=1, numerator=700, denominator=17, debug=False)
    print(f"\nResult for x=700: {result}")

    result = fake_exponential(factor=1, numerator=720, denominator=17, debug=False)
    print(f"\nResult for x=720: {result}")

    result = fake_exponential(factor=1, numerator=753, denominator=17, debug=False)
    print(f"\nResult for x=753: {result}")
```

3.2.2 Incorrect Gas Refund Calculation in `state_transition::execute` Under Prague Rules

Submitted by 0xmahdirostami

Severity: Informational

Context: state_transition.go#L515

Summary: The `state_transition::execute` function returns an `ExecutionResult` containing `RefundedGas`. However, when `IsPrague` is enabled and `floorDataGas` exceeds `st.gasUsed()`, the function incorrectly retains the `gasRefund` value, leading to an incorrect gas refund.

Finding Description: Within `state_transition::execute`, `gasRefund` is initially calculated using `st.calcRefund()`, and then added to `st.gasRemaining`. If `IsPrague` is enabled and `st.gasUsed() < floorDataGas`, the function adjusts `st.gasRemaining` by setting it to `st.initialGas - floorDataGas`, effectively disregarding the previously calculated `gasRefund`. However, `gasRefund` is not reset before being returned, which leads to an incorrect refund value.

Impact Explanation: Since `execute` returns the incorrect `RefundedGas` value, components relying on it, such as `gasestimator.go`, may overestimate gas limits. Specifically, `optimisticGasLimit` is computed as:

```
optimisticGasLimit := (result.UsedGas + result.RefundedGas + params.CallStipend) * 64 / 63
```

In cases where `IsPrague` is enabled and `floorDataGas` exceeds `st.gasUsed()`, `RefundedGas` should be zero, but it remains nonzero, leading to an overestimated `optimisticGasLimit`.

Likelihood Explanation: This issue occurs whenever `IsPrague` is enabled and `floorDataGas` is greater than `st.gasUsed()`.

Recommendation: Reset `gasRefund` when `IsPrague` is enabled to ensure correct gas calculations:

```
// Compute refund counter, capped to a refund quotient.
gasRefund := st.calcRefund() //@audit
st.gasRemaining += gasRefund
if rules.IsPrague {
    // After EIP-7623: Data-heavy transactions pay the floor gas.
    if st.gasUsed() < floorDataGas {
+        if st.gasUsed() + gasRefund < floorDataGas {
+       gasRefund = 0
+         }
        prev := st.gasRemaining
        st.gasRemaining = st.initialGas - floorDataGas
        if t := st.evm.Config.Tracer; t != nil && t.OnGasChange != nil {
            t.OnGasChange(prev, st.gasRemaining, tracing.GasChangeTxDataFloor)
        }
    }
}
st.returnGas()
```

3.2.3 EIP-7702: Delegated accounts can still send multiple blob transactions

Submitted by Haxatron

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Delegated accounts have a restriction of only having only one pending transaction in the mempool. However the geth mempool only checks this restriction for the `legacypool`, which only consists of Type 1, Type 2 or Type 4 transactions. We can see the check is only present in the legacypool (legacypool.go#L613-L635):

```
// Allow at most one in-flight tx for delegated accounts or those with a
// pending authorization.
if pool.currentState.GetCodeHash(from) != types.EmptyCodeHash || len(pool.all.auths[from]) != 0 {
    var (
        count  int
        exists bool
    )
    pending := pool.pending[from]
    if pending != nil {
        count += pending.Len()
        exists = pending.Contains(tx.Nonce())
    }
    queue := pool.queue[from]
    if queue != nil {
        count += queue.Len()
        exists = exists || queue.Contains(tx.Nonce())
    }
    // Replace the existing in-flight transaction for delegated accounts
    // are still supported
    if count >= 1 && !exists {
        return ErrInflightTxLimitReached
    }
}
```

The restriction is not enforced for the Type-3 transactions added to the blob pool, which allows a delegated account to bypass restrictions by sending multiple blob transactions.

Recommendation: The check should also be enforced on the blob pool.