CANTINA

Ethereum Pectra: Erigon

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 erigon implementation. The participants identified a total of **11** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 0
- Low Risk: 5
- Gas Optimizations: 0
- Informational: 6

3 Findings

3.1 Low Risk

3.1.1 Erigon incorrectly handles delegation removal

Submitted by zigtur

Severity: Low Risk

Context: state_transition.go#L437-L441

Description: The erigon node does not expose the correct code when an EOA sets a delegation and then remove this delegation. When removing a delegation, the code of the account is set to `nil`:

```
// 7. set authority code
if auth.Address == (libcommon.Address{}) {
  if err := st.state.SetCode(authority, nil); err != nil {
    return nil, fmt.Errorf("%w: %w", ErrStateTransitionFailed, err)
  }
} else {
```

However, due to this `nil` argument, the code is not set correctly to the account. The `GetCode` function used for the `eth_getCode` RPC interface retrieves the code of an account. However, due to the issue, the returned the code of an account does not match the code hash of this account. By adding logs capabilities in this `GetCode` function, the following logs were obtained:

```
2025-03-03 10:18:54 [WARN] [03-03|09:18:54.301] ZIGTUR: GetCode
2025-03-03 10:18:54 [WARN] [03-03|09:18:54.301] ZIGTUR: GetCode - ReadAccountData
↪  acc="&{Initialised:true Nonce:4 Balance:[11515771281038350272 54210108 0 0]
↪  Root:0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421
↪  CodeHash:0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470 Incarnation:0
↪  PrevIncarnation:0}" err=nil
2025-03-03 10:18:54 [WARN] [03-03|09:18:54.301] ZIGTUR: GetCode - ReadAccountCode        res="[239 1 0 180 180
↪  107 218 168 53 248 228 180 216 226 8 182 85 156 210 103 133 16 81]"
```

As we can see, the `CodeHash` of the account corresponds to the `EMPTY_CODEHASH`. So, we should expected the account code to be empty (i.e. `[]`). However, the account code retrieved here is the previous delegation, it is not empty code.

Proof of Concept:

- Initial setup: Start a localnet with Erigon and Go-ethereum nodes and configure it so Prague is active. Download Erigon repository, then run in this repo:

  ```
  git clone https://github.com/erigontech/erigon
  cd erigon
  docker build . -t erigon/erigon:pectra-zigtur
  ```

  Download Go-ethereum repository, then run in this repo:

  ```
  git clone https://github.com/ethereum/go-ethereum
  cd go-ethereum
  docker build . -t local/geth:pectra-zigtur
  ```

  Clone the ethereum-package repository to start a kurtosis enclave. Replace the existing `network_params.yaml` file by the one in Appendix. Follow the quickstart if not installed. Finally, run the following in this repository:

  ```
  kurtosis run --enclave my-testnet . --args-file network_params.yaml
  ```

  Once all docker containers are started, the RPC ports for each node should be available. Take the RPC ones of execution clients and set them in the Python script.

  Note: the Erigon rpc port is printed as `ws-rpc` while others node rpc ports are named `rpc` in the kurtosis output.

- `network_params.yaml`:

  ```
  participants:
  # EL
  ```

```yaml
- el_type: erigon
    el_image: erigon/erigon:pectra-zigtur
    el_log_level: ""
    el_extra_env_vars: {}
    el_extra_labels: {}
    el_extra_params: []
    el_tolerations: []
    el_volume_size: 0
    el_min_cpu: 0
    el_max_cpu: 0
    el_min_mem: 0
    el_max_mem: 0
    # CL
    cl_type: lighthouse
    cl_image: sigp/lighthouse:latest-unstable
    cl_log_level: ""
    cl_extra_env_vars: {}
    cl_extra_labels: {}
    cl_extra_params: []
    cl_tolerations: []
    cl_volume_size: 0
    cl_min_cpu: 0
    cl_max_cpu: 0
    cl_min_mem: 0
    cl_max_mem: 0
    supernode: false
    use_separate_vc: true
    # Validator
    vc_type: lighthouse
    vc_image: sigp/lighthouse:latest-unstable
    vc_log_level: ""
    vc_extra_env_vars: {}
    vc_extra_labels: {}
    vc_extra_params: []
    vc_tolerations: []
    vc_min_cpu: 0
    vc_max_cpu: 0
    vc_min_mem: 0
    vc_max_mem: 0
    validator_count: null
    use_remote_signer: false
    # Remote signer
    remote_signer_type: web3signer
    remote_signer_image: consensys/web3signer:latest
    remote_signer_extra_env_vars: {}
    remote_signer_extra_labels: {}
    remote_signer_extra_params: []
    remote_signer_tolerations: []
    remote_signer_min_cpu: 0
    remote_signer_max_cpu: 0
    remote_signer_min_mem: 0
    remote_signer_max_mem: 0
    # participant specific
    node_selectors: {}
    tolerations: []
    count: 1
    snooper_enabled: false
    ethereum_metrics_exporter_enabled: false
    xatu_sentry_enabled: false
    prometheus_config:
    scrape_interval: 15s
    labels: {}
    blobber_enabled: false
    blobber_extra_params: []
    builder_network_params: null
    keymanager_enabled: false

# SECOND config
- el_type: geth
    el_image: local/geth:pectra-zigtur
    el_log_level: "DEBUG"
    el_extra_env_vars: {}
    el_extra_labels: {}
    el_extra_params: []
    el_tolerations: []
    el_volume_size: 0
    el_min_cpu: 0
    el_max_cpu: 0
    el_min_mem: 0
    el_max_mem: 0
    # CL
    cl_type: lighthouse
    cl_image: sigp/lighthouse:latest-unstable
    cl_log_level: ""
    cl_extra_env_vars: {}
    cl_extra_labels: {}
    cl_extra_params: []
    cl_tolerations: []
    cl_volume_size: 0
    cl_min_cpu: 0
    cl_max_cpu: 0
    cl_min_mem: 0
    cl_max_mem: 0
    supernode: false
    use_separate_vc: true
    # Validator
    vc_type: lighthouse
    vc_image: sigp/lighthouse:latest-unstable
    vc_log_level: ""
    vc_extra_env_vars: {}
    vc_extra_labels: {}
    vc_extra_params: []
    vc_tolerations: []
    vc_min_cpu: 0
    vc_max_cpu: 0
    vc_min_mem: 0
    vc_max_mem: 0
    validator_count: null
    use_remote_signer: false
    # Remote signer
    remote_signer_type: web3signer
    remote_signer_image: consensys/web3signer:latest
    remote_signer_extra_env_vars: {}
    remote_signer_extra_labels: {}
    remote_signer_extra_params: []
    remote_signer_tolerations: []
    remote_signer_min_cpu: 0
    remote_signer_max_cpu: 0
    remote_signer_min_mem: 0
    remote_signer_max_mem: 0
    # participant specific
    node_selectors: {}
    tolerations: []
    count: 1
    snooper_enabled: false
    ethereum_metrics_exporter_enabled: false
    xatu_sentry_enabled: false
    prometheus_config:
    scrape_interval: 15s
    labels: {}
    blobber_enabled: false
    blobber_extra_params: []
    builder_network_params: null
    keymanager_enabled: false
network_params:
network: kurtosis
network_id: "3151908"
deposit_contract_address: "0x4242424242424242424242424242424242424242"
seconds_per_slot: 12
num_validator_keys_per_node: 64
preregistered_validator_keys_mnemonic:
    "giant issue aisle success illegal bike spike
    question tent bar rely arctic volcano long crawl hungry vocal artwork sniff fantasy
    very lucky have athlete"
preregistered_validator_count: 0
genesis_delay: 20
genesis_gaslimit: 30000000
max_per_epoch_activation_churn_limit: 8
churn_limit_quotient: 65536
ejection_balance: 16000000000
eth1_follow_distance: 2048
min_validator_withdrawability_delay: 256
```

```
shard_committee_period: 256
deneb_fork_epoch: 0
electra_fork_epoch: 1
fulu_fork_epoch: 2000000000
network_sync_base_url: https://snapshots.ethpandaops.io/
data_column_sidecar_subnet_count: 128
samples_per_slot: 8
custody_requirement: 4
max_blobs_per_block_electra: 9
target_blobs_per_block_electra: 6
max_blobs_per_block_fulu: 12
target_blobs_per_block_fulu: 9
additional_preloaded_contracts: {}
devnet_repo: ethpandaops
prefunded_accounts: {}
additional_services: []
dora_params:
image: ""
tx_spammer_params:
tx_spammer_extra_args: []
spamoor_blob_params:
spamoor_extra_args: []
prometheus_params:
storage_tsdb_retention_time: "1d"
storage_tsdb_retention_size: "512MB"
min_cpu: 10
max_cpu: 1000
min_mem: 128
max_mem: 2048
grafana_params:
additional_dashboards: []
min_cpu: 10
max_cpu: 1000
min_mem: 128
max_mem: 2048
assertoor_params:
image: ""
run_stability_check: false
run_block_proposal_check: false
run_transaction_test: false
run_blob_transaction_test: false
run_opcodes_transaction_test: false
run_lifecycle_test: false
tests: []
wait_for_finalization: false
global_log_level: info
snooper_enabled: false
ethereum_metrics_exporter_enabled: false
parallel_keystore_generation: false
disable_peer_scoring: false
persistent: false
mev_type: null
mev_params:
mev_relay_image: ethpandaops/mev-boost-relay:main
mev_builder_image: ethpandaops/flashbots-builder:main
mev_builder_cl_image: sigp/lighthouse:latest
mev_boost_image: ethpandaops/mev-boost:develop
mev_boost_args: ["mev-boost", "--relay-check"]
mev_relay_api_extra_args: []
mev_relay_housekeeper_extra_args: []
mev_relay_website_extra_args: []
mev_builder_extra_args: []
mev_builder_prometheus_config:
    scrape_interval: 15s
    labels: {}
mev_flood_image: flashbots/mev-flood
mev_flood_extra_args: []
mev_flood_seconds_per_bundle: 15
custom_flood_params:
    interval_between_transactions: 1
xatu_sentry_enabled: false
xatu_sentry_params:
xatu_sentry_image: ethpandaops/xatu-sentry
xatu_server_addr: localhost:8000
xatu_server_tls: false
xatu_server_headers: {}
```

```
beacon_subscriptions:
    - attestation
    - block
    - chain_reorg
    - finalized_checkpoint
    - head
    - voluntary_exit
    - contribution_and_proof
    - blob_sidecar
apache_port: 40000
global_tolerations: []
global_node_selectors: {}
keymanager_enabled: false
checkpoint_sync_enabled: false
checkpoint_sync_url: ""
ethereum_genesis_generator_params:
image: ethpandaops/ethereum-genesis-generator:3.7.0
port_publisher:
nat_exit_ip: KURTOSIS_IP_ADDR_PLACEHOLDER
el:
    enabled: false
    public_port_start: 32000
cl:
    enabled: false
    public_port_start: 33000
vc:
    enabled: false
    public_port_start: 34000
remote_signer:
    enabled: false
    public_port_start: 35000
additional_services:
    enabled: false
    public_port_start: 36000
```

- Monitoring: The code of an account can be checked with:

```
cast code 0x8943545177806ED17B9F23F0a21ee5948eCaa776 --rpc-url $RPC_URL
```

To facilitate monitoring, the following python script can be used. Create a python localenv with following commands:

```
python3 -m venv localenv
source localenv/bin/activate
pip3 install web3
```

Then start the following python script by replacing the given RPC urls with yours:

```
from web3 import Web3
import time
import logging

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s - %(message)s",
    handlers=[
        logging.FileHandler("code_monitor.log"),  # Save logs to a file
        logging.StreamHandler()  # Print logs to the console (CLI)
    ]
)

# Ethereum node RPC URLs (Replace these with actual node URLs)
NODE_URLS = [
    "http://127.0.0.1:50986",
    "http://127.0.0.1:51008"
]

# Ethereum account to monitor (Replace with the actual contract address)
ACCOUNT_ADDRESS = "0x8943545177806ED17B9F23F0a21ee5948eCaa776"

# Connect to nodes
web3_nodes = [Web3(Web3.HTTPProvider(url)) for url in NODE_URLS]

# Function to get the smart contract code at an address from a node
```

```
def get_account_code(w3, address):
    try:
        return w3.eth.get_code(address).hex()
    except Exception as e:
        logging.error(f"Error fetching code from node {w3.provider.endpoint_uri}: {e}")
        return None

def monitor_account_code():
    last_code = None
    while True:
        try:
            code_entries = {}

            # Fetch the contract code from all nodes
            for i, w3 in enumerate(web3_nodes):
                code = get_account_code(w3, ACCOUNT_ADDRESS)
                if code is not None:
                    code_entries[f"Node {i+1}"] = code

            # Ensure at least one node responded
            if not code_entries:
                logging.error("No nodes responded with code data.")
                time.sleep(3)
                continue

            # Check if all nodes report the same code
            unique_codes = set(code_entries.values())

            if len(unique_codes) > 1:
                logging.warning(f"Code mismatch detected for account {ACCOUNT_ADDRESS}: {code_entries}")
            else:
                logging.info(f"Account {ACCOUNT_ADDRESS} code verified successfully:
                ↪  {list(unique_codes)[0]}")

            # Check for changes in the code
            current_code = list(unique_codes)[0]
            if last_code is not None and last_code != current_code:
                logging.warning(f"Code change detected at {ACCOUNT_ADDRESS}: Old: {last_code}, New:
                ↪  {current_code}")

            last_code = current_code
        except Exception as e:
            logging.error(f"Unexpected error in monitoring loop: {e}")

        time.sleep(3)  # Check every 3 seconds

if __name__ == "__main__":
    logging.info(f"Ethereum Account Code Monitoring Started for {ACCOUNT_ADDRESS}...")
    monitor_account_code()
```

- Delegation: Then, start the following Go code twice. Please wait to have block number greater than 32 as Pectra activates in epoch 1:

```
go run script.go -step 1
go run script.go -step 2
```

Clone go-ethereum, go in the repo, create a `script` folder, go to this folder and import the following file as `script.go`.

```
package main

import (
    "context"
    "crypto/ecdsa"
    "encoding/hex"
    "flag"
    "fmt"
    "log"

    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/crypto"
    "github.com/ethereum/go-ethereum/ethclient"
    "github.com/holiman/uint256"
)
```

```go
func main() {
	option := flag.Int("step", 1, "Select option 1 or 2 for authDel1 address")
	flag.Parse()
	rpcURL := "http://127.0.0.1:51022" // CHANGE PORT HERE
	privateKeyHex := "bcdf20249abf0ed6d944c0288fad489e33f66b3960d9e6229c1cd214ed3bbe31"
	toAddressHex := "0x72bCbB3f339aF622c28a26488Eed9097a2977404"

	// Connect to Ethereum client
	client, err := ethclient.Dial(rpcURL)
	if err != nil {
		log.Fatalf("Failed to connect to the Ethereum client: %v", err)
	}
	defer client.Close()

	// Load the private key
	privateKeyBytes, err := hex.DecodeString(privateKeyHex)
	if err != nil {
		log.Fatalf("Invalid private key: %v", err)
	}

	privateKey, err := crypto.ToECDSA(privateKeyBytes)
	if err != nil {
		log.Fatalf("Failed to parse private key: %v", err)
	}

	publicKey := privateKey.Public()
	publicKeyECDSA, ok := publicKey.(*ecdsa.PublicKey)
	if !ok {
		log.Fatal("Failed to assert public key type")
	}

	fromAddress := crypto.PubkeyToAddress(*publicKeyECDSA)
	nonce, err := client.PendingNonceAt(context.Background(), fromAddress)
	if err != nil {
		log.Fatalf("Failed to get nonce: %v", err)
	}
	log.Printf("Nonce: %d\n", nonce)
	data := common.FromHex("f8a8fd6d")

	gasTipCap := uint256.NewInt(1000000000)  // 1 Gwei
	gasFeeCap := uint256.NewInt(20000000000) // 20 Gwei
	value := uint256.NewInt(0)               // 1 Ether
	toAddress := common.HexToAddress(toAddressHex)
	chainId := uint64(3151908)

	var delegatee string
	if *option == 1 {
		delegatee = "0x373E0B8B80A15cdf587C1263654c6B5edd195a43"
	} else {
		delegatee = "0x0000000000000000000000000000000000000000"
	}
	log.Printf("Delegatee: %s\n", delegatee)
	auth := types.SetCodeAuthorization{
		ChainID: *uint256.NewInt(chainId),
		Address: common.HexToAddress(delegatee),
		Nonce:   nonce + 1,
		V:       0,
		R:       *uint256.NewInt(0),
		S:       *uint256.NewInt(0),
	}

	signedAuth, err := types.SignSetCode(privateKey, auth)
	if err != nil {
		log.Fatalf("Failed to sign authorization: %v", err)
	}

	tx := &types.SetCodeTx{
		ChainID:    uint256.NewInt(chainId),
		Nonce:      nonce,
		GasTipCap:  gasTipCap,
		GasFeeCap:  gasFeeCap,
		Gas:        2000000,
		To:         toAddress,
		Value:      value,
		Data:       data,
        AccessList: types.AccessList{},
        AuthList:   []types.SetCodeAuthorization{signedAuth},
        V:          nil,
        R:          nil,
        S:          nil,
    }
    // Sign the transaction
    chainID, err := client.NetworkID(context.Background())
    if err != nil {
        log.Fatalf("Failed to get network ID: %v", err)
    }
    preTx := types.NewTx(tx)
    signedTx, err := types.SignTx(preTx, types.NewPragueSigner(chainID), privateKey)
    if err != nil {
        log.Fatalf("Failed to sign transaction: %v", err)
    }

    // Send the transaction
    err = client.SendTransaction(context.Background(), signedTx)
    if err != nil {
        log.Fatalf("Failed to send transaction: %v", err)
    }

    fmt.Printf("Transaction sent: %s\n", signedTx.Hash().Hex())
}
```

- Results: The monitoring shows the following:

```
2025-02-28 09:55:10,296 - INFO - Account 0x8943545177806ED17B9F23F0a21ee5948eCaa776 code verified
↪ successfully:
2025-02-28 09:55:13,311 - INFO - Account 0x8943545177806ED17B9F23F0a21ee5948eCaa776 code verified
↪ successfully:
2025-02-28 09:55:16,330 - INFO - Account 0x8943545177806ED17B9F23F0a21ee5948eCaa776 code verified
↪ successfully:
2025-02-28 09:55:19,353 - INFO - Account 0x8943545177806ED17B9F23F0a21ee5948eCaa776 code verified
↪ successfully: ef0100373e0b8b80a15cdf587c1263654c6b5edd195a43
2025-02-28 09:55:19,353 - WARNING - Code change detected at 0x8943545177806ED17B9F23F0a21ee5948eCaa776:
↪ Old: , New: ef0100373e0b8b80a15cdf587c1263654c6b5edd195a43
2025-02-28 09:55:22,376 - INFO - Account 0x8943545177806ED17B9F23F0a21ee5948eCaa776 code verified
↪ successfully: ef0100373e0b8b80a15cdf587c1263654c6b5edd195a43
2025-02-28 09:55:25,399 - INFO - Account 0x8943545177806ED17B9F23F0a21ee5948eCaa776 code verified
↪ successfully: ef0100373e0b8b80a15cdf587c1263654c6b5edd195a43
2025-02-28 09:55:28,422 - INFO - Account 0x8943545177806ED17B9F23F0a21ee5948eCaa776 code verified
↪ successfully: ef0100373e0b8b80a15cdf587c1263654c6b5edd195a43
2025-02-28 09:55:31,434 - INFO - Account 0x8943545177806ED17B9F23F0a21ee5948eCaa776 code verified
↪ successfully: ef0100373e0b8b80a15cdf587c1263654c6b5edd195a43
2025-02-28 09:55:34,469 - INFO - Account 0x8943545177806ED17B9F23F0a21ee5948eCaa776 code verified
↪ successfully: ef0100373e0b8b80a15cdf587c1263654c6b5edd195a43
2025-02-28 09:55:37,488 - INFO - Account 0x8943545177806ED17B9F23F0a21ee5948eCaa776 code verified
↪ successfully: ef0100373e0b8b80a15cdf587c1263654c6b5edd195a43
2025-02-28 09:55:40,516 - INFO - Account 0x8943545177806ED17B9F23F0a21ee5948eCaa776 code verified
↪ successfully: ef0100373e0b8b80a15cdf587c1263654c6b5edd195a43
2025-02-28 09:55:43,541 - WARNING - Code mismatch detected for account
↪ 0x8943545177806ED17B9F23F0a21ee5948eCaa776: {'Node 1':
↪ 'ef0100373e0b8b80a15cdf587c1263654c6b5edd195a43', 'Node 2': '', 'Node 3': '', 'Node 4': ''}
2025-02-28 09:55:43,541 - WARNING - Code change detected at 0x8943545177806ED17B9F23F0a21ee5948eCaa776:
↪ Old: ef0100373e0b8b80a15cdf587c1263654c6b5edd195a43, New:
2025-02-28 09:55:46,559 - WARNING - Code mismatch detected for account
↪ 0x8943545177806ED17B9F23F0a21ee5948eCaa776: {'Node 1':
↪ 'ef0100373e0b8b80a15cdf587c1263654c6b5edd195a43', 'Node 2': '', 'Node 3': '', 'Node 4': ''}
2025-02-28 09:55:49,581 - WARNING - Code mismatch detected for account
↪ 0x8943545177806ED17B9F23F0a21ee5948eCaa776: {'Node 1':
↪ 'ef0100373e0b8b80a15cdf587c1263654c6b5edd195a43', 'Node 2': '', 'Node 3': '', 'Node 4': ''}
2025-02-28 09:55:52,603 - WARNING - Code mismatch detected for account
↪ 0x8943545177806ED17B9F23F0a21ee5948eCaa776: {'Node 1':
↪ 'ef0100373e0b8b80a15cdf587c1263654c6b5edd195a43', 'Node 2': '', 'Node 3': '', 'Node 4': ''}
```

We can see that Erigon's node does not show zero code.

Recommendation: Erigon's node must be fixed to expose the correct code for EOAs. The following patch seems to fix the issue:

```
diff --git a/core/state_transition.go b/core/state_transition.go
index 98ef146189..290d4d3d62 100644
--- a/core/state_transition.go
+++ b/core/state_transition.go
@@ -436,7 +436,7 @@ func (st *StateTransition) TransitionDb(refunds bool, gasBailout bool) (*evmtype

                        // 7. set authority code
                        if auth.Address == (libcommon.Address{}) {
-                               if err := st.state.SetCode(authority, nil); err != nil {
+                               if err := st.state.SetCode(authority, []byte{}); err != nil {
                                        return nil, fmt.Errorf("%w: %w", ErrStateTransitionFailed, err)
                                }
                        } else {
```

3.1.2 EIP-7702: Incorrect `AuthRaw` in transaction parsing from the mempool

Submitted by Haxatron

Severity: Low Risk

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

Description: To compute the authorization hash to sign we do the following:

```
0x05 ++ rlp([chain_id, address_ nonce])
```

In Erigon, the `AuthRaw`, which aims to correspond to `rlp([chain_id, address_ nonce])` is extracted incorrectly from the transaction (pool_txn_parser.go#L471-L503):

```
for authPos < dataPos+dataLen {
    var authLen int
    authPos, authLen, err = rlp.ParseList(payload, authPos)
    if err != nil {
        return 0, fmt.Errorf("%w: authorization: %s", ErrParseTxn, err) //nolint
    }
    var sig Signature
    p2 := authPos
    rawStart := p2
    p2, err = rlp.ParseU256(payload, p2, &sig.ChainID)
    if err != nil {
        return 0, fmt.Errorf("%w: authorization chainId: %s", ErrParseTxn, err) //nolint
    }
    if !sig.ChainID.IsUint64() {
        // https://github.com/ethereum/EIPs/pull/8929
        return 0, fmt.Errorf("%w: authorization chainId is too big: %s", ErrParseTxn, &sig.ChainID)
    }
    p2, err = rlp.StringOfLen(payload, p2, length.Addr) // address
    if err != nil {
        return 0, fmt.Errorf("%w: authorization address: %s", ErrParseTxn, err) //nolint
    }
    p2 += length.Addr
    p2, _, err = rlp.ParseU64(payload, p2) // nonce
    if err != nil {
        return 0, fmt.Errorf("%w: authorization nonce: %s", ErrParseTxn, err) //nolint
    }
    rawEnd := p2
    p2, _, err = parseSignature(payload, p2, false /* legacy */, nil /* cfgChainId */, &sig)
    if err != nil {
        return 0, fmt.Errorf("%w: authorization signature: %s", ErrParseTxn, err) //nolint
    }
    slot.Authorizations = append(slot.Authorizations, sig)
    slot.AuthRaw = append(slot.AuthRaw, common.CopyBytes(payload[rawStart:rawEnd]))
```

Here, the `AuthRaw` is extracted from the transaction is just the slice of bytes from the start of the `chain_id` (`rawStart`) in the RLP list to the start of the `signature` (`rawEnd`) in the RLP list. In essence, this is how the `AuthRaw` is being computed in Erigon:

```
rlp(chain_id) ++ rlp(address) ++ rlp(nonce)
```

Note this is very different from:

```
rlp([chain_id, address_ nonce])
```

Which includes the corresponding length prefix of the list. The impact here is that the the signer recovered in the TXPool is incorrect (pool.go#L1487):

```
signer, err := types.RecoverSignerFromRLP(mt.TxnSlot.AuthRaw[i], uint8(signature.V.Uint64()), signature.R,
↪  signature.S)
```

Which also results in the transaction check from a user with pending authority failing (pool.go#L1505-L1513):

```
// Do not allow transaction from reserved authority
addr, ok := p.senders.getAddr(mt.TxnSlot.SenderID)
if !ok {
    p.logger.Info("senderID not registered, discarding transaction for safety")
    return txpoolcfg.InvalidSender
}
if _, ok := p.auths[addr]; ok {
    return txpoolcfg.ErrAuthorityReserved
}
```

Recommendation: Need to RLP encode the list `[chain_id, address_ nonce]` to correctly obtain `AuthRaw`, as done in authorization.go#L44-L63:

```
authLen := (1 + rlp.Uint256LenExcludingHead(&ath.ChainID))
authLen += 1 + length.Addr
authLen += rlp.U64Len(ath.Nonce)

if err := rlp.EncodeStructSizePrefix(authLen, data, b); err != nil {
    return nil, err
}

// chainId, address, nonce
if err := rlp.EncodeUint256(&ath.ChainID, data, b); err != nil {
    return nil, err
}

if err := rlp.EncodeOptionalAddress(&ath.Address, data, b); err != nil {
    return nil, err
}

if err := rlp.EncodeInt(ath.Nonce, data, b); err != nil {
    return nil, err
}

return RecoverSignerFromRLP(data.Bytes(), ath.YParity, ath.R, ath.S)
```

3.1.3 EIP-7702: Dangling authorizations in `p.auths` map in the mempool

Submitted by Haxatron

Severity: Low Risk

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

Description: Erigon tracks any pending delegations in the mempool using a `p.auths` map as shown (pool.go#L1505-L1513):

```go
// Check if we have txn with same authorization in the pool
if mt.TxnSlot.Type == SetCodeTxnType {
    numAuths := len(mt.TxnSlot.AuthRaw)
    foundDuplicate := false
    for i := range numAuths {
        signature := mt.TxnSlot.Authorizations[i]
        signer, err := types.RecoverSignerFromRLP(mt.TxnSlot.AuthRaw[i], uint8(signature.V.Uint64()),
        ↪  signature.R, signature.S)
        if err != nil {
            continue
        }

        if _, ok := p.auths[*signer]; ok {
            foundDuplicate = true
            break
        }

        p.auths[*signer] = mt
    }

    if foundDuplicate {
        return txpoolcfg.ErrAuthorityReserved
    }
}
```

As seen if a transaction has an authorization that is duplicate of another authorization already in the mempool, then `foundDuplicate` will be true and the transaction will be rejected with `txpoolcfg.ErrAuthorityReserved` error.

But if an EIP-7702 transaction has multiple authorization tuples, then previously created authorizations will be created in `p.auths` map before the transaction is rejected. This authorization will be dangling in the `p.auths` map as they will not be deleted when the transaction is rejected.

Recommendation: Consider tracking any newly added authorizations in a separate array and map before adding to `p.auths` map when the transaction is fully successful.

3.1.4 EIP-7702: Permanent DoS of an EOA that has performed a `SetCodeTx` due to improper handling in mempool

Submitted by Haxatron, also found by guhu95

Severity: Low Risk

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

Description: The new EIP-7702 transaction is handled differently in the mempool, with strict limit on how many pending transactions an EOA can have. However, how Erigon handles this transaction is problematic (pool.go#L1505-L1513):

```
// Don't add blob txn to queued if it's less than current pending blob base fee
if mt.TxnSlot.Type == BlobTxnType && mt.TxnSlot.BlobFeeCap.LtUint64(p.pendingBlobFee.Load()) {
    return txpoolcfg.FeeTooLow
}

// Check if we have txn with same authorization in the pool
if mt.TxnSlot.Type == SetCodeTxnType {
    numAuths := len(mt.TxnSlot.AuthRaw)
    foundDuplicate := false
    for i := range numAuths {
        signature := mt.TxnSlot.Authorizations[i]
        signer, err := types.RecoverSignerFromRLP(mt.TxnSlot.AuthRaw[i], uint8(signature.V.Uint64()),
        ↪  signature.R, signature.S)
        if err != nil {
            continue
        }

        if _, ok := p.auths[*signer]; ok {
            foundDuplicate = true
            break
        }

        p.auths[*signer] = mt
    }

    if foundDuplicate {
        return txpoolcfg.ErrAuthorityReserved
    }
}

// Do not allow transaction from reserved authority
addr, ok := p.senders.getAddr(mt.TxnSlot.SenderID)
if !ok {
    p.logger.Info("senderID not registered, discarding transaction for safety")
    return txpoolcfg.InvalidSender
}
if _, ok := p.auths[addr]; ok {
    return txpoolcfg.ErrAuthorityReserved
}
```

In particular in the above code, if the account has a pending authorization (which means a EIP-7702 transaction that is inside the mempool but has not been confirmed), which is stored in `p.auths`, then the account cannot submit anymore transactions until that transaction is confirmed. Here is the problem associated with this:

1. Firstly, anyone can submit a valid authorization tuple for an account that is different from the transaction sender. This means that it is possible to submit even an old valid authorization tuple, which will be added to a `p.auths` map.
2. If this occurs, the victim account will be stored in `p.auths`.
3. Effectively, the victim EOA can be permanently DOS from submitting transactions until the attacker's transaction that contains the victim authorization is removed from the mempool, the attacker can set extremely low base fee which effectively DoS it forever.

Note: Currently, the `AuthRaw` bug is blocking this bug from occurring, however when that bug is fixed, this bug will appear.

Also, when the `AuthRaw` bug is fixed, then self-sponsored SetCode transactions (where the sender includes their own authorization) will not even be possible because `p.auths` will be set before it is checked again and the transaction becomes rejected.

Recommendation: If the EOA has a pending delegation, then allow any transaction of the next nonce, this match the GETH behaviour.

3.1.5 Missing `authorizations` in `NewMessage()` When Converting Transaction of Type `SetCodeTransaction` into Message

Submitted by codegpt

Severity: Low Risk

Context: transaction.go#L376-L404

Summary: Missing `authorizations` in `NewMessage()` when converting transaction of type `SetCodeTransaction` into message would incorrectly process the transaction of type `SetCodeTransaction` (introduced in EIP-7702). Any invocation to this function will not work as expected. Consequently, all the relevant API calls (for example, `eth_call`) do not work for the transaction type `SetCodeTransaction`.

Finding Description: The function `NewMessage()` is intended to convert a transaction into a `message` of the following structure:

- `core/types/transaction.go#L357`:

  ```
  // Message is a fully derived transaction and implements core.Message
  type Message struct {
      to               *libcommon.Address
      from             libcommon.Address
      nonce            uint64
      amount           uint256.Int
      gasLimit         uint64
      gasPrice         uint256.Int
      feeCap           uint256.Int
      tip              uint256.Int
      maxFeePerBlobGas uint256.Int
      data             []byte
      accessList       AccessList
      checkNonce       bool
      isFree           bool
      blobHashes       []libcommon.Hash
      authorizations   []Authorization
  }
  ```

  However, it misses the inclusion of authorizations (last field in `Message`) when the transaction is of type `SetCodeTransaction` that is introduced in the EIP-7702. As a result, the resulting message does not have the `authorizations`, which could be problematic in any execution flow that invokes the function `NewMessage()` without `authorizations` in the transaction of type `SetCodeTransaction`.

- `core/types/transaction.go#L376`:

  ```
  func NewMessage(from libcommon.Address, to *libcommon.Address, nonce uint64, amount *uint256.Int,
  ↪   gasLimit uint64,
      gasPrice *uint256.Int, feeCap, tip *uint256.Int, data []byte, accessList AccessList, checkNonce
      ↪   bool,
      isFree bool, maxFeePerBlobGas *uint256.Int,
  ) *Message {
      m := Message{
          from:       from,
          to:         to,
          nonce:      nonce,
          amount:     *amount,
          gasLimit:   gasLimit,
          data:       data,
          accessList: accessList,
          checkNonce: checkNonce,
          isFree:     isFree,
      }
      if gasPrice != nil {
          m.gasPrice.Set(gasPrice)
      }
      if tip != nil {
          m.tip.Set(tip)
      }
      if feeCap != nil {
          m.feeCap.Set(feeCap)
      }
      if maxFeePerBlobGas != nil {
          m.maxFeePerBlobGas.Set(maxFeePerBlobGas)
      }
      return &m
  }
  ```

  For instance, the function `NewMessage()` is called in function `ToMessage()` when performing the `eth_call` API call:

- `turbo/adapter/ethapi/api.go#L65`:

```go
// ToMessage converts CallArgs to the Message type used by the core evm
func (args *CallArgs) ToMessage(globalGasCap uint64, baseFee *uint256.Int) (*types.Message, error) {
    // Reject invalid combinations of pre- and post-1559 fee styles
    if args.GasPrice != nil && (args.MaxFeePerGas != nil || args.MaxPriorityFeePerGas != nil) {
        return nil, errors.New("both gasPrice and (maxFeePerGas or maxPriorityFeePerGas) specified")
    }
    // Set sender address or use zero address if none specified.
    addr := args.from()

    // Set default gas & gas price if none were set
    gas := globalGasCap
    if gas == 0 {
        gas = uint64(math.MaxUint64 / 2)
    }
    if args.Gas != nil {
        gas = uint64(*args.Gas)
    }
    if globalGasCap != 0 && globalGasCap < gas {
        log.Warn("Caller gas above allowance, capping", "requested", gas, "cap", globalGasCap)
        gas = globalGasCap
    }

    var (
        gasPrice         *uint256.Int
        gasFeeCap        *uint256.Int
        gasTipCap        *uint256.Int
        maxFeePerBlobGas *uint256.Int
    )
    if baseFee == nil {
        // If there's no basefee, then it must be a non-1559 execution
        gasPrice = new(uint256.Int)
        if args.GasPrice != nil {
            overflow := gasPrice.SetFromBig(args.GasPrice.ToInt())
            if overflow {
                return nil, errors.New("args.GasPrice higher than 2^256-1")
            }
        }
        gasFeeCap, gasTipCap = gasPrice, gasPrice
    } else {
        // A basefee is provided, necessitating 1559-type execution
        if args.GasPrice != nil {
            // User specified the legacy gas field, convert to 1559 gas typing
            gasPrice = new(uint256.Int)
            overflow := gasPrice.SetFromBig(args.GasPrice.ToInt())
            if overflow {
                return nil, errors.New("args.GasPrice higher than 2^256-1")
            }
            gasFeeCap, gasTipCap = gasPrice, gasPrice
        } else {
            // User specified 1559 gas fields (or none), use those
            gasFeeCap = new(uint256.Int)
            if args.MaxFeePerGas != nil {
                overflow := gasFeeCap.SetFromBig(args.MaxFeePerGas.ToInt())
                if overflow {
                    return nil, errors.New("args.GasPrice higher than 2^256-1")
                }
            }
            gasTipCap = new(uint256.Int)
            if args.MaxPriorityFeePerGas != nil {
                overflow := gasTipCap.SetFromBig(args.MaxPriorityFeePerGas.ToInt())
                if overflow {
                    return nil, errors.New("args.GasPrice higher than 2^256-1")
                }
            }
            // Backfill the legacy gasPrice for EVM execution, unless we're all zeroes
            gasPrice = new(uint256.Int)
            if !gasFeeCap.IsZero() || !gasTipCap.IsZero() {
                gasPrice = math.U256Min(new(uint256.Int).Add(gasTipCap, baseFee), gasFeeCap)
            }
        }
        if args.MaxFeePerBlobGas != nil {
            blobFee, overflow := uint256.FromBig(args.MaxFeePerBlobGas.ToInt())
            if overflow {
                return nil, errors.New("args.MaxFeePerBlobGas higher than 2^256-1")
            }
            maxFeePerBlobGas = blobFee
        }
    }

    value := new(uint256.Int)
    if args.Value != nil {
        overflow := value.SetFromBig(args.Value.ToInt())
        if overflow {
            return nil, errors.New("args.Value higher than 2^256-1")
        }
    }
    var data []byte
    if args.Input != nil {
        data = *args.Input
    } else if args.Data != nil {
        data = *args.Data
    }
    var accessList types.AccessList
    if args.AccessList != nil {
        accessList = *args.AccessList
    }

    msg := types.NewMessage(addr, args.To, 0, value, gas, gasPrice, gasFeeCap, gasTipCap, data,
    ↪   accessList, false /* checkNonce */, false /* isFree */, maxFeePerBlobGas)
    return msg, nil
}
```

In which the function `ToMessage()` is called by function `DoCall()`:

- `turbo/transactions/call.go#L43`:

```
func DoCall(
    ctx context.Context,
    engine consensus.EngineReader,
    args ethapi2.CallArgs,
    tx kv.Tx,
    blockNrOrHash rpc.BlockNumberOrHash,
    header *types.Header,
    overrides *ethapi2.StateOverrides,
    gasCap uint64,
    chainConfig *chain.Config,
    stateReader state.StateReader,
    headerReader services.HeaderReader,
    callTimeout time.Duration,
) (*evmtypes.ExecutionResult, error) {
    // todo: Pending state is only known by the miner
    /*
        if blockNrOrHash.BlockNumber != nil && *blockNrOrHash.BlockNumber == rpc.PendingBlockNumber {
            block, state, _ := b.eth.miner.Pending()
            return state, block.Header(), nil
        }
    */

    state := state.New(stateReader)

    // Override the fields of specified contracts before execution.
    if overrides != nil {
        if err := overrides.Override(state); err != nil {
            return nil, err
        }
    }

    // Setup context so it may be cancelled the call has completed
    // or, in case of unmetered gas, setup a context with a timeout.
    var cancel context.CancelFunc
    if callTimeout > 0 {
        ctx, cancel = context.WithTimeout(ctx, callTimeout)
    } else {
        ctx, cancel = context.WithCancel(ctx)
    }

    // Make sure the context is cancelled when the call has completed
    // this makes sure resources are cleaned up.
    defer cancel()

    // Get a new instance of the EVM.
    var baseFee *uint256.Int
    if header != nil && header.BaseFee != nil {
        var overflow bool
        baseFee, overflow = uint256.FromBig(header.BaseFee)
        if overflow {
            return nil, errors.New("header.BaseFee uint256 overflow")
        }
    }
    msg, err := args.ToMessage(gasCap, baseFee)
    if err != nil {
        return nil, err
    }
    blockCtx := NewEVMBlockContext(engine, header, blockNrOrHash.RequireCanonical, tx, headerReader,
    ↪   chainConfig)
    txCtx := core.NewEVMTxContext(msg)

    evm := vm.NewEVM(blockCtx, txCtx, state, chainConfig, vm.Config{NoBaseFee: true})

    // Wait for the context to be done and cancel the evm. Even if the
    // EVM has finished, cancelling may be done (repeatedly)
    go func() {
        <-ctx.Done()
        evm.Cancel()
    }()

    gp := new(core.GasPool).AddGas(msg.Gas()).AddBlobGas(msg.BlobGas())
    result, err := core.ApplyMessage(evm, msg, gp, true /* refunds */, false /* gasBailout */, engine)
    if err != nil {
        return nil, err
    }

    // If the timer caused an abort, return an appropriate error message
    if evm.Cancelled() {
        return nil, fmt.Errorf("execution aborted (timeout = %v)", callTimeout)
    }
    return result, nil
}
```

This function is invoked in the API eth_call, `Call()`:

- `turbo/jsonrpc/eth_call.go#L64`:

```
// Call implements eth_call. Executes a new message call immediately without creating a transaction on
↪   the block chain.
func (api *APIImpl) Call(ctx context.Context, args ethapi2.CallArgs, blockNrOrHash
↪   rpc.BlockNumberOrHash, overrides *ethapi2.StateOverrides) (hexutil.Bytes, error) {
    tx, err := api.db.BeginTemporalRo(ctx)
    if err != nil {
        return nil, err
    }
    defer tx.Rollback()

    chainConfig, err := api.chainConfig(ctx, tx)
    if err != nil {
        return nil, err
    }
    engine := api.engine()

    if args.Gas == nil || uint64(*args.Gas) == 0 {
        args.Gas = (*hexutil.Uint64)(&api.GasCap)
    }

    blockNumber, hash, _, err := rpchelper.GetCanonicalBlockNumber(ctx, blockNrOrHash, tx,
    ↪   api._blockReader, api.filters) // DoCall cannot be executed on non-canonical blocks
    if err != nil {
        return nil, err
    }
    block, err := api.blockWithSenders(ctx, tx, hash, blockNumber)
    if err != nil {
        return nil, err
    }
    if block == nil {
        return nil, nil
    }
```

```
        stateReader, err := rpchelper.CreateStateReader(ctx, tx, api._blockReader, blockNrOrHash, 0,
        ↪  api.filters, api.stateCache, chainConfig.ChainName)
        if err != nil {
            return nil, err
        }
        header := block.HeaderNoCopy()
        result, err := transactions.DoCall(ctx, engine, args, tx, blockNrOrHash, header, overrides,
        ↪  api.GasCap, chainConfig, stateReader, api._blockReader, api.evmCallTimeout)
        if err != nil {
            return nil, err
        }

        if len(result.ReturnData) > api.ReturnDataLimit {
            return nil, fmt.Errorf("call returned result on length %d exceeding --rpc.returndata.limit %d",
            ↪  len(result.ReturnData), api.ReturnDataLimit)
        }

        // If the result contains a revert reason, try to unpack and return it.
        if len(result.Revert()) > 0 {
            return nil, ethapi2.NewRevertError(result)
        }

        return result.Return(), result.Err
    }
```

Note that the input `args` does not contain any `Authorization` that makes these APIs unusable for the transaction `SetCodeTransaction`:

- `turbo/adapter/ethapi/api.go#L41`:

```
    // CallArgs represents the arguments for a call.
    type CallArgs struct {
        From                 *libcommon.Address `json:"from"`
        To                   *libcommon.Address `json:"to"`
        Gas                  *hexutil.Uint64    `json:"gas"`
        GasPrice             *hexutil.Big       `json:"gasPrice"`
        MaxPriorityFeePerGas *hexutil.Big       `json:"maxPriorityFeePerGas"`
        MaxFeePerGas         *hexutil.Big       `json:"maxFeePerGas"`
        MaxFeePerBlobGas     *hexutil.Big       `json:"maxFeePerBlobGas"`
        Value                *hexutil.Big       `json:"value"`
        Nonce                *hexutil.Uint64    `json:"nonce"`
        Data                 *hexutil.Bytes     `json:"data"`
        Input                *hexutil.Bytes     `json:"input"`
        AccessList           *types.AccessList  `json:"accessList"`
        ChainID              *hexutil.Big       `json:"chainId,omitempty"`
    }
```

In case that user wants to perform the `eth_call` for transaction of type `SetCodeTransaction` , it will not work as expected due to the missing of `authorizations` in the resulting message.

Impact Explanation: The flawed function `NewMessage()` is invoked by the following APIs, which makes them unusable for transaction `SetCodeTransaction`:

- `CreateAccessList()`.
- `CallMany()`.
- `TraceCall()`.
- `TraceCallMany()`.
- `EstimateGas()`.
- `Call()`.

Additionally, it will also impact the Erigon node when running for Polygon network.

Likelihood Explanation: The case that all the aforementioned APIs do not work for the transaction `Set-CodeTransaction` definitely happens when user makes these API calls as they do not contain the `autho-rizations` in the transaction.

Proof of Concept: For simplicity, a simple unit test is provided for the function `NewMessage()`:

```
package types
```

```
import (
    "bytes"
    "crypto/ecdsa"
    "encoding/json"
    "errors"
    "fmt"
    "io"
    "math/big"
    "math/rand"
    "reflect"
    "testing"
    "time"

    "github.com/holiman/uint256"
    "github.com/stretchr/testify/assert"

    libcommon "github.com/erigontech/erigon-lib/common"
    "github.com/erigontech/erigon-lib/common/fixedgas"
    "github.com/erigontech/erigon-lib/common/u256"
    "github.com/erigontech/erigon-lib/crypto"
    "github.com/erigontech/erigon-lib/rlp"
    "github.com/erigontech/erigon/core/types/typestest"
)

func TestNewMessage(t *testing.T) {
    var (
        fromAddr         = libcommon.HexToAddress("b94f5374fce5edbc8e2a8697c15331677e6ebf0b")
        toAddr           = libcommon.HexToAddress("095e7baea6a6c7c4c2dfeb977efac326af552d87")
        nonce            = uint64(0)
        amount           = uint256.NewInt(10)
        gasLimit         = uint64(1000)
        gasPrice         = uint256.NewInt(10)
        feeCap           = uint256.NewInt(10)
        tip              = uint256.NewInt(10)
        data             = []byte{}
        maxFeePerBlobGas = uint256.NewInt(10)
    )

    msg := NewMessage(fromAddr, &toAddr, nonce, amount, gasLimit,
        gasPrice, feeCap, tip, data, nil, true,
        true, maxFeePerBlobGas)

    fmt.Printf("Authorizations in msg is empty? %t\n", len(msg.Authorizations()) == 0)

}
```

Test result:

```
=== RUN   TestNewMessage
Authorizations in msg is empty? true
--- PASS: TestNewMessage (0.00s)
PASS

Process finished with the exit code 0
```

Since there is not input of `authorization`, the resulting message cannot contain any `authorization`.

Recommendation: Adding the `authorization` to all the workflow with `SetCodeTransaction` transaction to ensure the resulting message contain them.

3.2 Informational

3.2.1 EIP-7702: `ErrAuthorityReserved` discard reason is not handled

Submitted by Haxatron

Severity: Informational

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

Description: We can see the following discard reason `ErrAuthorityReserved` (txpoolcfg.go#L120):

```
ErrAuthorityReserved DiscardReason = 34 // EIP-7702 transaction with authority already reserved
```

The discard reason is not handled during `String()` when converted to string to display to transaction submitter during RPC call (txpoolcfg.go#L123-L188):

```
func (r DiscardReason) String() string {
    // ...
    case GasLimitTooHigh:
        return "gas limit is too high"
    default:
        panic(fmt.Sprintf("discard reason: %d", r))
    }
```

The result is that the server returns `method handler crashed` when the error is triggered via multiple conflicting pending authorizations in the mempool.

```
Error: server returned an error response: error code -32000: method handler crashed
```

Recommendation: Handle it in the `switch-case` statement.

3.2.2 EIP-7702: Recommend preventing more than one pending transaction from accounts with a delegation designator

Submitted by Haxatron

Severity: Informational

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

Description: As per EIP-7702#Transaction Propagation:

> Allowing EOAs to behave as smart contracts via the delegation designation poses some challenges for transaction propagation. Traditionally, EOAs have only be able to send value via a transaction. This invariant allows nodes to statically determine the validity of transactions for that account. In other words, a single transaction has only been able to invalidate transactions pending from the senders account.
> With this EIP, it becomes possible to cause transactions from other accounts to become stale. This is due to the fact that once an EOA has delegated to code, that code can be called by anyone at any point in a transaction. It becomes impossible to know if the balance of the account has been sweeped in a static manner.
> While there are a few mitigations for this, the authors recommend that clients do not accept more than one pending transaction for any EOA with a non-zero delegation designator.

Erigon attempts to do this, but is flawed. In particular, Erigon only tracks any pending authorization of a pending transaction in the mempool, basically transaction that has not been confirmed (pool.go#L1481):

```
// Check if we have txn with same authorization in the pool
if mt.TxnSlot.Type == SetCodeTxnType {
    numAuths := len(mt.TxnSlot.AuthRaw)
    foundDuplicate := false
    for i := range numAuths {
        signature := mt.TxnSlot.Authorizations[i]
        signer, err := types.RecoverSignerFromRLP(mt.TxnSlot.AuthRaw[i], uint8(signature.V.Uint64()),
        ↪  signature.R, signature.S)
        if err != nil {
            continue
        }

        if _, ok := p.auths[*signer]; ok {
            foundDuplicate = true
            break
        }

        p.auths[*signer] = mt
    }

    if foundDuplicate {
        return txpoolcfg.ErrAuthorityReserved
    }
}

// Do not allow transaction from reserved authority
addr, ok := p.senders.getAddr(mt.TxnSlot.SenderID)
if !ok {
    p.logger.Info("senderID not registered, discarding transaction for safety")
    return txpoolcfg.InvalidSender
}
if _, ok := p.auths[addr]; ok {
    return txpoolcfg.ErrAuthorityReserved
}
```

If the transaction is confirmed in the block then the authorization is deleted from `p.auths` during `removeMined` which calls `discardLocked` to delete the authorization from `p.auths`. This means an account that have a delegation designator can send more than one pending transaction. In summary, Erigon only tracks the pending delegations in the `p.auths` map. However, both pending delegations and the current delegation designator of the account should be tracked as recommended by the EIP and done also by Nethermind and Geth.

Recommendation: Here is how Geth does the above by checking if the codehash of the account matches the empty code hash (legacypool.go#L613-L615):

```
// 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 {
```

3.2.3 `Erigon` computes `IntrinsicGas` after applying EIP-7702 authorizations

Submitted by alexfilippov314

Severity: Informational

Context: state_transition.go#L456

Description: Processing `EIP-7702` authorizations is computationally intensive, as it requires signature verification, state reading, and state updates. Therefore, it is crucial to ensure that a transaction provides sufficient gas before execution to prevent potential DoS issues.The issue arises because erigon applies EIP-7702 authorizations before computing `IntrinsicGas`, as shown in the following code snippet:

```
// set code tx
auths := msg.Authorizations()
verifiedAuthorities := make([]libcommon.Address, 0)
if len(auths) > 0 {
    if contractCreation {
        return nil, errors.New("contract creation not allowed with type4 txs")
    }
    var b [32]byte
    data := bytes.NewBuffer(nil)
```

```go
    for i, auth := range auths {
        data.Reset()

        // 1. chainId check
        if !auth.ChainID.IsZero() && rules.ChainID.String() != auth.ChainID.String() {
            log.Debug("invalid chainID, skipping", "chainId", auth.ChainID, "auth index", i)
            continue
        }

        // 2. authority recover
        authorityPtr, err := auth.RecoverSigner(data, b[:])
        if err != nil {
            log.Debug("authority recover failed, skipping", "err", err, "auth index", i)
            continue
        }
        authority := *authorityPtr

        // 3. add authority account to accesses_addresses
        verifiedAuthorities = append(verifiedAuthorities, authority)
        // authority is added to accessed_address in prepare step

        // 4. authority code should be empty or already delegated
        codeHash, err := st.state.GetCodeHash(authority)
        if err != nil {
            return nil, fmt.Errorf("%w: %w", ErrStateTransitionFailed, err)
        }
        if codeHash != emptyCodeHash && codeHash != (libcommon.Hash{}) {
            // check for delegation
            _, ok, err := st.state.GetDelegatedDesignation(authority)
            if err != nil {
                return nil, fmt.Errorf("%w: %w", ErrStateTransitionFailed, err)
            }
            if !ok {
                log.Debug("authority code is not empty or not delegated, skipping", "auth index", i)
                continue
            }
            // noop: has delegated designation
        }

        // 5. nonce check
        authorityNonce, err := st.state.GetNonce(authority)
        if err != nil {
            return nil, fmt.Errorf("%w: %w", ErrStateTransitionFailed, err)
        }
        if authorityNonce != auth.Nonce {
            log.Debug("invalid nonce, skipping", "auth index", i)
            continue
        }

        // 6. Add PER_EMPTY_ACCOUNT_COST - PER_AUTH_BASE_COST gas to the global refund counter if authority
        ↪  exists in the trie.
        exists, err := st.state.Exist(authority)
        if err != nil {
            return nil, fmt.Errorf("%w: %w", ErrStateTransitionFailed, err)
        }
        if exists {
            st.state.AddRefund(fixedgas.PerEmptyAccountCost - fixedgas.PerAuthBaseCost)
        }

        // 7. set authority code
        if auth.Address == (libcommon.Address{}) {
            if err := st.state.SetCode(authority, nil); err != nil {
                return nil, fmt.Errorf("%w: %w", ErrStateTransitionFailed, err)
            }
        } else {
            if err := st.state.SetCode(authority, types.AddressToDelegation(auth.Address)); err != nil {
                return nil, fmt.Errorf("%w: %w", ErrStateTransitionFailed, err)
            }
        }

        // 8. increase the nonce of authority
        if err := st.state.SetNonce(authority, authorityNonce+1); err != nil {
            return nil, fmt.Errorf("%w: %w", ErrStateTransitionFailed, err)
        }
    }
}
```

```
// Check clauses 4-5, subtract intrinsic gas if everything is correct
gas, floorGas7623, overflow := fixedgas.IntrinsicGas(st.data, uint64(len(accessTuples)),
↪  uint64(accessTuples.StorageKeys()), contractCreation, rules.IsHomestead, rules.IsIstanbul, isEIP3860,
↪  rules.IsPrague, uint64(len(auths)))
if overflow {
    return nil, ErrGasUintOverflow
}
if st.gasRemaining < gas || st.gasRemaining < floorGas7623 {
    return nil, fmt.Errorf("%w: have %d, want %d", ErrIntrinsicGas, st.gasRemaining, max(gas, floorGas7623))
}
```

This means a transaction could include numerous arbitrary authorizations while setting a low gas limit. Despite this, `erigon` will attempt to apply all authorizations before ultimately rejecting the transaction due to running out of gas.

That said, in practice, such transactions would be rejected before reaching the transaction pool. This suggests that the only viable way to execute this attack is by being a proposer. Even if the attack was carried out, the worst-case impact would likely be that some validators relying on `erigon` might fail to propose blocks on time or miss attestations.

Recommendation: Consider verifying intrinsic gas before applying `EIP-7702` authorizations.

3.2.4 `FlatRequests.Hash` implementation doesn't follow the specification

Submitted by alexfilippov314

Severity: Informational

Context: eip7685_requests.go#L54-L65

Description: EIP-7686 provides the following reference implementation for computing the requests hash:

```
def compute_requests_hash(block_requests: Sequence[bytes]):
    m = sha256()
    for r in block_requests:
        if len(r) > 1:
            m.update(sha256(r).digest())
    return m.digest()
```

`Erigon` uses the following implementation:

```
func (r FlatRequests) Hash() *libcommon.Hash {
    if r == nil {
        return nil
    }
    sha := sha256.New()
    for i, t := range r {
        hi := sha256.Sum256(append([]byte{t.Type}, r[i].RequestData...))
        sha.Write(hi[:])
    }
    h := libcommon.BytesToHash(sha.Sum(nil))
    return &h
}
```

This implementation does not strictly follow the specification, as it doesn't filter out empty requests. Here, an empty request refers to a request with a non-empty request type but empty request data. That said, at present, such requests do not occur during normal execution, so there is no impact.

Recommendation: Consider filtering out requests with empty `RequestData`.

3.2.5 `Erigon` is unable to estimate gas for transactions involving `EIP-7623` or `EIP-7702`

Submitted by alexfilippov314

Severity: Informational

Context: eth_call.go#L304

Description: The `eth_estimateGas` call fails for transactions involving `EIP-7623` or `EIP-7702`. The issue arises due to the following reasons:

1. `eth_estimateGas` is implemented in the `APIImpl.EstimateGas` function.
2. Initially, this function attempts to execute the transaction with either the user-provided gas value or a default "high" gas value:

```
// First try with highest gas possible
result, err := caller.DoCallWithNewGas(ctx, hi, engine, overrides)
if err != nil || result == nil {
    return 0, err
}
```

3. It then uses the gas spent in this call as the lower bound for the gas estimate:

```
// Assuming a contract can freely run all the instructions, we have
// the true amount of gas it wants to consume to execute fully.
// We want to ensure that the gas used doesn't fall below this
trueGas := result.EvmGasUsed // Must not fall below this
lo = min(trueGas+result.EvmRefund-1, params.TxGas-1)
```

4. The `lo` value here will likely always be set to 20999.
5. `Erigon` then performs a binary search to refine the gas estimate:

```
// Execute the binary search and hone in on an executable gas limit
for lo+1 < hi {
    mid := (hi + lo) / 2
    result, err := caller.DoCallWithNewGas(ctx, mid, engine, overrides)
    // If the error is not nil(consensus error), it means the provided message
    // call or transaction will never be accepted no matter how much gas it is
    // assigened. Return the error directly, don't struggle any more.
    if err != nil {
        return 0, err
    }
    if result.Failed() || result.EvmGasUsed < trueGas {
        lo = mid
    } else {
        hi = mid
    }
    i++
}
```

6. The issue arises because the next gas limit value is computed as `mid := (hi + lo) / 2`. For transactions involving `EIP-7702` and `EIP-7623`, this value might result in an intrinsic gas error:

```
if st.gasRemaining < gas || st.gasRemaining < floorGas7623 {
    return nil, fmt.Errorf("%w: have %d, want %d", ErrIntrinsicGas, st.gasRemaining, max(gas,
    ↪  floorGas7623))
}
```

7. As a result, `eth_estimateGas` execution fails with an error.

Proof of Concept: Run the following test with `Erigon` and other execution clients:

```go
package main

import (
    "context"
    "fmt"
    "testing"

    "github.com/ethereum/go-ethereum"
    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/ethclient"
    "github.com/stretchr/testify/require"
)

func TestPocErigonEstimateGasEip7623(t *testing.T) {
    // 1. Generate huge calldata
    data := make([]byte, 40000)
    for i := range data {
        data[i] = byte(i)
    }

    // 2. Prepare message
    msg := ethereum.CallMsg{
        To:   &common.Address{},
        From: common.Address{},
        Data: data,
    }

    // 3. Check the gas estimate for all RPCs. The RPCs correspond to Geth, Reth, and Erigon nodes,
    ↪  respectively.
    rpcs := []string{"http://127.0.0.1:32913", "http://127.0.0.1:32918", "http://127.0.0.1:32923"}
    for _, rpc := range rpcs {
        client, err := ethclient.Dial(rpc)
        require.NoError(t, err)
        gas, err := client.EstimateGas(context.Background(), msg)
        require.NoError(t, err)
        fmt.Println(gas)
    }
}
```

The request to an `Erigon` RPC should fail with an error.

Recommendation: Consider updating the `lo` value to `lo = mid` in case of an intrinsic gas error.

3.2.6 `Erigon` doesn't verify that `executionRequests` is not `nil` after Pectra

Submitted by alexfilippov314

Severity: Informational

Context: engine_server.go#L154-L161

Description: The `engine_newPayloadV4` method has an `executionRequests` parameter representing `EIP-7685` execution requests. While the specification does not explicitly mention this, all other clients reject requests where `executionRequests` is `nil/null/None`.

Example from geth:

```go
if executionRequests == nil {
    return engine.PayloadStatusV1{Status: engine.INVALID}, engine.InvalidParams.With(errors.New("nil
    ↪  executionRequests post-prague"))
}
```

The issue arises because `Erigon` accepts `nil` and effectively interprets it as `[]`. On its own, this behavior does not cause issues if the consensus client functions correctly. However, if a bug in the consensus client causes it to send `nil` instead of `[]`, all `Erigon` nodes paired with this consensus client will split from the rest of the network.

Recommendation: Consider verifying that `executionRequests` is not nil in the `checkRequestsPresence` function.