Blockchain Security Audit Report

Table Of Contents

1 Executive Summary

On 2025.03.19, the SlowMist security team received the ABFoundationGlobal team's security audit application for abcore, developed the audit plan according to the agreement of both parties and the characteristics of the project, and finally issued the security audit report.

The SlowMist security team adopts the strategy of "black, grey box lead, white box assists" to conduct a complete security test on the project in the way closest to the real attack.

The test method information:

Test method	Description
Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code modules through the scripting tool, observing the internal running status, mining weaknesses.
White box testing	Based on the open source code, non-open source code, to detect whether there are vulnerabilities in programs such as nodes, SDK, etc.

The vulnerability severity level information:

Level	Description
Critical	Critical severity vulnerabilities will have a significant impact on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High	High severity vulnerabilities will affect the normal operation of the DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium	Medium severity vulnerability will affect the operation of the DeFi project. It is recommended to fix medium-risk vulnerabilities.

Low	Low severity vulnerabilities may affect the operation of the DeFi project in certain scenarios. It is suggested that the project party should evaluate and consider whether these vulnerabilities need to be fixed.
Weakness	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.
Suggestion	There are better practices for coding or architecture.

2 Audit Methodology

The security audit process of SlowMist security team for the chain includes two steps:

Chain codes are scanned/tested for commonly known and more specific vulnerabilities using automated analysis tools.

Manual audit of the codes for security issues. The codes are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the chain:

NO.	Audit Items	Result
1	Cryptographic Security Audit	Passed
2	Account and Transaction Security Audit	Some Risks
3	RPC Security Audit	Passed
4	P2P Security Audit	Passed
5	Consensus Security Audit	Some Risks

3 Project Overview

3.1 Project Introduction

ABCore is based on official golang implementation of the Ethereum protocol.

3.2 Coverage

Target Code and Revision:

https://github.com/ABFoundationGlobal/abcore

Audit Commit: 2c08edb4b37daf1e33649299e003d2f67b1dce4d

Cryptographic Security Audit

(1) Insufficient entropy of private key random numbers audit

Generate encrypted accounts using the “crypto/rand” library to ensure that private keys cannot be predicted or cracked.

- accounts/keystore/keystore.go

```
import (
   "crypto/ecdsa"
   crand "crypto/rand"


//...
// NewAccount generates a new key and stores it into the key directory,
// encrypting it with the passphrase.
func (ks *KeyStore) NewAccount(passphrase string) (accounts.Account, error) {
   _, account, err := storeNewKey(ks.storage, crand.Reader, passphrase)
```

```
    if err != nil {
        return accounts.Account{}, err
    }
    // Add the account to the cache immediately rather
    // than waiting for file system notifications to pick it up.
    ks.cache.add(account)
    ks.refreshWallets()
    return account, nil
}
```

(2) Precision loss in private key seed conversion

Use ecdsa from the Go standard library to generate private keys, which are defined to be of type `big.Int`, with enough length to preserve random seeds without causing the strength of the private key to degrade.

```
// PrivateKey is a representation of an elliptic curve private key.
type PrivateKey struct {
    PublicKey
    D *big.Int
}
```

- accounts/keystore/key.go

```
func newKey(rand io.Reader) (*Key, error) {
    privateKeyECDSA, err := ecdsa.GenerateKey(crypto.S256(), rand)
    if err != nil {
        return nil, err
    }
    return newKeyFromECDSA(privateKeyECDSA), nil
}
```

(3) Theoretical reliability assessment of symmetric encryption algorithms

Scrypt key management:

- accounts/keystore/passphrase.go

ECIES encryption:

- crypto/ecies/params.go

(4) Theoretical reliability assessment of hash algorithms

```
- SHA-1/224/256/384/512: for message digests
- Keccak256: Address generation
- HMAC: Message Authentication Code
```

(5) Theoretical reliability assessment of signature algorithms

```
secp256k1
```

(6) Supply chain security of symmetric crypto algorithm reference libraries

The main task is to check the supply chain security of the reference libraries of the symmetric encryption algorithms used.

AB BlockChain's cryptographic libraries use the Ethereum library, which is mature, reliable and widely used.

Reference

https://github.com/ethereum/go-ethereum/tree/master/crypto

(7) Keystore encryption strength detection

The main purpose is to check the encryption strength of the Keystore.

The password strength of passphrase is not detected and there is a risk of weak passwords.

- accounts/keystore/keystore.go

```
// NewAccount generates a new key and stores it into the key directory,
// encrypting it with the passphrase.
func (ks *KeyStore) NewAccount(passphrase string) (accounts.Account, error) {
   _, account, err := storeNewKey(ks.storage, crand.Reader, passphrase)
   if err != nil {
       return accounts.Account{}, err
   }
   // Add the account to the cache immediately rather
   // than waiting for file system notifications to pick it up.
   ks.cache.add(account)
   ks.refreshWallets()
   return account, nil
}
```

(8) Hash algorithm length extension attack

The main purpose is to check the length extension attack of the hash algorithm.

Using Keccak256 algorithm to calculate transaction hash, there is no hash length extension attack problem.

- core/types/block.go

```
// rlpHash encodes x and hashes the encoded bytes.
func rlpHash(x interface{}) (h common.Hash) {
   sha := hasherPool.Get().(crypto.KeccakState)
   defer hasherPool.Put(sha)
   sha.Reset()
   rlp.Encode(sha, x)
   sha.Read(h[:])
   return h
}
```

(9) Merkle-tree Malleability attack

The main purpose is to check the modifiability of the Merkle Tree in the blockchain system.

In some merkle-tree algorithms, if there are an odd number of nodes, the last node is automatically copied to spell an even number, and an attacker may try to double-flush by including two identical transactions at the end of the block.

The Merkle-tree scheme used by AB BlockChain is consistent with that of Ethereum and there are no security issues.

(10) secp256k1 k-value randomness security

AB BlockChain has used the secp256k1 signature algorithm. The K value is generated using the RFC-6979 standard.

(11) secp256k1 r-value reuse private key extraction attack

AB BlockChain has used the secp256k1 signature algorithm.

(12) ECC signature malleability attack

The main task is to examine the malleability of signatures in Elliptic Curve Digital Signature (ECC signature), as well as the related security risks and potential attacks.

AB BlockChain has used the secp256k1 signature algorithm.

(13) Ed25519 private key extraction attack

The main purpose is to check the risks and impacts of the private key extraction attack in the Ed25519 elliptic curve digital signature algorithm.

AB BlockChain has not used the ed25519 signature algorithm.

(14) Schnorr private key extraction attack

The main task is to check the risks and impacts of the private key extraction attack existing in the Schnorr signature algorithm.

AB BlockChain has not used the ed25519 signature algorithm.

(15) ECC twist attack

The main task is to check the risks and impacts of the elliptic curve twist attack existing in Elliptic Curve Cryptography (ECC).

AB BlockChain does not use the elliptic curve cryptography library with security issues.

Account and Transaction Security Audit

(1) Native characteristic false recharge

The main objective is to examine the risk and impact of possible native feature false top-up vulnerabilities in blockchain systems.

Native feature false recharge refers to attackers exploiting the characteristics or vulnerabilities of the blockchain system to forge recharge records on the blockchain. This kind of attack may cause losses to users, exchanges or blockchain platforms. It should be noted that AB BlockChain has a notable feature: failed transactions may still be submitted to the chain, and the status of each transaction needs to be confirmed.

(2) Contract call-based false recharge

The main check is the risks and impacts of potential false recharge vulnerabilities based on contract calls that may exist in blockchain smart contracts.

AB BlockChain adopts an EVM-compatible smart contract architecture, which brings powerful functions while also inheriting certain complex characteristics. One of the particularly notable ones is the behavior of cross-contract calls. In a complex transaction, even if an internal cross-contract call fails, the entire transaction may still be marked as successful. This feature may lead to potential security risks, such as vulnerabilities like 'false top-up'. Therefore, it is necessary to deeply verify the execution results of all internal calls in the transaction. Only when all related internal transactions are successfully executed can the validity of the entire transaction be truly confirmed.

(3) Native chain transaction replay attack

The main purpose is to assess the risks and impacts of possible local chain transaction replay attacks in the blockchain network.

Transaction replay attacks refer to a type of attack where the attacker resubmits previously valid transaction data to the blockchain network, deceiving network nodes and participants, causing the transaction to be executed repeatedly. This may result in unnecessary financial losses, transaction delays, or other negative impacts.

For each address in AB BlockChain, a Nonce is added as a parameter for transactions. After a successful transaction, the Nonce is incremented by one. Therefore, there is no problem of transaction replay.

(4) Heterogeneous chain transaction replay attack

The main task is to examine the risks and impacts of potential transaction replay attacks that may exist between heterogeneous chains.

Transaction replay attacks between heterogeneous chains refer to a type of attack where attackers exploit the interoperability between different blockchain networks to repeat a valid transaction that was successfully executed on one chain on another chain, in order to gain improper benefits or cause system damage. In this type of attack, the attacker copies previously valid transaction data across chains and submits it to another chain, deceiving nodes and participants on the chain, causing the transaction to be executed repeatedly. This may result in financial losses, transaction delays, or other negative impacts.

Although AB BlockChain is designed to be fully compatible with the EVM, a specific chainID is embedded in the signature data of each transaction to prevent the transaction from being directly re-executed on another chain.

(5) Transaction lock attack

The main task is to check the risks and impacts of possible transaction locking attacks in the blockchain system.

Transaction locking attack refers to the manipulation of blockchain transactions by malicious users or attackers to make certain funds or resources remain locked for a long period of time, in order to achieve the purpose of causing damage to the system, interfering with it, or obtaining undue benefits. In this type of attack, the attacker may take advantage of incompletely understood smart contract logic, blockchain protocol vulnerabilities, or transaction sequencing rules to make specific transactions unable to be confirmed or executed, resulting in funds or resources being locked for a long period of time, affecting the normal operation of the system and user experience.

Transactions in AB BlockChain do not have a locking function.

RPC Security Audit

(1) RPC remote key theft attack

The main task is to examine the risks and impacts of potential RPC remote key theft attacks that may exist in the blockchain system.

The interface follows the Ethernet JSON-RPC specification.

The API provides interfaces such as remote signatures, and there is a risk of remote coin theft. An attacker may remotely initiate signature operations to steal node assets during the interval when the node's account is unlocked.

- internal/ethapi/api.go

```
func (s *TransactionAPI) Sign(addr common.Address, data hexutil.Bytes) (hexutil.Bytes,
error) {
   // Look up the wallet containing the requested signer
   account := accounts.Account{Address: addr}


   wallet, err := s.b.AccountManager().Find(account)
   if err != nil {
       return nil, err
   }
   // Sign the requested hash with the wallet
   signature, err := wallet.SignText(account, data)
   if err == nil {
       signature[64] += 27 // Transform V from 0/1 to 27/28 according to the yellow
paper
   }
   return signature, err
}


func (s *TransactionAPI) SignTransaction(ctx context.Context, args TransactionArgs)
(*SignTransactionResult, error) {
   if args.Gas == nil {
       return nil, errors.New("gas not specified")
   }
   if args.GasPrice == nil && (args.MaxPriorityFeePerGas == nil || args.MaxFeePerGas
== nil) {
       return nil, errors.New("missing gasPrice or maxFeePerGas/maxPriorityFeePerGas")
   }
   if args.IsEIP4844() {
       return nil, errBlobTxNotSupported
   }
   if args.Nonce == nil {
       return nil, errors.New("nonce not specified")
   }
   if err := args.setDefaults(ctx, s.b, false); err != nil {
       return nil, err
   }
   // Before actually sign the transaction, ensure the transaction fee is reasonable.
   tx := args.toTransaction()
   if err := checkTxFee(tx.GasPrice(), tx.Gas(), s.b.RPCTxFeeCap()); err != nil {
       return nil, err
   }
   signed, err := s.sign(args.from(), tx)
   if err != nil {
       return nil, err
   }
   data, err := signed.MarshalBinary()
   if err != nil {
       return nil, err
   }
   return &SignTransactionResult{data, signed}, nil
}
```

It is necessary to pay attention to blocking such sensitive interfaces when deploying nodes.

(2) RPC port identifiability

Port 8545/8546 is used by default, and can also be specified with a command line parameter, or through a configuration file.

- node/defaults.go

```
const (
    DefaultHTTPHost = "localhost" // Default host interface for the HTTP RPC server
    DefaultHTTPPort = 8545        // Default TCP port for the HTTP RPC server
    DefaultWSHost   = "localhost" // Default host interface for the websocket RPC
server
    DefaultWSPort   = 8546         // Default TCP port for the websocket RPC server
    DefaultAuthHost = "localhost" // Default host interface for the authenticated apis
    DefaultAuthPort = 8551         // Default port for the authenticated apis
)
```

(3) RPC open cross-domain vulnerability to local phishing attacks

The main check is to assess the cross-domain vulnerabilities of RPC interfaces in the blockchain system to determine whether the system is vulnerable to local phishing attacks.

The hacker tricks the victim into opening a malicious webpage, connects to the cryptocurrency wallet RPC port through a cross-domain request, and then steals crypto assets.

Test with a public RPC:

```
curl -s -v -D- 'https://rpc.core.testnet.ab.org' \
  -H 'content-type: application/json' \
  --data-raw '{
 "id":1,
 "jsonrpc":"2.0",
 "method":"eth_getTransactionByHash",
 "params":["0xe7fb5bee682ace0b9a9f4d9d08f93a06ece5a33809dfb901d1752cf7acaf7a98"]
}'
```

`Access-Control-Allow-Origin: *` Found cross-origin issues on the test network.

(4) JsonRPC malformed packet denial-of-service attack

The main check is the security of the JsonRPC interface in the blockchain system to determine whether the system is vulnerable to Denial of Service (DoS) attacks caused by maliciously constructed abnormal JSON data packets.

Constructing malformed data for testing node RPCs:

```
                    datainfo[str(i)] = '{slowmist}' * 0x101000 + '":' + '{"x":' * 0x10000
+ '"}'
                    response = requests.post(url, json=datainfo, headers=headers)
```

And

```
g_req += 'Content-Type: application/json\r\n'
g_req += 'Content-length: %d\r\n\r\n' % 0xffffffffffffffff
g_req += '{"method":"'
```

Returned results normally and did not cause the node to crash.

```
http.client.RemoteDisconnected: Remote end closed connection without response
```

(5) RPC database injection

The main check is whether there is a database injection problem.

There are no database queries using SQL statements and there are no RPC injection issues.

(6) RPC communication encryption

The main purpose is to check whether the RPC (Remote Procedure Call) communication in the blockchain system has appropriate encryption protection.

RPC does not use HTTPS encryption by default, but the node operator can encrypt the communication by adding a proxy such as Nginx in the front.

Related Code

- rpc/*

P2P Security Audit

(1) P2P communication encryption

The main check is whether the P2P (peer-to-peer) communication between nodes in the blockchain network uses an appropriate encryption mechanism to protect the privacy and security of the communication content.

The P2P encryption implementation is divided into the following main parts:

The handshake process uses the RLPx encrypted transmission protocol

- p2p/server.go

```
func (srv *Server) setupConn(c *conn, flags connFlag, dialDest *enode.Node)
//...
   // Run the RLPx handshake.
   remotePubkey, err := c.doEncHandshake(srv.PrivateKey)
//...
```

- p2p/rlpx/rlpx.go

```
// Secrets represents the connection secrets which are negotiated during the
handshake.
type Secrets struct {
   AES, MAC               []byte
   EgressMAC, IngressMAC hash.Hash
   remote                *ecdsa.PublicKey
}
```

The cryptographic handshake process uses ECDH for key exchange and ECIES for message encryption.

- p2p/transport.go

```
func (t *rlpxTransport) doEncHandshake(prv *ecdsa.PrivateKey) (*ecdsa.PublicKey,
error) {
   t.conn.SetDeadline(time.Now().Add(handshakeTimeout))
   return t.conn.Handshake(prv)
}
```

Symmetric encryption using AES-CTR mode and message integrity verification using MAC.

- p2p/rlpx.go

```
func (c *Conn) InitWithSecrets(sec Secrets) {
   if c.session != nil {
       panic("can't handshake twice")
   }
   macc, err := aes.NewCipher(sec.MAC)
   if err != nil {
       panic("invalid MAC secret: " + err.Error())
   }
   encc, err := aes.NewCipher(sec.AES)
   if err != nil {
       panic("invalid AES secret: " + err.Error())
   }
   // we use an all-zeroes IV for AES because the key used
   // for encryption is ephemeral.
   iv := make([]byte, encc.BlockSize())
   c.session = &sessionState{
       enc:        cipher.NewCTR(encc, iv),
       dec:        cipher.NewCTR(encc, iv),
       egressMAC:  newHashMAC(macc, sec.EgressMAC),
       ingressMAC: newHashMAC(macc, sec.IngressMAC),
    }
}
```

v4 uses ECDSA-based signatures to verify identity

v5 uses AES/GCM mode for message encryption

- p2p/discover/v5wire/encoding.go

```
func (c *Codec) encryptMessage(s *session, p Packet, head *Header, headerData []byte)
([]byte, error) {
   // Encode message plaintext.
   c.msgbuf.Reset()
   c.msgbuf.WriteByte(p.Kind())
   if err := rlp.Encode(&c.msgbuf, p); err != nil {
       return nil, err
   }
   messagePT := c.msgbuf.Bytes()


   // Encrypt into message ciphertext buffer.
   messageCT, err := encryptGCM(c.msgctbuf[:0], s.writeKey, head.Nonce[:], messagePT,
headerData)
   if err == nil {
       c.msgctbuf = messageCT
   }
   return messageCT, err
}
```

(2) Insufficient number of core nodes

The main check is whether the number of core nodes in the blockchain network is sufficient to ensure the security and stability of the network.

Utilizing the PoA consensus algorithm, a small number of nodes hold the rights to produce blocks, with fewer nodes, there is a higher risk of centralization.

(3) Excessive concentration of core node physical locations

The main check is whether the physical location distribution of the core nodes in the blockchain network is too concentrated.

(4) P2P node maximum connection limit

The main check is the maximum connection limit of a blockchain node to other nodes.

The `MaxPeers` parameter is defined in config to limit the maximum number of connections to peer nodes, preventing the system from experiencing performance degradation or even denial of service due to too many connections.

- p2p/server.go

```
// Config holds Server options.
type Config struct {


   //...
   // MaxPeers is the maximum number of peers that can be
   // connected. It must be greater than zero.
   MaxPeers int


//...
```

- node/defaults.go

```
    P2P: p2p.Config{
        ListenAddr: ":30303",
        MaxPeers:   50,
        NAT:        nat.Any(),
    },
```

(5) P2P node independent IP connection limit

The main check is the limit on the number of independent connections of the blockchain node to the same IP address.

Use `inboundHistory` to record connected nodes, while checking whether the same IP has an existing connection when a new connection is made, avoiding the malicious construction of a large number of nodes by one IP node to occupy the connection pool of the target node.

- p2p/server.go

```
func (srv *Server) checkInboundConn(remoteIP net.IP) error {
    if remoteIP == nil {
        return nil
    }
    // Reject connections that do not match NetRestrict.
    if srv.NetRestrict != nil && !srv.NetRestrict.Contains(remoteIP) {
        return errors.New("not in netrestrict list")
    }
    // Reject Internet peers that try too often.
    now := srv.clock.Now()
    srv.inboundHistory.expire(now, nil)
    if !netutil.IsLAN(remoteIP) && srv.inboundHistory.contains(remoteIP.String()) {
        return errors.New("too many attempts")
    }
    srv.inboundHistory.add(remoteIP.String(), now.Add(inboundThrottleTime))
    return nil
}
```

(6) P2P inbound/outbound connection limit

The main check is the limit on the number of incoming and outgoing connections of the blockchain node.

DialRatio controls the ratio of inbound to dialed connections.

- p2p/server.go

```
// Config holds Server options.
type Config struct {


   // DialRatio controls the ratio of inbound to dialed connections.
   // Example: a DialRatio of 2 allows 1/2 of connections to be dialed.
   // Setting DialRatio to zero defaults it to 3.
   DialRatio int `toml:",omitempty"`


   //...
}
//...
func (srv *Server) postHandshakeChecks(peers map[enode.ID]*Peer, inboundCount int, c
*conn) error {
   switch {
   case !c.is(trustedConn) && len(peers) >= srv.MaxPeers:
       return DiscTooManyPeers
   case !c.is(trustedConn) && c.is(inboundConn) && inboundCount >=
srv.maxInboundConns():
       return DiscTooManyPeers
   case peers[c.node.ID()] != nil:
       return DiscAlreadyConnected
   case c.node.ID() == srv.localnode.ID():
       return DiscSelf
```

```
    default:
        return nil
    }
}
```

(7) P2P Alien attack

Both Discv4 and Discv5 are enabled on the node discovery protocol, but the protocol does not support verifying that peer nodes are on the same chain during handshaking, which may cause nodes on the current chain and the address pools of similar chains, such as Ethereum, to pollute each other, resulting in degraded network performance or even congestion.

- p2p/server.go

```
func (srv *Server) setupDiscovery() error {
  //...
   // If both versions of discovery are running, setup a shared
   // connection, so v5 can read unhandled messages from v4.
   if srv.DiscoveryV4 && srv.DiscoveryV5 {
       unhandled = make(chan discover.ReadPacket, 100)
       sconn = &sharedUDPConn{conn, unhandled}
   }


   // Start discovery services.
//...
       ntab, err := discover.ListenV4(conn, srv.localnode, cfg)
       if err != nil {
           return err
       }
       srv.ntab = ntab
       srv.discmix.AddSource(ntab.RandomNodes())
   }
```

```
//...
        srv.DiscV5, err = discover.ListenV5(sconn, srv.localnode, cfg)
        if err != nil {
            return err
        }
    }
//...

}
```

- p2p/discover/v4wire/v4wire.go

```
    Ping struct {
        Version    uint
        From, To   Endpoint
        Expiration uint64
        ENRSeq     uint64 `rlp:"optional"` // Sequence number of local record, added by
EIP-868.


        // Ignore additional fields (for forward compatibility).
        Rest []rlp.RawValue `rlp:"tail"`
    }


    // Pong is the reply to ping.
    Pong struct {
        // This field should mirror the UDP envelope address
        // of the ping packet, which provides a way to discover the
        // external address (after NAT).
        To         Endpoint
        ReplyTok   []byte // This contains the hash of the ping packet.
        Expiration uint64 // Absolute timestamp at which the packet becomes invalid.
        ENRSeq     uint64 `rlp:"optional"` // Sequence number of local record, added by
EIP-868.
```

```
        // Ignore additional fields (for forward compatibility).
        Rest []rlp.RawValue `rlp:"tail"`
    }


    // Findnode is a query for nodes close to the given target.
    Findnode struct {
        Target     Pubkey
        Expiration uint64
        // Ignore additional fields (for forward compatibility).
        Rest []rlp.RawValue `rlp:"tail"`
    }
```

Reference:

https://mp.weixin.qq.com/s/UmricgYGUakAlZTb0ihqdw

(8) P2P port identifiability

The main purpose is to check whether the ports used for P2P (peer-to-peer) communication between nodes in the blockchain network are easy to be identified and tracked.

Port 38311 is used by default, and can also be specified with the command line parameter --port, or through a configuration file.

Consensus Security Audit

(1) Excessive administrator privileges

The main task is to check whether the system has administrator permissions or special beneficiary accounts to ensure the rationality, minimization and decentralization of permission control, thereby guaranteeing that there is no fraudulent behavior in the system.

There is not special administrator permission in AB BlockChain.

(2) Transaction fees not dynamically adjusted

The main check is whether the transaction fees in the blockchain system are dynamically adjusted according to the network conditions and demands.

In the AB BlockChain, transaction fees are dynamically adjusted based on the network congestion situation to ensure the efficiency and fairness of transaction processing.

(3) Miner grinding attack

The main purpose is to assess the potential risk of grinding attacks by miners in the blockchain system.

The PoA consensus algorithm will not have the grinding block attack problem.

(4) Final confirmation conditions

Block time is 3 seconds, so block finalized time is 75 hours

- params/network_params.go

```
// FullImmutabilityThreshold is the number of blocks after which a chain segment is
// considered immutable (i.e. soft finality). It is used by the downloader as a
// hard limit against deep ancestors, by the blockchain against deep reorgs, by
// the freezer as the cutoff threshold and by clique as the snapshot trust limit.
FullImmutabilityThreshold = 90000
```

```
    // LightImmutabilityThreshold is the number of blocks after which a header chain
    // segment is considered immutable for light client(i.e. soft finality). It is used
by
    // the downloader as a hard limit against deep ancestors, by the blockchain against
deep
    // reorgs, by the light pruner as the pruning validity guarantee.
    LightImmutabilityThreshold = 30000
```

(5) Double-signing penalty

Using the PoA consensus algorithm, the security of the network is guaranteed by the credibility of authoritative nodes.

If the authoritative node can double sign to construct another fork chain, then the fork will be inevitable.

(6) Block refusal penalty

Using the PoA consensus algorithm, there is no node punishment mechanism.

(7) Block time offset attack

The time deviation of the current block must not exceed `maxTimeFutureBlocks`, which is set to 30s on the mainnet.

- core/blockchain.go

```
func (bc *BlockChain) addFutureBlock(block *types.Block) error {
    max := uint64(time.Now().Unix() + maxTimeFutureBlocks)
    if block.Time() > max {
        return fmt.Errorf("future block timestamp %v > allowed %v", block.Time(), max)
    }
    if block.Difficulty().Cmp(common.Big0) == 0 {
        // Never add PoS blocks into the future queue
        return nil
    }
    bc.futureBlocks.Add(block.Hash(), block)
    return nil
}
```

(8) Consensus algorithm potential risk assessment

- consensus/clique/clique.go

  This file contains the implementation of the Clique PoA consensus algorithm. Key Points of the Clique PoA Algorithm:

```
Signers: Authorized accounts that can sign and create new blocks.
Extra Data: The header extra data field is split into the vanity section and the seal
section.
Nonces: Special nonce values are used to vote on adding or removing signers.
Difficulty: The difficulty field is used to encode whether the signer is in-turn or
out-of-turn.
Epochs: Periodic checkpoints reset the pending votes and signer set.
Signing and Sealing: Blocks are signed by authorized signers and are verified by other
nodes.
```

Utilizing the PoA consensus algorithm, a small number of nodes hold the rights to produce blocks, with fewer nodes, there is a higher risk of centralization.

3.3 Vulnerability Information

The following is the status of the vulnerabilities found in this audit:

NO	Title	Category	Level	Status

N1	False top-up risk of exchanges	Account and Transaction Security Audit	Information	Acknowledged
N2	The centralization risk of the PoA consensus algorithm	Consensus Security Audit	Information	Acknowledged
N3	Errors unhandled	SAST	Low	Acknowledged
N4	Implicit memory aliasing in for loop.	SAST	Low	Acknowledged
N5	Potential Slowloris Attack because ReadHeaderTimeout is not configured in the http.Server	SAST	Low	Acknowledged

4 Findings

4.1 Vulnerability Summary

[N1] [Information] False top-up risk of exchanges

Category: Account and Transaction Security Audit

Content

In a complex transaction, even if an internal cross-contract call fails, the entire transaction may still be marked as successful. This feature may lead to potential security risks like 'false top-up'

Solution

Exchanges need to pay attention to the risk of fake top-up and strictly check deposit transactions.

Status

Acknowledged

[N2] [Information] The centralization risk of the PoA consensus algorithm

Category: Consensus Security Audit

Content

1.Centralized control: Only a few pre-selected nodes are responsible for block generation and verification, which may lead to excessive centralized control over network decision-making and operations.

2.Trust risk: Network security relies on the trustworthiness of authoritative nodes. If some nodes have security vulnerabilities or are attacked, the entire network may face serious security threats.

3.Single point of failure: If an authoritative node fails or is attacked, it may have a significant impact on block generation, leading to service interruptions or data tampering risks.

4.Insider threats: The centralized management authority of nodes may lead to internal collusion, abuse of power, and other issues, making the network no longer fully decentralized and transparent.

Solution

Introducing hybrid consensus solutions that incorporate other consensus mechanisms (such as PoS, PBFT, etc.), further dispersing risks.

Status

Acknowledged

[N3] [Low] Errors unhandled

Category: SAST

Content

The following code does not process the returned error information during the calling process, which may cause the program to not terminate in time when an error occurs, resulting in a logical error.

Provide a separate document for the discovered part for reference.

Solution

Check the return value of a function call.

Status

Acknowledged

[N4] [Low] Implicit memory aliasing in for loop.

Category: SAST

Content

- abcore/p2p/simulations/http.go:437

```
  436:                  for _, node := range snap.Nodes {
> 437:                          event := NewEvent(&node.Node)
  438:                          if err := writeEvent(event); err != nil {
```

- abcore/internal/ethapi/transaction_args.go:353

```
352:    for i, c := range args.Commitments {
> 353:                  hashes[i] = kzg4844.CalcBlobHashV1(hasher, &c)
  354:          }
```

- abcore/cmd/geth/accountcmd.go:214

```
    213:                 for _, account := range wallet.Accounts() {
  > 214:                         fmt.Printf("Account #%d: {%x} %s\n", index,
account.Address, &account.URL)
    215:                         index++
```

Solution

By creating a copy of the variable, you ensure that each iteration receives an independent copy of the current variable, thus avoiding implicit memory aliasing problems.

Status

Acknowledged

[N5] [Low] Potential Slowloris Attack because ReadHeaderTimeout is not configured in the http.Server

Category: SAST

Content

- abcore/node/rpcstack.go:141

```
    140:         // Initialize the server.
  > 141:         h.server = &http.Server{Handler: h}
    142:         if h.timeouts != (rpc.HTTPTimeouts{}) {
```

Solution

Status

Acknowledged

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002503240001	SlowMist Security Team	2025.03.19 - 2025.03.24	Low Risk

Summary conclusion: The SlowMist security team use a manual and SlowMist team's analysis tool to audit the project, during the audit work we found 3 low risk. All the findings were fixed.

6 Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.

SLOWMIST

Official Website
www.slowmist.com

E-mail
team@slowmist.com

Twitter
@SlowMist_Team

Github

https://github.com/slowmist