Blockchain Security Audit Report

Table Of Contents

1 Executive Summary

On 2024.12.10, the SlowMist security team received the team's security audit application for go-u2u, 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	SAST	Some Risks
2	Cryptographic Security Audit	Some Risks
3	Account and Transaction Security Audit	Some Risks
4	RPC Security Audit	Passed
5	P2P Security Audit	Passed
6	Consensus Security Audit	Passed

3 Project Overview

3.1 Project Introduction

Implementation of U2U Network in Golang.

3.2 Coverage

Target Code and Revision:

https://github.com/unicornultrafoundation/go-u2u

commit: 082a048e5e0aa43e88bbf6be5b163fb117442f88

SAST

Scan *.go with Gosec, found 589 issues

Cryptographic Security Audit

(1) Insufficient entropy of private key random numbers audit

The main point is to ensure that there is sufficient randomness and unpredictability when generating the private key of the cryptocurrency wallet.

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

The main task is to check whether there is any precision loss during the conversion process from the seed to the address of the private key.

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 represents an ECDSA 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

The main check is to evaluate the theoretical reliability of symmetric encryption algorithms.

The following cryptographic algorithms are used in U2U:

```
AES-128/192/256: for account private key encryption storage
CTR Mode: AES operating mode
```

(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

```
ECDSA signature
```

(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.

U2U'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/hashing.go

```
// hasherPool holds LegacyKeccak256 hashers for rlpHash.
var hasherPool = sync.Pool{
   New: func() interface{} { return sha3.NewLegacyKeccak256() },
}


//...


// 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 U2U is consistent with that of Ethereum and there are no security issues.

(10) secp256k1 k-value randomness security

The main purpose is to examine the randomness and security of the k value in the secp256k1 elliptic curve encryption algorithm.

The `k` value of `Secp256k1` in u2u is generated based on the RFC-6979 standard. It is not random but kept confidential.

- crypto/signature_cgo.go

```
// Sign calculates an ECDSA signature.
//
// This function is susceptible to chosen plaintext attacks that can leak
// information about the private key that is used for signing. Callers must
// be aware that the given digest cannot be chosen by an adversery. Common
// solution is to hash any input before calculating the signature.
//
// The produced signature is in the [R || S || V] format where V is 0 or 1.
func Sign(digestHash []byte, prv *ecdsa.PrivateKey) (sig []byte, err error) {
    if len(digestHash) != DigestLength {
        return nil, fmt.Errorf("hash is required to be exactly %d bytes (%d)",
DigestLength, len(digestHash))
    }
    seckey := math.PaddedBigBytes(prv.D, prv.Params().BitSize/8)
    defer zeroBytes(seckey)
    return secp256k1.Sign(digestHash, seckey)
}
```

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

The main purpose is to examine the influence of r-value reuse in the secp256k1 elliptic curve encryption algorithm on private key extraction attacks.

The security of the r value stems from the random security of the k value. Since the k value is secure, then the r value is also secure here.

- secp256k1.c

```
int secp256k1_ecdsa_sign(const secp256k1_context* ctx, secp256k1_ecdsa_signature
*signature, const unsigned char *msg32, const unsigned char *seckey,
secp256k1_nonce_function noncefp, const void* noncedata) {
   secp256k1_scalar r, s;
   secp256k1_scalar sec, non, msg;
   int ret = 0;
   int overflow = 0;
   VERIFY_CHECK(ctx != NULL);
   ARG_CHECK(secp256k1_ecmult_gen_context_is_built(&ctx->ecmult_gen_ctx));
   ARG_CHECK(msg32 != NULL);
   ARG_CHECK(signature != NULL);
   ARG_CHECK(seckey != NULL);
   if (noncefp == NULL) {
       noncefp = secp256k1_nonce_function_default;
   }

   secp256k1_scalar_set_b32(&sec, seckey, &overflow);
   /* Fail if the secret key is invalid. */
   if (!overflow && !secp256k1_scalar_is_zero(&sec)) {
       unsigned char nonce32[32];
       unsigned int count = 0;
       secp256k1_scalar_set_b32(&msg, msg32, NULL);
       while (1) {
           ret = noncefp(nonce32, msg32, seckey, NULL, (void*)noncedata, count);
           if (!ret) {
               break;
           }
           secp256k1_scalar_set_b32(&non, nonce32, &overflow);
           if (!overflow && !secp256k1_scalar_is_zero(&non)) {
               if (secp256k1_ecdsa_sig_sign(&ctx->ecmult_gen_ctx, &r, &s, &sec, &msg,
&non, NULL)) {
                   break;
               }
           }
           count++;
       }
       memset(nonce32, 0, 32);
       secp256k1_scalar_clear(&msg);
```

```
        secp256k1_scalar_clear(&non);
        secp256k1_scalar_clear(&sec);
    }
    if (ret) {
        secp256k1_ecdsa_signature_save(signature, &r, &s);
    } else {
        memset(signature, 0, sizeof(*signature));
    }
    return ret;
}
```

(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.

There is an extensibility problem in secp256k1. By changing the s value in the signature to s - N or N - s, the signature result can be changed. This led to the attack on the mt.gox in the early version. However, this only affects the security of off-chain applications and does not affect the security of U2U itself.

(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.

Ed25519 is a digital signature algorithm based on elliptic curves, which is widely used in the fields of blockchain and cryptocurrencies. The private key extraction attack refers to an attacker's attempt to recover the signer's private key from known signatures by exploiting vulnerabilities or weaknesses in the system. If successful, the attacker can forge valid signatures, thereby performing malicious operations such as tampering with transactions and impersonating identities.

U2U does not use the library of ed25519 with security issues.

(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.

U2U has not used the Schnorr 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).

U2U 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 U2U 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.

U2U 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 U2U, 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 U2U is designed to be fully compatible with the Ethereum Virtual Machine (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 U2U 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 Ethereum 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.

- ethapi/api.go

```
func (s *PublicTransactionPoolAPI) 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
}
```

But in the configuration file of docker, the default open modules of open api not contain the insecurity functions.

- deployment/join-mainnet/docker-compose.yml#L24

```
     --http.api="eth,debug,net,web3,txpool,dag"
```

(2) RPC port identifiability

The main check is to assess the identifiability of RPC ports in the blockchain system to determine whether the system is vulnerable to attacks targeting RPC ports.

Port 18545/18546 is used by default, and can also be specified with a command line parameter, or through a configuration file.

- cmd/u2u/launcher/defaults.go

```
    DefaultHTTPPort = 18545 // Default TCP port for the HTTP RPC server
    DefaultWSPort   = 18546 // Default TCP port for the websocket RPC server
```

(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-mainnet.u2u.xyz' \
  -H 'content-type: application/json' \
  --data-raw '{
 "id":101,
 "jsonrpc":"2.0",
 "method":"eth_getTransactionByHash",
 "params":["0xfac58116b8483cb3a2e40a545f4f9b7e8caea6fb316f2c9a4c077f545bd8079b"]
}'
```

`Access-Control-Allow-Origin: *` was not found in the return header, there is no cross-domain request issue.

(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:

```
    data = '{"' + '}' * 0x101000 + '":' + '{"x":' * 0x10000 + '"}'
    print(post(posturl, data))
```

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

```
{"jsonrpc":"2.0","id":null,"error":{"code":-32700,"message":"parse error"}}
```

(5) RPC database injection

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

U2U uses LevelDB by default as the database for its blockchain data. LevelDB is a key-value pair repository and does not operate using SQL statements, so there is no database injection problem.

(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

```
    // 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/rlpx/rlpx.go

```
// Handshake performs the handshake. This must be called before any data is written
// or read from the connection.
func (c *Conn) Handshake(prv *ecdsa.PrivateKey) (*ecdsa.PublicKey, error) {
   var (
       sec Secrets
       err error
       h   handshakeState
   )
   if c.dialDest != nil {
       sec, err = h.runInitiator(c.conn, prv, c.dialDest)
   } else {
       sec, err = h.runRecipient(c.conn, prv)
   }
   if err != nil {
       return nil, err
   }
   c.InitWithSecrets(sec)
   c.session.rbuf = h.rbuf
   c.session.wbuf = h.wbuf
   return sec.remote, err
}
```

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

- p2p/rlpx/rlpx.go

```
func (c *Conn) InitWithSecrets(sec Secrets) {
   //...
   // 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) {
    // ...
    // 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.

U2U utilizes aBFT and DPoS mechanism to enhanced security, improve scalability and efficient validation.

There are only 13 validator nodes in the current mainnet, which is too small and poses a certain centralization risk.

Reference: https://staking.u2u.xyz/validators

(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 !netutil.IsLAN(remoteIP) && srv.inboundHistory.contains(remoteIP.String()) {
       return fmt.Errorf("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

```
type Config struct {
   //...
   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
   //...
   }
}
```

(7) P2P Alien attack

The Alien attack vulnerability was first discovered by the SlowMist team and is also known as peer pool pollution. It refers to an attack method that induces mutual invasion and pollution among similar chain nodes. The main reason for this vulnerability is that similar chain systems fail to identify dissimilar nodes in the communication protocol.

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 {
         //...
       ntab, err := discover.ListenV4(conn, srv.localnode, cfg)
       //...
       if sconn != nil {
           srv.DiscV5, err = discover.ListenV5(sconn, srv.localnode, cfg)
       } else {
           srv.DiscV5, err = discover.ListenV5(conn, srv.localnode, cfg)
       }
       //...
   }
   return nil
}
```

- p2p/discover/v4_udp.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"`
    }
```

- p2p/discover/v5wire/msg.go

```
    // PING is sent during liveness checks.
    Ping struct {
        ReqID  []byte
        ENRSeq uint64
    }


    // PONG is the reply to PING.
    Pong struct {
        ReqID  []byte
        ENRSeq uint64
        ToIP   net.IP // These fields should mirror the UDP envelope address of the
ping
        ToPort uint16 // packet, which provides a way to discover the the external
address (after NAT).
    }


    // FINDNODE is a query for nodes in the given bucket.
    Findnode struct {
        ReqID     []byte
        Distances []uint
    }
```

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 5050 is used by default, and can also be specified with the command line parameter --port, or through a configuration file.

TCP port is used for P2P network communication.

UDP port is used for node discovery (default is the same as TCP port, can be specified via the discport parameter).

- cmd/u2u/launcher/defaults.go

```
    DefaultP2PPort  = 5050
```

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 U2U project.

(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 U2U, through the EIP-1559 mechanism, 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.

Due to the use of aBFT and DPoS algorithms, the block production order mainly depends on the sorting of the validators' weights and has nothing to do with the block hash, so there is no grinding attack problem.

(4) PoS/BFT final confirmation conditions

The `MaxEpochDuration` of one epoch is 7 minutes, and usually 2 epochs are required to make the block irreversible.

- u2u/rules.go

```
func DefaultEpochsRules() EpochsRules {
   return EpochsRules{
       MaxEpochGas:      300000000,
       MaxEpochDuration: native.Timestamp(7 * time.Minute),
   }
}
```

(5) PoS/BFT double-signing penalty

The system will detect and handle the double-signing behavior of the verifier and mark the double-signing verifier as a cheater.

- gossip/c_block_callbacks.go

```
func consensusCallbackBeginBlockFn(
        parallelTasks *workers.Workers,
        wg *sync.WaitGroup,
        blockBusyFlag *uint32,
        store *Store,
        blockProc BlockProc,
        txIndex bool,
        feed *ServiceFeed,
        emitters *[]*emitter.Emitter,
        verWatcher *verwatcher.VerWarcher,
        bootstrapping *bool,
) utypes.BeginBlockFn {
                //...

                reportCheater := func(reporter, cheater idx.ValidatorID) {
```

```
                                mpsCheatersMap[cheater] = struct{}{}
                        }
                        //...
                                        if e.AnyMisbehaviourProofs() {
                                                mps :=
store.GetEventPayload(e.ID()).MisbehaviourProofs()
                                                for _, mp := range mps {
                                                        // self-contained parts of proofs
are already checked by the checkers
                                                        if proof := mp.BlockVoteDoublesign;
proof != nil {
                                                                reportCheater(e.Creator(),
proof.Pair[0].Signed.Locator.Creator)
                                                        }
                                                        if proof := mp.EpochVoteDoublesign;
proof != nil {
                                                                reportCheater(e.Creator(),
proof.Pair[0].Signed.Locator.Creator)
                                                        }
                                                        if proof := mp.EventsDoublesign;
proof != nil {
                                                                reportCheater(e.Creator(),
proof.Pair[0].Locator.Creator)
                                                        }
                        //...
}
```

The penalty code for nodes is in the u2u-sfc project.

- contracts/sfc/SFCLib.sol#L214

```
    function getSlashingPenalty(uint256 amount, bool isCheater, uint256 refundRatio)
internal pure returns (uint256 penalty) {
        if (!isCheater || refundRatio >= Decimal.unit()) {
            return 0;
        }
        // round penalty upwards (ceiling) to prevent dust amount attacks
        penalty = amount.mul(Decimal.unit() -
refundRatio).div(Decimal.unit()).add(1);
        if (penalty > amount) {
            return amount;
        }
        return penalty;
    }
```

(6) PoS/BFT block refusal penalty

When the Epoch switches, cheaters are further dealt with, such as adjusting their stake or removing them from the validator list.

```
// SealEpoch is called after pre-internal transactions are executed
func (s *U2UEpochsSealer) SealEpoch() (iblockproc.BlockState, iblockproc.EpochState) {
        // Select new validators
        //...
}
```

(7) Block time offset attack

A block time manipulation attack occurs when a malicious miner deliberately timestamps a block with an incorrect time during its production. Blockchain systems typically tolerate a certain degree of time deviation, but if there are no limits on the permissible range, the production of subsequent blocks may not occur within the expected time frame.

The time deviation of the current block must not exceed `MaxEmptyBlockSkipPeriod`, which is set to 1s on the mainnet.

- gossip/c_block_callbacks.go

```
func consensusCallbackBeginBlockFn(
//...
            skipBlock = skipBlock || (emptyBlock && blockCtx.Time <
bs.LastBlock.Time+es.Rules.Blocks.MaxEmptyBlockSkipPeriod)
//...
```

(8) Consensus algorithm potential risk assessment

Double signing may be used by malicious validators for long-range attacks, which is a serious consensus attack. The cheating validators should be fined.

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	secp256k1 signature malleability risk	Cryptographic Security Audit	Information	Acknowledged
N3	Errors unhandled	SAST	Low	Acknowledged
N4	Errors unhandled	SAST	Medium	Fixed
N5	Implicit memory aliasing in for loop	SAST	Medium	Fixed
N6	Use of unsafe calls	SAST	Information	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] secp256k1 signature malleability risk

Category: Cryptographic Security Audit

Content

The `secp256k1` signature algorithm has malleability issues. Exchanges or other users should check whether the signature has been tampered with when using it.

Solution

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.

- valkeystore/encryption/io.go

```
	}
	f.Close()
	return f.Name(), nil
```

- valkeystore/encryption/io.go

```
		f.Close()
		os.Remove(f.Name())
		return "", err
```

- valkeystore/encryption/io.go

```
	if _, err := f.Write(content); err != nil {
		f.Close()
		os.Remove(f.Name())
```

- trie/sync.go

```
		if len(requests) == 0 && req.deps == 0 {
			s.commit(req)
		} else {
```

- trie/sync.go

```
        req.data = result.Data
        s.commit(req)
    }
```

- trie/proof.go

```
    for index, key := range keys {
        tr.TryUpdate(key, values[index])
    }
```

- trie/proof.go

```
        for index, key := range keys {
            tr.TryUpdate(key, values[index])
        }
```

- trie/proof.go

```
            }
            proofDb.Put(hash, enc)
        }
```

- trie/database.go

```
        case <-ticker.C:
            db.saveCache(dir, 1)
        case <-stopCh:
```

- trie/database.go

```
        db.lock.Lock()
        batch.Replay(uncacher)
        db.lock.Unlock()
```

- trie/database.go

```

    batch.Replay(uncacher)

```

- trie/committer.go

```
                        if n.Children[16] != nil {
                                c.onleaf(nil, nil, n.Children[16].(valueNode), hash)
                        }
```

- trie/committer.go

```
                        if child, ok := n.Val.(valueNode); ok {
                                c.onleaf(nil, nil, child, hash)
                        }
```

- topicsdb/search_parallel.go

```
        }
        onMatched(nil)
}
```

- rpc/websocket.go

```
                        wc.jsonCodec.encMu.Lock()
                        wc.conn.SetWriteDeadline(time.Now().Add(wsPingWriteTimeout))
                        wc.conn.WriteMessage(websocket.PingMessage, nil)
                        wc.jsonCodec.encMu.Unlock()
```

- rpc/subscription.go

```
        if unsubscribe {
                sub.requestUnsubscribe()
        }
```

- rpc/subscription.go

```
                id := make([]byte, 16)
                rng.Read(id)
                return encodeID(id)
```

- rpc/server.go

```
                if err != io.EOF {
                        codec.writeJSON(ctx, errorMessage(&invalidMessageError{"parse error"}))
                }
```

- rpc/server.go

```
        rpcService := &RPCService{server}
        server.RegisterName(MetadataApi, rpcService)
        return server
```

- rpc/ipc_unix.go

```
        }
        os.Chmod(endpoint, 0600)
        return l, nil
```

- rpc/ipc_unix.go

```
        }
        os.Remove(endpoint)
        l, err := net.Listen("unix", endpoint)
```

- rpc/handler.go

```
                for _, n := range cp.notifiers {
                        n.activate()
                }
```

- rpc/handler.go

```
                if answer != nil {
                        h.conn.writeJSON(cp.ctx, answer)
                }
```

- rpc/handler.go

```
                for _, n := range cp.notifiers {
                        n.activate()
```

```
		}
```

- rpc/handler.go

```
		if len(answers) > 0 {
			h.conn.writeJSON(cp.ctx, answers)
		}
```

- rpc/handler.go

```
		h.startCallProc(func(cp *callProc) {
			h.conn.writeJSON(cp.ctx, errorMessage(&invalidRequestError{"empty batch"}))
		})
```

- rpc/client.go

```
		if _, ok := err.(*json.SyntaxError); ok {
			codec.writeJSON(context.Background(), errorMessage(&parseError{err.Error()}))
		}
```

- p2p/transport.go

```
		var reason [1]DiscReason
		rlp.Decode(msg.Payload, &reason)
		return nil, reason[0]
```

- 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)
```

- p2p/transport.go

```
	}
	t.conn.Close()
}
```

- p2p/transport.go

```
				rlp.Encode(&t.wbuf, []DiscReason{r})
				t.conn.Write(discMsg, t.wbuf.Bytes())
			}
```

- p2p/transport.go

```
				t.wbuf.Reset()
				rlp.Encode(&t.wbuf, []DiscReason{r})
				t.conn.Write(discMsg, t.wbuf.Bytes())
```

- p2p/transport.go

```
	// Write the message.
	t.conn.SetWriteDeadline(time.Now().Add(frameWriteTimeout))
	size, err := t.conn.Write(msg.Code, t.wbuf.Bytes())
```

- p2p/transport.go

```
	var msg Msg
	t.conn.SetReadDeadline(time.Now().Add(frameReadTimeout))
	code, data, wireSize, err := t.conn.Read()
```

- p2p/server.go

```
		go func() {
			srv.SetupConn(fd, inboundConn, nil)
			slots <- struct{}{}
```

- p2p/server.go

```
			srv.log.Debug("Rejected inbound connection", "addr", fd.RemoteAddr(), "err", err)
			fd.Close()
			slots <- struct{}{}
```

- p2p/server.go

```
        // this unblocks listener Accept
        srv.listener.Close()
    }
```

- p2p/peer.go

```
        // check errors because, the connection will be closed after it.
        rlp.Decode(msg.Payload, &reason)
        return reason[0]
```

- p2p/peer.go

```
    case msg.Code == pingMsg:
        msg.Discard()
        go SendItems(p.rw, pongMsg)
```

- p2p/peer.go

```
    if p.testPipe != nil {
        p.testPipe.Close()
    }
```

- p2p/nodestate/nodestate.go

```
    t.timer = ns.clock.AfterFunc(timeout, func() {
        ns.SetState(n, Flags{}, Flags{mask: t.mask, setup: ns.setup}, 0)
    })
```

- p2p/nodestate/nodestate.go

```
func (ns *NodeStateMachine) deleteNode(id enode.ID) {
    ns.db.Delete(append(ns.dbNodeKey, id[:]...))
}
```

- p2p/nat/natupnp.go

```
    lifetimeS := uint32(lifetime / time.Second)
    n.DeleteMapping(protocol, extport, intport)

```

- p2p/nat/natupnp.go67)

```
63:     var ipString string
64:     n.withRateLimit(func() error {
65:             ipString, err = n.client.GetExternalIPAddress()
66:             return err
67:     })
68:
```

- p2p/nat/natupnp.go58)

```
54:     var err error
55:     n.withRateLimit(func() error {
56:             _, ok, err = n.client.GetNATRSIPStatus()
57:             return err
58:     })
59:     return err == nil && ok
```

- p2p/nat/nat.go

```
104:            log.Debug("Deleting port mapping")
105:            m.DeleteMapping(protocol, extport, intport)
106:    }()
```

- p2p/enr/enr.go

```
269:    var id ID
270:    r.Load(&id)
271:    return string(id)
```

- p2p/enode/urlv4.go

```
175:    n.Load(&scheme)
176:    n.Load((*Secp256k1)(&key))
177:    switch {
```

- p2p/enode/urlv4.go

```
174:    )
175:    n.Load(&scheme)
```

```
	n.Load((*Secp256k1)(&key))
```

- p2p/enode/nodedb.go

```
	close(db.quit)
	db.lvl.Close()
}
```

- p2p/enode/nodedb.go

```
		ctr := id[0]
		rand.Read(id[:])
		id[0] = ctr + id[0]%16
```

- p2p/enode/nodedb.go

```
func (db *DB) storeLocalSeq(id ID, n uint64) {
	db.storeUint64(localItemKey(id, dbLocalSeq), n)
}
```

- p2p/enode/nodedb.go

```
	for it.Next() {
		db.Delete(it.Key(), nil)
	}
```

- p2p/enode/nodedb.go

```
		if !bytes.Equal(blob, currentVer) {
			db.Close()
			if err = os.RemoveAll(path); err != nil {
```

- p2p/enode/nodedb.go

```
		if err := db.Put([]byte(dbVersionKey), currentVer, nil); err != nil {
			db.Close()
			return nil, err
```

- p2p/enode/node.go

```
    var port enr.TCP
    n.Load(&port)
    return int(port)
```

- p2p/enode/node.go

```
    var port enr.UDP
    n.Load(&port)
    return int(port)
```

- p2p/enode/idscheme.go

```
    var id ID
    r.Load(enr.WithEntry("nulladdr", &id))
    return id[:]
```

- p2p/enode/idscheme.go

```
    h := sha3.NewLegacyKeccak256()
    rlp.Encode(h, r.AppendElements(nil))
    if !crypto.VerifySignature(entry, h.Sum(nil), sig) {
```

- p2p/enode/idscheme.go

```
    h := sha3.NewLegacyKeccak256()
    rlp.Encode(h, cpy.AppendElements(nil))
    sig, err := crypto.Sign(h.Sum(nil), privkey)
```

- p2p/dnsdisc/tree.go

```
    h := sha3.NewLegacyKeccak256()
    io.WriteString(h, e.String())
    return b32format.EncodeToString(h.Sum(nil)[:16])
```

- p2p/discover/v5_udp.go

```
    resp := &v5wire.TalkResponse{ReqID: p.ReqID, Message: response}
    t.sendResponse(fromID, fromAddr, resp)
}
```

- p2p/discover/v5_udp.go

```
795:    for _, resp := range packNodes(p.ReqID, nodes) {
796:            t.sendResponse(fromID, fromAddr, resp)
797:    }
```

- p2p/discover/v5_udp.go789)

```
783:    }
784:    t.sendResponse(fromID, fromAddr, &v5wire.Pong{
785:            ReqID:  p.ReqID,
786:            ToIP:   remoteIP,
787:            ToPort: uint16(fromAddr.Port),
788:            ENRSeq: t.localNode.Node().Seq(),
789:    })
790: }
```

- p2p/discover/v5_udp.go

```
738:    }
739:    t.sendResponse(fromID, fromAddr, challenge)
740: }
```

- p2p/discover/v5_udp.go

```
529:            case p := <-t.packetInCh:
530:                    t.handlePacket(p.Data, p.Addr)
531:                    // Arm next read.
```

- p2p/discover/v5_udp.go

```
188:            t.cancelCloseCtx()
189:            t.conn.Close()
190:            t.wg.Wait()
```

- p2p/discover/v4_udp.go793)

```
789: func (t *UDPv4) handleENRRequest(h *packetHandlerV4, from *net.UDPAddr, fromID
enode.ID, mac []byte) {
790:    t.send(from, fromID, &v4wire.ENRResponse{
791:            ReplyTok: mac,
		Record:   *t.localNode.Node().Record(),
	})
}
```

- p2p/discover/v4_udp.go

```
	if len(p.Nodes) > 0 || !sent {
		t.send(from, fromID, &p)
	}
```

- p2p/discover/v4_udp.go

```
		if len(p.Nodes) == v4wire.MaxNeighbors {
			t.send(from, fromID, &p)
			p.Nodes = p.Nodes[:0]
```

- p2p/discover/v4_udp.go

```
	t.localNode.UDPEndpointStatement(from, &net.UDPAddr{IP: req.To.IP, Port: int(req.To.UDP)})
	t.db.UpdateLastPongReceived(fromID, from.IP, time.Now())
	return nil
```

- p2p/discover/v4_udp.go

```
	// Update node database and endpoint predictor.
	t.db.UpdateLastPingReceived(n.ID(), from.IP, time.Now())
	t.localNode.UDPEndpointStatement(from, &net.UDPAddr{IP: req.To.IP, Port: int(req.To.UDP)})
```

- p2p/discover/v4_udp.go674)

```
	// Reply.
	t.send(from, fromID, &v4wire.Pong{
		To:         v4wire.NewEndpoint(from, req.From.TCP),
		ReplyTok:   mac,
		Expiration: uint64(time.Now().Add(expiration).Unix()),
		ENRSeq:     t.localNode.Node().Seq(),
	})

```

- p2p/discover/v4_udp.go

```
    // Send the packet and wait for the reply.
    t.write(addr, n.ID(), req.Name(), packet)
    if err := <-rm.errc; err != nil {
```

- p2p/discover/v4_udp.go329)

```
325:    })
326:    t.send(toaddr, toid, &v4wire.Findnode{
327:            Target:     target,
328:            Expiration: uint64(time.Now().Add(expiration).Unix()),
329:    })
330:    // Ensure that callers don't see a timeout if the node actually responded.
Since
```

- p2p/discover/v4_udp.go

```
    t.localNode.UDPContact(toaddr)
    t.write(toaddr, toid, req.Name(), packet)
    return rm
```

- p2p/discover/v4_udp.go

```
            t.cancelCloseCtx()
            t.conn.Close()
            t.wg.Wait()
```

- p2p/discover/table.go

```
                    if n.livenessChecks > 0 && now.Sub(n.addedAt) >= seedMinTableTime {
                            tab.db.UpdateNode(unwrapNode(n))
                    }
```

- p2p/discover/ntp.go

```
            // Retrieve the reply and calculate the elapsed time
            conn.SetDeadline(time.Now().Add(5 * time.Second))

```

- p2p/discover/lookup.go

```
                // Reset failure counter because it counts _consecutive_ failures.
                it.tab.db.UpdateFindFails(n.ID(), n.IP(), 0)
        }
```

- p2p/discover/lookup.go

```
                fails++
                it.tab.db.UpdateFindFails(n.ID(), n.IP(), fails)
                // Remove the node from the local table if it fails to return
anything useful too
```

- p2p/dial.go

```
                        if t.resolve(d) {
                                t.dial(d, t.dest)
                        }
```

- node/rpcstack.go

```
        h.server.Shutdown(context.Background())
        h.listener.Close()
        h.log.Info("HTTP server stopped", "endpoint", h.listener.Addr())
```

- node/rpcstack.go

```
        }
        h.server.Shutdown(context.Background())
        h.listener.Close()
```

- node/node.go

```
        n.ws.stop()
        n.ipc.stop()
        n.stopInProc()
```

- node/node.go

```
            n.stopServices(started)
            n.doClose(nil)
    }
```

- node/node.go

```
    if err != nil {
            n.stopServices(started)
            n.doClose(nil)
```

- node/node.go

```
    if err != nil {
            n.doClose(nil)
            return err
```

- node/defaults.go

```
    names, _ := f.Readdir(1)
    f.Close()
    return len(names) > 0
```

- node/api.go

```
                    case event := <-events:
                            notifier.Notify(rpcSub.ID, event)
                    case <-sub.Err():
```

- log/logger.go146)

```
func (l *logger) write(msg string, lvl Lvl, ctx []interface{}, skip int) {
    l.h.Log(&Record{
            Time: time.Now(),
            Lvl:  lvl,
            Msg:  msg,
            Ctx:  newContext(l.ctx, ctx),
            Call: stack.Caller(skip),
            KeyNames: RecordKeyNames{
                    Time: timeKey,
                    Msg:  msgKey,
                    Lvl:  lvlKey,
            Ctx:  ctxKey,
        },
    })
}
```

- log/handler.go

```
            // what to do about failures?
            h.Log(r)
        }
```

- light/nodeset.go

```
    for _, node := range n {
        db.Put(crypto.Keccak256(node), node)
    }
```

- light/nodeset.go

```
    for key, value := range db.nodes {
        target.Put([]byte(key), value)
    }
```

- internal/jsre/jsre.go

```
func (re *JSRE) Set(ns string, v interface{}) (err error) {
    re.Do(func(vm *goja.Runtime) { vm.Set(ns, v) })
    return err
```

- internal/jsre/jsre.go

```
            }
            call(goja.Null(), timer.call.Arguments...)

```

- internal/jsre/jsre.go

```
    re.vm.Set("clearTimeout", clearTimeout)
    re.vm.Set("clearInterval", clearTimeout)

```

- internal/jsre/jsre.go

```
165:    }`)
166:    re.vm.Set("clearTimeout", clearTimeout)
167:    re.vm.Set("clearInterval", clearTimeout)
```

- internal/jsre/jsre.go165)

```
159:    }`)
160:    re.vm.RunString(`var setInterval = function(args) {
161:            if (arguments.length < 1) {
162:                    throw TypeError("Failed to execute 'setInterval': 1 argument required, but only 0 present.");
163:            }
164:            return _setInterval.apply(this, arguments);
165:    }`)
166:    re.vm.Set("clearTimeout", clearTimeout)
```

- internal/jsre/jsre.go159)

```
153:    re.vm.Set("_setInterval", setInterval)
154:    re.vm.RunString(`var setTimeout = function(args) {
155:            if (arguments.length < 1) {
156:                    throw TypeError("Failed to execute 'setTimeout': 1 argument required, but only 0 present.");
157:            }
158:            return _setTimeout.apply(this, arguments);
159:    }`)
160:    re.vm.RunString(`var setInterval = function(args) {
```

- internal/jsre/jsre.go

```
152:    re.vm.Set("_setTimeout", setTimeout)
153:    re.vm.Set("_setInterval", setInterval)
154:    re.vm.RunString(`var setTimeout = function(args) {
```

- internal/jsre/jsre.go

```
151:    }
152:    re.vm.Set("_setTimeout", setTimeout)
153:    re.vm.Set("_setInterval", setInterval)
```

- internal/jsre/jsre.go

```
	re.Set("loadScript", MakeCallback(re.vm, re.loadScript))
	re.Set("inspect", re.prettyPrintJS)
	return re
```

- internal/jsre/jsre.go

```
	go re.runEventLoop()
	re.Set("loadScript", MakeCallback(re.vm, re.loadScript))
	re.Set("inspect", re.prettyPrintJS)
```

- internal/debug/trace.go

```
	log.Info("Done writing Go trace", "dump", h.traceFile)
	h.traceW.Close()
	h.traceW = nil
```

- internal/debug/trace.go

```
	if err := trace.Start(f); err != nil {
		f.Close()
		return err
```

- internal/debug/flags.go

```
	Handler.StopCPUProfile()
	Handler.StopGoTrace()
}
```

- internal/debug/flags.go

```
func Exit() {
	Handler.StopCPUProfile()
	Handler.StopGoTrace()
```

- internal/debug/flags.go

```
	}
	glogger.BacktraceAt(backtrace)
```

```

```

- internal/debug/flags.go

```
    vmodule := ctx.GlobalString(vmoduleFlag.Name)
    glogger.Vmodule(vmodule)

```

- internal/debug/api.go

```
    buf := new(bytes.Buffer)
    pprof.Lookup("goroutine").WriteTo(buf, 2)
    return buf.String()
```

- internal/debug/api.go

```
    time.Sleep(time.Duration(nsec) * time.Second)
    h.StopGoTrace()
    return nil
```

- internal/debug/api.go

```
    log.Info("Done writing CPU profile", "dump", h.cpuFile)
    h.cpuW.Close()
    h.cpuW = nil
```

- internal/debug/api.go

```
    if err := pprof.StartCPUProfile(f); err != nil {
            f.Close()
            return err
```

- internal/debug/api.go

```
    time.Sleep(time.Duration(nsec) * time.Second)
    h.StopCPUProfile()
    return nil
```

- internal/build/util.go

```
    }
    stdin.Close()
    return sftp.Wait()
```

- internal/build/download.go

```
    _, err = io.Copy(dst, resp.Body)
    dst.Close()
    if err != nil {
```

- internal/build/archive.go

```
        file.Close()
        os.Remove(target)
        return err
```

- internal/build/archive.go

```
    if _, err := io.Copy(file, data); err != nil {
        file.Close()
        os.Remove(target)
```

- internal/build/archive.go

```
        err = extractFile(zf.Name, zf.Mode(), data, dest)
        data.Close()
        if err != nil {
```

- internal/build/archive.go

```
        if err != nil {
            os.Remove(name)
        }
```

- internal/build/archive.go

```
    defer func() {
        archfd.Close()
        // Remove the half-written archive on failure.
```

- integration/assembly.go

```
            gdb.Close()
            cdb.Close()
            dbs.Close()
        }
```

- gossip/protocols/snap/snapstream/snapleecher/peer.go

```
    delete(ps.peers, id)
    ps.rates.Untrack(id)
    ps.lock.Unlock()
```

- gossip/protocols/snap/api.go

```
            case status := <-statuses:
                notifier.Notify(rpcSub.ID, status)
            case <-rpcSub.Err():
```

- gossip/evmstore/store.go

```
    if nodes > limit+ethdb.IdealBatchSize || imgs > 4*1024*1024 {
        triedb.Cap(limit)
    }
```

- gossip/evmstore/store.go

```
        triedb := s.EvmState.TrieDB()
        triedb.SaveCache(s.cfg.Cache.TrieCleanJournal)
    }
```

- gossip/evmstore/evmpruner/pruner.go

```
            if !bytes.Equal(acc.CodeHash, evmstore.EmptyCode) {
                stateBloom.Put(acc.CodeHash, nil)
            }
```

- gossip/evmstore/evmpruner/pruner.go

```
                if hash != (common.Hash{}) {
                    stateBloom.Put(hash.Bytes(), nil)
                }
```

- gossip/evmstore/evmpruner/pruner.go

```
    if hash != (common.Hash{}) {
        stateBloom.Put(hash.Bytes(), nil)
    }
```

- gossip/evmstore/evmpruner/pruner.go

```
// the things.
os.RemoveAll(bloomPath)

```

- gossip/evmstore/evmpruner/pruner.go

```
if batch.ValueSize() > 0 {
    batch.Write()
    batch.Reset()
```

- gossip/evmstore/evmpruner/pruner.go

```
        if batch.ValueSize() >= ethdb.IdealBatchSize {
            batch.Write()
            batch.Reset()
```

- gossip/evmstore/evmpruner/pruner.go

```
        size += common.StorageSize(len(key) + len(iter.Value()))
        batch.Delete(key)

```

- gossip/evmstore/evmpruner/bloom.go

```
}
f.Close()

```

- gossip/evmstore/evmpruner/bloom.go

```
        if err := f.Sync(); err != nil {
                f.Close()
                return err
```

- evmcore/txtracer/tx_tracer.go

```
                tracesBytes, _ := json.Marshal(tr.rootTrace.Actions)
                tr.store.SetTxTrace(tr.tx, tracesBytes)
                log.Debug("Added tx trace", "txHash", tr.tx.String())
```

- evmcore/tx_pool.go

```
                                // Internal shuffle shouldn't touch the lookup set.
                                pool.enqueueTx(hash, tx, false, false)
                        }
```

- evmcore/tx_pool.go

```
                        // Internal shuffle shouldn't touch the lookup set.
                        pool.enqueueTx(hash, tx, false, false)
                }
```

- evmcore/tx_pool.go

```
                                // Internal shuffle shouldn't touch the lookup set.
                                pool.enqueueTx(tx.Hash(), tx, false, false)
                        }
```

- evmcore/tx_pool.go

```
        if pool.journal != nil {
                pool.journal.close()
        }
```

- evmcore/tx_journal.go

```
        }
        replacement.Close()
```

```

```

- evmcore/tx_journal.go

```
                    if err = rlp.Encode(replacement, tx); err != nil {
                            replacement.Close()
                            return err
```

- evmcore/tx_cacher.go

```
            for i := 0; i < len(task.txs); i += task.inc {
                    types.Sender(task.signer, task.txs[i])
            }
```

- ethclient/signer.go

```
        // Use types.Sender for side-effect to store our signer into the cache.
        types.Sender(&senderFromServer{addr, block}, tx)
}
```

- ethapi/tx_tracer.go

```
                                                jsonTraceBytes, _ :=
json.Marshal(txTraces)
                                                s.b.TxTraceSave(ctx, tx.Hash(),
jsonTraceBytes)
                                        }
```

- ethapi/tx_tracer.go

```
                                        jsonTraceBytes, _ := json.Marshal(&at)
                                        s.b.TxTraceSave(ctx, tx.Hash(),
jsonTraceBytes)
                                } else {
```

- eth/tracers/tracer.go

```
        // Set up builtins for this environment
        tracer.vm.PushGlobalGoFunction("toHex", func(ctx *duktape.Context) int {
..
        })
```

```
    tracer.vm.PushGlobalGoFunction("toWord", func(ctx *duktape.Context) int {
..
    })
    tracer.vm.PushGlobalGoFunction("toAddress", func(ctx *duktape.Context) int {
..
    })
    tracer.vm.PushGlobalGoFunction("toContract", func(ctx *duktape.Context) int {
..
    })
    tracer.vm.PushGlobalGoFunction("toContract2", func(ctx *duktape.Context) int {
..
    })
    tracer.vm.PushGlobalGoFunction("isPrecompiled", func(ctx *duktape.Context) int {
..
    })
    tracer.vm.PushGlobalGoFunction("slice", func(ctx *duktape.Context) int {
..
    })
    // Push the JavaScript tracer as object #0 onto the JSVM stack and validate it
```

- eth/protocols/snap/sync.go

```
    delete(s.peers, id)
    s.rates.Untrack(id)

```

- eth/protocols/snap/sync.go

```
    s.peers[id] = peer
    s.rates.Track(id, msgrate.NewTracker(s.rates.MeanCapacities(), s.rates.MedianRoundTrip()))

```

- debug/trace.go

```
    log.Info("Done writing Go trace", "dump", h.traceFile)
    h.traceW.Close()
    h.traceW = nil
```

- debug/trace.go

```
    if err := trace.Start(f); err != nil {
        f.Close()
        return err
```

- debug/flags.go

```
    Handler.StopCPUProfile()
    Handler.StopGoTrace()
}
```

- debug/flags.go

```
func Exit() {
    Handler.StopCPUProfile()
    Handler.StopGoTrace()
```

- debug/flags.go

```
    }
    glogger.BacktraceAt(backtrace)

```

- debug/flags.go

```
    vmodule := ctx.GlobalString(vmoduleFlag.Name)
    glogger.Vmodule(vmodule)

```

- debug/api.go

```
    buf := new(bytes.Buffer)
    pprof.Lookup("goroutine").WriteTo(buf, 2)
    return buf.String()
```

- debug/api.go

```
    time.Sleep(time.Duration(nsec) * time.Second)
    h.StopGoTrace()
    return nil
```

- debug/api.go

```
	log.Info("Done writing CPU profile", "dump", h.cpuFile)
	h.cpuW.Close()
	h.cpuW = nil
```

- debug/api.go

```
	if err := pprof.StartCPUProfile(f); err != nil {
		f.Close()
		return err
```

- debug/api.go

```
	time.Sleep(time.Duration(nsec) * time.Second)
	h.StopCPUProfile()
	return nil
```

- core/state/trie_prefetcher.go

```
					} else {
						sf.trie.TryGet(task)
						sf.seen[string(task)] = struct{}{}
```

- core/state/snapshot/wipe.go

```
		// Delete the key and periodically recreate the batch and iterator
		batch.Delete(key)
		items++
```

- core/state/snapshot/snapshot.go

```
			if key := it.Key(); len(key) == 65 { // TODO(karalabe): Yuck, we should move this into the iterator
				batch.Delete(key)
				base.cache.Del(key[1:])
```

- core/state/snapshot/generate.go

```
                root, _ := snapTrie.Commit(nil)
                snapTrieDb.Commit(root, false, nil)
        }
```

- core/state/snapshot/generate.go

```
                for i, key := range keys {
                        stackTr.TryUpdate(key, vals[i])
                }
```

- core/state/snapshot/conversion.go

```
        for leaf := range in {
                t.TryUpdate(leaf.key[:], leaf.value)
        }
```

- core/state/pruner/pruner.go

```
        }
        os.RemoveAll(path)
        log.Info("Deleted trie clean cache", "path", path)
```

- core/state/pruner/pruner.go

```
                        if !bytes.Equal(acc.CodeHash, emptyCode) {
                                stateBloom.Put(acc.CodeHash, nil)
                        }
```

- core/state/pruner/pruner.go

```
                                        if hash != (common.Hash{}) {
                                                stateBloom.Put(hash.Bytes(), nil)
                                        }
```

- core/state/pruner/pruner.go

```
                if hash != (common.Hash{}) {
                        stateBloom.Put(hash.Bytes(), nil)
                }
```

- core/state/pruner/pruner.go

```
    // the things.
    os.RemoveAll(bloomPath)

```

- core/state/pruner/pruner.go

```
    if batch.ValueSize() > 0 {
        batch.Write()
        batch.Reset()
```

- core/state/pruner/pruner.go

```
            if batch.ValueSize() >= ethdb.IdealBatchSize {
                batch.Write()
                batch.Reset()
```

- core/state/pruner/pruner.go

```
            size += common.StorageSize(len(key) + len(iter.Value()))
            batch.Delete(key)

```

- core/state/pruner/bloom.go

```
    }
    f.Close()

```

- core/state/pruner/bloom.go

```
    if err := f.Sync(); err != nil {
        f.Close()
        return err
```

- core/rawdb/freezer_table.go

```
    // Write indexEntry
    t.index.Write(idx.marshallBinary())
```

```

```

- core/rawdb/freezer_table.go

```
        t.releaseFile(t.headId)
        t.openFile(t.headId, openFreezerFileForReadOnly)

```

- core/rawdb/freezer_table.go

```
            if remove {
                os.Remove(f.Name())
            }
```

- core/rawdb/freezer_table.go

```
            delete(t.files, fnum)
            f.Close()
            if remove {
```

- core/rawdb/freezer_table.go

```
        delete(t.files, num)
        f.Close()
    }
```

- core/rawdb/freezer_table.go

```
            offsetsSize -= indexEntrySize
            t.index.ReadAt(buffer, offsetsSize-indexEntrySize)
            var newLastIndex indexEntry
```

- core/rawdb/freezer_table.go

```
    if overflow := stat.Size() % indexEntrySize; overflow != 0 {
        truncateFreezerFile(t.index, stat.Size()-overflow) // New file can't trigger this path
    }
```

- core/rawdb/freezer_table.go

```
    if err != nil {
        tab.Close()
        return nil, err
```

- core/rawdb/freezer_table.go

```
    if err := tab.repair(); err != nil {
        tab.Close()
        return nil, err
```

- core/rawdb/freezer.go

```
        }
        lock.Release()
        return nil, err
```

- core/rawdb/freezer.go

```
        for _, table := range freezer.tables {
            table.Close()
        }
```

- core/rawdb/freezer.go

```
            }
            lock.Release()
            return nil, err
```

- core/rawdb/freezer.go

```
            for _, table := range freezer.tables {
                table.Close()
            }
```

- core/rawdb/database.go

```
    if err != nil {
        kvdb.Close()
        return nil, err
```

- core/rawdb/accessors_indexes.go

```
		}
		db.Delete(it.Key())
	}
```

- core/genesis.go

```
	statedb.Commit(false)
	statedb.Database().TrieDB().Commit(root, true, nil)

```

- core/genesis.go

```
	}
	statedb.Commit(false)
	statedb.Database().TrieDB().Commit(root, true, nil)
```

- cmd/utils/flags.go

```
			if ctx.IsSet(name) {
				ctx.GlobalSet(name, ctx.String(name))
			}
```

- cmd/utils/flags.go

```
	}
	w.Flush()
}
```

- cmd/utils/customflags.go

```
func (f *DirectoryFlag) Set(value string) {
	f.Value.Set(value)
}
```

- cmd/u2u/launcher/txtracer_cmd.go

```
				counter++
				gdb.TxTraceStore().RemoveTxTrace(tx.Hash())
```

```
280:                            if time.Since(reported) >= statsReportLimit {
```

- cmd/u2u/launcher/txtracer_cmd.go

```
230:                            counter++
231:                            rlp.Encode(w, TracePayload{tx.Hash(), traces})
232:                    }
```

- cmd/u2u/launcher/txtracer_cmd.go

```
214:    if from == 1 && to == gdb.GetLatestBlockIndex() {
215:            exportAllTraceTo(w, gdb)
216:            return
```

- cmd/u2u/launcher/txtracer_cmd.go

```
91:             } else {
92:                     gdb.TxTraceStore().SetTxTrace(e.Key, e.Traces)
93:                     counter++
```

- cmd/u2u/launcher/tracing/tracing.go

```
40:     stop = func() {
41:             closer.Close()
42:     }
```

- cmd/u2u/launcher/launcher.go

```
432:                            log.Info("Old wallet dropped", "url",
event.Wallet.URL())
433:                            event.Wallet.Close()
434:                    }
```

- cmd/u2u/launcher/launcher.go

```
240:            debug.Exit()
241:            prompt.Stdin.Close() // Resets terminal mode.
242:
```

- cmd/u2u/launcher/defaults.go

```
    names, _ := f.Readdir(1)
    f.Close()
    return len(names) > 0
```

- cmd/u2u/launcher/config.go

```
    dump.WriteString(comment)
    dump.Write(out)

```

- cmd/u2u/launcher/config.go

```
    }
    dump.WriteString(comment)
    dump.Write(out)
```

- accounts/usbwallet/wallet.go

```
    // Close the device, clear everything, then return
    w.device.Close()
    w.device = nil
```

- accounts/usbwallet/wallet.go

```
            w.stateLock.Lock() // Lock state to tear the wallet down
            w.close()
            w.stateLock.Unlock()
```

- accounts/keystore/watch.go

```
        case <-debounce.C:
            w.ac.scanAccounts()
            rescanTriggered = false
```

- accounts/keystore/key.go

```
    }
    f.Close()
    return f.Name(), nil
```

- accounts/keystore/key.go

```
                f.Close()
                os.Remove(f.Name())
                return "", err
```

- accounts/keystore/key.go

```
        if _, err := f.Write(content); err != nil {
                f.Close()
                os.Remove(f.Name())
```

- accounts/keystore/account_cache.go

```
        ac.mu.Unlock()
        ac.scanAccounts()
}
```

Solution

Check the return value of a function call.

Status

Acknowledged

[N4] [Medium] 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.

- cmd/cmdtest/test_cmd.go

```
                buf = append(buf, make([]byte, tt.stdout.Buffered())...)
                tt.stdout.Read(buf[n:])
                // Find the mismatch position.
```

- ethdb/dbtest/testsuite.go

```
                }
                db.Close()
        }
```

- ethdb/dbtest/testsuite.go

```
        // Mix writes and deletes in batch
        b.Put([]byte("5"), nil)
        b.Delete([]byte("1"))
        b.Put([]byte("6"), nil)
        b.Delete([]byte("3"))
        b.Put([]byte("3"), nil)
```

- gossip/ethapi_backend.go

```
traces := make([]txtracer.ActionTrace, 0)
json.Unmarshal(txBytes, &traces)
if len(traces) == 0 {
```

- p2p/simulations/adapters/exec.go

```
}
conf.Node.initEnode(nodeTcpConn.IP, nodeTcpConn.Port, nodeTcpConn.Port)
conf.Stack.P2P.PrivateKey = conf.Node.PrivateKey
```

- p2p/simulations/adapters/exec.go

```
wg.Wait()
conn.Close()
return nil
```

- p2p/simulations/adapters/types.go

```
}
l.Close()
_, port, err := net.SplitHostPort(l.Addr().String())
```

- p2p/simulations/http.go

```
		response, _ := ioutil.ReadAll(res.Body)
		res.Body.Close()
		return nil, fmt.Errorf("unexpected HTTP status: %s: %s", res.Status, response)
```

- p2p/simulations/pipes/pipes.go

```
	if err := <-aerr; err != nil {
		dconn.Close()
		return nil, nil, err
```

- p2p/simulations/test.go

```
				}
				rw.ReadMsg()
				return nil
```

Solution

Check the return value of a function call.

Status

Fixed; PR: https://github.com/unicornultrafoundation/go-u2u/pull/134

[N5] [Medium] Implicit memory aliasing in for loop

Category: SAST

Content

In these code snippets, implicit memory aliasing is used in the for loop, which may lead to potential concurrency issues or data inconsistency. To avoid this, a new variable can be created inside the loop to hold the value of the current iteration.

- p2p/simulations/http.go

```
		for _, conn := range snap.Conns {
			event := NewEvent(&conn)
			if err := writeEvent(event); err != nil {
```

- p2p/simulations/http.go

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

- p2p/discover/v4_udp.go

```
        // Don't advertise the private nodes
        if len(t.privateNodes) > 0 && containsEnode(t.privateNodes, &n.Node) {
                continue
```

- gossip/filters/api.go

```
                        for _, log := range logs {
                                _ = notifier.Notify(rpcSub.ID, &log)
                        }
```

- evmcore/txtracer/tx_tracer.go

```
        if traceIndex == nil || equalContent(traceIndex, trace.TraceAddress) {
                callTrace.AddTrace(&trace)
        }
```

- ethapi/tx_tracer.go

```
                                                if traceCount >= args.After {
                                                        callTrace.AddTrace(&trace)
                                                        traceAdded++
```

- core/rawdb/freezer.go

```
    for _, table := range f.tables {
            items := atomic.LoadUint64(&table.items)
            if min > items {
```

- cmd/u2u/launcher/accountcmd.go

```
            if err := ks.Unlock(a, auth); err == nil {
                    match = &a
                    break
```

- cmd/u2u/launcher/accountcmd.go

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

Fixed; PR: https://github.com/unicornultrafoundation/go-u2u/pull/133

[N6] [Information] Use of unsafe calls

Category: SAST

Content

Converting `[]byte` to `uint64` directly through `unsafe.Pointer` is dangerous and may cause memory alignment problems and undefined behavior, and may produce different results on different system architectures.

The code does not consider the system's byte order (big endian/little endian), which may lead to incorrect results on different platforms

- utils/memory/memsysctl.go

```
        }
        return *(*uint64)(unsafe.Pointer(&b[0])), nil
}
```

The following codes also have similar unsafe call problem.

- rlp/unsafe.go

```
	var s []byte
	hdr := (*reflect.SliceHeader)(unsafe.Pointer(&s))
	hdr.Data = v.UnsafeAddr()
```

- p2p/nodestate/nodestate.go

```
	}
	if len(setup.flags) > 8*int(unsafe.Sizeof(bitMask(0))) {
		panic("Too many node state flags")
```

- eth/tracers/tracer.go

```

	return *(*[]byte)(unsafe.Pointer(&sl))
}
```

- core/types/receipt.go

```
	size := common.StorageSize(unsafe.Sizeof(*r)) +
common.StorageSize(len(r.PostState))
	size += common.StorageSize(len(r.Logs)) *
common.StorageSize(unsafe.Sizeof(Log{}))
	for _, log := range r.Logs {
```

- core/types/receipt.go

```
func (r *Receipt) Size() common.StorageSize {
	size := common.StorageSize(unsafe.Sizeof(*r)) +
common.StorageSize(len(r.PostState))
	size += common.StorageSize(len(r.Logs)) *
common.StorageSize(unsafe.Sizeof(Log{}))
```

- common/bitutil/bitutil.go

```
	if w > 0 {
		pw := *(*[]uintptr)(unsafe.Pointer(&p))
		for i := 0; i < w; i++ {
```

- common/bitutil/bitutil.go

```
		aw := *(*[]uintptr)(unsafe.Pointer(&a))
		bw := *(*[]uintptr)(unsafe.Pointer(&b))
		for i := 0; i < w; i++ {
```

- common/bitutil/bitutil.go

```
		dw := *(*[]uintptr)(unsafe.Pointer(&dst))
		aw := *(*[]uintptr)(unsafe.Pointer(&a))
		bw := *(*[]uintptr)(unsafe.Pointer(&b))
```

- common/bitutil/bitutil.go

```
	if w > 0 {
		dw := *(*[]uintptr)(unsafe.Pointer(&dst))
		aw := *(*[]uintptr)(unsafe.Pointer(&a))
```

- common/bitutil/bitutil.go

```
		aw := *(*[]uintptr)(unsafe.Pointer(&a))
		bw := *(*[]uintptr)(unsafe.Pointer(&b))
		for i := 0; i < w; i++ {
```

- common/bitutil/bitutil.go

```
		dw := *(*[]uintptr)(unsafe.Pointer(&dst))
		aw := *(*[]uintptr)(unsafe.Pointer(&a))
		bw := *(*[]uintptr)(unsafe.Pointer(&b))
```

- common/bitutil/bitutil.go

```
	if w > 0 {
		dw := *(*[]uintptr)(unsafe.Pointer(&dst))
		aw := *(*[]uintptr)(unsafe.Pointer(&a))
```

- common/bitutil/bitutil.go

```
		aw := *(*[]uintptr)(unsafe.Pointer(&a))
		bw := *(*[]uintptr)(unsafe.Pointer(&b))
		for i := 0; i < w; i++ {
```

- common/bitutil/bitutil.go

```
		dw := *(*[]uintptr)(unsafe.Pointer(&dst))
		aw := *(*[]uintptr)(unsafe.Pointer(&a))
		bw := *(*[]uintptr)(unsafe.Pointer(&b))
```

- common/bitutil/bitutil.go

```
	if w > 0 {
		dw := *(*[]uintptr)(unsafe.Pointer(&dst))
		aw := *(*[]uintptr)(unsafe.Pointer(&a))
```

- common/bitutil/bitutil.go

```

const wordSize = int(unsafe.Sizeof(uintptr(0)))
const supportsUnaligned = runtime.GOARCH == "386" || runtime.GOARCH == "amd64" || runtime.GOARCH == "ppc64" || runtime.GOARCH == "ppc64le" || runtime.GOARCH == "s390x"
```

Solution

Stop using unsafe calls

Status

Acknowledged

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002412200001	SlowMist Security Team	2024.12.10 - 2024.12.20	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 2 medium risk, 1 low risk vulnerabilities.

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.