Competitive Security Assessment

NeoX_Bridge_Relayer

Aug 7th, 2024

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3′s Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.
The comprehensive examination and auditing scope includes:
• Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
• Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
• Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
• Verify the code base is compliant with the most up-to-date industry standards and security best practices.
• Comprehensive line-by-line manual code review of the entire codebase by industry experts.
The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

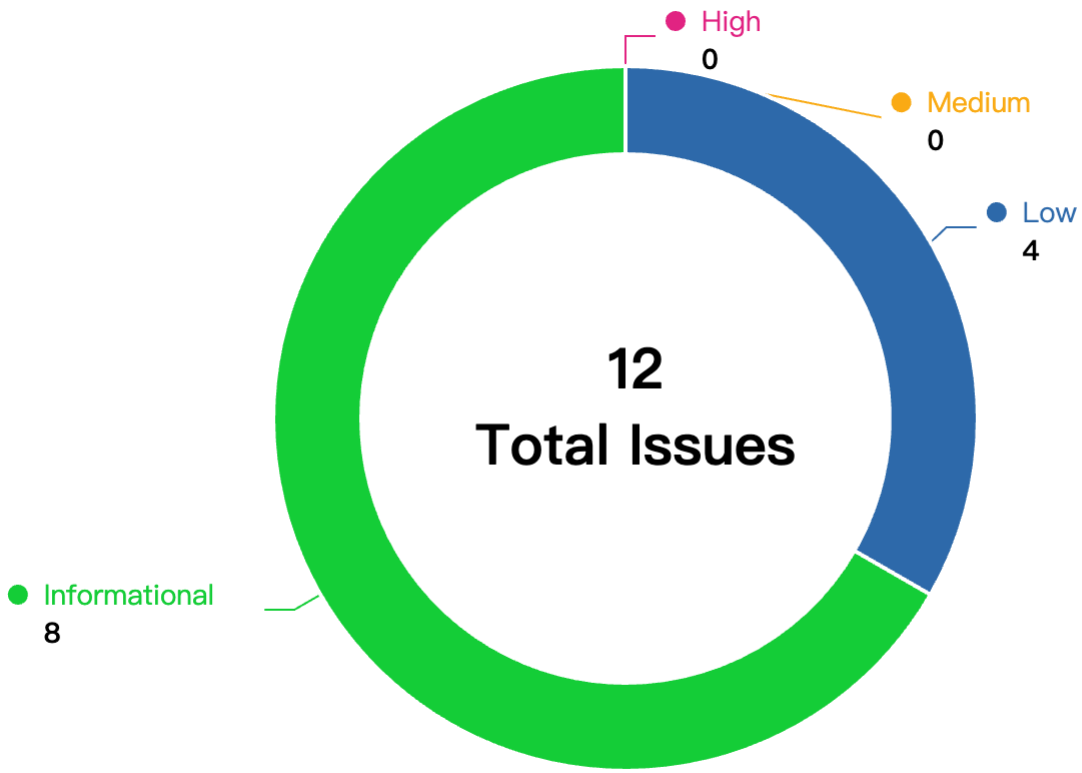
Overview

Project Name	NeoX_Bridge_Relayer
Language	solidity
Codebase	• https://github.com/bane-labs/bridge/commit/76744947e868378e21d02d5a943e47a07a00b1e6 • audit version-76744947e868378e21d02d5a943e47a07a00b1e6 • final version-ae1ac8e3d7e9c9c1eeb192594da01c3cec8a42eb

Audit Scope

File	SHA256 Hash
relayer/withdrawal.go	4acfda565674287248f546618416bfe88d787dcc1edf7a158852814c30eba09a
signatureprocessor/deposit.go	2448a40f26eef86d0bd33fa61d5e63c8814286900c3f316925a0cb0b681a2dd1
signatureprocessor/withdrawal.go	1d2a18d44ed5167bf0170fee4687d12672e203fccfbf1a99367cf3b083d34d60
relayer/deposit.go	71dea38a520f41137d770e2e24aa092808de59f1d80c9b5e89dd06b36f969af2
relayer/model.go	076f81a2881cf0ffeaa4df87e6b2faf9bdf2dae8e4eb65c89d2c62b9ccf4032c
wallet/neox.go	f60854c3514b7272d3eb471b5e075c471ca0ec5bd2c9fe9ebd788dcf481f1b2c
wallet/n3.go	ebdbe22dd763ef2eeca758f5db3f29107ba66c6215d406d03c41ba5cd4b209a5
signatureprocessor/model.go	b8b549a87578ce311be46585ffa85ab04f351bc6cb0b5fbed7adb9c699505dbc
sig/neox.go	fe429a7a38cc4f42b341e71553e1f2452bc1bf4dbb984caaa665b4462eead657
sig/n3.go	55e4ea00857af9554112ab1ab45b20ca3094258633c42ea58de636f626e71ded
sig/model.go	9d85ff8d8dc100993df70fa76ecfc889fb88c6d205e55220b3eca010856991da

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
NBR-1	The program should exit when it encounters a non-connection exception	Logical	Low	Fixed	***
NBR-2	Potential DoS in signature processors due to lack of timeout	DOS	Low	Fixed	***
NBR-3	Insecure Signature Verification	Logical	Low	Acknowledged	***
NBR-4	Hardcoded Signature Count Threshold in Relayer Contracts	Logical	Low	Acknowledged	***
NBR-5	The decrypted account is not closed after it signs the data	Code Style	Informational	Acknowledged	***
NBR-6	Reusing Password Input	Logical	Informational	Fixed	***
NBR-7	Return error instead of direct Exit	Logical	Informational	Acknowledged	***
NBR-8	Potential invalid `keyDir` in `NewEthKeyStore()`	Code Style	Informational	Fixed	***
NBR-9	Potential Runtime Panic	Logical	Informational	Fixed	***

NBR-10	Password should be encrypted	Logical	Informational	Acknowledged	***
NBR-11	Lack of handling for empty account arrays	Language Specific	Informational	Fixed	***
NBR-12	Goroutines in loops can cause race conditions	Race condition	Informational	Acknowledged	***

NBR-1:The program should exit when it encounters a non-connection exception

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/relayer/withdrawal.go#L21-L78

```
21: func (r *WithdrawalRelayer) Start() {
22:     r.Logger.Info("Starting withdrawal relayer")
23:     neoN3TokenHash := r.NeoN3TokenHash
24:     neoXTokenHash := r.NeoXTokenHash
25:     for {
26:         go func() {
27:             // In case the relayer crashes after storing 5 signatures but before successfully rel
aying the distribution, the relayer should trigger a relay on startup before waiting for new messages
in the queue.
28:             if r.relayTriggered() {
29:                 relayError := r.relayWithdrawals()
30:                 if relayError != nil {
31:                     r.Logger.Error("Failed to relay withdrawal", relayError.Error(), relayError)
32:                     os.Exit(1)
33:                 }
34:             }
35:
36:             var nextNonce uint64
37:             var brokerError error
38:             var relayError error
39:             for {
40:                 nextNonce = r.getLastNonce() + 1
```

```
41:                 brokerError = r.Broker.ConsumeWithdrawals(nextNonce, neoN3TokenHash, neoXTokenHa
sh)
42:                 if brokerError != nil {
43:                     r.Logger.Error("Failed to consume withdrawals", brokerError.Error(), brokerEr
ror)
44:                     // Todo: Wrap errors to check if it is a connection error. If it's not, issue
a graceful shutdown. Otherwise, return.
45:                     if strings.Contains(brokerError.Error(), "connection error") {
46:                         r.ConnectionErr <- brokerError
47:                         return
48:                     }
49:                     return
50:                 }
51:
52:                 relayError = r.relayWithdrawals()
53:                 if relayError != nil {
54:                     r.Logger.Error("Failed to relay withdrawals", relayError.Error(), relayError)
55:                     os.Exit(1)
56:                 }
57:
58:                 // r.config.QueueCleanupEnabled should be set to false by default in order to kee
p queues for debugging reasons.
59:                 // If the relayer is running in production, it should be set to true.
60:                 if *r.Config.QueueCleanupEnabled {
```

```
61:                     // Run queue cleanup routine
62:                     go func() {
63:                         r.Logger.Debug("Queue cleanup started")
64:                         r.Broker.CleanupQueues(nextNonce, neoN3TokenHash, neoXTokenHash)
65:                         r.Logger.Debug("Queue cleanup finished")
66:                     }()
67:                 }
68:             }
69:         }()
70:         if err := <-r.ConnectionErr; err != nil {
71:             r.Logger.Error("Withdrawal relayer connection error", err.Error(), err)
72:             err = r.Broker.Reconnect()
73:             if err != nil {
74:                 return // reconnect error, just return
75:             }
76:         }
77:     }
78: }
```

Description

***: In `withdrawal.go`, the `Start()` function will start an infinite `for` loop. In this `for` loop, it will call an inner `for` loop to consume withdrawals:

```
//outer loop
for {
        go func() {
            // In case the relayer crashes after storing 5 signatures but before successfully rela
ying the distribution, the relayer should trigger a relay on startup before waiting for new messag
es in the queue.
            if r.relayTriggered() {
                relayError := r.relayDeposits()
                if relayError != nil {
                    r.Logger.Error("Failed to relay deposits", relayError.Error(), relayError)
                    os.Exit(1)
                }
            }

            var nextNonce uint64
            var brokerError error
            var relayError error
      // inner loop
            for {
                nextNonce = r.getLastNonce() + 1
                brokerError = r.Broker.ConsumeDeposits(nextNonce, neoN3TokenHash, neoXTokenHash)
                if brokerError != nil {
                    r.Logger.Error("Failed to consume deposits", brokerError.Error(), brokerError)
                    // Todo: Wrap errors to check if it is a connection error. If it's not, issue
a graceful shutdown. Otherwise, return.
                    if strings.Contains(brokerError.Error(), "connection error") {
                        r.ConnectionErr <- brokerError
                        return
                    }
                    os.Exit(1) // replace with a graceful shutdown when available
                }

                relayError = r.relayDeposits()
                if relayError != nil {
                    r.Logger.Error("Failed to relay deposits", relayError.Error(), relayError)
                    os.Exit(1)
                }

                // r.config.QueueCleanupEnabled should be set to false by default in order to keep
queues for debugging reasons.
                // If the relayer is running in production, it should be set to true.
                if *r.Config.QueueCleanupEnabled {
                    // Run queue cleanup routine
                    go func() {
                        r.Logger.Debug("Queue cleanup started")
                        r.Broker.CleanupQueues(nextNonce, neoN3TokenHash, neoXTokenHash)
                        r.Logger.Debug("Queue cleanup finished")
```

If the `brokerError` is not nil and it is not a connection error, it will call `return`:

```
if brokerError != nil {
                    r.Logger.Error("Failed to consume withdrawals", brokerError.Error(), brokerErr
or)
                    // Todo: Wrap errors to check if it is a connection error. If it's not, issue
a graceful shutdown. Otherwise, return.
                    if strings.Contains(brokerError.Error(), "connection error") {
                        r.ConnectionErr <- brokerError
                        return
                    }
                    return
                }
```

That means the inner `for` loop will return and the outer `for` loop continues:

```
// outer loop
for {
        go func() {
              ...
        ...
        //inner loop
        ...
        }()
        if err := <-r.ConnectionErr; err != nil {
            r.Logger.Error("Withdrawal relayer connection error", err.Error(), err)
            err = r.Broker.Reconnect()
            if err != nil {
                return // reconnect error, just return
            }
        }
    }
```

Since the `brokerError` is not a connection error, the `r.ConnectionErr` will be `nil` and it will ignore the reconnection handling. The outer `for` loop will continue. As a result, the outer `for` loop will execute indefinitely, even if the broker keeps returning errors, which can lead to resource exhaustion.

Recommendation

***: Consider following fix:

```
if brokerError != nil {
                    r.Logger.Error("Failed to consume withdrawals", brokerError.Error(), brokerErr
or)
                    // Todo: Wrap errors to check if it is a connection error. If it's not, issue
a graceful shutdown. Otherwise, return.
                    if strings.Contains(brokerError.Error(), "connection error") {
                        r.ConnectionErr <- brokerError
                        return
                    }
                    os.Exit(1)
                }
```

Client Response

client response : Fixed. This issue was fixed in commit a559804cfd2bd3a98553eb77f3dd94e3594d0824.

NBR-2:Potential DoS in signature processors due to lack of timeout

Category	Severity	Client Response	Contributor
DOS	Low	Fixed	***

Code Reference

- code/signatureprocessor/deposit.go#L72
- code/signatureprocessor/deposit.go#L147

```
72: <-awaitAckChannel
```

```
147: <-awaitAckChannel
```

- code/signatureprocessor/withdrawal.go#L72
- code/signatureprocessor/withdrawal.go#L146

```
72: <-awaitAckChannel
```

```
146: <-awaitAckChannel
```

Description

***: In signature processors for both deposit and withdrawal, the code creates a channel called `awaitAckChannel` and calls `p.broker.PublishTokenDeposits` in a goroutine. After the call, the code waits acknowledgement from broker:

```
<-awaitAckChannel
```

The problem here is that there is no timeout for this wait. If the broker does not ACK this call, signature processor is going to wait for this response forever and put the system into a DoS state.

Recommendation

***: Add timeout to this code. Timeout can be implemented with `time.After()`. After a specific amount of time, the signature processor should throw error or exit from the call stack gracefully.

Client Response

client response : Fixed. This has been fixed in commit a7d2952f4715b9a4ac8fd4c4a8de3b8489fc4d2a.

NBR-3:Insecure Signature Verification

Category	Severity	Client Response	Contributor
Logical	Low	Acknowledged	***

Code Reference

- code/relayer/deposit.go#L130-L133

```
130: } else if len(depositSig.Signatures) < 5 {
131:         r.Logger.Error(fmt.Sprintf("Not enough signatures found for nonce %v, only %v signatures
present.", fmt.Sprint(nextNonce), len(depositSig.Signatures)))
132:         os.Exit(1)
133:     }
```

- code/relayer/withdrawal.go#L143-L146

```
143: } else if len(withdrawalSig.Signatures) < 5 {
144:         r.Logger.Error(fmt.Sprintf("Not enough signatures found for nonce %v, only %v signatures
present.", fmt.Sprint(nextNonce), len(withdrawalSig.Signatures)))
145:         os.Exit(1)
146:     }
```

Description

***: The deposit signature verification function only checks if the number of signatures is sufficient, but does not verify the uniqueness of the signatures. As shown in the following code snippet:

```
if depositSig == nil {
    r.Logger.Error(fmt.Sprintf("No deposit signature found for nonce %v", nextNonce))
    os.Exit(1)
} else if len(depositSig.Signatures) < 5 {
    r.Logger.Error(fmt.Sprintf("Not enough signatures found for nonce %v, only %v signatures prese
nt.", fmt.Sprint(nextNonce), len(depositSig.Signatures)))
    os.Exit(1)
}
```

This allows a valid account key to be used by multiple validator IDs, generating the same signature, which can bypass the signature verification process.

Recommendation

***: To prevent this vulnerability, it is recommended to add a check for duplicate signatures in the verification process.

Client Response

client response : Acknowledged. We acknowledge this issue and we will resolve it as soon as possible.

NBR-4:Hardcoded Signature Count Threshold in Relayer Contracts

Category	Severity	Client Response	Contributor
Logical	Low	Acknowledged	***

Code Reference

- code/relayer/deposit.go#L114

```
114: return sigCount >= 5
```

- code/relayer/withdrawal.go#L122

```
122: return sigCount >= 5
```

Description

***: The relayer contracts for deposit and withdrawal operations in the bridge protocol currently have the signature count threshold hardcoded. Specifically, the threshold for the number of signatures required to relay deposits and withdrawals is set to 5 within the code. This practice lacks flexibility and can lead to maintenance challenges, especially if the threshold needs to be adjusted in the future.
Impact:

- Hardcoding the signature count threshold means any change to this value requires modifying the source code and redeploying the contracts. This process is cumbersome and error-prone.
- As the protocol evolves, the required number of signatures may change based on security needs or network conditions. Hardcoding this value makes it difficult to adapt to such changes quickly.
- If the threshold needs to be increased due to a security vulnerability, the time required to update and redeploy the contracts could expose the protocol to attacks.
- Frequent updates to the threshold would necessitate multiple redeployments, increasing operational overhead and potential downtime.

Recommendation

***: To address the issue, the signature count threshold should be stored in a state variable within the relayer contracts. Additionally, an admin function should be implemented to allow authorized personnel to update this threshold as needed.
Implementation Steps

1. Introduce State Variable: Add a state variable to store the signature count threshold.
2. Admin Function: Implement an admin function to update the threshold.
3. Update Logic: Modify the existing logic to use the state variable instead of the hardcoded value.

State Variable

```
type Relayer struct {
    // Other fields...
    SignatureThreshold uint64
}
```

Admin Function

```
func (r *Relayer) SetSignatureThreshold(newThreshold uint64) {
    if !r.isAdmin(msg.sender) {
        r.Logger.Error("Unauthorized access to set signature threshold")
        return
    }
    r.SignatureThreshold = newThreshold
    r.Logger.Info(fmt.Sprintf("Signature threshold updated to %d", newThreshold))
}

func (r *Relayer) isAdmin(sender address) bool {
    // Implement admin check logic
    return sender == r.adminAddress
}
```

Usage in relayTriggered Replace the hardcoded value with the state variable:

```
func (r *DepositRelayer) relayTriggered() bool {
    // Other logic...
    return sigCount >= r.SignatureThreshold
}
```

Client Response

client response : Acknowledged. This issue is valid and we acknowledge it. We will resolve it as soon as possible.

NBR-5:The decrypted account is not closed after it signs the data

Category	Severity	Client Response	Contributor
Code Style	Informational	Acknowledged	***

Code Reference

- code/wallet/n3.go#L31-L40

```
31: func (w *N3Wallet) SignData(data []byte, pwd string) ([]byte, error) {
32:     acc := w.wallet.GetAccount(w.wallet.GetChangeAddress())
33:     err := acc.Decrypt(pwd, w.wallet.Scrypt)
34:     if err != nil {
35:         return nil, err
36:     }
37:     sigData := acc.PrivateKey().Sign(data)
38:
39:     return sigData, nil
40: }
```

Description

***: The function `SignData` decrypts the EncryptedWIF with the given passphrase:

```
  acc := w.wallet.GetAccount(w.wallet.GetChangeAddress())
    err := acc.Decrypt(pwd, w.wallet.Scrypt)
    if err != nil {
        return nil, err
    }
```

If no error returns, the decryption Account will be used to sign the data:

```
sigData := acc.PrivateKey().Sign(data)
```

As per the doc of Decrypt(), it is better to call `Close` after use for maximum safety:

```
// Decrypt decrypts the EncryptedWIF with the given passphrase returning error
// if anything goes wrong. After the decryption Account can be used to sign
// things unless it's locked. Don't decrypt the key unless you want to sign
// something and don't forget to call Close after use for maximum safety.
func (a *Account) Decrypt(passphrase string, scrypt keys.ScryptParams) error {
    var err error

    if a.EncryptedWIF == "" {
        return errors.New("no encrypted wif in the account")
    }
    a.privateKey, err = keys.NEP2Decrypt(a.EncryptedWIF, passphrase, scrypt)
    if err != nil {
        return err
    }

    return nil
}
```

Recommendation

***: Consider calling `Close` after the data is signed:

```
func (w *N3Wallet) SignData(data []byte, pwd string) ([]byte, error) {
    acc := w.wallet.GetAccount(w.wallet.GetChangeAddress())
    err := acc.Decrypt(pwd, w.wallet.Scrypt)
    if err != nil {
        return nil, err
    }
    sigData := acc.PrivateKey().Sign(data)
  acc.Close()
    return sigData, nil
}
```

Client Response

client response : Acknowledged. We acknowledge this issue and will review the necessity to apply the recommendation.

NBR-6:Reusing Password Input

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/wallet/neox.go#L74

```
74: pwd = string(input)
```

Description

***: In `VerifyOrPromptForPwdInput` same password input is reused for unlocking the account and signing transactions.

```
func (w *N3Wallet) VerifyOrPromptForPwdInput(pwd string) (string, error) {
    for !w.VerifyPassword(pwd) {
        fmt.Print("Please enter the validator N3 wallet password:")
        input, err := term.ReadPassword(int(os.Stdin.Fd()))
        pwd = string(input)
        if err != nil {
            return pwd, err
        }
        fmt.Println()
    }
    fmt.Println("N3 wallet unlocked successfully.")
    return pwd, nil
}
```

If the password is incorrect, the user is prompted again but does not have the option to enter a new password securely.
It would be better to clear the input buffer after each read to ensure no sensitive information remains in memory.

Recommendation

***: The recommendation is made to clear the password after each read.

```
func (w *N3Wallet) VerifyOrPromptForPwdInput(pwd string) (string, error) {
    for !w.VerifyPassword(pwd) {
        fmt.Print("Please enter the validator N3 wallet password:")
        input, err := term.ReadPassword(int(os.Stdin.Fd()))
-       pwd = string(input)
        if err != nil {
            return pwd, err
        }
+       pwd = string(input)
        fmt.Println()
    }
    fmt.Println("N3 wallet unlocked successfully.")
    return pwd, nil
}
```

Client Response

client response : Fixed. This issue has been fixed in commit ae1ac8e3d7e9c9c1eeb192594da01c3cec8a42eb.

NBR-7:Return error instead of direct Exit

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/wallet/n3.go#L25

```
25: os.Exit(1)
```

Description

***: In `NewN3KeyStore` function direct call to `os.Exit(1)` is made upon encountering an error while loading the wallet file.

```
func NewN3KeyStore(cfg *config.Config, logger *slog.Logger) *N3Wallet {
    logger.Info("Initializing N3 wallet...")
    service := &N3Wallet{}
    w, err := wallet.NewWalletFromFile(*cfg.NeoN3KeyPairFile)
    if err != nil {
        logger.Error("Load N3 wallet file failed:", err)
        os.Exit(1)
    }
    service.wallet = w
    return service
}
```

This can lead to issues as it abruptly terminates the program without performing any cleanup operations. It also prevents the caller from handling the error, potentially leading to data loss or corruption.

Recommendation

***: Instead of exiting `NewN3KeyStore` function directly, consider returning the error to the caller so it can be handle more efficiently situation.

```
func NewN3KeyStore(cfg *config.Config, logger *slog.Logger) (*N3Wallet, error) {
    logger.Info("Initializing N3 wallet...")
    service := &N3Wallet{}
    w, err := wallet.NewWalletFromFile(*cfg.NeoN3KeyPairFile)
    if err != nil {
        logger.Error("Load N3 wallet file failed:", err)
-       os.Exit(1)
+       return nil, err // Return the error instead of exiting
    }
    service.wallet = w
    return service, nil
}
```

If immediate termination is required as per design, ensure all resources are properly released before doing so.

Client Response

client response : Acknowledged. This issue will be acknowledged. We already have an issue open to implement a proper graceful shutdown, and this issue is touching this topic. However, this needs some time for the codebase to be adapted properly, which is why I mark this as acknowledged and we will fix this in the near future.

NBR-8:Potential invalid `keyDir` in `NewEthKeyStore()`

Category	Severity	Client Response	Contributor
Code Style	Informational	Fixed	***

Code Reference

- code/wallet/neox.go#L24

```
24: capitalKeyStore := keystore.NewKeyStore(*cfg.NeoXKeyPairFile, keystore.StandardScryptN, keystor
e.StandardScryptP)
```

Description

***: The **`NewEthKeyStore`** function passes a file path as the first argument **`cfg.NeoXKeyPairFile`** to the **`keystore.NewKeyStore`** function, which expects a directory path. This mismatch can lead to an error, causing the application to exit.
The **`keystore.NewKeyStore`** function is defined as:

```
func NewKeyStore(keydir string, scryptN, scryptP int) *KeyStore {
   ...
}
```

However, the **`NewEthKeyStore`** function passes a file path instead of a directory path, which can cause an error:

```
// incorrect argument passing
capitalKeyStore := keystore.NewKeyStore(*cfg.NeoXKeyPairFile, keystore.StandardScryptN, keystore.S
tandardScryptP)
```

Recommendation

***: It is recommended to pass the correct directory path to the **`keystore.NewKeyStore`** function or update the parameter name **`cfg.NeoXKeyPairFile`** to avoid confusion.

Client Response

client response : Fixed. This has been fixed in commit ee5c3bea8b47bed22a0d1d5bf90794354c1ff651.

NBR-9:Potential Runtime Panic

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/signatureprocessor/deposit.go#L37-L55
- code/signatureprocessor/deposit.go#L111-L129

```
37: for {
38:         var deposits []store.Deposit
39:         if highestNonceToProcess >= highestNonceSigned+100 {
40:             // get (the first) 100 deposits to process
41:             if found, err := p.depositStore.GetDeposits(highestNonceSigned+1, highestNonceSigned
+100, &deposits, nil); !found || err != nil {
42:                 p.logger.Error("GetDeposits error:", err)
43:                 errorChan <- err
44:                 return
45:             }
46:         } else {
47:             // get all deposits to process
48:             if found, err := p.depositStore.GetDeposits(highestNonceSigned+1, highestNonceToProc
ess, &deposits, nil); !found || err != nil {
49:                 p.logger.Error("GetDeposits error:", err)
50:                 errorChan <- err
51:                 return
52:             }
53:         }
54:         // Get the last deposit's root hash
55:         topRootHash := deposits[len(deposits)-1].Root
```

```
111: for {
112:         var deposits []store.Deposit
113:         if highestNonceToProcess >= highestNonceSigned+100 {
114:             // get (the first) 100 deposits to process
115:             if found, err := p.depositStore.GetDeposits(highestNonceSigned+1, highestNonceSigned
+100, &deposits, neoN3TokenHash); !found || err != nil {
116:                 p.logger.Error("GetDeposits error:", err)
117:                 errorChan <- err
118:                 return
119:             }
120:         } else {
121:             // get all deposits to process
122:             if found, err := p.depositStore.GetDeposits(highestNonceSigned+1, highestNonceToProc
ess, &deposits, neoN3TokenHash); !found || err != nil {
123:                 p.logger.Error("GetDeposits error:", err)
124:                 errorChan <- err
125:                 return
126:             }
127:         }
128:         // Get the last deposit's root hash
129:         topRootHash := deposits[len(deposits)-1].Root
```

- code/signatureprocessor/withdrawal.go#L38-L56
- code/signatureprocessor/withdrawal.go#L111-L129

```
38: for {
39:         var withdrawals []store.Withdrawal
40:         if highestNonceToProcess >= highestNonceSigned+100 {
41:             // get (the first) 100 withdrawals to process
42:             if found, err := p.withdrawalStore.GetWithdrawals(highestNonceSigned+1, highestNonce
Signed+100, &withdrawals, nil); !found || err != nil {
43:                 p.logger.Error("GetWithdrawals error:", err)
44:                 errorChan <- err
45:                 return
46:             }
47:         } else {
48:             // get all withdrawals to process
49:             if found, err := p.withdrawalStore.GetWithdrawals(highestNonceSigned+1, highestNonce
ToProcess, &withdrawals, nil); !found || err != nil {
50:                 p.logger.Error("GetWithdrawals error:", err)
51:                 errorChan <- err
52:                 return
53:             }
54:         }
55:         // Get the last withdrawal's root hash
56:         topRootHash := withdrawals[len(withdrawals)-1].Root
```

```
111: for {
112:         var withdrawals []store.Withdrawal
113:         if highestNonceToProcess >= highestNonceSigned+100 {
114:             // get (the first) 100 withdrawals to process
115:             if found, err := p.withdrawalStore.GetWithdrawals(highestNonceSigned+1, highestNonce
Signed+100, &withdrawals, neoXTokenHash); !found || err != nil {
116:                 p.logger.Error("GetWithdrawals error:", err)
117:                 errorChan <- err
118:                 return
119:             }
120:         } else {
121:             // get all withdrawals to process
122:             if found, err := p.withdrawalStore.GetWithdrawals(highestNonceSigned+1, highestNonce
ToProcess, &withdrawals, neoXTokenHash); !found || err != nil {
123:                 p.logger.Error("GetWithdrawals error:", err)
124:                 errorChan <- err
125:                 return
126:             }
127:         }
128:         // Get the last withdrawal's root hash
129:         topRootHash := withdrawals[len(withdrawals)-1].Root
```

Description

***: In `DepositSignatureProcessor`, the function `ProcessDepositSignature` is used to retrieve deposits to process. The code checks if `highestNonceToProcess` is greater than or equal to `highestNonceSigned + 100`. If true, it retrieves the first 100 deposits to process. Otherwise, it retrieves all deposits up to `highestNonceToProcess`:

```
if highestNonceToProcess >= highestNonceSigned+100 {
            // get (the first) 100 deposits to process
            if found, err := p.depositStore.GetDeposits(highestNonceSigned+1, highestNonceSigned+1
00, &deposits, nil); !found || err != nil {
                p.logger.Error("GetDeposits error:", err)
                errorChan <- err
                return
            }
        } else {
            // get all deposits to process
            if found, err := p.depositStore.GetDeposits(highestNonceSigned+1, highestNonceToProces
s, &deposits, nil); !found || err != nil {
                p.logger.Error("GetDeposits error:", err)
                errorChan <- err
                return
            }
        }
```

However, it does not take into account the case of `highestNonceToProcess<highestNonceSigned`.
If `highestNonceToProcess < highestNonceSigned`, the code will go to `else` block too. It will call `p.depositStore.GetDeposits(highestNonceSigned+1, highestNonceToProcess, &deposits, nil);`. Let's say the `highestNonceSigned` is 100, the `highestNonceToProcess` is 99, in `GetDeposits`, it will call the following for loop:

```
func (s *DepositStore) GetDeposits(startNonce, endNonce uint64, deposits *[]Deposit, tokenHash int
erface{}) (bool, error) {
    s.dataStore.mu.Lock()
    defer s.dataStore.mu.Unlock()
    for i := startNonce; i <= endNonce; i++ {
        var deposit Deposit
        prefix, err := BuildPrefixedKey(s.config.prefix, i, tokenHash)
        if err != nil {
            return false, err
        }
        if found, err := s.dataStore.store.Get(prefix, &deposit); !found || err != nil {
            return false, err
        } else {
            *deposits = append(*deposits, deposit)
        }
    }
    return true, nil
}
```

Since `startNonce` is 101 and the `endNonce` is 99, the for loop will be ignored and the function `GetDeposits` will return `(true, nil)`. In `ProcessDepositSignature` function, it will then go to the following code:

```
topRootHash := deposits[len(deposits)-1].Root
```

The code attempts to access `deposits[len(deposits)-1]`, which is `deposits[-1]`. Since deposits is empty, there is no element at index -1, resulting in a runtime panic.
The same issue exits in `ProcessTokenDepositSignature`, `ProcessWithdrawalSignature` and `ProcessTokenWithdrawalSignature`.

Recommendation

***: Consider adding following check:

```
if highestNonceToProcess < highestNonceSigned {
      p.logger.Error("highestNonceToProcess is less than highestNonceSigned")
                return
}
```

Client Response

client response : Fixed. This has been fixed in commit e7f266512351b8f07d0cc0b69d049627807b8de2.

NBR-10:Password should be encrypted

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/sig/n3.go#L9

```
9: func NewN3Signer(wallet *wallet.N3Wallet, password string) *N3Signer {
```

- code/sig/neox.go#L11

```
11: func NewNeoXSigner(wallet *wallet.EthWallet, password string) *NeoXSigner {
```

Description

***:

`NewN3Signer` function in `n3` contract and `NewNeoXSigner` function of `neox` store password `pwd` directly as string within the struct. If this password is used for authentication or encryption purposes, it should not be stored in plain text within the code.

```
func NewN3Signer(wallet *wallet.N3Wallet, password string) *N3Signer {
    return &N3Signer{
        wallet: wallet,
        pwd:    password,
    }
}
```

The password parameter is directly stored in memory within the `N3Signer` and `NeoXSigner` struct without any form of encryption or hashing. This could potentially expose the password to unauthorized access if the memory is not properly secured.

Recommendation

***:

Store hashed versions of passwords instead of plain text. Use a secure hash function like bcrypt, scrypt, or Argon2.

```
func NewN3Signer(wallet *wallet.N3Wallet, password string) *N3Signer {
+   hashedPassword, err := argon2.IDKey(password, rand.Reader)
+   if err != nil {
+       log.Printf("Error hashing password: %v", err)
+       return nil, err
+   }

    return &N3Signer{
        wallet: wallet,
        pwd:    hashedPassword,
    }
}
```

Client Response

client response : Acknowledged. This issue will be acknowledged.
We will address this in the near future. However, it needs to be said that only encryption can resolve this issue as stated. Hashing is not an option for the obvious reason that it is a one-way function.

NBR-11:Lack of handling for empty account arrays

Category	Severity	Client Response	Contributor
Language Specific	Informational	Fixed	***

Code Reference

- code/sig/neox.go#L18C1-L24C2

```
NaN: func (s *NeoXSigner) Sign(data []byte) ([]byte, error) {
NaN:     return s.wallet.SignData(data, s.wallet.GetAccounts()[0], s.pwd)
NaN: }
NaN:
NaN: func (s *NeoXSigner) SignTxWithPassphrase(tx *types.Transaction, chainId *big.Int) (*types.Tran
saction, error) {
NaN:     return s.wallet.SignTxWithPassphrase(s.wallet.GetAccounts()[0], s.pwd, tx, chainId)
NaN: }
```

Description

***:

`Sign` and `SignTxWithPassphrase` have code accesses the first account returned by GetAccounts() without checking if there are any accounts available. This could lead to runtime panics if no accounts are found.

```
func (s *NeoXSigner) Sign(data []byte) ([]byte, error) {
    return s.wallet.SignData(data, s.wallet.GetAccounts()[0], s.pwd)
}

func (s *NeoXSigner) SignTxWithPassphrase(tx *types.Transaction, chainId *big.Int) (*types.Transac
tion, error) {
    return s.wallet.SignTxWithPassphrase(s.wallet.GetAccounts()[0], s.pwd, tx, chainId)
}
```

Recommendation

***: The recommendation is made to have check to ensure existence of accounts and returning error if no account existed.

```
func (s *NeoXSigner) Sign(data []byte) ([]byte, error) {
+    accounts := s.wallet.GetAccounts()
+    if len(accounts) == 0 {
+        return nil, errors.New("no accounts found")
+    }
    return s.wallet.SignData(data, accounts[0], s.pwd)
}

func (s *NeoXSigner) SignTxWithPassphrase(tx *types.Transaction, chainId *big.Int) (*types.Transac
tion, error) {
+    accounts := s.wallet.GetAccounts()
+    if len(accounts) == 0 {
+        return nil, errors.New("no accounts found")
+    }
    return s.wallet.SignTxWithPassphrase(accounts[0], s.pwd, tx, chainId)
}
```

Client Response

client response : Fixed. This has been fixed in commit 51b4fb00dfa7c41650ac88249039b5f552034b88.

NBR-12:Goroutines in loops can cause race conditions

Category	Severity	Client Response	Contributor
Race condition	Informational	Acknowledged	***

Code Reference

- code/relayer/withdrawal.go#L41

```
41: brokerError = r.Broker.ConsumeWithdrawals(nextNonce, neoN3TokenHash, neoXTokenHash)
```

Description

***: The use of goroutines within loops in line#26-68 could introduce race conditions, especially when accessing shared resources like r.Broker.

```
func (r *WithdrawalRelayer) Start() {
    r.Logger.Info("Starting withdrawal relayer")
    neoN3TokenHash := r.NeoN3TokenHash
    neoXTokenHash := r.NeoXTokenHash
    for {
        go func() {
            // In case the relayer crashes after storing 5 signatures but before successfully rela
ying the distribution, the relayer should trigger a relay on startup before waiting for new messag
es in the queue.
            if r.relayTriggered() {
                relayError := r.relayWithdrawals()
                if relayError != nil {
                    r.Logger.Error("Failed to relay withdrawal", relayError.Error(), relayError)
                    os.Exit(1)
                }
            }

            var nextNonce uint64
            var brokerError error
            var relayError error
            for {
                nextNonce = r.getLastNonce() + 1
                brokerError = r.Broker.ConsumeWithdrawals(nextNonce, neoN3TokenHash, neoXTokenHas
h)
                if brokerError != nil {
                    r.Logger.Error("Failed to consume withdrawals", brokerError.Error(), brokerErr
or)
                    // Todo: Wrap errors to check if it is a connection error. If it's not, issue
a graceful shutdown. Otherwise, return.
                    if strings.Contains(brokerError.Error(), "connection error") {
                        r.ConnectionErr <- brokerError
                        return
                    }
                    return
                }

                relayError = r.relayWithdrawals()
                if relayError != nil {
                    r.Logger.Error("Failed to relay withdrawals", relayError.Error(), relayError)
                    os.Exit(1)
                }

                // r.config.QueueCleanupEnabled should be set to false by default in order to keep
queues for debugging reasons.
                // If the relayer is running in production, it should be set to true.
                if *r.Config.QueueCleanupEnabled {
                    // Run queue cleanup routine
```

Recommendation

***: It is recommedned to have synchronization primitives such as mutexes or channels to control access to shared resources.
Introduce a mutex to protect the shared resource `r.Broker` from concurrent access. A mutex ensures that only one goroutine can access the protected section of code at a time.
First, define a mutex in the WithdrawalRelayer struct:

```
type WithdrawalRelayer struct {
    Config        config.RelayerConfig
    Broker        *broker.Broker
    ConnectionErr chan error

    RelayerStore *store.RelayerStore
    Logger       *slog.Logger

    BridgeAddress util.Uint160
    Signer        sig.N3Signer
    Client        *node.N3Client

    NeoN3TokenHash *util.Uint160
    NeoXTokenHash  *util.Uint160

+   BrokerMutex     sync.Mutex

}
```

Then apply by locking the mutex before accessing r.Broker and unlock it afterward:

```
func (r *WithdrawalRelayer) Start() {
    r.Logger.Info("Starting withdrawal relayer")
    neoN3TokenHash := r.NeoN3TokenHash
    neoXTokenHash := r.NeoXTokenHash
+   r.BrokerMutex.Lock()
+   defer r.BrokerMutex.Unlock()

    for {
        go func() {
            // In case the relayer crashes after storing 5 signatures but before successfully rela
ying the distribution, the relayer should trigger a relay on startup before waiting for new messag
es in the queue.
            if r.relayTriggered() {
                relayError := r.relayWithdrawals()
                if relayError != nil {
                    r.Logger.Error("Failed to relay withdrawal", relayError.Error(), relayError)
                    os.Exit(1)
                }
            }

            var nextNonce uint64
            var brokerError error
            var relayError error
            for {
                nextNonce = r.getLastNonce() + 1
                brokerError = r.Broker.ConsumeWithdrawals(nextNonce, neoN3TokenHash, neoXTokenHas
h)
                if brokerError != nil {
                    r.Logger.Error("Failed to consume withdrawals", brokerError.Error(), brokerErr
or)
                    // Todo: Wrap errors to check if it is a connection error. If it's not, issue
a graceful shutdown. Otherwise, return.
                    if strings.Contains(brokerError.Error(), "connection error") {
                        r.ConnectionErr <- brokerError
                        return
                    }
                    return
                }

                relayError = r.relayWithdrawals()
                if relayError != nil {
                    r.Logger.Error("Failed to relay withdrawals", relayError.Error(), relayError)
                    os.Exit(1)
                }

                // r.config.QueueCleanupEnabled should be set to false by default in order to keep
queues for debugging reasons.
```

Client Response

client response : Acknowledged. This issue will be acknowledged.
The broker in question is not shared outside of the WithdrawalRelayer and the functions within the WithdrawalRelayer do not write to the value at this pointer except at initialization. Thus, there is no need for it. However, it is a valid statement to be cautious with such pointer values. We will take that into consideration. Especially, once we decide to share a single broker instance among different relayer instances, this topic will arise and we will introduce a mutex eventually.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3's prior written consent in each instance.

This report is not an "endorsement" or "disapproval" of any particular project or team. This report is not an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3's position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.