Forwarding Module

Noble

HALBORN

Forwarding Module - Noble

Prepared by: HALBORN

Last Updated 03/20/2024

Date of Engagement by: March 1st, 2024 - March 15th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
9	0	0	2	1	6

TABLE OF CONTENTS

Introduction

`The Noble team` engaged Halborn to conduct a security assessment on their modules, beginning on Halborn to conduct a security assessment on the forwarding module, beginning on *03/01/2024* and ending on *03/15/2024*. The security assessment was scoped to the sections of code that pertain to the forwarding module. Commit hashes and further details can be found in the Scope section of this report.

Assessment Summary

Halborn was provided 2 weeks for the engagement and assigned 1 full-time security engineer to review the security of the smart contracts in scope. The engineer is a blockchain and smart contract security experts with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.
The purpose of the assessment is to:
- Ensure that the **Noble Forwarding Module** operates as intended.
- Identify potential security issues with the custom modules used in the Noble Chain.
In summary, Halborn identified several security concerns that were mostly addressed by the Noble team. The main ones were the following:
1. **Channel State Verification Missing**: No check on channel state during account registration, raising risks.
2. **Potential Misforwarding of Tokens**: Lack of channel state verification in the ExecuteForwards function could misdirect tokens.
3. **Lack of Specifications**: Absence of detailed module specifications.
4. **No Retry Mechanism**: Failed automatic forwards lack a retry feature.
5. **No Blocklisting for Forward Recipients**: Missing mechanism to block unwanted recipient addresses.
6. **Absence of Account Deregistration**: No feature to deregister accounts in the forwarding module.
7. **Missing CLI Command Descriptions**: Lack of usage descriptions for transaction commands in the CLI.
8. **Insufficient Recipient Validation**: Inadequate validation of recipient addresses in the MsgRegisterAccount operation.

Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the custom modules. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of structures and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment:
- Research into architecture and purpose.
- Static Analysis of security for scoped repository, and imported functions. (e.g., `staticcheck`, `gosec`, `unconvert`, `codeql`, `ineffassign` and `semgrep`)
- Manual Assessment for discovering security vulnerabilities on the codebase.
- Ensuring the correctness of the codebase.
- Dynamic Analysis of files and modules related to the **Forwarding Module.**

Out-Of-Scope

- External libraries and financial-related attacks.
- New features/implementations after/with the **remediation commit IDs**.

1. RISK METHODOLOGY

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

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

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

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

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

1.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

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

ATTACK COMPLEXITY (AX):

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

METRICS:

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

EXPLOITABILIY METRIC (m_e)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

1.2 IMPACT

CONFIDENTIALITY (C):

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

INTEGRITY (I):

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

AVAILABILITY (A):

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

DEPOSIT (D):

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

YIELD (Y):

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

METRICS:

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

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

Impact I is calculated using the following formula:

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

1.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

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

SCOPE (S):

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

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

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

The score is rounded up to 1 decimal places.

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

2. SCOPE

FILES AND REPOSITORY

(a) Repository: noble

(b) Assessed Commit ID: 9c1e703

(c) Items in scope:

- forwarding

Out-of-Scope:

REMEDIATION COMMIT ID:

- bb9ee09bb9ee09
- 3165e713165e71
- 5bfd1485bfd148
- 29e2a4429e2a44

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

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	2	1	6

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-04 - CHANNEL STATE VERIFICATION MISSING IN ACCOUNT REGISTRATION PROCESS	Medium	SOLVED - 03/18/2024
HAL-05 - POTENTIAL MISFORWARDING OF TOKENS DUE TO UNVERIFIED CHANNEL STATE IN EXECUTEFORWARDS FUNCTION	Medium	SOLVED - 03/18/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-02 - LACK OF SPEC ON THE MODULES	Low	RISK ACCEPTED - 03/18/2024
HAL-01 - IMPLEMENT RETRY MECHANISM FOR FAILED AUTOMATIC FORWARDS	Informational	ACKNOWLEDGED - 03/12/2024
HAL-03 - LACK OF BLOCKLISTING MECHANISM FOR FORWARD RECIPIENT ADDRESSES IN AUTOMATIC FORWARDS FUNCTION	Informational	ACKNOWLEDGED - 03/18/2024
HAL-06 - ABSENCE OF ACCOUNT DEREGISTRATION FEATURE IN FORWARDING MODULE	Informational	SOLVED - 03/18/2024
HAL-07 - MISSING USAGE DESCRIPTION FOR ALL TRANSACTION COMMANDS CLI	Informational	SOLVED - 03/18/2024
HAL-08 - INSUFFICIENT RECIPIENT VALIDATION IN MSGREGISTERACCOUNT OPERATION	Informational	ACKNOWLEDGED - 03/18/2024
HAL-09 - IMPROPER ACCOUNT EXISTENCE VERIFICATION IN ACCOUNT REGISTRATION PROCESS	Informational	SOLVED - 03/18/2024

4. FINDINGS & TECH DETAILS

4.1 (HAL-04) CHANNEL STATE VERIFICATION MISSING IN ACCOUNT REGISTRATION PROCESS

// MEDIUM

Description

The current implementation of the `RegisterAccount` function in the Keeper module does not validate the state of the channel when registering a new account. It only checks for the existence of the channel without considering its operational state. This oversight could potentially lead to the registration of accounts linked to inactive or closed channels, resulting in operational inconsistencies and complications in the handling of forwards.

```
func (k *Keeper) RegisterAccount(goCtx context.Context, msg
*types.MsgRegisterAccount) (*types.MsgRegisterAccountResponse, error) {
    ctx := sdk.UnwrapSDKContext(goCtx)
    address := types.GenerateAddress(msg.Channel, msg.Recipient)

    if _, found := k.channelKeeper.GetChannel(ctx, transfertypes.PortID,
msg.Channel); !found {
        return nil, fmt.Errorf("channel does not exist: %s", msg.Channel)
    }

    if k.authKeeper.HasAccount(ctx, address) {
        switch account := k.authKeeper.GetAccount(ctx, address).(type) {
        case *authtypes.BaseAccount:
            k.authKeeper.SetAccount(ctx, &types.ForwardingAccount{
                BaseAccount: account,
                Channel:     msg.Channel,
                Recipient:   msg.Recipient,
                CreatedAt:   ctx.BlockHeight(),
            })

            k.IncrementNumOfAccounts(ctx, msg.Channel)
        case *types.ForwardingAccount:
            return nil, errors.New("account has already been registered")
        default:
            break
        }

        if !k.bankKeeper.GetAllBalances(ctx, address).IsZero() {
            rawAccount := k.authKeeper.GetAccount(ctx, address)
            account, ok := rawAccount.(*types.ForwardingAccount)
```

```
            if ok {
                k.SetPendingForward(ctx, account)
            }
        }

        return &types.MsgRegisterAccountResponse{Address: address.String()}, nil
    }

    account := types.ForwardingAccount{
        BaseAccount: authtypes.NewBaseAccountWithAddress(address),
        Channel:     msg.Channel,
        Recipient:   msg.Recipient,
        CreatedAt:   ctx.BlockHeight(),
    }

    k.authKeeper.SetAccount(ctx, &account)
    k.IncrementNumOfAccounts(ctx, msg.Channel)

    return &types.MsgRegisterAccountResponse{Address: address.String()}, nil
}
```

Registering accounts without validating the channel's state may lead to the accumulation of accounts associated with non-operational channels. This can cause confusion, hinder the efficient management of accounts and channels, and potentially impact the system's reliability. Moreover, it increases the complexity of troubleshooting and managing the state of accounts and their corresponding channels.

Proof of Concept

```
func TestForwarding_RegisterOnNoble(t *testing.T) {
    t.Parallel()

    ctx, wrapper, gaia, sender, receiver := ForwardingSuite(t)
    validator := wrapper.chain.Validators[0]

    address, exists := ForwardingAccount(t, ctx, validator, receiver)
    require.False(t, exists)

    _, err := validator.ExecTx(ctx, sender.KeyName(), "forwarding", "register-
account", "channel-999999", receiver.FormattedAddress())
    require.NoError(t, err)

    _, exists = ForwardingAccount(t, ctx, validator, receiver)
    require.True(t, exists)

    require.NoError(t, validator.SendFunds(ctx, sender.KeyName(), ibc.WalletAmount{
        Address: address,
```

```
        Denom:   "uusdc",
        Amount:  1_000_000,
    }))
    require.NoError(t, testutil.WaitForBlocks(ctx, 10, wrapper.chain, gaia))

    senderBalance, err := wrapper.chain.AllBalances(ctx, sender.FormattedAddress())
    require.NoError(t, err)
    require.True(t, senderBalance.IsZero())

    balance, err := wrapper.chain.AllBalances(ctx, address)
    require.NoError(t, err)
    require.True(t, balance.IsZero())

    receiverBalance, err := gaia.GetBalance(ctx, receiver.FormattedAddress(),
transfertypes.DenomTrace{
        Path:      "transfer/channel-0",
        BaseDenom: "uusdc",
    }.IBCDenom())
    require.NoError(t, err)
    require.Equal(t, int64(1_000_000), receiverBalance)
}
```

BVSS

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:N/Y:N/R:P/S:C (4.7)

Recommendation

To mitigate potential issues arising from the registration of accounts linked to non-operational channels, it is recommended to enhance the account registration validation process by including a check for the channel's state. Specifically, the registration process should proceed only if the channel is found and its state is `OPEN`. This ensures that accounts are only registered and associated with active and operational channels.

Remediation Plan

SOLVED : The **Noble team** solved the issue by adding channel status check.

Remediation Hash

https://github.com/noble-assets/noble/commit/bb9ee097f09ce3e923242e9bac3ac997f0d44f8b

4.2 (HAL-05) POTENTIAL MISFORWARDING OF TOKENS DUE TO UNVERIFIED CHANNEL STATE IN EXECUTEFORWARDS FUNCTION

// MEDIUM

Description

The current implementation of the `ExecuteForwards` function in our forwarding module does not verify the state of the IBC channel before attempting to forward tokens. This oversight may lead to situations where the `transferKeeper.SendTransfer` method is called even if the channel is not in an **OPEN** state, potentially causing token misforwarding or lost funds due to the optimistic sending mechanism described in the IBC specification (https://github.com/cosmos/ibc/blob/main/spec/core/ics-004-channel-and-packet-semantics/README.md#sending-packets).

```
// ExecuteForwards is an end block hook that clears all pending forwards from
transient state.
func (k *Keeper) ExecuteForwards(ctx sdk.Context) {
    forwards := k.GetPendingForwards(ctx)
    if len(forwards) > 0 {
        k.Logger(ctx).Info(fmt.Sprintf("executing %d automatic forward(s)",
len(forwards)))
    }

    for _, forward := range forwards {
        balances := k.bankKeeper.GetAllBalances(ctx, forward.GetAddress())

        for _, balance := range balances {
            timeout := uint64(ctx.BlockTime().UnixNano()) +
transfertypes.DefaultRelativePacketTimeoutTimestamp
            err := k.transferKeeper.SendTransfer(ctx, transfertypes.PortID,
forward.Channel, balance, forward.GetAddress(), forward.Recipient,
clienttypes.ZeroHeight(), timeout)
            if err != nil {
                // TODO: Consider moving to persistent store in order to retry in
future blocks?
                k.Logger(ctx).Error("unable to execute automatic forward", "channel",
forward.Channel, "address", forward.GetAddress().String(), "amount",
balance.String(), "err", err)
            }
        }
    }

    // NOTE: As pending forwards are stored in transient state, they are
automatically cleared at the end of the block lifecycle. No further action is
```

```
required.
}
```

Proof of Concept

```
func TestForwarding_RegisterViaTransfer(t *testing.T) {
    t.Parallel()

    ctx, wrapper, gaia, _, receiver := ForwardingSuite(t)
    validator := wrapper.chain.Validators[0]

    address, exists := ForwardingAccount(t, ctx, validator, receiver)
    require.False(t, exists)

    _, err := gaia.SendIBCTransfer(ctx, "channel-0", receiver.KeyName(),
ibc.WalletAmount{
        Address: address,
        Denom:   "uatom",
        Amount:  100000,
    }, ibc.TransferOptions{
        Memo: fmt.Sprintf("{\"noble\":{\"forwarding\":{\"recipient\":\"%s\"}}}",
receiver.FormattedAddress()),
    })
    require.NoError(t, err)

    require.NoError(t, testutil.WaitForBlocks(ctx, 10, wrapper.chain, gaia))

    _, exists = ForwardingAccount(t, ctx, validator, receiver)
    require.True(t, exists)

    balance, err := wrapper.chain.AllBalances(ctx, address)
    require.NoError(t, err)
    require.True(t, balance.IsZero())

    receiverBalance, err := gaia.GetBalance(ctx, receiver.FormattedAddress(),
"uatom")
    require.NoError(t, err)
    require.Equal(t, int64(998000), receiverBalance)
}
```

BVSS

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:N/Y:N/R:P/S:C (4.7)

Recommendation

To mitigate this issue and ensure the integrity of token forwarding, it's recommended to add a channel state check before executing the forwarding logic. Specifically, the system should query the state of the specified channel using the `channelKeeper.GetChannel` method and proceed with the forwarding operation only if the channel is in an OPEN state. This approach will prevent attempts to forward tokens through channels that are not ready to handle them, safeguarding against potential issues related to token transfer and channel states.

Remediation Plan

SOLVED : The **Noble team** solved the issue by adding a channel status check.

Remediation Hash

https://github.com/noble-assets/noble/commit/bb9ee097f09ce3e923242e9bac3ac997f0d44f8b

4.3 (HAL-02) LACK OF SPEC ON THE MODULES

// LOW

Description

The spec file intends to outline the common structure for specifications within this directory. The `forwarding` module is missing spec. This documentation is segmented into developer-focused messages and end-user-facing messages. These messages may be shown to the end user (the human) at the time that they will interact with the module.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:C (3.1)

Recommendation

It is recommended that modules are fully annotated using spec for all available functionalities.

Remediation Plan

RISK ACCEPTED : The **Noble team** accepted the risk of the issue.

4.4 (HAL-01) IMPLEMENT RETRY MECHANISM FOR FAILED AUTOMATIC FORWARDS

// INFORMATIONAL

Description

The `ExecuteForwards` function in the Keeper module is responsible for executing automatic forwards of balances across channels. Currently, if an error occurs during the execution of an automatic forward, the error is logged, but there is no mechanism for retrying or persisting these failed forwards for future attempts. This can potentially lead to missed opportunities for retrying in subsequent blocks, as pending forwards are automatically cleared from transient state at the end of the block lifecycle.

```
// ExecuteForwards is an end block hook that clears all pending forwards from
transient state.
func (k *Keeper) ExecuteForwards(ctx sdk.Context) {
    forwards := k.GetPendingForwards(ctx)
    if len(forwards) > 0 {
        k.Logger(ctx).Info(fmt.Sprintf("executing %d automatic forward(s)",
len(forwards)))
    }

    for _, forward := range forwards {
        balances := k.bankKeeper.GetAllBalances(ctx, forward.GetAddress())

        for _, balance := range balances {
            timeout := uint64(ctx.BlockTime().UnixNano()) +
transfertypes.DefaultRelativePacketTimeoutTimestamp
            err := k.transferKeeper.SendTransfer(ctx, transfertypes.PortID,
forward.Channel, balance, forward.GetAddress(), forward.Recipient,
clienttypes.ZeroHeight(), timeout)
            if err != nil {
                // TODO: Consider moving to persistent store in order to retry in
future blocks?
                k.Logger(ctx).Error("unable to execute automatic forward", "channel",
forward.Channel, "address", forward.GetAddress().String(), "amount",
balance.String(), "err", err)
            }
        }
    }

    // NOTE: As pending forwards are stored in transient state, they are
automatically cleared at the end of the block lifecycle. No further action is
required.
}
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:P/S:C (0.0)

Recommendation

To address this issue, it is recommended to implement a retry mechanism for failed automatic forwards.

Remediation Plan

ACKNOWLEDGED: The **Noble team** acknowledged the issue.

4.5 (HAL-03) LACK OF BLOCKLISTING MECHANISM FOR FORWARD RECIPIENT ADDRESSES IN AUTOMATIC FORWARDS FUNCTION

// INFORMATIONAL

Description

The current implementation of the `ExecuteForwards` function in the Keeper module processes pending forwards without evaluating the recipient addresses against a blocklist. This oversight could potentially allow malicious actors or compromised addresses to receive funds automatically, posing a security risk to the system. Without a mechanism to prevent transfers to undesirable addresses, the platform's integrity could be compromised.

```
// ExecuteForwards is an end block hook that clears all pending forwards from
transient state.
func (k *Keeper) ExecuteForwards(ctx sdk.Context) {
    forwards := k.GetPendingForwards(ctx)
    if len(forwards) > 0 {
        k.Logger(ctx).Info(fmt.Sprintf("executing %d automatic forward(s)",
len(forwards)))
    }

    for _, forward := range forwards {
        balances := k.bankKeeper.GetAllBalances(ctx, forward.GetAddress())

        for _, balance := range balances {
            timeout := uint64(ctx.BlockTime().UnixNano()) +
transfertypes.DefaultRelativePacketTimeoutTimestamp
            err := k.transferKeeper.SendTransfer(ctx, transfertypes.PortID,
forward.Channel, balance, forward.GetAddress(), forward.Recipient,
clienttypes.ZeroHeight(), timeout)
            if err != nil {
                // TODO: Consider moving to persistent store in order to retry in
future blocks?
                k.Logger(ctx).Error("unable to execute automatic forward", "channel",
forward.Channel, "address", forward.GetAddress().String(), "amount",
balance.String(), "err", err)
            }
        }
    }

    // NOTE: As pending forwards are stored in transient state, they are
automatically cleared at the end of the block lifecycle. No further action is
required.
}
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:P/S:C (0.0)

Recommendation

To mitigate this risk and enhance the security of the automatic forwards process, we recommend the introduction of a blocklisting feature to forward recipient addresses.

Remediation Plan

ACKNOWLEDGED: The **Noble team** acknowledged the issue.

4.6 (HAL-06) ABSENCE OF ACCOUNT DEREGISTRATION FEATURE IN FORWARDING MODULE

// INFORMATIONAL

Description

The current implementation of the `RegisterAccount` function within the forwarding module allows for the registration of new forwarding accounts but lacks a corresponding feature to deregister or clear an account. This limitation restricts the ability to manage forwarding accounts dynamically, potentially leading to outdated or unnecessary accounts persisting within the system.

```
func (k *Keeper) RegisterAccount(goCtx context.Context, msg
*types.MsgRegisterAccount) (*types.MsgRegisterAccountResponse, error) {
    ctx := sdk.UnwrapSDKContext(goCtx)
    address := types.GenerateAddress(msg.Channel, msg.Recipient)

    if _, found := k.channelKeeper.GetChannel(ctx, transfertypes.PortID,
msg.Channel); !found {
        return nil, fmt.Errorf("channel does not exist: %s", msg.Channel)
    }

    if k.authKeeper.HasAccount(ctx, address) {
        switch account := k.authKeeper.GetAccount(ctx, address).(type) {
        case *authtypes.BaseAccount:
            k.authKeeper.SetAccount(ctx, &types.ForwardingAccount{
                BaseAccount: account,
                Channel:     msg.Channel,
                Recipient:   msg.Recipient,
                CreatedAt:   ctx.BlockHeight(),
            })

            k.IncrementNumOfAccounts(ctx, msg.Channel)
        case *types.ForwardingAccount:
            return nil, errors.New("account has already been registered")
        default:
            break
        }

        if !k.bankKeeper.GetAllBalances(ctx, address).IsZero() {
            rawAccount := k.authKeeper.GetAccount(ctx, address)
            account, ok := rawAccount.(*types.ForwardingAccount)

            if ok {
                k.SetPendingForward(ctx, account)
            }
```

```
        }

        return &types.MsgRegisterAccountResponse{Address: address.String()}, nil
    }

    account := types.ForwardingAccount{
        BaseAccount: authtypes.NewBaseAccountWithAddress(address),
        Channel:     msg.Channel,
        Recipient:   msg.Recipient,
        CreatedAt:   ctx.BlockHeight(),
    }

    k.authKeeper.SetAccount(ctx, &account)
    k.IncrementNumOfAccounts(ctx, msg.Channel)

    return &types.MsgRegisterAccountResponse{Address: address.String()}, nil
}
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:P/S:C (0.0)

Recommendation

A potential implementation approach could involve adding a new message type, such as `MsgDeregisterAccount`, and its corresponding handler within the module.

Remediation Plan

SOLVED: The **Noble team** solved the issue by adding a message.

Remediation Hash

https://github.com/noble-assets/noble/commit/3165e716a8d4e57e22db3aca4c2a48ffffe562e3

4.7 (HAL-07) MISSING USAGE DESCRIPTION FOR ALL TRANSACTION COMMANDS CLI

// INFORMATIONAL

Description

All the transaction and query commands for all the modules are missing a long message to provide description about their usage, which will be helpful for users and external developers.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:P/S:C (0.0)

Recommendation

It is recommended to specify a long message for all transaction commands. Each command should provide a description of how to use the command correctly.

Remediation Plan

SOLVED: The **Noble team** solved the issue by implementing the suggested recommendations.

Remediation Hash

https://github.com/noble-assets/noble/commit/5bfd148a21fd22c3451dd024e2a8eac6d71b0df2

4.8 (HAL-08) INSUFFICIENT RECIPIENT VALIDATION IN MSGREGISTERACCOUNT OPERATION

// INFORMATIONAL

Description

The `MsgRegisterAccount` function within the forwarding module currently lacks comprehensive validation for the `recipient` field. This oversight means that recipients can be registered without thorough checks to ensure they conform to expected formats or validation criteria. Without proper validation, there's a risk of introducing invalid or malformed recipient addresses into the system, potentially leading to errors in forwarding operations or complications in account management.

```
func (msg *MsgRegisterAccount) ValidateBasic() error {
    _, err := sdk.AccAddressFromBech32(msg.Signer)
    if err != nil {
        return errors.New("invalid signer")
    }

    if !channeltypes.IsValidChannelID(msg.Channel) {
        return errors.New("invalid channel")
    }

    return nil
}
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:P/S:C (0.0)

Recommendation

To mitigate these risks and enhance the robustness of the forwarding module, it is recommended to implement comprehensive validation for the `recipient` field within the `MsgRegisterAccount` operation. This validation should check for adherence to expected address formats and potentially other criteria to ensure the recipient's validity and integrity.

Remediation Plan

ACKNOWLEDGED: The **Noble team** acknowledged the issue.

4.9 (HAL-09) IMPROPER ACCOUNT EXISTENCE VERIFICATION IN ACCOUNT REGISTRATION PROCESS

// INFORMATIONAL

Description

The account registration process in the forwarding module does not perform comprehensive checks on the existence and state of an account before registration. Specifically, it overlooks verifying whether the account has a sequence number of `uint64(0)` and a `nil` public key. This limited verification could lead to scenarios where accounts that have previously engaged in transactions or those that are not entirely new (indicated by a non-zero sequence number or a non-nil public key) are inaccurately treated as fresh accounts and subjected to registration logic that may not be appropriate for their state.

```
func (k *Keeper) RegisterAccount(goCtx context.Context, msg
*types.MsgRegisterAccount) (*types.MsgRegisterAccountResponse, error) {
    ctx := sdk.UnwrapSDKContext(goCtx)
    address := types.GenerateAddress(msg.Channel, msg.Recipient)

    if _, found := k.channelKeeper.GetChannel(ctx, transfertypes.PortID,
msg.Channel); !found {
        return nil, fmt.Errorf("channel does not exist: %s", msg.Channel)
    }

    if k.authKeeper.HasAccount(ctx, address) {
        switch account := k.authKeeper.GetAccount(ctx, address).(type) {
        case *authtypes.BaseAccount:
            k.authKeeper.SetAccount(ctx, &types.ForwardingAccount{
                BaseAccount: account,
                Channel:     msg.Channel,
                Recipient:   msg.Recipient,
                CreatedAt:   ctx.BlockHeight(),
            })

            k.IncrementNumOfAccounts(ctx, msg.Channel)
        case *types.ForwardingAccount:
            return nil, errors.New("account has already been registered")
        default:
            break
        }

        if !k.bankKeeper.GetAllBalances(ctx, address).IsZero() {
            rawAccount := k.authKeeper.GetAccount(ctx, address)
            account, ok := rawAccount.(*types.ForwardingAccount)

            if ok {
```

```
                k.SetPendingForward(ctx, account)
            }
        }

        return &types.MsgRegisterAccountResponse{Address: address.String()}, nil
    }

    account := types.ForwardingAccount{
        BaseAccount: authtypes.NewBaseAccountWithAddress(address),
        Channel:     msg.Channel,
        Recipient:   msg.Recipient,
        CreatedAt:   ctx.BlockHeight(),
    }

    k.authKeeper.SetAccount(ctx, &account)
    k.IncrementNumOfAccounts(ctx, msg.Channel)

    return &types.MsgRegisterAccountResponse{Address: address.String()}, nil
}
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:P/S:C (0.0)

Recommendation

Introduce additional checks within the `RegisterAccount` function to verify the sequence number and public key of the account. Specifically, ensure that only accounts with a sequence number of `uint64(0)` and a `nil` public key are eligible for registration as new forwarding accounts.

Remediation Plan

SOLVED: The **Noble team** solved the issue with adding validations on the account.

Remediation Hash

https://github.com/noble-assets/noble/commit/29e2a44a3553fe24df2cd2ebfb8796c1ac6580e6

5. AUTOMATED TESTING

Halborn used automated testing techniques to enhance coverage of certain areas of the scoped component. Among the tools used were **staticcheck**, **gosec**, **semgrep**, **unconvert**, **codeql** and **nancy**. After Halborn verified all the code and scoped structures in the repository and was able to compile them correctly, these tools were leveraged on scoped structures. With these tools, Halborn can statically verify security related issues across the entire codebase.

Staticcheck

```
types/query.pb.gw.go:16:2: "github.com/golang/protobuf/descriptor" is deprecated: See the "google.golang.org/protobuf/reflect/protoreflect" package for how to obtain an EnumDescriptor or MessageDescriptor in order to programatically interact with the protobuf type system.  (SA1019)
types/query.pb.gw.go:17:2: "github.com/golang/protobuf/proto" is deprecated: Use the "google.golang.org/protobuf/proto" package instead.  (SA1019)
types/query.pb.gw.go:33:9: descriptor.ForMessage is deprecated: Not all concrete message types satisfy the Message interface. Use MessageDescriptorProto instead. If possible, the calling code should be rewritten to use protobuf reflection instead. See package "google.golang.org/protobuf/reflect/protoreflect" for details.  (SA1019)
```

Gosec

No result.

Errcheck

No result.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.