EURe Monerium

Noble

HALBORN

EURe Monerium - Noble

Prepared by: HALBORN

Last Updated 09/06/2024

Date of Engagement by: August 19th, 2024 - September 5th, 2024

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
3	0	0	1	1	1

TABLE OF CONTENTS

1. Introduction

`Noble` engaged Halborn to conduct a security assessment on their florin module, beginning on Halborn to conduct a security assessment on the forwarding module, beginning on *08/19/2024* and ending on *09/05/2024*. The security assessment was scoped to the sections of code that pertain to the halo module. Commit hashes and further details can be found in the Scope section of this report.

2. Assessment Summary

Halborn was provided 2 weeks for the engagement and assigned 1 full-time security engineer to review the security of the module 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 Florin Module** operate as intended.

- Identify potential security issues with the **Noble Florin Module** in the Noble Chain.

In summary, Halborn identified some security concerns that were addressed and acknowledged by the `Noble team`.

3. 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 in the codebase.

- Ensuring the correctness of the codebase.

- Dynamic Analysis of files and modules related to the **Florin Module.**

4. RISK METHODOLOGY

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

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

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

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

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

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

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

ATTACK COMPLEXITY (AX):

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

METRICS:

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:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

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

INTEGRITY (I):

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

AVAILABILITY (A):

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

DEPOSIT (D):

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

YIELD (Y):

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

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (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:

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

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

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

SCOPE (S):

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

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) 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

5. SCOPE

FILES AND REPOSITORY

(a) Repository: florin

(b) Assessed Commit ID: 2bb5b3b

(c) Items in scope:

- x/florin

Out-of-Scope:

REMEDIATION COMMIT ID:

- ae736bb

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	1	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MSGRECOVER NOT REGISTERED IN CODEC	MEDIUM	SOLVED - 09/04/2024
SIGNATURE REUSE ISSUE IN BURN AND RECOVER OPERATIONS	LOW	RISK ACCEPTED
INSUFFICIENT ENFORCEMENT OF MINIMUM BLACKLIST ADMINISTRATORS	INFORMATIONAL	ACKNOWLEDGED

7. FINDINGS & TECH DETAILS

7.1 MSGRECOVER NOT REGISTERED IN CODEC

// MEDIUM

Description

The `MsgRecover` message type is implemented in the `msgServer` but is not registered in the codec in the `types` package. This omission can lead to serialization and deserialization issues.In the provided codec registration code, we can see registrations for various message types, but `MsgRecover` is notably absent:

```
func RegisterLegacyAminoCodec(cdc codec.LegacyAmino) {
    // ... other registrations ...
    cdc.RegisterConcrete(&MsgBurn{}, "florin/Burn", nil)
    cdc.RegisterConcrete(&MsgMint{}, "florin/Mint", nil)
    // MsgRecover is missing here
    // ...
}func RegisterInterfaces(registry codectypes.InterfaceRegistry) {
    // ... other registrations ...
    registry.RegisterImplementations((sdk.Msg)(nil), &MsgBurn{})
    registry.RegisterImplementations((*sdk.Msg)(nil), &MsgMint{})
    // MsgRecover is missing here
    // ...
}
```

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:C (6.3)

Recommendation

Update the codec registration in the `types` package **to** include `MsgRecover`:1. In the `RegisterLegacyAminoCodec` function, add:

```
   cdc.RegisterConcrete(&MsgRecover{}, "florin/Recover", nil)
   registry.RegisterImplementations((*sdk.Msg)(nil), &MsgRecover{})
```

Remediation Progress

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

Remediation Hash

https://github.com/noble-assets/florin/commit/ae736bb16f83df9b397c510d9400aa6c5ed23f76

7.2 SIGNATURE REUSE ISSUE IN BURN AND RECOVER OPERATIONS

// LOW

Description

The `Burn` and `Recover` functions in the module are susceptible to signature reuse attacks due to the use of a static message for signature verification.

1. Static Message Problem:

The current implementation uses a fixed message for all signature verifications:
`[]byte("I hereby declare that I am the address owner.")` This static message is problematic because once a user signs this message, the signature remains valid indefinitely for that account.

2. Signature Verification Process:

The code verifies the signature using:
`if !adr36.VerifySignature( account.GetPubKey(), []byte("I hereby declare that I am the address owner."), msg.Signature, ) { return nil, types.ErrInvalidSignature }`
This verification only checks if the signature is valid for the given public key and message, without any additional context or freshness checks.

3. Reuse Scenario:

If an attacker gains access to a valid signature (through interception, compromise of a signing device, or social engineering), they can reuse this signature to repeatedly execute burn or recover operations without needing ongoing access to the user's private key.

BVSS

AO:A/AC:H/AX:L/C:L/I:N/A:N/D:N/Y:M/R:N/S:C (2.3)

Recommendation

Implement a Nonce System:

- Maintain a nonce for each account.
- Include this nonce in the message to be signed.
- Increment the nonce after each successful operation.

Remediation Progress

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

7.3 INSUFFICIENT ENFORCEMENT OF MINIMUM BLACKLIST ADMINISTRATORS

// INFORMATIONAL

Description

The current implementation of the blacklist module allows for the removal of admin accounts without ensuring a minimum number of administrators remain. This could potentially lead to a situation where all admin accounts are removed, leaving the blacklist functionality inoperable.

```
func (k blacklistMsgServer) RemoveAdminAccount(goCtx context.Context, msg
*blacklist.MsgRemoveAdminAccount) (*blacklist.MsgRemoveAdminAccountResponse, error) {
    ctx := sdk.UnwrapSDKContext(goCtx)
    _, err := k.EnsureOwner(ctx, msg.Signer)
    if err != nil {
        return nil, err
    }

    k.DeleteBlacklistAdmin(ctx, msg.Account)

    return &blacklist.MsgRemoveAdminAccountResponse{},
ctx.EventManager().EmitTypedEvent(&blacklist.AdminAccountRemoved{
        Account: msg.Account,
    })
}
```

Score

Impact:

Likelihood:

Recommendation

Implement a check to ensure a minimum number of admin accounts always exist.

Remediation Progress

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