// HALBORN

Money Mates - Smart Contract

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **October 11th, 2023 - October 13th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	10/12/2023
0.2	Document Update	10/13/2023
0.3	Draft Review	10/13/2023
1.0	Remediation Plan	10/17/2023
1.1	Remediation Plan Update	10/18/2023
1.2	Remediation Plan Review	10/18/2023

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Piotr Cielas	Halborn	Piotr.Cielas@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

The MoneyMates smart contract enables users to buy and sell shares of a subject. The system also has a referral mechanism to encourage user acquisition. When shares are bought or sold, various fees are levied and distributed to different stakeholders.

Money Mates engaged Halborn to conduct a security assessment on their smart contracts beginning on October 11th, 2023 and ending on October 13th, 2023. The security assessment was scoped to the smart contracts provided in the JediKunnow/moneymates-contracts GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

1.2 ASSESSMENT SUMMARY

Halborn was provided 3 days for the engagement and assigned a team of one full-time security engineer to review the security of the smart contracts in scope. The security team consists of 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:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some security risks, which were mostly addressed by Money Mates. The main ones were the following:
- Add a minimum amount parameter to the sellShares() function to allow users to control the slippage.
- Modify the verification checking of payment received from the buyer in the buyShares() function to also consider the referral fee.
- Deduct the refFee from the payment sent to the seller in the sellShares() function.
- Send back any overpayments to the share buyers.

- Instead of reverting the buyShares() and sellShares() functions, if any of the fee transfers fail, transfer the amount to the protocol's fee address.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (solgraph).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions (Slither).
- Testnet deployment (Foundry, Brownie).

2. 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.

2.1 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:

Exploitability 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
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:

$$E = \prod m_e$$

2.2 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)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

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

2.3 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.

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

2.4 SCOPE

Code repositories:

1. Project Name

- Repository: JediKunnow/moneymates-contracts
- Commit ID: c63d1ffbd30ef47e7c0bef30709d6bb91386afbc
- Smart contracts in scope:
    1. MoneyMates.sol (contracts/MoneyMates.sol)
- Fix commit ID: 6e3d82bc3bea884f6ed19495735aad59ea637305

Out-of-scope

- Third-party libraries and dependencies.
- Economic attacks.

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	4	0	4	6

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) MISSING SLIPPAGE CONTROL	Critical (10)	SOLVED - 10/14/2023
(HAL-02) REFFEE iS MISSING FROM PAYMENT CHECK	High (7.5)	SOLVED - 10/12/2023
(HAL-03) REFFEE IS NOT DEDUCTED FROM SELLPRICE	High (7.5)	SOLVED - 10/12/2023
(HAL-04) OVERPAYMENT IS NOT TRANSFERRED BACK TO BUYERS	High (7.5)	SOLVED - 10/18/2023
(HAL-05) DENIAL OF SERVICE	High (7.5)	SOLVED - 10/13/2023
(HAL-06) MISSING ZERO ADDRESS CHECKS	Low (2.5)	SOLVED - 10/12/2023
(HAL-07) LACK OF FEE PERCENTAGE VALIDATION	Low (2.0)	SOLVED - 10/12/2023
(HAL-08) SINGLE STEP OWNERSHIP TRANSFER PROCESS	Low (2.0)	SOLVED - 10/12/2023
(HAL-09) OWNER CAN RENOUNCE OWNERSHIP	Low (2.0)	RISK ACCEPTED
(HAL-10) LACK OF EMERGENCY STOP PATTERN IMPLEMENTATION	Informational (0.8)	ACKNOWLEDGED
(HAL-11) MISSING EVENTS FOR CONTRACT OPERATIONS	Informational (0.8)	SOLVED - 10/12/2023
(HAL-12) SHARE PRICE MANIPULATION EXPOSURE	Informational (0.0)	ACKNOWLEDGED
(HAL-13) IMPROPER OWNER CONFIGURATION	Informational (0.0)	SOLVED - 10/12/2023
(HAL-14) USING REVERT STRINGS INSTEAD OF CUSTOM ERRORS	Informational (0.0)	ACKNOWLEDGED
(HAL-15) INCOMPLETE NATSPEC DOCUMENTATION	Informational (0.0)	ACKNOWLEDGED

FINDINGS & TECH DETAILS

4.1 (HAL-01) MISSING SLIPPAGE CONTROL - CRITICAL(10)

Description:

It was identified that slippage control is missing from the sellShares() function. In the contract, the share price can change dynamically based on their popularity. Unlike the buyShares() function, where users can control the slippage by the amount of the Ether they send, in sellShares(), users can only specify how many shares they would like to sell, but not the minimum price they would want to receive after the sale.

Code Location:

There is no slippage control in the sellShares() function:

Listing 1: contracts/MoneyMates.sol

```
111     function sellShares(address sharesSubject, uint256 amount)
 ↳ public payable {
112         require(initialized == true, 'NOT_INITIALIZED_YET');
113         require(amount > 0, 'ZERO_AMOUNT');
114         require(refs[msg.sender].active == true, "Signup first");
115         address ref = refs[msg.sender].referrer;
116         uint256 supply = sharesSupply[sharesSubject];
117         require(supply > amount, "Cannot sell the last share");
118         uint256 price = getPrice(supply - amount, amount);
119         uint256 protocolFee = price * protocolFeePercent / 1 ether
 ↳ ;
120         uint256 subjectFee = price * subjectFeePercent / 1 ether;
```

Proof of Concept:

As the number of shares falls, the price of shares decreases:

```
>>> tx = moneyMates.sellShares(alice, 10, {'from': carl})
Transaction sent: 0x053bf8f3a265dc4998e425031df85255a027cea568392d50054601576d6761d1
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 4
  MoneyMates.sellShares confirmed   Block: 18339872   Gas used: 77422 (0.26%)

>>> moneyMates.getSellPriceAfterFee(alice, 10)
6265406250000000000
>>> tx = moneyMates.sellShares(alice, 10, {'from': carl})
Transaction sent: 0xbba5474bd736eb4323c331f6fb1291b89efc123eba162a3b9c35563ebaf50e65
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 5
  MoneyMates.sellShares confirmed   Block: 18339873   Gas used: 77422 (0.26%)

>>> moneyMates.getSellPriceAfterFee(alice, 10)
5134781250000000000
>>> tx = moneyMates.sellShares(alice, 10, {'from': carl})
Transaction sent: 0xd731c85dd37732deee71b72dfbf87190bcbddacc7d4f93e3056d35e8203bd5c9
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 6
  MoneyMates.sellShares confirmed   Block: 18339874   Gas used: 77422 (0.26%)

>>> moneyMates.getSellPriceAfterFee(alice, 10)
4116656250000000000
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:C/Y:N/R:N/S:U (10)

Recommendation:

A minimum amount parameter should be added to the sellShares() function to allow users to control the slippage.

Remediation Plan:

SOLVED: The Money Mates team solved the issue in commit 672f8dd by allowing users to control the slippage with the new sellMinPrice parameter.

4.2 (HAL-02) REFFEE iS MISSING FROM PAYMENT CHECK - HIGH (7.5)

Description:

It was identified that in the buyShares() function, the require statement, which checks the amount of payment sent, does not take into account whether the referral is also included in the price. As users are not obliged to pay the full price, less Ether than necessary may be transferred to the contract when people buy shares. As the shortage increases, not everybody will be able to sell their shares because there will not be enough Ether in the contract to fulfill all payments.

Code Location:

Listing 2: contracts/MoneyMates.sol (Lines 110,111)

```
100 function buyShares(address sharesSubject, uint256 amount) public
 ↳ payable {
101     require(initialized == true, 'NOT_INITIALIZED_YET');
102     require(amount > 0, 'ZERO_AMOUNT');
103     require(refs[msg.sender].active == true, "Signup first");
104     address ref = refs[msg.sender].referrer;
105     uint256 supply = sharesSupply[sharesSubject];
106     require(supply > 0 || sharesSubject == msg.sender, "Only the
 ↳ shares' subject can buy the first share");
107     uint256 price = getPrice(supply, amount);
108     uint256 protocolFee = price * protocolFeePercent / 1 ether;
109     uint256 subjectFee = price * subjectFeePercent / 1 ether;
110     uint256 refFee = price * refFeePercent / 1 ether;
111     require(msg.value >= price + protocolFee + subjectFee, "
 ↳ Insufficient payment");
112     sharesBalance[sharesSubject][msg.sender] = sharesBalance[
 ↳ sharesSubject][msg.sender] + amount;
```

Proof of Concept:

It is possible to buy shares without paying the referral fee:

```
>>> moneyMates.sharesSupply(alice)
11
>>> price = moneyMates.getPrice(11,5)
>>> payment = moneyMates.getBuyPriceAfterFee(alice, 5) - price * moneyMates.refFeePercent()//10**18
>>> tx = moneyMates.buyShares(alice, 5, {'from': carl, 'value': payment})
Transaction sent: 0xd8044f70d9b70eb82e516738dc5bc3f272bf3cfebb9cb7f71b1a625c6300e269
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 1
  MoneyMates.buyShares confirmed   Block: 18339480   Gas used: 83886 (0.28%)

>>> tx.info()
Transaction was Mined
---------------------
Tx Hash: 0xd8044f70d9b70eb82e516738dc5bc3f272bf3cfebb9cb7f71b1a625c6300e269
From: 0xD5D0a77c6cf0061e203C0b3EF070370236dF9776
To: 0x842D1E7E82B970Cc2F3f7cCfe5BA43B6785BF9c5
Value: 58246875000000000
Function: MoneyMates.buyShares
Block: 18339480
Gas Used: 83886 / 30000000 (0.3%)

Events In This Transaction
--------------------------
└── MoneyMates (0x842D1E7E82B970Cc2F3f7cCfe5BA43B6785BF9c5)
    └── Trade
        ├── trader: 0xD5D0a77c6cf0061e203C0b3EF070370236dF9776
        ├── subject: 0xE73a82998660181b314085a6421A4EA124436ae6
        ├── isBuy: True
        ├── shareAmount: 5
        ├── ethAmount: 53437500000000000
        ├── protocolEthAmount: 2404687500000000
        ├── subjectEthAmount: 2404687500000000
        ├── refEthAmount: 534375000000000
        └── supply: 16
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:H/Y:N/R:N/S:U (7.5)

Recommendation:

The payment check should also include the referral fee in the verification.

Remediation Plan:

SOLVED: The Money Mates team solved the issue in commit e8f3a16 by including the referral fee in the verification.

4.3 (HAL-03) REFFEE IS NOT DEDUCTED FROM SELLPRICE - HIGH (7.5)

Description:

It was identified that in the sellShares() function, the refFee is not deducted from the payment sent to the seller. As users are receiving more payment than they supposed, less Ether than necessary will remain in the contract when people sell shares. As the shortage increases, not everybody will be able to sell their shares because there will not be enough Ether in the contract to fulfill all payments.

Code Location:

Listing 3: contracts/MoneyMates.sol (Lines 131,136)

```
129     uint256 protocolFee = price * protocolFeePercent / 1 ether;
130     uint256 subjectFee = price * subjectFeePercent / 1 ether;
131     uint256 refFee = price * refFeePercent / 1 ether;
132     require(sharesBalance[sharesSubject][msg.sender] >= amount, "
 ↳ Insufficient shares");
133     sharesBalance[sharesSubject][msg.sender] = sharesBalance[
 ↳ sharesSubject][msg.sender] - amount;
134     sharesSupply[sharesSubject] = supply - amount;
135     emit Trade(msg.sender, sharesSubject, false, amount, price,
 ↳ protocolFee, subjectFee, refFee, supply - amount);
136     (bool success1, ) = msg.sender.call{value: price - protocolFee
 ↳  - subjectFee}("");
137     (bool success2, ) = protocolFeeDestination.call{value:
 ↳ protocolFee}("");
138     (bool success3, ) = sharesSubject.call{value: subjectFee}("");
139     (bool success4, ) = ref.call{value: refFee}("");
```

Proof of Concept:

It is possible to buy shares without paying the ref fee:

```
>>> moneyMates.sharesSupply(alice)
16
>>> balance_before = carl.balance()
>>> tx = moneyMates.sellShares(alice, 5, {'from': carl})
Transaction sent: 0xd81223844351e4c4eb89f22fa072f466dceaa7a7a9a93077773183c52e7f6982
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 2
  MoneyMates.sellShares confirmed   Block: 18339481   Gas used: 62422 (0.21%)

>>> received = carl.balance() - balance_before
>>> tx.info()
Transaction was Mined
---------------------
Tx Hash: 0xd81223844351e4c4eb89f22fa072f466dceaa7a7a9a93077773183c52e7f6982
From: 0xD5D0a77c6cf0061e203C0b3EF070370236dF9776
To: 0x842D1E7E82B970Cc2F3f7cCfe5BA43B6785BF9c5
Value: 0
Function: MoneyMates.sellShares
Block: 18339481
Gas Used: 62422 / 30000000 (0.2%)

Events In This Transaction
--------------------------
└── MoneyMates (0x842D1E7E82B970Cc2F3f7cCfe5BA43B6785BF9c5)
    └── Trade
        ├── trader: 0xD5D0a77c6cf0061e203C0b3EF070370236dF9776
        ├── subject: 0xE73a82998660181b314085a6421A4EA124436ae6
        ├── isBuy: False
        ├── shareAmount: 5
        ├── ethAmount: 53437500000000000
        ├── protocolEthAmount: 2404687500000000
        ├── subjectEthAmount: 2404687500000000
        ├── refEthAmount: 534375000000000
        └── supply: 11

>>> received == 53437500000000000 - 2404687500000000 - 2404687500000000
True
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:H/Y:N/R:N/S:U (7.5)

Recommendation:

The `refFee` should also be deducted from the payment sent to the seller.

Remediation Plan:

SOLVED: The Money Mates team solved the issue in commit `8dfdb91` by deducting the referral fee from the payment sent to the seller.

4.4 (HAL-04) OVERPAYMENT IS NOT TRANSFERRED BACK TO BUYERS - HIGH (7.5)

Description:

It was identified that in the buyShares() function, the overpayment sent by the user is not transferred back. In the contract, the share price can change dynamically based on their popularity.
Buyers are incentivized to send higher payments than the actual calculated price to make sure their order will still be fulfilled if somebody buys shares before them. The price can also be decreased if someone sells shares before them. However, it was identified that if the user sends more payment than necessary, the remaining amounts will be locked in the contract instead of transferred back to the buyer.

Code Location:

Overpayment is not sent back to the buyer:

Listing 4: contracts/MoneyMates.sol

```
111     require(msg.value >= price + protocolFee + subjectFee, "
 ↳ Insufficient payment");
112     sharesBalance[sharesSubject][msg.sender] = sharesBalance[
 ↳ sharesSubject][msg.sender] + amount;
113     sharesSupply[sharesSubject] = supply + amount;
114     emit Trade(msg.sender, sharesSubject, true, amount, price,
 ↳ protocolFee, subjectFee, refFee, supply + amount);
115     (bool success1, ) = protocolFeeDestination.call{value:
 ↳ protocolFee}("");
116     (bool success2, ) = sharesSubject.call{value: subjectFee}("");
117     (bool success3, ) = ref.call{value: refFee}("");
118     require(success1 && success2 && success3, "Unable to send
 ↳ funds");
```

Proof of Concept:

The test user paid three times the price and did not get the overpayment back:

```
>>> moneyMates.sharesSupply(alice)
11
>>> balance_before = carl.balance()
>>> payment = moneyMates.getBuyPriceAfterFee(alice, 5)
>>> tx = moneyMates.buyShares(alice, 5, {'from': carl, 'value': payment * 3})
Transaction sent: 0x67f2f6171db4a9029c30419c51f39cd6b7e8cd14f2e0f54d8594e553029a0fe2
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 1
  MoneyMates.buyShares confirmed   Block: 18339869   Gas used: 83886 (0.28%)

>>> tx.info()
Transaction was Mined
---------------------
Tx Hash: 0x67f2f6171db4a9029c30419c51f39cd6b7e8cd14f2e0f54d8594e553029a0fe2
From: 0x39fF3f4B9CFAC2E4284d55cA703F3C520dDb5D7D
To: 0x1E77319eF8e86d9067Da0D8841ADB46B12a7eAFC
Value: 176343750000000000
Function: MoneyMates.buyShares
Block: 18339869
Gas Used: 83886 / 30000000 (0.3%)

Events In This Transaction
--------------------------
└── MoneyMates (0x1E77319eF8e86d9067Da0D8841ADB46B12a7eAFC)
    └── Trade
        ├── trader: 0x39fF3f4B9CFAC2E4284d55cA703F3C520dDb5D7D
        ├── subject: 0x161EFa1bb67f344787af3Bf2E4200F679826F6Be
        ├── isBuy: True
        ├── shareAmount: 5
        ├── ethAmount: 53437500000000000
        ├── protocolEthAmount: 2404687500000000
        ├── subjectEthAmount: 2404687500000000
        ├── refEthAmount: 534375000000000
        └── supply: 16

>>> payment
58781250000000000
>>> balance_before - payment * 3 == carl.balance()
True
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:H/Y:N/R:N/S:U (7.5)

Recommendation:

Any overpayments should be sent back to the share buyers.

Remediation Plan:

SOLVED: The Money Mates team solved the issue in commit 6e3d82b by sending the overpayment back to the msg.sender.

4.5 (HAL-05) DENIAL OF SERVICE - HIGH (7.5)

Description:

Users have to sign up using a referrer before buying or selling shares. The referrer receives a fee after the buy and sell transactions made by the referred users. If the Ether transfer to the referrer fails, the corresponding buy or sell transaction will revert. A malicious user might exploit this behavior, deploy a contract, and use its address to invite people. The contract can prevent referred users from buying or selling shares if it stops accepting Ether transfers. The malicious user can potentially ask for ransom from the referred users to allow them to trade.

Note that the share's subject can also revert the transactions similarly.

Code Location:

The buyShares() and sellShares() functions revert if any of the Ether transfers transaction fails:

Listing 5: contracts/MoneyMates.sol (Lines 116-118)

```
115     (bool success1, ) = protocolFeeDestination.call{value:
 ↳ protocolFee}("");
116     (bool success2, ) = sharesSubject.call{value: subjectFee}("");
117     (bool success3, ) = ref.call{value: refFee}("");
118     require(success1 && success2 && success3, "Unable to send
 ↳ funds");
```

Listing 6: contracts/MoneyMates.sol (Lines 138-140)

```
136         (bool success1, ) = msg.sender.call{value: price -
 ↳ protocolFee - subjectFee}("");
137         (bool success2, ) = protocolFeeDestination.call{value:
 ↳ protocolFee}("");
```

```
138         (bool success3, ) = sharesSubject.call{value: subjectFee}(
 ↳ "");
139         (bool success4, ) = ref.call{value: refFee}("");
140         require(success1 && success2 && success3 && success4, "
 ↳ Unable to send funds");
141 }
```

Proof of Concept:

The malicious referrer can turn on and off the referred user's trading:

```
>>> moneyMates.refs(carl)['referrer'] == malicious
True
>>> payment = moneyMates.getBuyPriceAfterFee(alice, 10)
>>> tx = moneyMates.buyShares(alice, 10, {'from': carl, 'value': payment})
Transaction sent: 0x2f4577da23dfd4ba3fc10b90a3fcecbef219de4977da20c186edec30c952385e
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 2
  MoneyMates.buyShares confirmed   Block: 18340203   Gas used: 84776 (0.28%)

>>> tx = moneyMates.sellShares(alice, 1, {'from': carl})
Transaction sent: 0xbb9a606db449b8ae1d408863e314f20ecbca7dae1f1d628aba78947f33f19dcb
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 3
  MoneyMates.sellShares confirmed   Block: 18340204   Gas used: 78312 (0.26%)

>>> malicious.lock()
Transaction sent: 0xa3ed6f3af40a246e6ca906743c7796fd72cbb55d8637b41f007cd5e43b52cc0d
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 1
  Malicious.lock confirmed   Block: 18340205   Gas used: 13507 (0.05%)

<Transaction '0xa3ed6f3af40a246e6ca906743c7796fd72cbb55d8637b41f007cd5e43b52cc0d'>
>>> tx = moneyMates.sellShares(alice, 1, {'from': carl})
Transaction sent: 0x5d4848098c5cb1bf1be6832aa10893e9dfdc686a6c87e2478a7b2dcf8ab03a8f
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 4
  MoneyMates.sellShares confirmed (Unable to send funds)   Block: 18340206   Gas used: 78584 (0.26%)

>>> malicious.unlock()
Transaction sent: 0x4e12ed242a8e677c62d7b0691ed2c2cabf7d51effaf2cf2b318158575c2a50af
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 2
  Malicious.unlock confirmed   Block: 18340207   Gas used: 41998 (0.14%)

<Transaction '0x4e12ed242a8e677c62d7b0691ed2c2cabf7d51effaf2cf2b318158575c2a50af'>
>>> tx = moneyMates.sellShares(alice, 1, {'from': carl})
Transaction sent: 0x55e51e148c0edb351401e4167e53aefef733a0b67c633c9014ba2999e8ef531d
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 5
  MoneyMates.sellShares confirmed   Block: 18340208   Gas used: 78312 (0.26%)
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:H/D:N/Y:N/R:N/S:U (7.5)

Recommendation:

Instead of reverting the buyShares() and sellShares() functions, if any of the fee transfers fail, the amount should be transferred to the protocolFeeDestination address.

Remediation Plan:

SOLVED: The Money Mates team solved the issue in commit 7b24cce by sending the fee to the protocolFeeDestination address if the transfers fail.

4.6 (HAL-06) MISSING ZERO ADDRESS CHECKS - LOW (2.5)

Description:

It was identified that the parameters of the initialize() and setFeeDestination() functions lack zero address validation.

Code Location:

Listing 7: contracts/MoneyMates.sol

```
34 function initialize(address _feeDestination) public initializer {
35     __Ownable_init(_feeDestination);
36     refs[_feeDestination].active = true;
37     setFeeDestination(_feeDestination);
```

Listing 8: contracts/MoneyMates.sol

```
44 function setFeeDestination(address _feeDestination) public
↳ onlyOwner {
45     protocolFeeDestination = _feeDestination;
46 }
```

Code Location:

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:N/S:U (2.5)

Recommendation:

It is recommended to add zero address validation for the `_feeDestination` parameter.

Remediation Plan:

SOLVED: The Money Mates team solved the issue in commit `e8f3a16` by adding zero address validation to the above parameters.

4.7 (HAL-07) LACK OF FEE PERCENTAGE VALIDATION - LOW (2.0)

Description:

It was identified that the `setProtocolFeePercent()`, `setSubjectFeePercent()`, and `setRefFeePercent()` functions lack validation of the fee percentage parameter. For this reason, it is possible to set fee values higher than 100%.

Code Location:

Listing 9: contracts/MoneyMates.sol

```
48 function setProtocolFeePercent(uint256 _feePercent) public
↳ onlyOwner {
49     protocolFeePercent = _feePercent;
50 }
51
52 function setSubjectFeePercent(uint256 _feePercent) public
↳ onlyOwner {
53     subjectFeePercent = _feePercent;
54 }
55
56 function setRefFeePercent(uint256 _feePercent) public onlyOwner {
57     refFeePercent = _feePercent;
58 }
```

Code Location:

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:C/Y:N/R:N/S:U (2.0)

Recommendation:

It is recommended to set the maximum value for these fee percentages. It is also recommended to add another check to ensure that the sum of the fees does not exceed a maximum value.

Remediation Plan:

SOLVED: The Money Mates team solved the issue in commit e8f3a16 by setting the maximum value for the fee percentages.

4.8 (HAL-08) SINGLE STEP OWNERSHIP TRANSFER PROCESS - LOW (2.0)

Description:

The ownership of the contracts can be lost as the MoneyMates contract is inherited from the OwnableUpgradeable contract and their ownership can be transferred in a single-step process. The address the ownership is changed to should be verified to be active or willing to act as the owner.

Code Location:

Listing 10: @openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol

```
102 function transferOwnership(address newOwner) public virtual
 ↳ onlyOwner {
103     if (newOwner == address(0)) {
104         revert OwnableInvalidOwner(address(0));
105     }
106     _transferOwnership(newOwner);
107 }
```

Listing 11: @openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol

```
113 function _transferOwnership(address newOwner) internal virtual {
114     OwnableStorage storage $ = _getOwnableStorage();
115     address oldOwner = $._owner;
116     $._owner = newOwner;
117     emit OwnershipTransferred(oldOwner, newOwner);
118 }
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:C/Y:N/R:N/S:U (2.0)

Recommendation:

Consider using the Ownable2StepUpgradeable library over the OwnableUpgradeable library or implementing similar two-step ownership transfer logic into the contract.

Remediation Plan:

SOLVED: The Money Mates team solved the issue in commit e8f3a16 by using the Ownable2StepUpgradeable library over the OwnableUpgradeable library.

4.9 (HAL-09) OWNER CAN RENOUNCE OWNERSHIP - LOW (2.0)

Description:

The MoneyMates contract is inherited from the OwnableUpgradeable contract. The Owner of the contract is usually the account that deploys the contract. As a result, the Owner can perform some privileged functions. In the Ownable contracts, the renounceOwnership() function is used to renounce the Owner permission. Renouncing ownership before transferring would result in the contract having no Owner, eliminating the ability to call privileged functions.

Code Location:

Listing 12: @openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol

```
94 function renounceOwnership() public virtual onlyOwner {
95     _transferOwnership(address(0));
96 }
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:C/Y:N/R:N/S:U (2.0)

Recommendation:

It is recommended that the Owner cannot call renounceOwnership() without first transferring Ownership to another address. In addition, if a multi-signature wallet is used, the call to the renounceOwnership() function should be confirmed for two or more users.

Remediation Plan:

RISK ACCEPTED: The Money Mates team made a business decision to accept the risk of this finding, to leave the possibility to renounce the ownership and leave the contract decentralized.

4.10 (HAL-10) LACK OF EMERGENCY STOP PATTERN IMPLEMENTATION - INFORMATIONAL (0.8)

Description:

It was identified that the Pausable module is not used in the MoneyMates contract. Emergency stop patterns allow the project team to pause crucial functionalities, while being in a state of emergency, e.g., being under adversary attack. In the case the emergency stop pattern is not used, critical functions cannot be temporarily disabled.

BVSS:

AO:A/AC:L/AX:H/C:N/I:N/A:N/D:L/Y:N/R:N/S:U (0.8)

Recommendation:

It is recommended to apply the emergency stop pattern in the MoneyMates contract.

Remediation Plan:

ACKNOWLEDGED: The Money Mates team acknowledged this finding.

4.11 (HAL-11) MISSING EVENTS FOR CONTRACT OPERATIONS - INFORMATIONAL (0.8)

Description:

It was identified that the setFeeDestination(), setProtocolFeePercent(), setSubjectFeePercent(), setRefFeePercent() and signup() functions from the MoneyMates contract do not emit any events. As a result, blockchain monitoring systems might not be able to timely detect suspicious behaviors related to these functions.

It is also noted that it is more difficult to track signups if an event is not emitted in the signup() function.

BVSS:

AO:A/AC:L/AX:H/C:N/I:N/A:N/D:L/Y:N/R:N/S:U (0.8)

Recommendation:

Adding events for all important operations is recommended to help monitor the contracts and detect suspicious behavior. A monitoring system that tracks relevant events would allow the timely detection of compromised system components.

Remediation Plan:

SOLVED: The Money Mates team solved the issue in commit e8f3a16 by adding events for all important operations.

4.12 (HAL-12) SHARE PRICE MANIPULATION EXPOSURE - INFORMATIONAL (0.0)

Description:

After signing up, users can buy their own first share and become a subject. This purchase initializes the price to a starting amount. However, as the number of shares held increases, so does the price of the shares. This encourages people to buy as many shares as they can immediately after the subject's first share is created. This allows the first buyers to have massive control over the supply at a very low price, get a huge advantage by owning many shares, and earn considerable profits by selling them.

It is also identified that the price is increasing very rapidly as more and more users buy shares, and the price has no cap. As a result, users who could not buy in time may be priced out of the market. This carries the risk that only a few users will be able to buy shares, which discourages users from the protocol.

Code Location:

The price calculation is based on the shares:

Listing 13: contracts/MoneyMates.sol

```
60 function getPrice(uint256 supply, uint256 amount) public pure
↳ returns (uint256) {
61     uint256 sum1 = supply == 0 ? 0 : (supply - 1 )* (supply) * (2
↳ * (supply - 1) + 1) / 6;
62     uint256 sum2 = supply == 0 && amount == 1 ? 0 : (supply - 1 +
↳ amount) * (supply + amount) * (2 * (supply - 1 + amount) + 1) / 6;
63     uint256 summation = sum2 - sum1;
64     return summation * 1 ether / 16000;
65 }
```

Proof of Concept:

The first 30 shares are cheaper to buy than the following 10.

```
>>> moneyMates.getPrice(1,30) / 10**18 # getprice(supply,amount)
0.5909375
>>> moneyMates.getPrice(31,10) / 10**18 # getprice(supply,amount)
0.7928125
```

The share price rises rapidly as their number increases:

```
>>> moneyMates.getPrice(1,30) / 10**18 # getprice(supply,amount)
0.5909375
>>> moneyMates.getPrice(31,10) / 10**18 # getprice(supply,amount)
0.7928125
>>> moneyMates.getPrice(100,10) / 10**18 # getprice(supply,amount)
6.8303125
>>> moneyMates.getPrice(200,10) / 10**18 # getprice(supply,amount)
26.1428125
>>> moneyMates.getPrice(300,10) / 10**18 # getprice(supply,amount)
57.9553125
```

BVSS:

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

Recommendation:

The current share price algorithm should be reviewed to lower the price manipulation risk and create a more inclusive system. Halborn recommends the following:

- Review the current price structure and the rate of price increase.
- Converge the share price to a target price or cap the prices.
- Enable the subject to buy more shares initially before opening their market.

Remediation Plan:

ACKNOWLEDGED: The Money Mates team acknowledged this finding. The team states that the pricing model is intended to be part of this protocol. To mitigate the impact, the team added the option to limit the maximum size of the trades in commit 7b24cce, the maximum size is 10 by default.

4.13 (HAL-13) IMPROPER OWNER CONFIGURATION - INFORMATIONAL (0.0)

Description:

It identified that the Owner role in the initializer() function is configured for the _feeDestination address instead of the msg.sender. This causes the function to revert if not the _feeDestination wallet initializes it, as the setter functions called from the initializer can only be called by the Owner.

Code Location:

The _feeDestination is initialized as Owner, not the msg.caller:

Listing 14: contracts/MoneyMates.sol

```
34 function initialize(address _feeDestination) public initializer {
35     __Ownable_init(_feeDestination);
36     refs[_feeDestination].active = true;
37     setFeeDestination(_feeDestination);
38     setProtocolFeePercent(45000000000000000); // 0.045
39     setSubjectFeePercent(45000000000000000); // 0.045
40     setRefFeePercent(10000000000000000); // 0.01
41     initialized = true;
42 }
```

Only the Owner can call the setter functions:

Listing 15: contracts/MoneyMates.sol

```
44     function setFeeDestination(address _feeDestination) public
↳ onlyOwner {
45         protocolFeeDestination = _feeDestination;
46     }
```

Code Location:

BVSS:

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

Recommendation:

The `msg.sender` should be initialized as an `Owner` instead of the `_feeDestination` address.

Remediation Plan:

SOLVED: The Money Mates team solved the issue in commit `e8f3a16` by initializing the `msg.sender` as an `Owner` instead of the `_feeDestination` address.

4.14 (HAL-14) USING REVERT STRINGS INSTEAD OF CUSTOM ERRORS - INFORMATIONAL (0.0)

Description:

Starting from Solidity v0.8.4, there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. If the revert string uses strings to provide additional information about failures (e.g. `require(supply > amount, "Cannot sell the last share");`), but they are rather expensive, especially when it comes to deploying cost, and it is difficult to use dynamic information in them.

BVSS:

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

Recommendation:

Consider implementing custom errors instead of reverting strings.

An example implementation of the initialization checks using custom errors:

Listing 16: Using Custom Errors

```
error CannotSellTheLastShare();

function sellShares(address sharesSubject, uint256 amount) public
↳ payable {
    ...
    // require(supply > amount, "Cannot sell the last share");
    if (supply <= amount) revert CannotSellTheLastShare();
    ...
```

Remediation Plan:

ACKNOWLEDGED: The Money Mates team acknowledged this finding.

4.15 (HAL-15) INCOMPLETE NATSPEC DOCUMENTATION - INFORMATIONAL (0.0)

Description:

Natspec documentation is useful for internal developers that need to work on the project, external developers that need to integrate with the project, security professionals that have to review it but also for end users given that many chain explorers have officially integrated the support for it directly on their site. It was detected that the MoneyMates contract has an incomplete **natspec** documentation.

BVSS:

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

Recommendation:

Consider adding the missing **natspec** documentation, adhering to the format guideline included in Solidity documentation.

Remediation Plan:

ACKNOWLEDGED: The Money Mates team acknowledged this finding.

AUTOMATED TESTING

5.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their ABIs and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.
The security team assessed all findings identified by the Slither software, however, findings with severity Information and Optimization are not included in the below results for the sake of report readability.

Results:

contracts/MoneyMates.sol

Slither results for MoneyMates	
Finding	**Impact**
MoneyMates.setSubjectFeePercent(uint256) (contracts/MoneyMates.sol#52-54) should emit an event for: - subjectFeePercent = _feePercent (contracts/MoneyMates.sol#53)	Low
MoneyMates.setProtocolFeePercent(uint256) (contracts/MoneyMates.sol#48-50) should emit an event for: - protocolFeePercent = _feePercent (contracts/MoneyMates.sol#49)	Low
MoneyMates.setRefFeePercent(uint256) (contracts/MoneyMates.sol#56-58) should emit an event for: - refFeePercent = _feePercent (contracts/MoneyMates.sol#57)	Low
MoneyMates.setFeeDestination(address)._feeDestination (contracts/MoneyMates.sol#44) lacks a zero-check on : - protocolFeeDestination = _feeDestination (contracts/MoneyMates.sol#45)	Low
End of table for MoneyMates	

Results summary:

The findings obtained as a result of the Slither scan were reviewed and added to the report.

THANK YOU FOR CHOOSING

// HALBORN