Truffles - NFT Invoice

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **July 24th, 2023 - July 31st, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY 4
CONTACTS 5
1 EXECUTIVE OVERVIEW 6
1.1 INTRODUCTION 7
1.2 ASSESSMENT SUMMARY 7
1.3 TEST APPROACH & METHODOLOGY 8
2 RISK METHODOLOGY 9
2.1 EXPLOITABILITY 10
2.2 IMPACT 11
2.3 SEVERITY COEFFICIENT 13
3 ASSESSMENT SUMMARY & FINDINGS OVERVIEW 16
4 FINDINGS & TECH DETAILS 17
4.1 (HAL-01) LACK OF THE TWO STEP OWNERSHIP TRANSFER PATTERN - LOW(3.0) 19
Description 19
Code Location 19
BVSS 19
Recommendation 20
Remediation Plan 20
4.2 (HAL-02) MISSING ZERO ADDRESS CHECK - LOW(2.0) 21
Description 21
Code Location 21
BVSS 22
Recommendation 22
Remediation Plan 22

4.3 (HAL-03) INVOICE TYPE CHECK MISSING - INFORMATIONAL(1.9) 23
Description 23
Code Location 23
BVSS 24
Recommendation 24
Remediation Plan 24
4.4 (HAL-04) LACK OF REENTRANCYGUARD - INFORMATIONAL(1.2) 25
Description 25
Code Location 26
BVSS 27
Recommendation 27
Remediation Plan 27
4.5 (HAL-05) REDUNDANT CHECK IN THE REMOVEAUTHORIZED FUNCTION - INFORMATIONAL(0.0) 28
Description 28
Code Location 28
BVSS 29
Recommendation 29
Remediation Plan 29
4.6 (HAL-06) CONTRACT PAUSE FEATURE MISSING - INFORMATIONAL(0.0) 30
Description 30
Code Location 30
BVSS 30
Recommendation 30
Remediation Plan 30
4.7 (HAL-07) FLOATING PRAGMA - INFORMATIONAL(0.0) 31
Description 31

	Code Location	31
	BVSS	31
	Recommendation	31
	Remediation Plan	32
5	MANUAL TESTING	33
5.1	ACCESS CONTROL	34
	Description	34
	Results	34
5.2	IMPROPER ADDITION AND DELETION OF AUTHORISED	36
	Description	36
	Results	36
5.3	MINTING INVOICE WITH SAME ID	37
	Description	37
	Results	37
6	AUTOMATED TESTING	38
6.1	STATIC ANALYSIS REPORT	39
	Description	39
	Results	39
6.2	AUTOMATED SECURITY SCAN	41
	Description	41
	Results	41

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	07/25/2023	Isabel Burruezo
0.2	Document Updates	07/27/2023	Isabel Burruezo
0.3	Final Draft	07/29/2023	Isabel Burruezo
0.4	Draft Review	07/30/2023	Piotr Cielas
0.5	Draft Review	07/31/2023	Gabi Urrutia
1.0	Remediation Plan	08/08/2023	Isabel Burruezo
1.1	Remediation Plan Review	08/09/2023	Piotr Cielas
1.2	Remediation Plan Review	08/09/2023	Gabi Urrutia

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
Isabel Burruezo	Halborn	Isabel.Burruezo@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Truffles is a platform that enables sellers to generate invoices and buyers to settle them via their internal payment partners. The invoices generated on the platform are minted as Non-Fungible Tokens (NFTs) for increased transparency, while preserving the privacy of both buyers and sellers by keeping their information confidential.

Truffles engaged Halborn to conduct a security assessment on their smart contracts beginning on July 24th, 2023 and ending on July 31st, 2023. The security assessment was scoped to the smart contracts provided in the truffles-nft-invoice GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

1.2 ASSESSMENT SUMMARY

The team at Halborn was provided 1.5 weeks for the engagement and assigned a full-time security engineer to verify the security of the smart contracts in scope. The security engineer is a blockchain and smart contract security expert 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
- Verify whether the smart contracts work as expected

In summary, Halborn did not identify any significant issues; however, some recommendations were given to reduce the likelihood and impact of risks, which were successfully addressed by Truffles .

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
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Testnet deployment (Foundry)

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)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

EXECUTIVE OVERVIEW

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

Code repositories:

1. Truffles NFT Invoice

- Repository: contracts
- Commit ID: 2d1a6334139ed9d6c60ff44e16c5a4198ebab737
- Branch: main
- Smart contracts in scope:
    1. contracts/Truffles.sol
    2. contracts/Authorizable.sol

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	2	5

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) LACK OF THE TWO STEP OWNERSHIP TRANSFER PATTERN	Low (3.0)	SOLVED - 08/04/2023
(HAL-02) MISSING ZERO ADDRESS CHECK	Low (2.0)	SOLVED - 08/04/2023
(HAL-03) INVOICE TYPE CHECK MISSING	Informational (1.9)	SOLVED - 08/04/2023
(HAL-04) LACK OF REENTRANCYGUARD	Informational (1.2)	SOLVED - 08/04/2023
(HAL-05) REDUNDANT CHECK IN THE REMOVEAUTHORIZED FUNCTION	Informational (0.0)	SOLVED - 08/04/2023
(HAL-06) CONTRACT PAUSE FEATURE MISSING	Informational (0.0)	SOLVED - 08/07/2023
(HAL-07) FLOATING PRAGMA	Informational (0.0)	SOLVED - 08/04/2023

FINDINGS & TECH DETAILS

4.1 (HAL-01) LACK OF THE TWO STEP OWNERSHIP TRANSFER PATTERN - LOW (3.0)

Description:

The Authorizable contract is inherited by the Truffles contract and implements the Ownable pattern. However, the assessment revealed that the solution does not support the two-step-ownership-transfer pattern. The ownership transfer might be accidentally set to an inactive EOA account. In the case of account hijacking, all functionalities get under permanent control of the attacker.

Code Location:

Listing 1: contracts/Authorizable.sol

```
contract Authorizable is Ownable {

mapping(address => bool) private authorized;
modifier onlyAuthorized() {
    require(authorized[msg.sender] ||  owner() == msg.sender, "Not
 ↳  authorized");
        _;
}
```

Listing 2: contracts/Truffles.sol

```
contract TRUFFLES is ERC721, ERC721Enumerable, ERC721URIStorage,
 ↳  Authorizable {
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:H/A:H/D:M/Y:M/R:N/S:C (3.0)

Recommendation:

It is recommended to implement a two-step process where the owner nominates an account and the nominated account needs to call an acceptOwnership() function for the transfer of the ownership to fully succeed. This ensures the nominated EOA account is a valid and active account.

Remediation Plan:

SOLVED: The Truffles team solved this finding in commit fc36014: the Ownable contract was replaced with the Ownable2Step.sol from the **Open-Zeppelin** library within the Authorizable contract to establish a secure approach for conducting two-step ownership transfers.

4.2 (HAL-02) MISSING ZERO ADDRESS CHECK - LOW (2.0)

Description:

The functions addEligibleHolder, removeEligibleHolder and isEligibleHolder do not perform verification to ensure that no addresses provided as parameters are the zero addresses. Consequently, there is a risk of accidentally setting an eligible holder address to the zero address, leading to unintended behavior or potential vulnerabilities in the future.

Code Location:

Listing 3: contracts/Truffles.sol (Lines 103,110)

```
102 function addEligibleHolder(address _org) public onlyAuthorized {
103     s_eligibleHolders[_org] = true;
104 }
105
106 /// @notice remove eligible holders for NFT
107 /// @dev remove eligible address to eligibleHolders maping :
    ↳ onlyAuthorized
108 /// @param _org address of already eligible holder
109 function removeEligibleHolder(address _org) public onlyAuthorized
    ↳ {
110     s_eligibleHolders[_org] = false;
111 }
```

Listing 4: contracts/Truffles.sol (Line 197)

```
192 function isEligibleHolder(address _holder)
193     public
194     view
195     returns (bool _isEligibleHolder)
196 {
197     return (s_eligibleHolders[_holder]);
198 }
```

BVSS:

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

Recommendation:

Consider adding a check to ensure that the provided address is not the zero address before modifying or checking holders' eligibility. It is also possible to add a modifier that performs the verification and then apply it to the above functions instead, to avoid repeating the zero address check in each function.

Remediation Plan:

SOLVED: The Truffles team solved this finding in commit fc36014: the function addEligibleHolder was updated with a validation step to ensure that the provided address is not the zero address before making modifications to the headlineholders' eligibility. Similarly, the removeEligibleHolder function now includes a validation check to confirm that the given address corresponds to an existing holder.

4.3 (HAL-03) INVOICE TYPE CHECK MISSING - INFORMATIONAL (1.9)

Description:

The settlePrivateInvoice function allows authorized addresses to mark the corresponding NFT of a private invoice as settled when the full amount of the invoice is paid. However, it lacks a check to verify that the provided NFT is of the private invoice type. Consequently, both private and public invoices could be settled indifferently. If the provided NFT ID corresponds to a public invoice, it is added to the s_isPrivateInvoicePaid mapping, even though its type does not match, leading to data inconsistency.

In addition, private invoices do not store the amount for the NFT unlike public ones, so the function also does not have a proper check to verify that the full amount of the invoice is paid.

Code Location:

Listing 5: contracts/Truffles.sol

```
function mintPrivateInvoice(
    address to,
    uint256 tokenId,
    string memory uri
) public onlyAuthorized {
    _safeMint(to, tokenId);
    _setTokenURI(tokenId, uri);
    s_nftType[tokenId] = NftType.PrivateInvoice;
}
```

Listing 6: contracts/Truffles.sol (Lines 142,143)

```
141 function settlePrivateInvoice(uint256 _nftID) public
    ↳ onlyAuthorized {
142     require(_exists(_nftID), "NFT not minted");
143     s_isPrivateInvoicePaid[_nftID] = true;
```

```

}
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:H/A:L/D:N/Y:M/R:N/S:U (1.9)

Recommendation:

It is recommended to include a check in the settlePrivateInvoice function to verify that the invoice being settled corresponds to the private NFT type. This additional check ensures that only invoices related to private NFTs can be settled within the function.

Remediation Plan:

SOLVED: The Truffles team solved this finding in commit fc36014: A validation step was introduced to ensure that the provided NFT ID is that of a private invoice before modifying the s_isPrivateInvoicePaid mapping.

4.4 (HAL-04) LACK OF REENTRANCYGUARD - INFORMATIONAL (1.2)

Description:

In the Truffles contract, the functions mintPublicInvoice and mintPrivateInvoice play a crucial role in minting public and private invoices as non-fungible tokens (NFTs).

To achieve this, these functions make use of the _safemint function from the ERC721 contract. This function is responsible for verifying whether the designated recipient can indeed receive ERC721 tokens. By performing this check, it ensures that an NFT is not minted to a contract incapable of handling ERC721 tokens, safeguarding the integrity of the token ecosystem.

However, a potential security loophole arises due to an external function call within these functions. When the to parameter refers to a smart contract, it must implement **IERC721Receiver.onERC721Received**, which is invoked during a safe transfer of the NFT. This external function call creates an opportunity for reentrancy attacks.

Specifically, the absence of a reentrancy guard in both the mintPublicInvoice and mintPrivateInvoice functions allows an attacker to exploit the onERC721Received callback by performing reentrant calls. This vulnerability enables the attacker to execute multiple, unintended operations during the callback, leading to unexpected and potentially harmful outcomes.

In this context, the possibility of a reentry attack is considered to be unlikely, since only authorized users and the administrator have the privilege to call the mintPublicInvoice and mintPrivateInvoice functions for token minting. Furthermore, as there are no limits on the number of invoices that can be created, and no balance transfers are involved, the potential impact of any such attack would be minimal.

Nevertheless, it is crucial to exercise caution, as even minor changes to the code could inadvertently introduce significant security vulnerabilities

Code Location:

Listing 7: contracts/Truffles.sol (Line 133)

```
128 function mintPrivateInvoice(
129     address to,
130     uint256 tokenId,
131     string memory uri
132 ) public onlyAuthorized {
133     _safeMint(to, tokenId);
134     _setTokenURI(tokenId, uri);
135     s_nftType[tokenId] = NftType.PrivateInvoice;
136 }
```

Listing 8: contracts/Truffles.sol (Line 179)

```
160 function mintPublicInvoice(
161     address _to,
162     uint256 _tokenId,
163     string memory _uri,
164     uint256 _invoiceAmount,
165     string calldata _currency,
166     string calldata _invoiceID,
167     bytes32 _merkleProofTransaction
168 ) public onlyAuthorized {
169     invoiceData memory iData = invoiceData({
170         Amount: _invoiceAmount,
171         nftID: _tokenId,
172         merkleProofTransaction: _merkleProofTransaction,
173         currency: _currency,
174         invoiceID: _invoiceID
175     });
176     s_invoiceDetails[_tokenId] = iData;
177     s_invoice_ID_to_Nft_ID[_invoiceID] = _tokenId;
178     s_nftType[_tokenId] = NftType.PublicInvoice;
179     _safeMint(_to, _tokenId);
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:M/R:N/S:U (1.2)

Recommendation:

To address this security concern to contract be resilient against reentrancy attacks providing a robust and secure NFT minting process, it is crucial to fortify both the mintPublicInvoice and mintPrivateInvoice functions with two effective measures:

- add a reentrancy guard
- the adoption of the check-effects pattern.

Remediation Plan:

SOLVED: The Truffles team solved this finding in commit fc36014: the reentrancy guard was added to mintPrivateInvoice and mintPublicInvoice functions to mitigate the risk of reentrancy attacks.

4.5 (HAL-05) REDUNDANT CHECK IN THE REMOVEAUTHORIZED FUNCTION - INFORMATIONAL (0.0)

Description:

In the Authorizable contract, the addAuthorized function allows the owner to add an address to the list of authorized addresses. It requires that the provided address is not the zero address, and it can only be executed by the contract owner because of the **onlyOwner** modifier.

The removeAuthorized function allows the owner to remove an address from the list of authorized addresses. It requires that the provided address is not the zero address and is different from the senders, which has to be the owner.

The checks performed in the latter function are redundant, since the addAuthorized function already prevents adding the zero address as an authorized address. Therefore, this check is unnecessary and result in extra gas overhead.

Code Location:

Listing 9: contracts/Authorizable.sol (Line 25)

```
24 function addAuthorized(address _toAdd) onlyOwner public {
25     require(_toAdd != address(0));
26     authorized[_toAdd] = true;
27 }
```

Listing 10: contracts/Authorizable.sol (Lines 27,28)

```
26 function removeAuthorized(address _toRemove) onlyOwner public {
27     require(_toRemove != address(0));
28     require(_toRemove != msg.sender);
29     authorized[_toRemove] = false;
30 }
```

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 removing the unnecessary check from the removeAuthorized function.

Remediation Plan:

SOLVED: The Truffles team solved this finding in commit fc36014: the redundant check was removed from the removeAuthorized function.

4.6 (HAL-06) CONTRACT PAUSE FEATURE MISSING - INFORMATIONAL (0.0)

Description:

It was identified that the Owner cannot pause the Truffles contract. In the case of a security incident, this means that the owner lacks the ability to halt the minting or settlement of Invoices, potentially leading to further complications.

Code Location:

- **contracts/Truffles.sol**

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 pause functionality to the contract.

Remediation Plan:

SOLVED: The Truffles team solved this finding in commit 2e2a367: The contract now incorporates a pausing mechanism, which introduces two additional functions: pauseContract and unpauseContract. This enhancement is complemented by the inclusion of the whenNotPaused modifier within several functions, including mintPrivateInvoice, settlePrivateInvoice, mintPublicInvoice, and _beforeTokenTransfer.

4.7 (HAL-07) FLOATING PRAGMA - INFORMATIONAL (0.0)

Description:

The Truffles contract uses the Solidity pragma **^0.8.9**. It's essential to deploy the contract with the exact compiler version and flags that have undergone thorough testing. Locking the pragma to a specific version helps to ensure that contracts are not accidentally deployed using outdated compiler versions, which might introduce bugs negatively impacting the contract system, or excessively new pragma versions that haven't undergone extensive testing.

Code Location:

Listing 11: contracts/Truffles.sol (Line 2)

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.9;
```

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 locking the pragma version with known bugs for the compiler version. When possible, do not use floating pragma in the final live deployment. Specifying a fixed compiler version ensures that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

FINDINGS & TECH DETAILS

Remediation Plan:

SOLVED: The Truffles team solved this finding in commit fc36014: the pragma version has been locked and upgraded to match the version used in the Authorizable contract.

Listing 12: contracts/Truffles.sol (Line 2)

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.17;
```

MANUAL TESTING

In the manual testing phase, the following scenarios were simulated. The scenarios listed below were selected based on the severity of the vulnerabilities Halborn was testing the program for.

5.1 ACCESS CONTROL

Description:

In the Truffles contract, the functions mintPublicInvoice and mintPrivateInvoice serve the purpose of minting non-fungible tokens for public and private invoices, respectively. It is crucial to note that these functions should only be accessible to the contract owner or authorized addresses, which have been added by the owner.

To ensure the integrity of the access control mechanism and prevent unauthorized minting by malicious users or any unauthorized addresses, rigorous testing has been conducted. The tests verify that the proper access controls are in place, guaranteeing that only the designated individuals can initiate the minting process.

By carrying out these tests and enforcing the necessary access controls, the Truffles contract maintains a secure and trusted environment for minting non-fungible tokens, safeguarding against potential security risks and unauthorized actions.

Results:

No vulnerabilities were identified.

```
Running 1 test for test/HalbornTrufflesTest.t.sol:HalbornTruffles
[FAIL. Reason: Not authorized] test_mintPublicInvoice_notAuth() (gas: 21912)
Logs:
  [+] Mint Public Invoice by:  0x0000000000000000000000000000000000000aA11

Test result: FAILED. 0 passed; 1 failed; finished in 5.97ms

Failing tests:
Encountered 1 failing test in test/HalbornTrufflesTest.t.sol:HalbornTruffles
[FAIL. Reason: Not authorized] test_mintPublicInvoice_notAuth() (gas: 21912)
```

MANUAL TESTING

```
Running 1 test for test/HalbornTrufflesTest.t.sol:HalbornTruffles
[FAIL. Reason: Not authorized] test_mintPrivateInvoice_NotAuthorised() (gas: 19096)
Logs:
  [+] Mint Private Invoice by:  0x000000000000000000000000000000000000aA11

Test result: FAILED. 0 passed; 1 failed; finished in 5.71ms

Failing tests:
Encountered 1 failing test in test/HalbornTrufflesTest.t.sol:HalbornTruffles
[FAIL. Reason: Not authorized] test_mintPrivateInvoice_NotAuthorised() (gas: 19096)
```

```
Running 1 test for test/HalbornTrufflesTest.t.sol:HalbornTruffles
[FAIL. Reason: Not authorized] test_notAuth_addEligigleHolders() (gas: 20608)
Logs:
  [+]Add Elegible Holder by:  0x000000000000000000000000000000000000AA22

Test result: FAILED. 0 passed; 1 failed; finished in 5.81ms

Failing tests:
Encountered 1 failing test in test/HalbornTrufflesTest.t.sol:HalbornTruffles
[FAIL. Reason: Not authorized] test_notAuth_addEligigleHolders() (gas: 20608)
```

```
Running 1 test for test/HalbornTrufflesTest.t.sol:HalbornTruffles
[FAIL. Reason: Not authorized] test_notAuth_removeEligigleHolders() (gas: 69548)
Logs:
  [+]Add Authorized :  0x000000000000000000000000000000000000aA11
  [+]Add Elegible Holder:  0x000000000000000000000000000000000000AA22
  [+]Remove Elegible Holder by:  0x000000000000000000000000000000000000AA22

Test result: FAILED. 0 passed; 1 failed; finished in 5.66ms

Failing tests:
Encountered 1 failing test in test/HalbornTrufflesTest.t.sol:HalbornTruffles
[FAIL. Reason: Not authorized] test_notAuth_removeEligigleHolders() (gas: 69548)
```

5.2 IMPROPER ADDITION AND DELETION OF AUTHORISED

Description:

In the Authorizable contract, the addAuthorized and removeAuthorized functions enable the owner to include or exclude authorized addresses. These authorized addresses gain the privilege to call functions in the Truffles contract, enabling those addresses to extract invoices and assist in contract management.

To ensure the integrity of the system, testing was run to verify that the provided address cannot be set to zero.

Results:

No vulnerabilities were identified.

```
Running 1 test for test/HalbornTrufflesTest.t.sol:HalbornTruffles
[FAIL. Reason: EvmError: Revert] test_addAuth_ZeroAdd() (gas: 10996)
Logs:
  [+]Add Authorized:  0x0000000000000000000000000000000000000000

Test result: FAILED. 0 passed; 1 failed; finished in 5.79ms

Failing tests:
Encountered 1 failing test in test/HalbornTrufflesTest.t.sol:HalbornTruffles
[FAIL. Reason: EvmError: Revert] test_addAuth_ZeroAdd() (gas: 10996)
```

```
Running 1 test for test/HalbornTrufflesTest.t.sol:HalbornTruffles
[FAIL. Reason: EvmError: Revert] test_removeAuth_ZeroAdd() (gas: 10977)
Logs:
  [+]Remove Authorized:  0x0000000000000000000000000000000000000000

Test result: FAILED. 0 passed; 1 failed; finished in 5.87ms

Failing tests:
Encountered 1 failing test in test/HalbornTrufflesTest.t.sol:HalbornTruffles
[FAIL. Reason: EvmError: Revert] test_removeAuth_ZeroAdd() (gas: 10977)
```

5.3 MINTING INVOICE WITH SAME ID

Description:

The Truffles contract incorporates the mintPublicInvoice and mintPrivateInvoice functions, both dedicated to minting non-fungible tokens for public and private invoices. These functions necessitate certain values as parameters, including the **tokenId**.

To uphold the uniqueness and integrity of the minted invoices, comprehensive tests have been conducted to verify that no two invoices can share the same **tokenId** value.

Results:

No vulnerabilities were identified.

```
[FAIL. Reason: ERC721: token already minted] test_mint_sameNFT_public_and_private_invoice() (gas: 350755)
Logs:
  [+] Mint Private Invoice
  Token id:  1
  total supply:  1
  Balance of auth after minting:  1
  [+] Mint Public Invoice
  Token id:  1

Test result: FAILED. 0 passed; 1 failed; finished in 6.26ms

Failing tests:
Encountered 1 failing test in test/HalbornTrufflesTest.t.sol:HalbornTruffles
[FAIL. Reason: ERC721: token already minted] test_mint_sameNFT_public_and_private_invoice() (gas: 350755)
```

```
[FAIL. Reason: ERC721: token already minted] test_mintPrivateInvoice_twice() (gas: 195573)
Logs:
  [+] Mint Private Invoice
  Token id:  4
  total supply:  1
  Balance of auth after minting:  1
  [+] Mint Private Invoice
  Token id:  4

Test result: FAILED. 0 passed; 1 failed; finished in 6.19ms

Failing tests:
Encountered 1 failing test in test/HalbornTrufflesTest.t.sol:HalbornTruffles
[FAIL. Reason: ERC721: token already minted] test_mintPrivateInvoice_twice() (gas: 195573)
```

AUTOMATED TESTING

6.1 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.

Results:

Truffles

```
INFO:Detectors:
ERC721._checkOnERC721Received(address,address,uint256,bytes) (node_modules/@openzeppelin/contracts/token/ERC721/ERC721.sol#429-451) ignores return value by IERC721Receiver(to).onERC721Received(_msgSender(
),from,tokenId,data) (node_modules/@openzeppelin/contracts/token/ERC721/ERC721.sol#436-447)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
INFO:Detectors:
Reentrancy in TRUFFLES.mintPrivateInvoice(address,uint256,string) (contracts/Truffles.sol#126-134):
        External calls:
        - _safeMint(to,tokenId) (contracts/Truffles.sol#131)
                - IERC721Receiver(to).onERC721Received(_msgSender(),from,tokenId,data) (node_modules/@openzeppelin/contracts/token/ERC721/ERC721.sol#436-447)
        State variables written after the call(s):
        - _setTokenURI(tokenId,uri) (contracts/Truffles.sol#132)
                - _tokenURIs[tokenId] = _tokenURI (node_modules/@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol#47)
        - s_nftType[tokenId] = NftType.PrivateInvoice (contracts/Truffles.sol#133)
Reentrancy in TRUFFLES.mintPublicInvoice(address,uint256,string,uint256,string,string,bytes32) (contracts/Truffles.sol#158-179):
        External calls:
        - _safeMint(_to,_tokenId) (contracts/Truffles.sol#177)
                - IERC721Receiver(to).onERC721Received(_msgSender(),from,tokenId,data) (node_modules/@openzeppelin/contracts/token/ERC721/ERC721.sol#436-447)
        State variables written after the call(s):
        - _setTokenURI(_tokenId,_uri) (contracts/Truffles.sol#178)
                - _tokenURIs[tokenId] = _tokenURI (node_modules/@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol#47)
```

AUTOMATED TESTING

Authorizable

```
INFO:Detectors:
Context._msgData() (node_modules/@openzeppelin/contracts/utils/Context.sol#21-23) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code
INFO:Detectors:
Pragma version0.8.17 (contracts/Authorizable.sol#3) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/access/Ownable.sol#4) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/utils/Context.sol#4) allows old versions
solc-0.8.17 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Parameter Authorizable.isAuthorised(address)._authAdd (contracts/Authorizable.sol#20) is not in mixedCase
Parameter Authorizable.addAuthorized(address)._toAdd (contracts/Authorizable.sol#24) is not in mixedCase
Parameter Authorizable.removeAuthorized(address)._toRemove (contracts/Authorizable.sol#29) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Slither:contracts/Authorizable.sol analyzed (3 contracts with 85 detectors), 9 result(s) found
```

All the issues flagged by Slither were found to be either false positives or issues already reported, like the reentrancy issue in **HAL-04** .

6.2 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and to identify low-hanging fruits on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the smart contracts and sent the compiled results to the analyzers in order to locate any vulnerabilities.

Results:

Line	SWC Title	Severity	Short Description
2	(SWC-103) Floating Pragma	Low	A floating pragma is set.

The issue flagged by MythX was reported in **HAL-07**.

THANK YOU FOR CHOOSING

HALBORN