One ID

Smart Contract Audit Report
Prepared for Coin98

Date Issued:	Aug 2, 2023
Project ID:	AUDIT2023011
Version:	v1.0
Confidentiality Level:	Public

inspex

CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2023011
Version	v1.0
Client	Coin98
Project	One ID
Auditor(s)	Wachirawit Kanpanluk Phitchakorn Apiratisakul
Author(s)	Puttimet Thammasaeng
Reviewer	Natsasit Jirathammanuwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Aug 2, 2023	Full report	Puttimet Thammasaeng

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by Coin98, Inspex team conducted an audit to verify the security posture of the One ID smart contracts between Jul 7, 2023 and Jul 13, 2023. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of One ID smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 2 critical, 3 medium, 1 low, 2 very low, and 2 info-severity issues. With the project team's prompt response in resolving the issues found by Inspex, all issues were resolved or mitigated in the reassessment. Therefore, Inspex trusts that One ID smart contracts have high-level protections in place to be safe from most attacks.

This smart contract passes Inspex's security verification standard, and is trustworthy.

Approved by Inspex on Aug 2, 2023

inspex
CYBERSECURITY PROFESSIONAL SERVICE
PASS

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

Coin98-One ID simplifies decentralized identity management with its user-friendly name system. Users can register, renew, and link their names to addresses across multiple blockchains. The project seamlessly integrates with Coin98wallets, providing a convenient experience for users managing their decentralized identities.

Scope Information:

Project Name	One ID
Website	https://coin98.com/
Smart Contract Type	Ethereum Smart Contract
Chain	TomoChain
Programming Language	Solidity
Category	Name Service

Audit Information:

Audit Method	Whitebox
Audit Date	Jul 7, 2023 - Jul 13, 2023
Reassessment Date	Aug 2, 2023

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit: (Commit: e8f07e384ba53416113193824dc2b7aaa9c06146)

Contract	Location (URL)
IPriceAggregator	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/IPriceAggregator.sol
IPriceConfig	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/IPriceConfig.sol
IRegistrar	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/IRegistrar.sol
IRegistrarController	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/IRegistrarController.sol
PriceConfig	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/PriceConfig.sol
PriceOracle	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/PriceOracle.sol
Registrar	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/Registrar.sol
RegistrarController	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/RegistrarController.sol
ResolverProxy	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/ResolverProxy.sol
IBaseRegistrar	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/registrar/IBaseRegistrar.sol
StringUtils	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/registrar/StringUtils.sol
IRegistry	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/registry/IRegistry.sol
Registry	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/registry/Registry.sol
IMulticallable	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/resolvers/IMulticallable.sol

Multicallable	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/resolvers/Multicallable.sol
PublicResolver	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/resolvers/PublicResolver.sol
ResolverBase	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/resolvers/ResolverBase.sol
AddrResolver	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/resolvers/profiles/AddrResolver.sol
BinaryResolver	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/resolvers/profiles/BinaryResolver.sol
IAddrResolver	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/resolvers/profiles/IAddrResolver.sol
IAddressResolver	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/resolvers/profiles/IAddressResolver.sol
IBinaryResolver	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/resolvers/profiles/IBinaryResolver.sol
INameResolver	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/resolvers/profiles/INameResolver.sol
ITextResolver	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/resolvers/profiles/ITextResolver.sol
IVersionableResolver	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/resolvers/profiles/IVersionableResolver.sol
NameResolver	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/resolvers/profiles/NameResolver.sol
TextResolver	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/resolvers/profiles/TextResolver.sol
AdvancedERC20	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/utils/AdvancedERC20.sol
ERC20Recoverable	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/utils/ERC20Recoverable.sol
NameEncoder	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/utils/NameEncoder.sol
BytesUtils	https://github.com/coin98/coin98-oneid-evm/blob/e8f07e384b/contracts/wrapper/BytesUtils.sol

Reassessment: (Commit: ae67619d072a6891b8ec8a2258bcda5aea18f6c7)

Contract	Location (URL)
IPriceAggregator	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/IPriceAggregator.sol
IPriceConfig	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/IPriceConfig.sol
IRegistrar	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/IRegistrar.sol
IRegistrarController	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/IRegistrarController.sol
PriceConfig	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/PriceConfig.sol
PriceOracle	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/PriceOracle.sol
Registrar	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/Registrar.sol
RegistrarController	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/RegistrarController.sol
ResolverProxy	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/ResolverProxy.sol
IBaseRegistrar	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/registrar/IBaseRegistrar.sol
StringUtils	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/registrar/StringUtils.sol
IRegistry	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/registry/IRegistry.sol
Registry	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/registry/Registry.sol
PublicResolver	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/resolvers/PublicResolver.sol
ResolverBase	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/resolvers/ResolverBase.sol
AddrResolver	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/reso

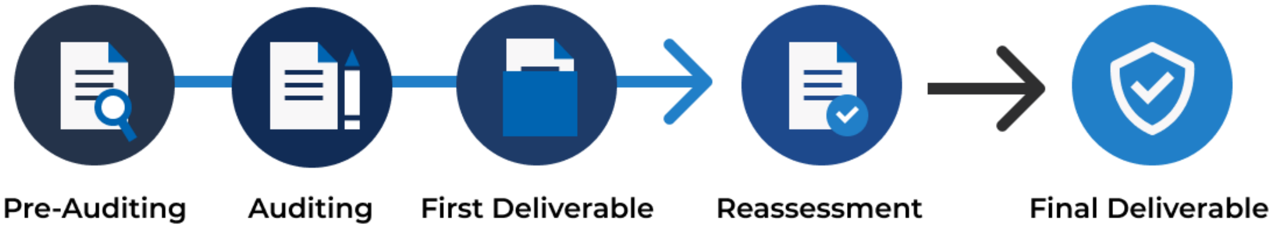
	lvers/profiles/AddrResolver.sol
BinaryResolver	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/resolvers/profiles/BinaryResolver.sol
IAddrResolver	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/resolvers/profiles/IAddrResolver.sol
IAddressResolver	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/resolvers/profiles/IAddressResolver.sol
IBinaryResolver	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/resolvers/profiles/IBinaryResolver.sol
INameResolver	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/resolvers/profiles/INameResolver.sol
ITextResolver	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/resolvers/profiles/ITextResolver.sol
IVersionableResolver	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/resolvers/profiles/IVersionableResolver.sol
NameResolver	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/resolvers/profiles/NameResolver.sol
TextResolver	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/resolvers/profiles/TextResolver.sol
AdvancedERC20	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/utils/AdvancedERC20.sol
ERC20Recoverable	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/utils/ERC20Recoverable.sol
NameEncoder	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/utils/NameEncoder.sol
BytesUtils	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/wrapper/BytesUtils.sol
FixedPriceOracle	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/FixedPriceOracle.sol
FixedPriceConfig	https://github.com/coin98/coin98-oneid-evm/blob/ae67619d07/contracts/FixedPriceConfig.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they inherit from.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The testing items checked are based on our Smart Contract Security Testing Guide (SCSTG) v1.0 (https://github.com/InspexCo/SCSTG/releases/download/v1.0/SCSTG_v1.0.pdf) which covers most prevalent risks in smart contracts. The latest version of the document can also be found at https://inspex.gitbook.io/testing-guide/.

The following audit items were checked during the auditing activity:

Testing Category	Testing Items
1. Architecture and Design	1.1. Proper measures should be used to control the modifications of smart contract logic 1.2. The latest stable compiler version should be used 1.3. The circuit breaker mechanism should not prevent users from withdrawing their funds 1.4. The smart contract source code should be publicly available 1.5. State variables should not be unfairly controlled by privileged accounts 1.6. Least privilege principle should be used for the rights of each role
2. Access Control	2.1. Contract self-destruct should not be done by unauthorized actors 2.2. Contract ownership should not be modifiable by unauthorized actors 2.3. Access control should be defined and enforced for each actor roles 2.4. Authentication measures must be able to correctly identify the user 2.5. Smart contract initialization should be done only once by an authorized party 2.6. tx.origin should not be used for authorization
3. Error Handling and Logging	3.1. Function return values should be checked to handle different results 3.2. Privileged functions or modifications of critical states should be logged 3.3. Modifier should not skip function execution without reverting
4. Business Logic	4.1. The business logic implementation should correspond to the business design 4.2. Measures should be implemented to prevent undesired effects from the ordering of transactions 4.3. msg.value should not be used in loop iteration
5. Blockchain Data	5.1. Result from random value generation should not be predictable 5.2. Spot price should not be used as a data source for price oracles 5.3. Timestamp should not be used to execute critical functions 5.4. Plain sensitive data should not be stored on-chain 5.5. Modification of array state should not be done by value 5.6. State variable should not be used without being initialized

Testing Category	Testing Items
6. External Components	6.1. Unknown external components should not be invoked 6.2. Funds should not be approved or transferred to unknown accounts 6.3. Reentrant calling should not negatively affect the contract states 6.4. Vulnerable or outdated components should not be used in the smart contract 6.5. Deprecated components that have no longer been supported should not be used in the smart contract 6.6. Delegatecall should not be used on untrusted contracts
7. Arithmetic	7.1. Values should be checked before performing arithmetic operations to prevent overflows and underflows 7.2. Explicit conversion of types should be checked to prevent unexpected results 7.3. Integer division should not be done before multiplication to prevent loss of precision
8. Denial of Services	8.1. State changing functions that loop over unbounded data structures should not be used 8.2. Unexpected revert should not make the whole smart contract unusable 8.3. Strict equalities should not cause the function to be unusable
9. Best Practices	9.1. State and function visibility should be explicitly labeled 9.2. Token implementation should comply with the standard specification 9.3. Floating pragma version should not be used 9.4. Builtin symbols should not be shadowed 9.5. Functions that are never called internally should not have public visibility 9.6. Assert statement should not be used for validating common conditions

3.3. Risk Rating

OWASP Risk Rating Methodology (https://owasp.org/www-community/OWASP_Risk_Rating_Methodology) is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker
- **Impact**: a measure of the damage caused by a successful attack

Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

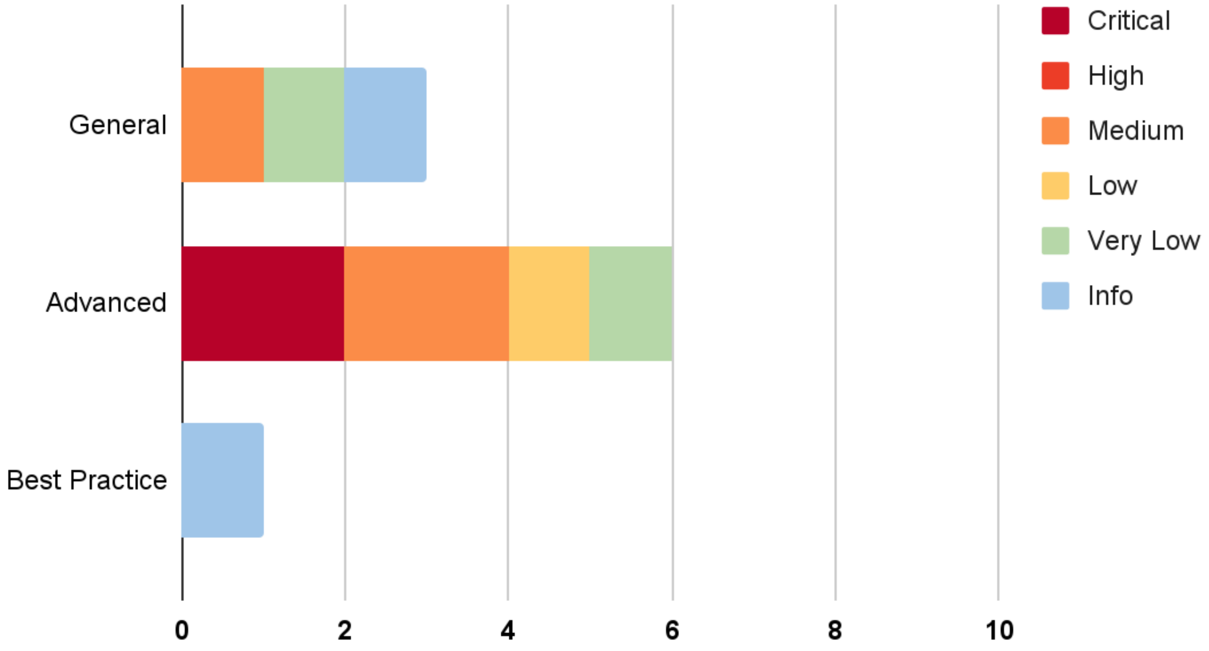
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Likelihood Impact	Low	Medium	High
Low	Very Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	Critical

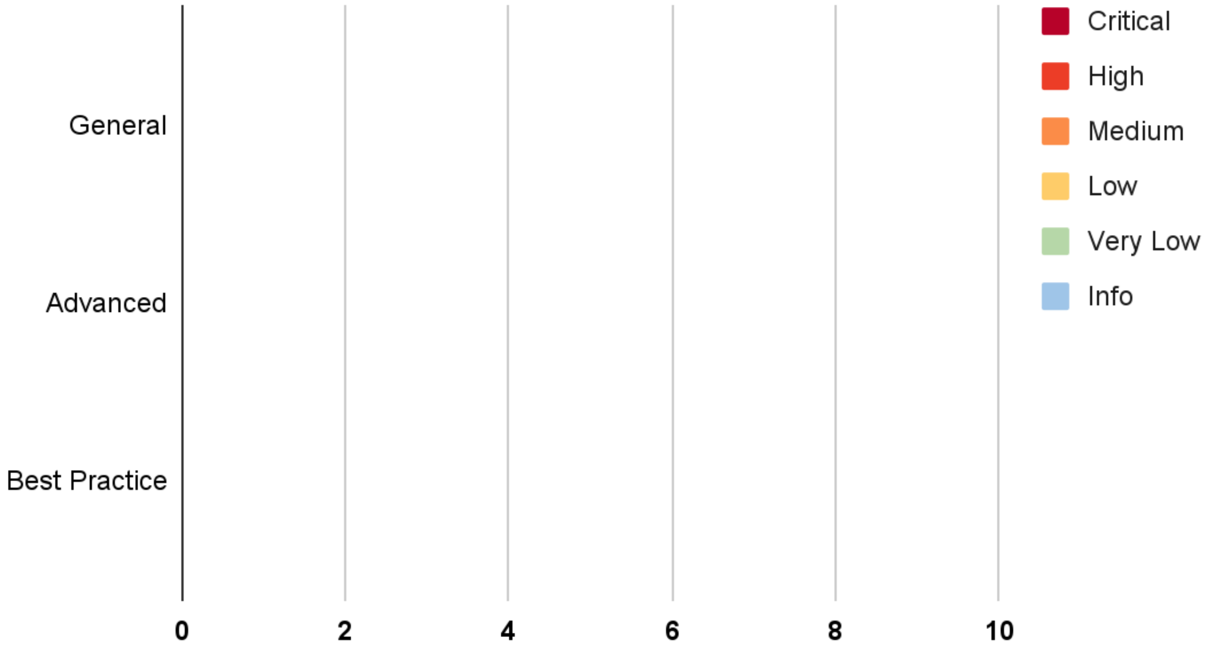
4. Summary of Findings

The following charts show the number of the issues found during the assessment and the issues acknowledged in the reassessment, categorized into three categories: **General**, **Advanced**, and **Best Practice**.

Assessment:



Reassessment:



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Missing _minCommitmentAge and _maxCommitmentAge States Assignment	Advanced	Critical	Resolved
IDX-002	Lack of ERC20 Token Withdrawal Function	Advanced	Critical	Resolved
IDX-003	Missing the Referrer Setting in the Registration Process	Advanced	Medium	Resolved
IDX-004	Centralized Control of State Variable	General	Medium	Resolved *
IDX-005	Improper Price Oracle Implementation	Advanced	Medium	Resolved *
IDX-006	Missing Live Modifier in Register Function	Advanced	Low	Resolved
IDX-007	Lack of Input Validation	Advanced	Very Low	Resolved
IDX-008	Insufficient Logging for Privileged Functions	General	Very Low	Resolved
IDX-009	Inexplicit Solidity Compiler Version	Best Practice	Info	Resolved
IDX-010	Incorrect Transfer Fee Logging	General	Info	Resolved

* The mitigations or clarifications by Coin98 can be found in Chapter 5.

5. Detailed Findings Information

5.1. Missing _minCommitmentAge and _maxCommitmentAge States Assignment

ID	IDX-001
Target	RegistrarController
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity: Critical** **Impact: High** The commit flow in the `RegistrarController` contract that rely on the `_minCommitmentAge` and `_maxCommitmentAge` states will be unusable, as the state cannot be set anywhere in the contract. resulting in the user being unable to use the `register()` function to register a new domain. **Likelihood: High** Due to the state that cannot be set, whenever the user calls the `register()` function, the function will always revert.
Status	Resolved The Coin98 team has fixed the issue by adding the function to configure the `_minCommitmentAge` and the `_maxCommitmentAge` state on commit `39fd74604e64e6b9ab01b33b49a130116b526380`.

5.1.1. Description

When a user registers a domain name, at first, the user has to use the `commit()` function to set the `commitments` state, then the `register()` function calls the `_consumeCommitment()` function to check the current commitment with `commitments` state from `commit()` function.

RegistrarController.sol

```
function commit(bytes32 commitment) public override {
    _charge(0);
    if (commitments[commitment] + _maxCommitmentAge >= block.timestamp) {
        revert UnexpiredCommitmentExists(commitment);
    }
    commitments[commitment] = block.timestamp;
}
```

RegistrarController.sol

```
function register(
    string calldata name,
    address owner,
    uint256 duration,
    bytes32 secret,
    address resolver,
    bytes32 referrer,
    address paymentToken
) public payable override {
    _charge(0);
    IPriceConfig.Price memory price = rentPrice(name, duration, paymentToken);
    uint256 paymentAmount = price.base + price.premium + price.service;
    if(paymentToken == address(0)) {
        if(msg.value < paymentAmount) {
            revert InsufficientValue();
        }
    }
    else {
        IERC20(paymentToken).safeTransferFrom(msg.sender, address(this), paymentAmount);
    }

    { // fix stack too deep
        bytes32 commitment = makeCommitment(name, owner, duration, secret, resolver);
        _consumeCommitment(name, duration, commitment);
    }

    uint256 tokenId = uint256(keccak256(bytes(name)));
    uint256 expires = _registrar.register(tokenId, owner, duration);
    emit NameRegistered(name, keccak256(bytes(name)), owner, price.base, price.premium, expires);

    _handleFee(referrer, price.base, paymentToken);
    if(paymentToken == address(0)) {
        if (msg.value > paymentAmount) {
            payable(msg.sender).transfer(msg.value - paymentAmount);
        }
    }
}
```

In the `_consumeCommitment()` function, in lines 221-223, it validates that the commitment date is not too old to use. However, this condition will never pass due to the fact that the `_maxCommitmentAge` state cannot be set anywhere and the value is always zero.

RegistrarController.sol

```
function _consumeCommitment(
    string memory name,
    uint256 duration,
    bytes32 commitment
) internal {
    // commitment is never registered
    if(commitments[commitment] == 0) {
        revert CommitmentNotFound(commitment);
    }

    // Require an old enough commitment.
    if (commitments[commitment] + _minCommitmentAge > block.timestamp) {
        revert CommitmentTooNew(commitment);
    }

    // If the commitment is too old, or the name is registered, stop
    if (commitments[commitment] + _maxCommitmentAge <= block.timestamp) {
        revert CommitmentTooOld(commitment);
    }
    if (!available(name)) {
        revert NameNotAvailable(name);
    }

    delete (commitments[commitment]);

    if (duration < _registrar.minDuration()) {
        revert DurationTooShort(duration);
    }
}
```

5.1.2. Remediation

Inspex suggests adding the `minCommitmentAge` and `maxCommitmentAge` parameter in the constructor to set the initial value.

RegistrarController.sol

```
constructor(IRegistrar registrar, string memory name, string memory symbol, uint8 decimals_, uint256 minCommitmentAge, uint256 maxCommitmentAge)
VRC25(name, symbol, decimals_) {
    require(maxCommitmentAge > minCommitmentAge);
    _minCommitmentAge = minCommitmentAge;
    _maxCommitmentAge = maxCommitmentAge;

    _registry = registrar.registry();
    _registrar = registrar;
}
```

5.2. Lack of ERC20 Token Withdrawal Function

ID	IDX-002
Target	RegistrarController
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity: Critical** **Impact: High** The tokens that platform users have paid to register or renew will become trapped within the contract. Consequently, the platform loses all tokens in the contract. **Likelihood: High** It is very likely to occur since the platform accepts ERC20 tokens as payment tokens.
Status	**Resolved** The Coin98 team has fixed the issue by adding the function to withdraw the register fee for both native token and ERC20 tokens on commit `39fd74604e64e6b9ab01b33b49a130116b526380`.

5.2.1. Description

The `RegistrarController` contract enables platform users to register domain names using either native tokens or ERC20 tokens. Once the registration process is complete, both native tokens and ERC20 tokens are stored within the `RegistrarController` contract.

RegistrarController.sol

```
function register(
    string calldata name,
    address owner,
    uint256 duration,
    bytes32 secret,
    address resolver,
    bytes32 referrer,
    address paymentToken
) public payable override {
    _charge(0);
    IPriceConfig.Price memory price = rentPrice(name, duration, paymentToken);
    uint256 paymentAmount = price.base + price.premium + price.service;
    if(paymentToken == address(0)) {
        if(msg.value < paymentAmount) {
            revert InsufficientValue();
        }
    }
    else {
        IERC20(paymentToken).safeTransferFrom(msg.sender, address(this), paymentAmount);
    }

    { // fix stack too deep
        bytes32 commitment = makeCommitment(name, owner, duration, secret, resolver);
        _consumeCommitment(name, duration, commitment);
    }

    uint256 tokenId = uint256(keccak256(bytes(name)));
    uint256 expires = _registrar.register(tokenId, owner, duration);
    emit NameRegistered(name, keccak256(bytes(name)), owner, price.base, price.premium, expires);

    _handleFee(referrer, price.base, paymentToken);
    if(paymentToken == address(0)) {
        if (msg.value > paymentAmount) {
            payable(msg.sender).transfer(msg.value - paymentAmount);
        }
    }
}
```

For the native tokens, the owner can transfer out through the `withdraw()` function without any problem.

RegistrarController.sol

```
function withdraw() public onlyOwner {
    payable(owner()).transfer(address(this).balance);
}
```

However, since the `RegistrarController` contract lacks a withdrawal function for withdrawing ERC20 tokens, they become permanently trapped within the contract.

5.2.2. Remediation

Inspex recommends implementing a new function in the contract that enables the platform owner to withdraw and utilize the payment tokens effectively. This addition will provide a mechanism for the platform owner to access and transfer the payment tokens out of the contract. For example, adding the `withdrawToken()` function.

RegistrarController.sol

```
function withdrawToken(
    address _token,
    address _to,
    uint256 _amount
) external onlyOwner {
    IERC20(_token).transfer(_to, _amount);
}
```

Alternatively, inheriting the `RegistrarController` contract from the `ERC20Recoverable` contract allows the owner to utilize the `recoverFunds()` function.

ERC20Recoverable.sol

```
contract ERC20Recoverable is Ownable {
    /**
    @notice Recover ERC20 tokens sent to the contract by mistake.
    @dev The contract is Ownable and only the owner can call the recover
function.
    @param _to The address to send the tokens to.
@param _token The address of the ERC20 token to recover
    @param _amount The amount of tokens to recover.
 */
    function recoverFunds(
        address _token,
        address _to,
        uint256 _amount
    ) external onlyOwner {
        IERC20(_token).transfer(_to, _amount);
    }
}
```

RegistrarController.sol

```
contract RegistrarController is VRC25, IERC165, IRegistrarController,
ERC20Recoverable {
    bytes32 constant ZERO_BYTES32 =
0x0000000000000000000000000000000000000000000000000000000000000000;
```

5.3. Missing the Referrer Setting in the Registration Process

ID	IDX-003
Target	RegistrarController
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Low The loss of incentives for the referrer during the domain name renewal process. **Likelihood:** High It is likely to occur due to the exclusion of a call to the `setReferrer()` function in the `register()` function within the `RegistrarController` contract.
Status	Resolved The Coin98 team has fixed the issue by adding the function set the referrer in the `register()` function on commit `39fd74604e64e6b9ab01b33b49a130116b526380`.

5.3.1. Description

In the `RegistrarController` contract, the `register()` and `renew()` functions are used to register a new domain name and extend the expiration date of a domain name, respectively.

The user can pass the referrer parameter to the `register()` function to enable it to receive incentives through the `_handleFee()` function as shown below in lines 132 and 161.

RegistrarController.sol

```
102 function register(
103     string calldata name,
104     address owner,
105     uint256 duration,
106     bytes32 secret,
107     address resolver,
108     bytes32 referrer,
109     address paymentToken
110 ) public payable override {
111     _charge(0);
112     IPriceConfig.Price memory price = rentPrice(name, duration, paymentToken);
113     uint256 paymentAmount = price.base + price.premium + price.service;
114     if(paymentToken == address(0)) {
115         if(msg.value < paymentAmount) {
116             revert InsufficientValue();
117         }
```

```
    }
    else {
        IERC20(paymentToken).safeTransferFrom(msg.sender, address(this), paymentAmount);
    }

    { // fix stack too deep
        bytes32 commitment = makeCommitment(name, owner, duration, secret, resolver);
        _consumeCommitment(name, duration, commitment);
    }

    uint256 tokenId = uint256(keccak256(bytes(name)));
    uint256 expires = _registrar.register(tokenId, owner, duration);
    emit NameRegistered(name, keccak256(bytes(name)), owner, price.base, price.premium, expires);

    _handleFee(referrer, price.base, paymentToken);
    if(paymentToken == address(0)) {
        if (msg.value > paymentAmount) {
            payable(msg.sender).transfer(msg.value - paymentAmount);
        }
    }
}

function renew(
    string calldata name,
    uint256 duration,
    address paymentToken
) external payable override {
    _charge(0);
    bytes32 labelhash = keccak256(bytes(name));
    uint256 tokenId = uint256(labelhash);
    IPriceConfig.Price memory price = rentPrice(name, duration, paymentToken);
    uint256 paymentAmount = price.base + price.service;
    if(paymentToken == address(0)) {
        if(msg.value < paymentAmount) {
            revert InsufficientValue();
        }
    }
    else {
        IERC20(paymentToken).safeTransferFrom(msg.sender, address(this), paymentAmount);
    }
    uint256 expires = _registrar.renew(tokenId, duration);
    emit NameRenewed(name, labelhash, price.base, expires);

```

```
161     _handleFee(_registrar.referrer(tokenId), price.base, paymentToken);
162     if(paymentToken == address(0)) {
163         if (msg.value > paymentAmount) {
164             payable(msg.sender).transfer(msg.value - paymentAmount);
165         }
166     }
167 }
```

The referrer data can only be stored using the `setReferrer()` function, which can only be called by the `onlyController` modifier in the `Registrar` contract.

Registrar.sol

```
211 function setReferrer(uint256 id, bytes32 referrerNode) external override
    onlyController {
212     _referrers[id] = referrerNode;
213 }
```

However, the `setReferrer()` function has never been called inside the `register()` function. As a result, the parameter passed to the `_handleFee()` function in line 161 would be `0`, leading to a loss of incentives for the referrer during the domain renewal process.

5.3.2. Remediation

Inspex suggests adding the `setReferrer()` function call into the registration process, as shown below in line 130.

RegistrarController.sol

```
102 function register(
103     string calldata name,
104     address owner,
105     uint256 duration,
106     bytes32 secret,
107     address resolver,
108     bytes32 referrer,
109     address paymentToken
110 ) public payable override {
111     _charge(0);
112     IPriceConfig.Price memory price = rentPrice(name, duration, paymentToken);
113     uint256 paymentAmount = price.base + price.premium + price.service;
114     if(paymentToken == address(0)) {
115         if(msg.value < paymentAmount) {
116             revert InsufficientValue();
117         }
118     }
119     else {
120         IERC20(paymentToken).safeTransferFrom(msg.sender, address(this),
paymentAmount);
    }

    { // fix stack too deep
        bytes32 commitment = makeCommitment(name, owner, duration, secret,
resolver);
        _consumeCommitment(name, duration, commitment);
    }

    uint256 tokenId = uint256(keccak256(bytes(name)));
    uint256 expires = _registrar.register(tokenId, owner, duration);
    _registrar.setReferrer(tokenId, referrer);
    emit NameRegistered(name, keccak256(bytes(name)), owner, price.base,
price.premium, expires);

    _handleFee(referrer, price.base, paymentToken);
    if(paymentToken == address(0)) {
        if (msg.value > paymentAmount) {
            payable(msg.sender).transfer(msg.value - paymentAmount);
        }
    }
}
```

5.4. Centralized Control of State Variable

ID	IDX-004
Target	PriceConfig PriceOracle Registrar RegistrarController VRC25
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Medium **Impact:** High The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood:** Low There is nothing to restrict the changes from being done; however, this action can only be done by the contract owner.
Status	Resolved * The Coin98 team has mitigated this issue by implementing a Timelock contract as the owner of all contracts on commit `e211833ea67af1f908792cfc06a5b85f6ce2d520` .

5.4.1. Description

Critical state variables can be updated at any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

Target	Contract	Function	Modifier
PriceConfig.sol (L:118)	PriceConfig	setPricing()	onlyOwner
PriceOracle.sol (L:37)	PriceOracle	setOperator()	onlyOwner
Registrar.sol (L:186)	Registrar	addController()	onlyOwner
Registrar.sol (L:194)	Registrar	removeController()	onlyOwner
Registrar.sol (L:202)	Registrar	setResolver()	onlyOwner

Registrar.sol (L:220)	Registrar	setGracePeriod()	onlyOwner
Registrar.sol (L:228)	Registrar	setMinDuration()	onlyOwner
RegistrarController.sol (L:169)	RegistrarController	setPriceConfig()	onlyOwner
RegistrarController.sol (L:179)	RegistrarController	setPartner()	onlyOwner
RegistrarController.sol (L:186)	RegistrarController	setReferralFee()	onlyOwner
VRC25.sol (L:206)	RegistrarController	transferOwnership()	onlyOwner
VRC25.sol (L:216)	RegistrarController	setFee()	onlyOwner

5.4.2. Remediation

In the ideal case, critical state variables should not be modifiable to maintain the integrity of the smart contract. However, if modifications are needed, Inspex suggests using the `onlyAllGovernance` modifier, which requires governance decisions instead of the `onlyOwner` modifier to control the use of these functions.

If removing the functions or using the `onlyAllGovernance` modifier is not possible, Inspex suggests mitigating the risk of this issue by implementing a timelock mechanism to delay the changes for a reasonable amount of time, such as at least 24 hours.

5.5. Improper Price Oracle Implementation

ID	IDX-005
Target	PriceConfig
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Medium **Impact:** High The platform can perform front-running registration or renewal transactions to change the registration price and gain more users' funds. **Likelihood:** Low There is nothing to restrict the changes from being done; however, this action can only be done by the contract owner.
Status	Resolved * The Coin98 team has mitigated this issue by removing the `PriceOracle` contract, and implementing a Timelock contract as the owner of the `FixedPriceOracle` contract, which can only be set the price via the `updatePrice()` function on commit `ae67619d072a6891b8ec8a2258bcda5aea18f6c7`.

5.5.1. Description

In the `PriceOracle` contract, the `updatePrice()` function is used to update the price of tokens in the registration or renewal process.

PriceOracle.sol

```
function updatePrice(int256 price, uint8 decimals) external onlyOperator {
    _latestPrice = price;
    _decimals = decimals;
}
```

However, the function can only be called by the `onlyOperator` modifier, which can be directly set by the platform and may have more than one operator. As a result, the platform has control over determining the price of tokens during the registration or renewal process. This centralized control can potentially raise concerns about transparency and fairness in the pricing mechanism.

5.5.2. Remediation

Inspex suggests modifying the operator role to allow only one operator at a time and assigning the operator role to a well-known third-party price oracle, such as Chainlink or TWAP oracle to reduce potential risks associated with the platform's control over the operator role, for example.

PriceOracle.sol

```solidity
event NewOperator(address);
address _operator;

modifier onlyOperator {
    if(msg.sender != _operator) {
        revert Unauthorized(msg.sender);
    }
    _;
}

function setOperator(address newOperator) external onlyOwner {
    require(newOperator != address(0), "Invalid operator address");
    _operator = newOperator;
    emit NewOperator(_operator);
}
```

5.6. Missing Live Modifier in Register Function

ID	IDX-006
Target	Registrar
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium The platform users will be unable to reclaim ownership of their domain name through the `reclaim()` function. **Likelihood:** Low It is unlikely to occur because it seems to happen when the platform owner does not set t he base node owner address in the `Registry` contract.
Status	Resolved The Coin98 team has fixed the issue by adding the live modifier on the `_register()` function on commit `39fd74604e64e6b9ab01b33b49a130116b526380`.

5.6.1. Description

The `_register()` function in the `Registrar` contract, which is called by both the `register()` and `registerOnly()` functions, does not include the `live` modifier to verify the contract's availability.

Registrar.sol

```
function register(uint256 id, address owner, uint256 duration) external override returns (uint256) {
    return _register(id, owner, duration, true);
}

/**
 * @notice Register a name, without modifying the registry.
 * @param id The token ID (keccak256 of the label).
 * @param owner The address that should own the registration.
 * @param duration Duration in seconds for the registration.
 */
function registerOnly(uint256 id, address owner, uint256 duration) external returns (uint256) {
    return _register(id, owner, duration, false);
}
```

When the `_register()` internal function is called from `registerOnly()`, it bypasses the flow that updates the registry sub-node owner. Consequently, the registration process will be completed regardless of whether the Registrar contract is the owner of the _baseNode state or not.

Registrar.sol

```
function _register(uint256 id, address owner, uint256 duration, bool updateRegistry) internal onlyController returns (uint256) {
    require(available(id), "Registar: Label is registered");
    require(block.timestamp + duration + _gracePeriod > block.timestamp + _gracePeriod, "Registar: Future overflow"); // Prevent integer overflow
    require(duration >= _minDuration, "Registar: Duration too short");

    _expiries[id] = block.timestamp + duration;
    if (_exists(id)) {
        _burn(id);
    }
    _mint(owner, id);
    if (updateRegistry) {
        _registry.setSubnodeOwner(_baseNode, bytes32(id), owner);
    }

    emit NameRegistered(id, owner, block.timestamp + duration);
    return block.timestamp + duration;
}
```

As a result, platform users will be unable to reclaim ownership of their domain name through the reclaim() function. This is due to the live modifier in the reclaim() function, which verifies the owner of the base node. If the base node owner is not the Registrar contract, the function will revert.

Registrar.sol

```
modifier live() {
      require(_registry.owner(_baseNode) == address(this), "Registar: Registrar is not authorized");
      _;
}
```

Registrar.sol

```
function reclaim(uint256 id, address owner) external override live {
    require(_isApprovedOrOwner(msg.sender, id), "Registar: Caller is not label owner");
    _registry.setSubnodeOwner(_baseNode, bytes32(id), owner);
}
```

5.6.2. Remediation

Inspex suggests adding the live modifier to the _register() function in the Registrar contract. This modifier is used to check the availability of the Registrar contract. By adding this modifier, the contract will ensure that it is live and operational before executing the logic within the _register() function. This check is important to prevent any undesired behavior or errors that could occur if the contract is not available. For example:

Registrar.sol

```
function _register(uint256 id, address owner, uint256 duration, bool updateRegistry) internal live onlyController returns (uint256) {
    require(available(id), "Registar: Label is registered");
    require(block.timestamp + duration + _gracePeriod > block.timestamp + _gracePeriod, "Registar: Future overflow"); // Prevent integer overflow
    require(duration >= _minDuration, "Registar: Duration too short");

    _expiries[id] = block.timestamp + duration;
    if (_exists(id)) {
        _burn(id);
    }
    _mint(owner, id);
    if (updateRegistry) {
        _registry.setSubnodeOwner(_baseNode, bytes32(id), owner);
    }

    emit NameRegistered(id, owner, block.timestamp + duration);
    return block.timestamp + duration;
}
```

5.7. Lack of Input Validation

ID	IDX-007
Target	PriceConfig
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Very Low **Impact:** Low The integer underflow will occur in the registration and renewal processes when the value is improperly set, resulting in denial of services. **Likelihood:** Low This issue occurs when the `_upsaleDiscountPercent` state value is greater than `10000`. There is nothing to prevent the changes from being made; however, this action can only be taken by the contract owner.
Status	Resolved The Coin98 team has fixed the issue by ensuring that the `_upsaleDiscountPercent` state is less or equal `10000` in the `_setPricing()` function on commit `39fd74604e64e6b9ab01b33b49a130116b526380`.

5.7.1. Description

In the `PriceConfig` contract, the `price()` function can be called to get the price of the registration or renewal process.

PriceConfig.sol

```
function price(
    string calldata name,
    uint256 expires,
    uint256 duration
) external view override returns (Price memory) {
    uint256 len = name.strlen();
    uint256 basePrice;

    if (len >= 5) {
        basePrice = _price5Letter * duration;
    } else if (len == 4) {
        basePrice = _price4Letter * duration;
    } else if (len == 3) {
        basePrice = _price3Letter * duration;
    } else if (len == 2) {
```

```
 87         basePrice = _price2Letter * duration;
 88     } else {
 89         basePrice = _price1Letter * duration;
 90     }
 91     if(duration >= _upsaleThreshold) {
 92         basePrice = basePrice * (10000 - _upsaleDiscountPercent) / 10000;
 93     }
 94 
 95     uint256 tokenPrice = uint256(_oracle.latestAnswer());
 96     uint8 priceDecimals = _oracle.decimals();
 97     return Price({
 98         base: _attoUSDToWei(basePrice, tokenPrice, priceDecimals),
 99         premium: _attoUSDToWei(_premium(name, expires, duration), tokenPrice,
priceDecimals),
100         service: _attoUSDToWei(_serviceFee, tokenPrice, priceDecimals)
101     });
102 }
```

When the duration parameter is greater than or equal to the `_upsaleThreshold` state in the `PriceConfig` contract, users can receive a discount from the platform based on the `_upsaleDiscountPercent` state. Both of these states can be set by the contract owner using the `setPricing()` function.

PriceConfig.sol

```
function setPricing(uint256[] memory pricing) external onlyOwner {
    _setPricing(pricing);
}
```

PriceConfig.sol

```
function _setPricing(uint256[] memory pricing) internal {
    _serviceFee = pricing[0];
    _price1Letter = pricing[1];
    _price2Letter = pricing[2];
    _price3Letter = pricing[3];
    _price4Letter = pricing[4];
    _price5Letter = pricing[5];
    _upsaleThreshold = pricing[6];
    _upsaleDiscountPercent = pricing[7];

    emit RentPriceChanged(pricing);
}
```

However, the `_setPricing()` function does not have input validation, which can allow the value of the `_upsaleDiscountPercent` state to exceed `10000`. This can lead to a denial of service when calling the `price()` function in line 92 during the registration or renewal process.

5.7.2. Remediation

Inspex suggests adding input validation in the `_setPricing()` function to prevent setting the `_upsaleDiscountPercent` state value exceeding `10000`, for example.

PriceConfig.sol

```
function _setPricing(uint256[] memory pricing) internal {
    _serviceFee = pricing[0];
    _price1Letter = pricing[1];
    _price2Letter = pricing[2];
    _price3Letter = pricing[3];
    _price4Letter = pricing[4];
    _price5Letter = pricing[5];
    _upsaleThreshold = pricing[6];
    _upsaleDiscountPercent = pricing[7];
    require(_upsaleDiscountPercent <= 10000, "Discount percent cannot exceed 10000");

    emit RentPriceChanged(pricing);
}
```

5.8. Insufficient Logging for Privileged Functions

ID	IDX-008
Target	PriceOracle Registrar RegistrarController VRC25
Category	General Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged functions will be a malicious action.
Status	Resolved The Coin98 team has fixed the issue by emitting all events of privileged functions on commit `e211833ea67af1f908792cfc06a5b85f6ce2d520` .

5.8.1. Description

Privileged functions that are executable by the controlling parties are not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of those privileged functions, allowing the controlling parties to perform actions that cause big impacts on the platform.

For example, the owner can withdraw the ether in the contract by executing the `withdraw()` function in the `RegistrarController` contract, and no events are emitted.

RegistrarController.sol

```
function withdraw() public onlyOwner {
    payable(owner()).transfer(address(this).balance);
}
```

The privileged functions without sufficient logging are as follows:

File	Contract	Function	Modifier
PriceOracle.sol (L:37)	PriceOracle	setOperator()	onlyOwner
Registrar.sol (L:220)	Registrar	setGracePeriod()	onlyOwner

Registrar.sol (L:228)	Registrar	setMinDuration()	onlyOwner
RegistrarController.sol (L:169)	RegistrarController	setPriceConfig()	onlyOwner
RegistrarController.sol (L:179)	RegistrarController	setPartner()	onlyOwner
RegistrarController.sol (L:186)	RegistrarController	setReferralFee()	onlyOwner
RegistrarController.sol (L:193)	RegistrarController	withdraw()	onlyOwner
VRC25.sol (L:216)	RegistrarController	setFee()	onlyOwner

5.8.2. Remediation

Inspex suggests emitting events for the execution of privileged functions, for example:

RegistrarController.sol

```
192 event Withdraw(address owner, uint256 amount);
193 function withdraw() public onlyOwner {
194     emit Withdraw(owner(), address(this).balance);
195     payable(owner()).transfer(address(this).balance);
196 }
```

5.9. Inexplicit Solidity Compiler Version

ID	IDX-009
Target	PriceConfig PriceOracle Registrar RegistrarController ResolverProxy StringUtils Registry Multicallable PublicResolver ResolverBase AddrResolver BinaryResolver NameResolver TextResolver AdvancedERC20 ERC20Recoverable NameEncoder VRC25 BytesUtils
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The Coin98 team has fixed the issue by using the explicit solidity version on commit `e211833ea67af1f908792cfc06a5b85f6ce2d520`.

5.9.1. Description

The Solidity compiler versions declared in the smart contracts were not explicit. Each compilation may be done using different compiler versions, which may potentially result in compatibility issues.

Registrar.sol

```
pragma solidity >=0.8.4;
```

The following tables contain contracts that use an inexplicit Solidity version.

File	Contract	Version
PriceConfig.sol (L:1)	PriceConfig	~0.8.4
PriceOracle.sol (L:1)	PriceOracle	~0.8.4
Registrar.sol (L:1)	Registrar	>=0.8.4
RegistrarController.sol (L:2)	RegistrarController	~0.8.17
ResolverProxy.sol (L:1)	ResolverProxy	~0.8.4
StringUtils.sol (L:1)	StringUtils	>=0.8.4
Registry.sol (L:1)	Registry	>=0.8.4
Multicallable.sol (L:2)	Multicallable	>=0.8.4
PublicResolver.sol (L:2)	PublicResolver	>=0.8.17 <0.9.0
ResolverBase.sol (L:2)	ResolverBase	>=0.8.4
AddrResolver.sol (L:2)	AddrResolver	>=0.8.4
BinaryResolver.sol (L:2)	BinaryResolver	>=0.8.4
NameResolver.sol (L:2)	NameResolver	>=0.8.4
TextResolver.sol (L:2)	TextResolver	>=0.8.4
AdvancedERC20.sol (L:1)	AdvancedERC20	>=0.8.4
ERC20Recoverable.sol (L:2)	ERC20Recoverable	>=0.8.17 <0.9.0
NameEncoder.sol (L:2)	NameEncoder	^0.8.13
VRC25.sol (L:1)	VRC25	^0.8.4
BytesUtils.sol (L:2)	BytesUtils	~0.8.17

5.9.2. Remediation

Inspex suggests fixing the Solidity compiler to the latest stable version. At the time of the audit, the latest stable version of Solidity compiler in major 0.8 is v0.8.20 (https://github.com/ethereum/solidity/releases), for example.

Registrar.sol

```
pragma solidity 0.8.20;
```

5.10. Incorrect Transfer Fee Logging

ID	IDX-010
Target	VRC25
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The Coin98 team has fixed the issue by adding an event emitted in the `_transferFee()` function on commit `39fd74604e64e6b9ab01b33b49a130116b526380`.

5.10.1. Description

The fee value in event logging is incorrect, which leads to misunderstandings for users. In the `VRC25` contract, the `_transferFee()` function is emitting event logs with the `Fee` event using the `_minFee` state as below:

VRC25.sol

```
function _transferFee(address from, address to, uint256 amount) internal {
    uint256 fee = estimateFee(amount);
    if(fee > 0) {
      _transfer(from, _owner, fee);
      emit Fee(from, to, _owner, _minFee);
    }
}
```

However, the `_minFee` state is used inside the `estimateFee()` function for comparing the fee value. The actual fee is returned back and set as the `fee` state inside the `_transferFee()` function, but the function emits the `_minFee` state, which may confuse the user who inspects the log.

VRC25.sol

```
function estimateFee(uint256 value) public view returns (uint256) {
    if(value > _minFee) {
      return value;
    }
    return _minFee;
}
```

5.10.2. Remediation

Inspex suggests changing the logging value from emitting the `_minFee` state to `fee` value.

VRC25.sol

```
function _transferFee(address from, address to, uint256 amount) internal {
    uint256 fee = estimateFee(amount);
    if(fee > 0) {
      _transfer(from, _owner, fee);
      emit Fee(from, to, _owner, fee);
    }
}
```

6. Appendix

6.1. About Inspex

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

inspex

CYBERSECURITY PROFESSIONAL SERVICE