WUSDMaster

Smart Contract Audit Report
Prepared for Wault Finance

Date Issued: Aug 19, 2021
Project ID: AUDIT2021013
Version: V2.0
Confidentiality Level: Public

inspex
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2021013
Version	V1.0
Client	Wault Finance
Project	WUSDMaster
Auditor(s)	Pongsakorn Sommalai
Author	Pongsakorn Sommalai
Reviewer	Weerawat Pawanawiwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
2.0	Aug 19, 2021	Update the reassessment information	Pongsakorn Sommalai
1.0	Aug 15, 2021	Full report	Pongsakorn Sommalai

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 Wault Finance, Inspex team conducted an audit to verify the security posture of the WUSDMaster smart contracts on Aug 11, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of WUSDMaster 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 3 high, 2 medium, 3 low, 1 very low, and 1 info-severity issues. With the project team's prompt response, 3 high, 2 medium, 3 low, and 1 very low-severity issues were resolved in the reassessment, while only 1 very low-severity issue was acknowledged by the team. Therefore, Inspex trusts that WUSDMaster smart contracts have sufficient protections to be safe for public use. However, in the long run, Inspex suggests resolving all issues found in this report.

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), Inpex 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

Wault Finance is a decentralized finance hub that connects all of the primary DeFi use-cases within one simple ecosystem. In short, an all-in-one DeFi Platform!

WUSD is a brand new stablecoin model that has never been done before, taking inspiration from modern stablecoin frameworks such as Frax and Olympus, and improving on their foundations by minimizing the element of uncertainty.

Scope Information:

Project Name	WUSDMaster
Website	https://app.wault.finance/bsc/index.html#wusd
Smart Contract Type	Ethereum Smart Contract
Chain	Binance Smart Chain
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Aug 11, 2021
Reassessment Date	Aug 19, 2021

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: 91c541c2f1c0ac781ddcfb2be6a62555a5e1e8d1)

Contract	Location (URL)
WUSD	https://github.com/WaultFinance/WUSD/blob/91c541c2f1/WUSD.sol
WUSDMaster	https://github.com/WaultFinance/WUSD/blob/91c541c2f1/WUSDMaster.sol
WexWithdrawer	https://github.com/WaultFinance/WUSD/blob/91c541c2f1/WexWithdrawer.sol

Reassessment: (Commit: 5f50a2c7ffff7828c70299e8a9217cfbb926b8c1)

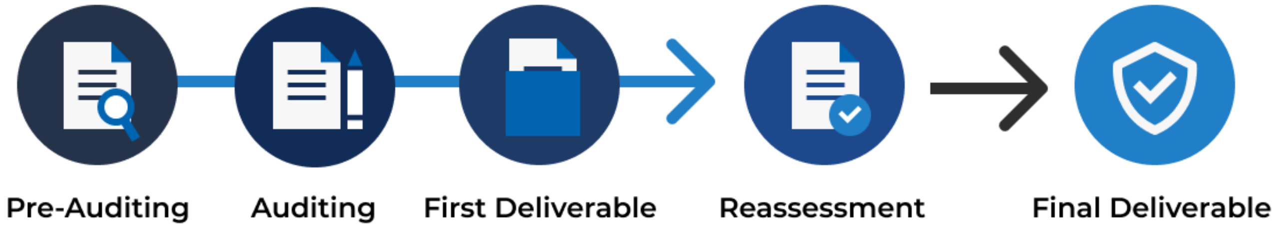
Contract	Location (URL)
WUSD	https://github.com/WaultFinance/WUSD/blob/5f50a2c7ff/WUSD.sol
WUSDMaster	https://github.com/WaultFinance/WUSD/blob/5f50a2c7ff/WUSDMaster.sol
WexWithdrawer	https://github.com/WaultFinance/WUSD/blob/5f50a2c7ff/WexWithdrawer.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they are inherited 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 following audit items were checked during the auditing activity.

General
Reentrancy Attack
Integer Overflows and Underflows
Unchecked Return Values for Low-Level Calls
Bad Randomness
Transaction Ordering Dependence
Time Manipulation
Short Address Attack
Outdated Compiler Version
Use of Known Vulnerable Component
Deprecated Solidity Features
Use of Deprecated Component
Loop with High Gas Consumption
Unauthorized Self-destruct
Redundant Fallback Function
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication
Upgradable Without Timelock
Improper Kill-Switch Mechanism
Improper Front-end Integration
Insecure Smart Contract Initiation

Denial of Service
Improper Oracle Usage
Memory Corruption
Best Practice
Use of Variadic Byte Array
Implicit Compiler Version
Implicit Visibility Level
Implicit Type Inference
Function Declaration Inconsistency
Token API Violation
Best Practices Violation

3.3. Risk Rating

OWASP Risk Rating Methodology[1] 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

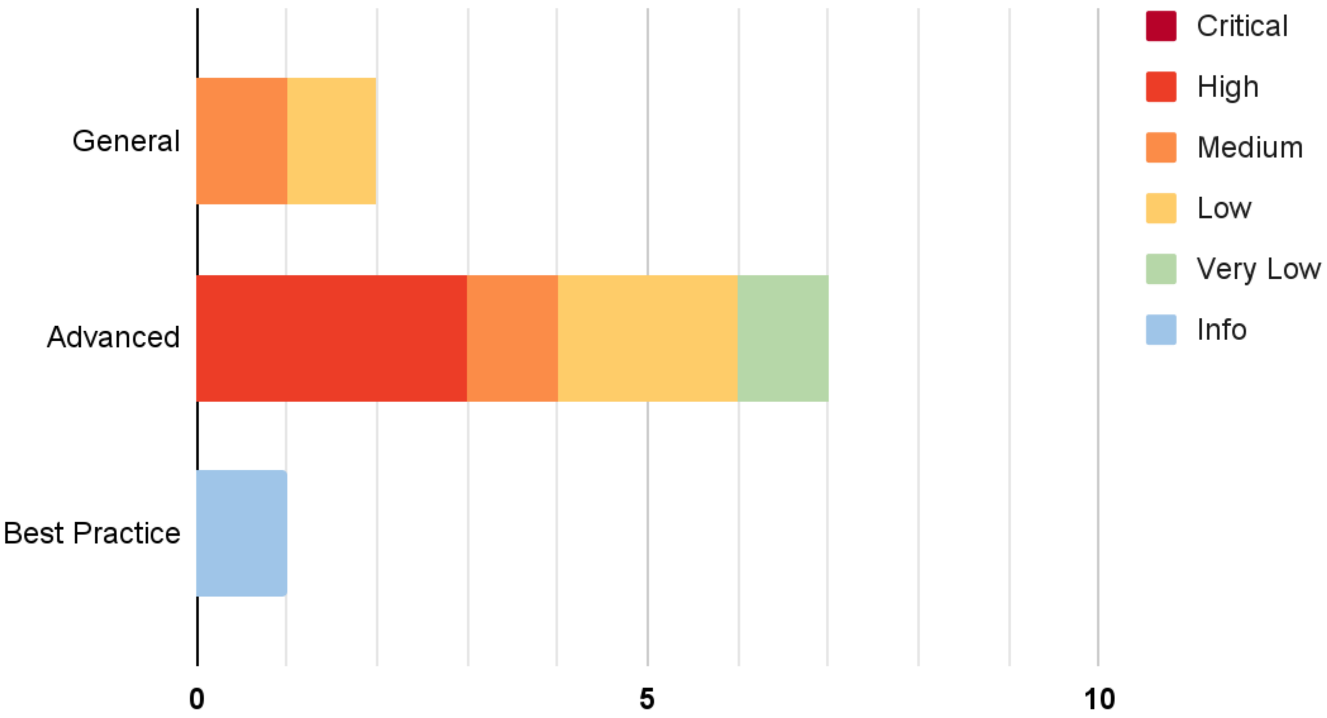
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

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

From the assessments, Inspex has found 10 issues in three categories. The following chart shows the number of the issues categorized into three categories: **General**, **Advanced**, and **Best Practice**.



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	Improper Share Calculation in Redeeming Process	Advanced	High	Resolved
IDX-002	USDT Draining with withdrawUsdt() function	Advanced	High	Resolved *
IDX-003	WUSD Arbitrary Minting with mint() function	Advanced	High	Resolved *
IDX-004	Transaction Ordering Dependence	General	Medium	Resolved
IDX-005	WEX Draining by WexWithdrawer Contract	Advanced	Medium	Resolved *
IDX-006	Improper Modification of Contract State	Advanced	Low	Resolved *
IDX-007	Improper Input Validation	Advanced	Low	Resolved
IDX-008	Centralized Control of State Variable	General	Low	Resolved *
IDX-009	Missing Kill-Switch Mechanism in WUSDMaster	Advanced	Very Low	Resolved
IDX-010	Inexplicit Solidity Compiler Version	Best Practice	Info	No Security Impact

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

5. Detailed Findings Information

5.1. Improper Share Calculation in Redeeming Process

ID	IDX-001
Target	WUSDMaster
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** High With a front-running attack, an attacker will gain an additional $USDT from the `WUSDMaster` while redeeming $WUSD. **Likelihood:** Low It is likely that an attacker can perform a front-running attack on a victim. However, a sufficient redeeming amount is required for the attack to be profitable.
Status	Resolved This issue has been fixed by sending the $WUSD to the dead address in the `redeem()` function and then burning them after calculating the share in the `claim()` function in commit `8e6fd69a78c543a51659ad47ba254b53ad0609d7`.

5.1.1. Description

For the redeeming process in the `WUSDMaster` contract, a user must execute the `redeem()` function to burn $WUSD token in line 745 and save redeeming amount in line 746 as shown in the following source code:

WUSDMaster.sol

```
741 function redeem(uint256 amount) external nonReentrant {
742     require(amount > 0, 'amount cant be zero');
743     require(usdtClaimAmount[msg.sender] == 0, 'you have to claim first');
744 
745     wusd.burn(msg.sender, amount);
746     usdtClaimAmount[msg.sender] = amount;
747     usdtClaimBlock[msg.sender] = block.number;
748 
749     emit Redeem(msg.sender, amount);
750 }
```

Then, in the next block, the user will be able to execute the `claimUsdt()` function for taking their $USDT back. In the `claimUsdt()` function, the $WEX amount is calculated with the share of $WUSD that users are redeeming in line 761 as shown below:

WUSDMaster.sol

```
function claimUsdt() external nonReentrant {
    require(usdtClaimAmount[msg.sender] > 0, 'there is nothing to claim');
    require(usdtClaimBlock[msg.sender] < block.number, 'you cant claim yet');

    uint256 amount = usdtClaimAmount[msg.sender];
    usdtClaimAmount[msg.sender] = 0;

    uint256 usdtTransferAmount = amount * (1000 - wexPermille - treasuryPermille) / 1000;
    uint256 usdtTreasuryAmount = amount * treasuryPermille / 1000;
    uint256 wexTransferAmount = wex.balanceOf(address(this)) * amount / (wusd.totalSupply() + amount);
    usdt.safeTransfer(treasury, usdtTreasuryAmount);
    usdt.safeTransfer(msg.sender, usdtTransferAmount);
    wex.approve(address(wswapRouter), wexTransferAmount);
    wswapRouter.swapExactTokensForTokensSupportingFeeOnTransferTokens(
        wexTransferAmount,
        0,
        swapPathReverse,
        msg.sender,
        block.timestamp
    );

    emit UsdtClaim(msg.sender, amount);
}
```

As described above, there is a gap between $WUSD burning and the `wexTransferAmount` calculation. With a front-running attack, an attacker can use this gap to gain an additional $USDT from the `WUSDMaster` contract. Due to the fact that the $WUSD is burned (`usd.totalSupply()` is decreased) but the balance of $WEX in the `WUSDMaster` is not transferred out (`wex.balanceOf(address(this))` is still unchanged.

Please consider the following attack scenario:

- **$WEX and $USDT:** 1 $WEX per 1 $USDT (for the ease of calculation)
- **Attacker's $WUSD balance:** 1,000
- **Victim's $WUSD balance:** 1,000
- **$WUSD total supply:** 3,000
- **WUSDMaster $WEX balance:** 300

First, the attacker detects the victim's redeeming transaction with 1,000 $WUSD from the transaction pool. Then, the attacker injects their redeeming transaction with $1,000 $WUSD in front of the victim's transaction. The $WUSD total supply will be changed as follows:

```
1st Attacker Tx: $WUSD total supply = 3,000 - 1,000 = 2,000
```

```
2nd Victim Tx: $WUSD total supply = 2,000 - 1,000 = 1,000
```

In the next block, the attacker executes the `claimUsdt()` function and then the following calculation will be performed.

```
wexTransferAmount = wex.balanceOf(address(this)) * amount / (wusd.totalSupply() +
amount)
wexTransferAmount = 300 * 1,000 / (1,000 + 1,000) = 150
```

As the swap rate is 1 $WEX per 1 $USDT, the attacker gains a total of 1,050 $USDT from the `WUSDMaster` contract instead of 1,000 $USDT.

5.1.2. Recommendation

Inspex suggests calculating everything in a single execution or transaction to close the calculation gap.

In this case, the `wexTransferAmount` must be calculated along with reserve the redeemed $WEX in the `redeem()` function as shown below:

WUSDMaster.sol

```
function redeem(uint256 amount) external nonReentrant {
    require(amount > 0, 'amount cant be zero');
    require(usdtClaimAmount[msg.sender] == 0, 'you have to claim first');

    uint256 wexTransferAmount = (wex.balanceOf(address(this)) -
wexReserveAmount) * amount / (wusd.totalSupply() + amount);
    usdtClaimAmount[msg.sender] = amount;
    wexClaimAmount[msg.sender] = wexTransferAmount
    wexReserveAmount =  wexReserveAmount + wexTransferAmount;
    usdtClaimBlock[msg.sender] = block.number;
    wusd.burn(msg.sender, amount);

    emit Redeem(msg.sender, amount);
}
```

Next, in the `claimUsdt()` function, the stored state must be used as shown in the following example:

WUSDMaster.sol

```
function claimUsdt() external nonReentrant {
    require(usdtClaimAmount[msg.sender] > 0, 'there is nothing to claim');
    require(usdtClaimBlock[msg.sender] < block.number, 'you cant claim yet');

    uint256 amount = usdtClaimAmount[msg.sender];
    usdtClaimAmount[msg.sender] = 0;
    uint256 wexTransferAmount = wexClaimAmount[msg.sender];
    wexClaimAmount[msg.sender] = 0;
```

```
    wexReserveAmount =  wexReserveAmount - wexTransferAmount;

    uint256 usdtTransferAmount = amount * (1000 - wexPermille - treasuryPermille) / 1000;
    uint256 usdtTreasuryAmount = amount * treasuryPermille / 1000;

    usdt.safeTransfer(treasury, usdtTreasuryAmount);
    usdt.safeTransfer(msg.sender, usdtTransferAmount);
    wex.approve(address(wswapRouter), wexTransferAmount);
    wswapRouter.swapExactTokensForTokensSupportingFeeOnTransferTokens(
        wexTransferAmount,
        0,
        swapPathReverse,
        msg.sender,
        block.timestamp
    );

    emit UsdtClaim(msg.sender, amount);
}
```

Please note that the remediations for other issues are not yet applied to the example above.

5.2. USDT Draining with withdrawUsdt() function

ID	IDX-002
Target	WUSDMaster
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** High $USDT stored in the `WUSDMaster` can be drained by the `WUSDMaster` contract owner. **Likelihood:** Medium Only the `WUSDMaster` contract owner can execute the `withdrawUsdt()` function. However, the `WUSDMaster` contract owner has a lot of motives to perform this attack.
Status	Resolved * The Wault team has confirmed that the timelock mechanism with a 1-day minimum delay will be set to the `WUSDMaster` contract. Although the timelock mechanism with 1 day has been set, some users might not be able to respond to this action and the manual minting without any limit can cause a high impact on them. Even when the timelock has already been implemented, The user must frequently monitor the timelock contract based on minimum delay.

5.2.1. Description

In the `WUSDMaster` contract, the $USDT can be withdrawn to the `strategist` address by the contract owner as shown in the following source code:

WUSDMaster.sol

```
function withdrawUsdt(uint256 amount) external onlyOwner {
    require(strategist != address(0), 'strategist not set');
    usdt.safeTransfer(strategist, amount);

    emit UsdtWithdrawn(amount);
}
```

Moreover, the contract owner can set the `strategist` state by using the `setStrategistAddress()` function as shown below:

WUSDMaster.sol

```
function setStrategistAddress(address _strategist) external onlyOwner {
    strategist = _strategist;

```

```
    emit StrategistAddressChanged(strategist);
}
```

Please consider the following attack scenario:

- The contract owner changes the `strategist` state to their wallet by using the `setStrategistAddress()` function.
- The contract owner executes the `withdrawUsdt()` function to drain all $USDT from the `WUSDMaster` contract.

5.2.2. Recommendation

Inspex suggests disabling the capability to transfer $USDT out from the `WUSDMaster` contract to prevent anyone from draining the collateral token by removing the `withdrawUsdt()` and `setStrategistAddress()` functions.

5.3. WUSD Arbitrary Minting with mint() function

ID	IDX-003
Target	WUSD
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** High The `WUSD` contract owner can arbitrarily mint the $WUSD token without any limit. **Likelihood:** Medium Only the `WUSD` contract owner can execute the `transferMintership()` function. However, the `WUSD` contract owner has a lot of motive to perform this attack.
Status	Resolved * The timelock mechanism with a 1-day minimum delay already has been set to the `WUSD` contract. Although the timelock mechanism with 1 day has been set, some users might not be able to respond to this action and the manual minting without any limit can cause a high impact on them. - `WUSD` contract: 0x3ff997eaea488a082fb7efc8e6b9951990d0c3ab - `Timelock` contract: 0x7a8d6c614635657660651db4802da08d17ddbbff Even when the timelock has already been implemented, the user must frequently monitor the timelock contract based on minimum delay.

5.3.1. Description

In the `WUSD` contract, the `mint()` function is protected by the `onlyMinter` modifier as shown below:

WUSD.sol

```
function mint(address account, uint256 amount) external onlyMinter {
    _mint(account, amount);
}
```

The `onlyMinter` only allows a specific address to perform the `mint()` function as follows:

WUSD.sol

```
modifier onlyMinter() {
    require(_minter == _msgSender(), "Mintable: caller is not the minter");
    _;
}
```

The current `_minter` state is set to `WUSDMaster` contract that will mint only necessary $WUSD. However, the `_minter` state can still be set by using `transferMintership()` function by the contract owner as shown below:

WUSD.sol

```
function transferMintership(address newMinter) public virtual onlyOwner {
    require(newMinter != address(0), "Mintable: new minter is the zero
address");
    emit MintershipTransferred(_minter, newMinter);
    _minter = newMinter;
}
```

Nevertheless, the timelock mechanism with a 1-day minimum delay already has been set to the `WUSD` contract:

- **WUSD contract:** 0x3ff997eaea488a082fb7efc8e6b9951990d0c3ab
- **Timelock contract:** 0x7a8d6c614635657660651db4802da08d17ddbbff

Although the timelock mechanism with 1 day has been set, some users might not be able to respond to this action and the manual minting without any limit can cause high impact to them.

5.3.2. Recommendation

Inspex suggests disabling the owner of the `WUSD` contract by executing the `renounceOwnership()` function to prevent the manual minting without any limiting action.

5.4. Transaction Ordering Dependence

ID	IDX-004
Target	WUSDMaster
Category	Advanced Smart Contract Vulnerability
CWE	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Risk	**Severity:** Medium **Impact:** Medium Attackers can perform a front-running attack to gain profit in the `stake()` and `claimUsdt()` functions. However, only a portion of the input amount, which can be set up to 50%, will face this issue. **Likelihood:** Medium It is very easy to perform the attack. Moreover, anyone that monitors the BSC's transaction pool can attack users with this issue. However, `maxStakeAmount` state is used to limit the staking amount, resulting in lower profit and motivation in exploiting the `stake()` function.
Status	Resolved This issue has been fixed as recommended in commit `de61d93cd7a35213484827cf32533919c34e732e`.

5.4.1. Description

When users want to mint the $WUSD, the `stake()` and `claimWusd()` functions of `WUSDMaster` contract will swap a portion of input $USDT or $WUSD amount which can be up to 50% to $WEX.

During the swapping of tokens, there is a potential bad-rate swapping since `wswapRouter.swapExactTokensForTokensSupportingFeeOnTransferTokens()` takes `0` as `amountOutMin` in the `stake()` function at line 718 and `claimUsdt()` function at line 767. This means that there is no price tolerance in the swapping process.

WUSDMaster.sol

```
703 function stake(uint256 amount) external nonReentrant {
704     require(amount > 0, 'amount cant be zero');
705     require(wusdClaimAmount[msg.sender] == 0, 'you have to claim first');
706     require(amount <= maxStakeAmount, 'amount too high');
707 
708     usdt.safeTransferFrom(msg.sender, address(this), amount);
709     if(feePermille > 0) {
710         uint256 feeAmount = amount * feePermille / 1000;
711         usdt.safeTransfer(treasury, feeAmount);
```

```
        amount = amount - feeAmount;
    }
    uint256 wexAmount = amount * wexPermille / 1000;
    usdt.approve(address(wswapRouter), wexAmount);
    wswapRouter.swapExactTokensForTokensSupportingFeeOnTransferTokens(
        wexAmount,
        0,
        swapPath,
        address(this),
        block.timestamp
    );

    wusdClaimAmount[msg.sender] = amount;
    wusdClaimBlock[msg.sender] = block.number;

    emit Stake(msg.sender, amount);
}
```

WUSDMaster.sol

```
function claimUsdt() external nonReentrant {
    require(usdtClaimAmount[msg.sender] > 0, 'there is nothing to claim');
    require(usdtClaimBlock[msg.sender] < block.number, 'you cant claim yet');

    uint256 amount = usdtClaimAmount[msg.sender];
    usdtClaimAmount[msg.sender] = 0;

    uint256 usdtTransferAmount = amount * (1000 - wexPermille - treasuryPermille) / 1000;
    uint256 usdtTreasuryAmount = amount * treasuryPermille / 1000;
    uint256 wexTransferAmount = wex.balanceOf(address(this)) * amount / (wusd.totalSupply() + amount);
    usdt.safeTransfer(treasury, usdtTreasuryAmount);
    usdt.safeTransfer(msg.sender, usdtTransferAmount);
    wex.approve(address(wswapRouter), wexTransferAmount);
    wswapRouter.swapExactTokensForTokensSupportingFeeOnTransferTokens(
        wexTransferAmount,
        0,
        swapPathReverse,
        msg.sender,
        block.timestamp
    );

    emit UsdtClaim(msg.sender, amount);
}
```

An example below demonstrates the impact of bad-rate swapping:

The formula to calculate the output price is as follows (swapping fee is ignored):

```
output = amountIn * reserveOut / (reserveIn + amountIn)
```

Assuming the reserve amounts of tokens in the pool before being manipulated are as follows:

reserveUSDT	reserveWEX
50	50

The contract swaps 5 $USDT to $WEX.

```
output = 5 * 50 / (50 + 5) = 4.54
```

As a result, swapping 5 $USDT will get 4.54 $WEX.

However, if this transaction is being front-run with 10 $USDT, the price will be worse as follows:

reserveUSDT	reserveWEX
60	41.67

The contract swaps 5 $USDT to $WEX.

```
output = 5 * 41.67 / (60 + 5) = 3.2053
```

After that, the current reserve amount of tokens in pool will be as follows:

reserveUSDT	reserveWEX
65	38.46

Finally, the front-runner can swap their 8.33 $WEX back to $USDT. They will gain 11.57 $USDT back as shown below:

```
output = 8.33 * 65 / (38.46 + 8.33) = 11.57
```

As a result, swapping 5 $USDT will get only 3.2053 $WEX instead of 4.45 $WEX. Moreover, the front-runner will gain 1.57 $USDT from the swap pool.

However, the `WUSDMaster` contract has the mechanism to limit the staking amount in line 706 as shown below:

WUSDMaster.sol

```
703 function stake(uint256 amount) external nonReentrant {
704     require(amount > 0, 'amount cant be zero');
```

```
    require(wusdClaimAmount[msg.sender] == 0, 'you have to claim first');
    require(amount <= maxStakeAmount, 'amount too high');

    usdt.safeTransferFrom(msg.sender, address(this), amount);
    if(feePermille > 0) {
        uint256 feeAmount = amount * feePermille / 1000;
        usdt.safeTransfer(treasury, feeAmount);
        amount = amount - feeAmount;
    }
    uint256 wexAmount = amount * wexPermille / 1000;
    usdt.approve(address(wswapRouter), wexAmount);
    wswapRouter.swapExactTokensForTokensSupportingFeeOnTransferTokens(
        wexAmount,
        0,
        swapPath,
        address(this),
        block.timestamp
    );

    wusdClaimAmount[msg.sender] = amount;
    wusdClaimBlock[msg.sender] = block.number;

    emit Stake(msg.sender, amount);
}
```

This mechanism reduces the attacker's profit and motivation in exploiting the `stake()` function.

This mechanism is implemented to only the `stake()` function and will work only when `maxStakeAmount` is set to a small amount based on the current TVL of the swap pool.

5.4.2. Recommendation

Inspex suggests calculating the `amountOutMin` from the front-end, forwarding it through the function parameters, and setting it as the price tolerance of swap function as shown in the following examples:

WUSDMaster.sol

```
function stake(uint256 amount, uint256 amountOutMin) external nonReentrant {
    require(amount > 0, 'amount cant be zero');
    require(wusdClaimAmount[msg.sender] == 0, 'you have to claim first');
    require(amount <= maxStakeAmount, 'amount too high');

    usdt.safeTransferFrom(msg.sender, address(this), amount);
    if(feePermille > 0) {
        uint256 feeAmount = amount * feePermille / 1000;
        usdt.safeTransfer(treasury, feeAmount);
        amount = amount - feeAmount;
    }
```

```
    uint256 wexAmount = amount * wexPermille / 1000;
    usdt.approve(address(wswapRouter), wexAmount);
    wswapRouter.swapExactTokensForTokensSupportingFeeOnTransferTokens(
        wexAmount,
        amountOutMin,
        swapPath,
        address(this),
        block.timestamp
    );

    wusdClaimAmount[msg.sender] = amount;
    wusdClaimBlock[msg.sender] = block.number;

    emit Stake(msg.sender, amount);
}
```

WUSDMaster.sol

```
function claimUsdt(uint256 amountOutMin) external nonReentrant {
    require(usdtClaimAmount[msg.sender] > 0, 'there is nothing to claim');
    require(usdtClaimBlock[msg.sender] < block.number, 'you cant claim yet');

    uint256 amount = usdtClaimAmount[msg.sender];
    usdtClaimAmount[msg.sender] = 0;

    uint256 usdtTransferAmount = amount * (1000 - wexPermille - treasuryPermille) / 1000;
    uint256 usdtTreasuryAmount = amount * treasuryPermille / 1000;
    uint256 wexTransferAmount = wex.balanceOf(address(this)) * amount / (wusd.totalSupply() + amount);
    usdt.safeTransfer(treasury, usdtTreasuryAmount);
    usdt.safeTransfer(msg.sender, usdtTransferAmount);
    wex.approve(address(wswapRouter), wexTransferAmount);
    wswapRouter.swapExactTokensForTokensSupportingFeeOnTransferTokens(
        wexTransferAmount,
        amountOutMin,
        swapPathReverse,
        msg.sender,
        block.timestamp
    );

    emit UsdtClaim(msg.sender, amount);
}
```

Please note that the remediations for other issues are not yet applied to the example above.

5.5. WEX Draining by WexWithdrawer Contract

ID	IDX-005
Target	WexWithdrawer
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium $WEX stored in the `WUSDMaster` can be drained by the `WexWithdrawer` contract owner. **Likelihood:** Medium Only `WexWithdrawer` contract owner can execute `withdraw()`, `deposit()`, `initiateMasterChange()`, and `changeMaster()` functions. However, the `WexWithdrawer` contract owner has a lot of motive to perform this attack.
Status	Resolved * The built-in timelock mechanism with 2 days minimum delay already has been set to the `changeMaster()` function of `WexWithdrawer` contract. However, some users might not be able to respond to this action and the token draining can cause a high impact on them. Even when the timelock has already been implemented, the user must frequently monitor the timelock contract based on minimum delay.

5.5.1. Description

In the `WexWithdrawer` contract, the `withdraw()` function can be used to withdraw all $WEX from the `WUSDMaster` contract as shown below:

WexWithdrawer.sol

```
function withdraw(uint256 amount) external onlyOwner {
    wusdMaster.withdrawWex(amount);

    emit Withdraw(amount);
}
```

Moreover, the $WEX can be transferred back to the `WUSDMaster` contract by using the `deposit()` function as follows:

WexWithdrawer.sol

```
function deposit(uint256 amount) external onlyOwner {
    wex.safeTransfer(address(wusdMaster), amount);

    emit Deposit(amount);
```

```
}
```

Unfortunately, the `wusdMaster` state can be changed by using `initiateMasterChange()` and `changeMaster()` functions as follows:

WexWithdrawer.sol

```
function initiateMasterChange(uint256 timestamp, IWUSDMaster _wusdMaster)
external onlyOwner {
    require(!isMasterChangeInitiated, 'change already initiated');
    require(timestamp >= block.timestamp + 48 hours, 'timestamp not valid!');
    require(address(_wusdMaster) != address(0),"zero address");

    isMasterChangeInitiated = true;
    masterChangeTimestamp = timestamp;
    newWusdMaster = _wusdMaster;

    emit InitiateMasterChange(timestamp, address(_wusdMaster));
}
```

WexWithdrawer.sol

```
function changeMaster() external onlyOwner {
    require(isMasterChangeInitiated, 'change not initiated');
    require(block.timestamp >= masterChangeTimestamp, 'not yet possible');

    wusdMaster = newWusdMaster;

    isMasterChangeInitiated = false;
    masterChangeTimestamp = 0;
    newWusdMaster = IWUSDMaster(address(0));

    emit MasterChanged(address(wusdMaster));
}
```

Please consider the following attack scenario:

- The attacker performs the `initiateMasterChange()` function in order to prepare the changing of `wusdMaster` state to their wallet.
- After waiting for 2 days, the attacker executes the `withdraw()` function to drain all $WEX from the `WUSDMaster` contract to the `WexWithdrawer` contract.
- The attacker executes the `chargemaster()` function to change the `wusdMaster` state to their wallet.
- The attacker executes the `deposit()` function to transfer all $WEX to their wallet.

As can be seen above, the timelock mechanism with 2 days minimum delay has already been set to protect the `changeMaster()` function of the `WexWithdrawer` contract. However, some users might not be able to respond to this action and the token draining can cause a high impact on them.

5.5.2. Recommendation

Inspex suggests disabling the capability to change the `wusdMaster` contract by removing the `initiateMasterChange()`, `cancelMasterChange()`, and `changeMaster()` functions from the `WexWithdrawer` contract.

In case that the `WexWithdrawer` cannot be modified and redeployed, Inspex suggests implementing a shield contract that forwards only the `withdraw()` and `deposit()` functions to the `WexWithdrawer` contract.

5.6. Improper Modification of Contract State

ID	IDX-006
Target	WUSDMaster
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium Changing the `wexPermille` or `treasuryPermille` states can cause the $WUSD to be unredeemable, or cause $USDT to be unusable and remain in the `WUSDMaster` contract. **Likelihood:** Low It is very unlikely that the `wexPermille` or `treasuryPermille` state will be changed.
Status	Resolved * The Wault team has clarified that these functions will be used only if it is governed by the holders. If such proposal is approved and the Wault team will decide to increase $WEX collateral to 15%, the Wault team will perform the following steps: 1. Withdraw a portion of $USDT from `WUSDMaster` contract 2. Buy $WEX with withdrawn $USDT 3. Deposit the $WEX acquired to `WUSDMaster` contract However, without performing the above steps, the risk still remains. The user should monitor the increasing collateral process when this process is performed.

5.6.1. Description

In the `WUSDMaster` contract, the `wexPermille` and `treasuryPermille` states are used to calculate the $USDT amount that will be sent to the user in lines 759 and 763 as shown below:

WUSDMaster.sol

```
752 function claimUsdt() external nonReentrant {
753     require(usdtClaimAmount[msg.sender] > 0, 'there is nothing to claim');
754     require(usdtClaimBlock[msg.sender] < block.number, 'you cant claim yet');
755 
756     uint256 amount = usdtClaimAmount[msg.sender];
757     usdtClaimAmount[msg.sender] = 0;
758 
759     uint256 usdtTransferAmount = amount * (1000 - wexPermille -
treasuryPermille) / 1000;
760     uint256 usdtTreasuryAmount = amount * treasuryPermille / 1000;
761     uint256 wexTransferAmount = wex.balanceOf(address(this)) * amount /
(wusd.totalSupply() + amount);
```

```
        usdt.safeTransfer(treasury, usdtTreasuryAmount);
        usdt.safeTransfer(msg.sender, usdtTransferAmount);
        wex.approve(address(wswapRouter), wexTransferAmount);
        wswapRouter.swapExactTokensForTokensSupportingFeeOnTransferTokens(
            wexTransferAmount,
            0,
            swapPathReverse,
            msg.sender,
            block.timestamp
        );

        emit UsdtClaim(msg.sender, amount);
    }
```

The `wexPermille` and `treasuryPermille` can be changed by using `setFeePermille()` and `setTreasuryPermille()` functions as follows:

WUSDMaster.sol

```
function setTreasuryPermille(uint _treasuryPermille) external onlyOwner {
    require(_treasuryPermille <= 50, 'treasuryPermille too high!');
    treasuryPermille = _treasuryPermille;

    emit TreasuryPermilleChanged(treasuryPermille);
}

function setFeePermille(uint _feePermille) external onlyOwner {
    require(_feePermille <= 20, 'feePermille too high!');
    feePermille = _feePermille;

    emit FeePermilleChanged(feePermille);
}
```

By changing the `wexPermille` or `treasuryPermille` states, the `transferred` $USDT amount will also be changed. Therefore, if the values of `wexPermille` or `treasuryPermille` states are reduced, some of $WUSD will be unclaimable. Vice versa, if the values of `wexPermille` or `treasuryPermille` states are increased, some of $USDT will be stuck and unusable in the `WUSDMaster` contract.

5.6.2. Recommendation

Inspex suggests making the `wexPermille` and `treasuryPermille` states `unchangeable` by removing `setTreasuryPermille()` and `setFeePermille()` functions from the `WUSDMaster` contract.

5.7. Improper Input Validation

ID	IDX-007
Target	WUSDMaster
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Low **Impact:** Medium With improper setting of swap path, the user's tokens will be unusable and stuck in the `WUSDMaster` contract. **Likelihood:** Low It is very unlikely that the swap path will be set as an improper value.
Status	Resolved This issue has been fixed as recommended in commit `de61d93cd7a35213484827cf32533919c34e732e`.

5.7.1. Description

The swap path in the `WUSDMaster` contract can be freely set to any value by using the `setSwapPath()` function as shown below:

WUSDMaster.sol

```
function setSwapPath(address[] calldata _swapPath) external onlyOwner {
    swapPath = _swapPath;

    emit SwapPathChanged(swapPath);
}
```

By setting the improper value to the `swapPath` state, when the user performs `stake()` function, the user's token will be swapped to an unexpected token (not $WEX) in line 716-722 and stuck in the `WUSDMaster` contract as shown below:

WUSDMaster.sol

```
703 function stake(uint256 amount) external nonReentrant {
704     require(amount > 0, 'amount cant be zero');
705     require(wusdClaimAmount[msg.sender] == 0, 'you have to claim first');
706     require(amount <= maxStakeAmount, 'amount too high');
707
708     usdt.safeTransferFrom(msg.sender, address(this), amount);
709     if(feePermille > 0) {
```

```
        uint256 feeAmount = amount * feePermille / 1000;
        usdt.safeTransfer(treasury, feeAmount);
        amount = amount - feeAmount;
    }
    uint256 wexAmount = amount * wexPermille / 1000;
    usdt.approve(address(wswapRouter), wexAmount);
    wswapRouter.swapExactTokensForTokensSupportingFeeOnTransferTokens(
        wexAmount,
        0,
        swapPath,
        address(this),
        block.timestamp
    );

    wusdClaimAmount[msg.sender] = amount;
    wusdClaimBlock[msg.sender] = block.number;

    emit Stake(msg.sender, amount);
}
```

5.7.2. Recommendation

Inspex suggests validating that the first element of `swapPath` must be $USDT and the last element must be $WEX as shown in the following example:

WUSDMaster.sol

```
function setSwapPath(address[] calldata _swapPath) external onlyOwner {
    require(_swapPath.length > 1 && _swapPath[0] == address(usdt) && _swapPath[_swapPath.length - 1] == address(wex), "invalid _swapPath")
    swapPath = _swapPath;

    emit SwapPathChanged(swapPath);
}
```

5.8. Centralized Control of State Variable

ID	IDX-008
Target	WUSDMaster
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standard
Risk	**Severity:** Low **Impact:** Low The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood:** Medium There is potentially nothing to restrict the changes from being done by the owner; however, the changes are limited by fixed values in the smart contracts.
Status	Resolved * The Wault team confirmed that the timelock mechanism with a 1-day minimum delay will be implemented when the `WUSDMaster` contract is deployed.

5.8.1. 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, as the contract is not yet deployed, there is potentially no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

File	Contract	Function	Modifier
WUSDMaster.sol (L:658)	WUSDMaster	setSwapPath()	onlyOwner
WUSDMaster.sol (L:664)	WUSDMaster	setWexPermille()	onlyOwner
WUSDMaster.sol (L:671)	WUSDMaster	setTreasuryPermille()	onlyOwner
WUSDMaster.sol (L:678)	WUSDMaster	setFeePermille()	onlyOwner
WUSDMaster.sol (L:685)	WUSDMaster	setTreasuryAddress()	onlyOwner
WUSDMaster.sol (L:691)	WUSDMaster	setStrategistAddress()	onlyOwner
WUSDMaster.sol (L:697)	WUSDMaster	setMaxStakeAmount()	onlyOwner

WUSDMaster.sol (L:776)	WUSDMaster	withdrawUsdt()	onlyOwner

5.8.2. Recommendation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract.

However, if modifications are needed, Inspex suggests limiting the use of these functions via the following options:

- Implementing community-run governance to control the use of these functions
- Using a timelock contract to delay the changes for a sufficient amount of time

5.9. Missing Kill-Switch Mechanism in WUSDMaster

ID	IDX-009
Target	WUSDMaster
Category	Advanced Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Low **Impact:** Low If an attack happens when the contract is unpassable, further damage cannot be prevented. **Likelihood:** Low It is unlikely for the kill-switch mechanism to be required.
Status	Resolved This issue has been fixed as recommended by adding a kill-switch mechanism and implementing an emergency redeeming process in commit `de61d93cd7a35213484827cf32533919c34e732e`.

5.9.1. Description

Immutability is one of the core principles of the blockchain. If the contract is designed to be non-upgradable, there is no mechanism to prevent contracts from potential failures.

For example, when the `WUSDMaster` contract is deployed, there is no mechanism to protect the contract from potential failures.

WUSDMaster.sol

```
function stake(uint256 amount) external nonReentrant {
    require(amount > 0, 'amount cant be zero');
    require(wusdClaimAmount[msg.sender] == 0, 'you have to claim first');
    require(amount <= maxStakeAmount, 'amount too high');

    usdt.safeTransferFrom(msg.sender, address(this), amount);
    if(feePermille > 0) {
        uint256 feeAmount = amount * feePermille / 1000;
        usdt.safeTransfer(treasury, feeAmount);
        amount = amount - feeAmount;
    }
    uint256 wexAmount = amount * wexPermille / 1000;
    usdt.approve(address(wswapRouter), wexAmount);
    wswapRouter.swapExactTokensForTokensSupportingFeeOnTransferTokens(
```

```
        wexAmount,
        0,
        swapPath,
        address(this),
        block.timestamp
    );

    wusdClaimAmount[msg.sender] = amount;
    wusdClaimBlock[msg.sender] = block.number;

    emit Stake(msg.sender, amount);
}
```

The kill-switch mechanism should be added to the following functions of `WUSDContract`:

- `stake()` function
- `claimWusd()` function
- `redeem()` function (the emergency redeeming function should be implemented)
- `claimUsdt()` function (the emergency redeeming function should be implemented)

5.9.2. Recommendation

Inspex recommends using the emergency stop pattern to protect the contract from potential failures.

In this case, it is recommended to inherit the `Pauseable` abstraction contract of OpenZeppelin to the `WUSDMaster` contract as follows:

WUSDMaster.sol

```
contract WUSDMaster is Ownable, Withdrawable, ReentrancyGuard, Pauseable {
```

Then, implement the **pause()** and **unpause()** function as shown below:

WUSDMaster.sol

```
function pause() external onlyOwner {
    _pause();
}

function unpause() external onlyOwner {
    _unpause();
}
```

Finally, add the `whenNotPaused` modifier to critical external functions, for example:

WUSDMaster.sol

```
function stake(uint256 amount) external whenNotPaused nonReentrant {
    require(amount > 0, 'amount cant be zero');
```

```
    require(wusdClaimAmount[msg.sender] == 0, 'you have to claim first');
    require(amount <= maxStakeAmount, 'amount too high');

    usdt.safeTransferFrom(msg.sender, address(this), amount);
    if(feePermille > 0) {
        uint256 feeAmount = amount * feePermille / 1000;
        usdt.safeTransfer(treasury, feeAmount);
        amount = amount - feeAmount;
    }
    uint256 wexAmount = amount * wexPermille / 1000;
    usdt.approve(address(wswapRouter), wexAmount);
    wswapRouter.swapExactTokensForTokensSupportingFeeOnTransferTokens(
        wexAmount,
        0,
        swapPath,
        address(this),
        block.timestamp
    );

    wusdClaimAmount[msg.sender] = amount;
    wusdClaimBlock[msg.sender] = block.number;

    emit Stake(msg.sender, amount);
}
```

Please note that the remediations for other issues are not yet applied to the example above.

5.10. Inexplicit Solidity Compiler Version

ID	IDX-010
Target	WUSD WUSDMaster WexWithdrawer
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact:** None **Likelihood:** None
Status	**No Security Impact** Only `WUSDMaster` contract has been fixed as recommended in the commit `de61d93cd7a35213484827cf32533919c34e732e`.

5.10.1. 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 the compatibility issues, for example:

WUSD.sol

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;
```

The following table contains all targets which the inexplicit compiler version is declared.

Contract	Version
WUSD	^0.8.0
WUSDMaster	^0.8.0
WexWithdrawer	^0.8.0

5.10.2. Recommendation

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 v0.8.6.

6. Appendix

6.1. About Inspex

CYBERSECURITY PROFESSIONAL SERVICE

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

6.2. References

[1] "OWASP Risk Rating Methodology." [Online]. Available: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology. [Accessed: 08-May-2021]

inspex

CYBERSECURITY PROFESSIONAL SERVICE