



Security Assessment

Hourglass

Audit Summary Report

Jun 07, 2023

Table of contents

Document revisions.	3
Introduction to Sublime Group:	4
Technical Security Assessment and Advisory Services:	4
Smart Contract Auditing:	4
Decentralized Finance (DeFi):	5
Quantitative Trading:	5
Market Making:	5
Audit Test and Reporting Disclaimer:	6
Audit summary	7
Findings Summary	10
List of Issues Found:	12
ISSUE-1 Lack of reentrancy protection	12
ISSUE-2 Missing pause functionality	14
ISSUE-3 DeployCustomTranche function is duplicated	15
ISSUE-4 transferOwnership functions are single step	17
ISSUE-5 depositedAmount not validated	18
ISSUE-6 Risk of centralization in ConvexFraxVault	19
ISSUE-7 Public burn functions	21
Test coverage:	22
TEST-1 Wrong test assertion	22
TEST-2 Wrong assertion description	23
TEST-3 Outdated test	24
Static analysis	25
Integration with Other Protocols	26
Enhancing Interoperability:	26
Risk Assessment:	26
Ongoing Risk Management:	26

Document revisions.

Revision	Description	Date
1.0	Initial report	Jun 07, 2023
1.1	Updated issues resolution status after Hourglass fixes	Jul 20, 2023

Introduction to Sublime Group:

Sublime Group is a leading organization revolutionizing the financial landscape through its expertise in Decentralized Finance (DeFi), quantitative trading, market making, and technical security assessment. With a commitment to innovation and trust, Sublime Group offers cutting-edge solutions and advisory services in the rapidly evolving digital asset ecosystem.

Technical Security Assessment and Advisory Services:

Sublime Group excels in providing cutting-edge technical security assessment and advisory services. Leveraging advanced tools and methodologies, we conduct comprehensive security assessments, penetration testing, and vulnerability analysis to fortify systems and infrastructure. Our team of highly skilled cybersecurity experts possesses extensive expertise in identifying and mitigating potential risks and ensuring compliance with regulatory frameworks. By partnering with Sublime Group, clients benefit from our industry-leading security solutions, enabling them to safeguard their digital assets and maintain a robust security posture that surpasses competitors.

Smart Contract Auditing:

At Sublime Group, we pride ourselves on delivering meticulous smart contract audits that go beyond industry standards. Our experienced auditors combine their deep understanding of blockchain technology with an arsenal of cutting-edge tools and advanced methodologies. In addition to manual code analysis, our auditors utilize static code analysis tools to perform automated checks, machine learning techniques to identify complex vulnerabilities, and comprehensive fuzz testing to detect potential security loopholes. By employing these advanced methods, we thoroughly review the reliability, functionality, and security aspects of smart contract code. Our auditors provide actionable recommendations to address identified vulnerabilities, ensuring the robustness and integrity of our clients' blockchain-based applications. Sublime Group's smart contract auditing process, powered by state-of-the-art LLM and fuzzing tools, puts Sublime ahead of the industry standard. By choosing Sublime Group, clients benefit from our unrivaled expertise and innovative approach, guaranteeing the utmost security for their smart contracts and staying ahead of the curve in the ever-evolving landscape of software security.

Decentralized Finance (DeFi):

Sublime Group leverages decentralized protocols and smart contracts to provide seamless access to decentralized lending, borrowing, yield farming, and decentralized exchanges. Our solutions bridge traditional finance with the blockchain ecosystem, empowering users in the DeFi space.

Quantitative Trading:

Sublime Group's skilled quantitative traders optimize trading strategies using advanced algorithms and data analysis techniques. Our expertise in market dynamics, liquidity, and risk management enables efficient execution and enhanced trading performance across digital asset markets.

Market Making:

Sublime Group ensures market liquidity, reduces spreads, and minimizes price volatility through our market-making services. Our proprietary algorithms and risk management systems contribute to fair and efficient price discovery, benefiting institutional and retail investors.

Audit Test and Reporting Disclaimer:

Sublime Group conducted activities for this project according to the statement of work. However, it is important to note that security assessments have time constraints and rely on client-provided information. Thus, the findings in this report may not encompass all security issues or flaws in the system or codebase.

Sublime Group employs automated testing techniques and manual security reviews to assess software controls. However, automated tools have limitations, such as not capturing all edge cases or incomplete analysis within time limits. These limitations are subject to project time and resource constraints.

Clients should understand that while Sublime Group's test coverage is comprehensive within the project's scope, it may not uncover all potential vulnerabilities or flaws. The audit scope does not cover code provided by third party libraries or protocols that hourglass integrates with. However, we have made due diligence when conducting our test to check for any known vulnerabilities and correct usage of said libraries. Ongoing security assessments and proactive measures are advised to maintain system integrity. Sublime Group remains dedicated to assisting clients in enhancing their security and providing expert guidance throughout the development process.

Audit summary

Hourglass engaged Sublime Group to review the security of its contracts. Our dedicated team has invested 1.5 person-weeks of effort into conducting a security review of the client-provided source code. We have conducted a thorough audit of the smart contracts code provided by Hourglass.

The audit commenced with revision 59ff2b0c5841843ea6cae09927bd24d5fbb8e2a3 on the main branch, which served as the starting point for our evaluation.

Throughout the audit process, our team has diligently reviewed the Hourglass smart contracts, carefully scrutinizing the codebase for potential security vulnerabilities, logical flaws, and adherence to best practices. Our objective was to assess the robustness and integrity of the contracts, identifying any potential issues that could compromise the security and functionality of the system.

As part of the audit work, we engaged in a collaborative effort with Hourglass, ensuring a smooth and effective communication channel between our team and theirs. This allowed for the timely resolution of any issues or concerns that arose during the audit. Hourglass actively participated in addressing the identified issues, promptly implementing fixes and enhancements based on our recommendations.

The audit process involved a comprehensive analysis of the codebase, including but not limited to the smart contracts and associated libraries or dependencies. Our experienced auditors meticulously reviewed the code, examining key aspects such as contract architecture, data handling, access controls, input validation, and adherence to industry best practices.

During the audit, we uncovered several issues, which were promptly communicated to the Hourglass team. We provided detailed reports outlining the identified vulnerabilities and flaws, along with recommended remediation measures. Hourglass demonstrated a strong commitment to security and swiftly addressed the identified issues by implementing the recommended fixes. This collaborative approach between our team and Hourglass ensured a robust and secure smart contract implementation.

It is important to note that the audit conducted on Hourglass smart contracts was focused solely on the code provided by Hourglass. While we diligently assessed the code for potential vulnerabilities, logical flaws, and adherence to best practices, it is essential to acknowledge that unforeseen risks or vulnerabilities may exist beyond the scope of the audit.

Hourglass's proactive engagement during the audit process, including addressing the identified issues, reflects their dedication to ensuring the security and reliability of their smart contracts. By actively participating in the audit and promptly resolving any identified issues, Hourglass showcases a commitment to delivering a secure and trustworthy platform for their users.

In conclusion, the audit conducted on the Hourglass smart contracts involved a comprehensive review of the codebase, analysis of potential vulnerabilities, and collaboration with the Hourglass team to address and rectify identified issues. The proactive engagement from Hourglass in fixing the identified issues demonstrates their commitment to maintaining a secure and robust smart contract implementation.

The contracts were updated by hourglass multiple times during the audit and re-checked by Sublime Group on later revision numbers. The list of all revisions checked during the audit can be found below:

1. 59ff2b0c5841843ea6cae09927bd24d5fbb8e2a3
2. 6e6b71c34dfea993b2376e15277b796f00720c8b
3. D8372925e02129243d429fcd051ea27957194454
4. a9dd559739ef89f7c6d83d896f3dbdcfcce27cba

Below is a list of smart contracts included in the audit scope along with their respective sha256 checksums of the latest revision tested:

Filename	SHA-256
FeeManager.sol	25cc8cdfa64979f678457a08d4001ea21a7fbd507591f488a8eb8e160513de7
HourglassCustodian.sol	69d00354b13b9a84cc0b6888edd34ba1e393fc990a90dc3f37daff6c5c772fc1

RewardsDistributor.sol	74a849e06408e939213c7310c625927d1b3f0eca4b90528c130cc008ee4006d4
ConvexFraxVault.sol	3aaa13b39e593cd84becf73098561aab08053ad1ce057582163ee771c14f4447
HourglassConvexFraxReceipt.sol	4ceb214340492be9c590f85cff8e46b2a8544664de453b4c0cd1d670abbace8b
ERC20Intermediary.sol	1a5ad6a16dc067ff97e91647d8dee9269bf9c4eb89e2da7b38853aa759f02917
GeneralMatureHoldingVault.sol	e7f12b6ba66159f3839b623191118a519bfce3527168378fca8831e7645850f6
HourglassCustodianTest.t.sol (later renamed to HourglassProtocolTest.t.sol)	a4f55ae5b20c71576aed7ae1644dc7cd80c7029e493e1463cf380358da14f5f2

ConvexFraxMaturedHoldings.sol was not verified due to work in progress on the side of Hourglass.

Findings Summary

Our primary focus during the audit was to identify potential vulnerabilities in the Hourglass smart contracts, specifically related to business logic, arithmetic operations, and integration with other protocols and tokens. Our objective was to ensure that the protocol remains secure, preventing any unauthorized access or fund theft. Additionally, we have provided Hourglass with guidance on clean code practices and industry best practices wherever applicable.

Throughout the audit process, we diligently analyzed the smart contracts codebase, resulting in the identification of several issues with varying severity levels. We provided Hourglass with comprehensive guidance and recommendations on security enhancements, best practices, and maintaining clean code standards. Collaboratively, we worked with Hourglass to address and rectify all the issues discovered during the audit.

While evaluating the overall security of the system, we are pleased to report that no critical issues were found that would directly endanger user funds. However, we did identify a range of other issues, with severity levels ranging from high to low. Although the absence of critical issues is commendable, we have advised Hourglass to enhance their clean code practices for better maintainability and security.

The test coverage of the protocol was determined to be satisfactory; however, we recommend improving the quality of the tests. A detailed section in the report outlines all the issues identified within the tests.

Additionally, we found the protocol documentation to be inadequate and in need of improvement. We advised Hourglass to update and expand the documentation to provide comprehensive coverage of the protocol's functionality.

In conclusion, the audit process has uncovered several issues of varying severity within the Hourglass smart contracts codebase. However, we are pleased to report that the overall security of the system is satisfactory, with no critical issues threatening user funds. We have collaborated closely with Hourglass to address and resolve the identified issues, reinforcing their commitment to maintaining a secure protocol.

Furthermore, we recommend Hourglass to enhance their clean code practices, improve the quality of tests, and update the protocol documentation to provide comprehensive information to users. By implementing these recommendations, Hourglass can further enhance the security, reliability, and usability of their protocol.

Below is a summary of all the issues found during the audit divided into respective severity categories ranging from critical to suggestion.

Severity	Number of issues	Remaining after Audit
● critical	0	0
● high	1	0
● medium	3	0
● low	3	0
● suggestion	3	2

List of Issues Found:

ISSUE-1 | Lack of reentrancy protection

Severity	Revision found	Status
● medium	59ff2b0c5841843ea6cae09927bd24d5fbb8e2a3	resolved by hourglass

Description:

Function `deployCustomTranche` in `HourglassCustodian.sol` does not implement reentrancy protection despite interacting with smart contracts allowing for arbitrary callback functions.

Recommendation:

Although the function in question properly implements Checks Effects & Interactions pattern inside, the `_createVault` internal function, it is still recommended to implement the `nonReentrant` check. This can help avoid problems especially in case of future upgrades to the contract that can accidentally break Checks Effects & Interactions pattern.

```
function deployCustomTranche(
    uint256 _assetId,
    uint256[] calldata _maturities,
    uint256 _amount,
    bytes calldata _depositBytes
) public returns (address[] memory) {
    if(!assetIds[_assetId].isActive) revert AssetNotActive();

    uint256 numMaturities = _maturities.length;
    address[] memory newVaults = new address[](numMaturities);

    for (uint256 i; i < numMaturities; i++) {
        // check that the vault doesn't already exist for this maturity
        if(assetIdToMaturityToVault[_assetId][_maturities[i]] != address(0))
            revert VaultAlreadyExists();
        // clone the vault
        newVaults[i] = _createVault(_assetId, _maturities[i], _amount, _depositBytes);
    }
    return newVaults;
}
```

```

function _createVault(
    uint256 _assetId,
    uint256 _maturity,
    uint256 _amount,
    bytes calldata _depositBytes
) internal returns (address newVault) {
    if(_maturity > block.timestamp + assetIds[_assetId].maxDuration)
        revert MaturityBeyondMaxDuration();

    // clone the vault implementation
    newVault = _clone(assetIds[_assetId].vaultImplementation);

    // now that the vault exists, have intermediary pull directly from user to the vault
    IIntermediary(assetIds[_assetId].intermediary).pullFundsFromUserToVault(
        msg.sender,
        newVault,
        assetIds[_assetId].depositToken,
        _amount,
        _depositBytes
    );

    // initialize the new cloned vault
    IProxyVault(newVault).initialize(
        feeManager,
        assetIds[_assetId].depositToken,
        _amount,
        _maturity,
        _assetId,
        assetIds[_assetId].erc1155Receipt,
        assetIds[_assetId].depositVaultInitData,
        _depositBytes
    );

    // store the deployed vault address by the asset id & maturity timestamp
    assetIdToMaturityToVault[_assetId][_maturity] = newVault;

    // set the token id parameters
    IERC1155Receipt(assetIds[_assetId].erc1155Receipt).initializeTokenId(
        _assetId,
        newVault,
        _maturity,
        assetIds[_assetId].erc1155ReceiptInitData
    );

    // store the token id mappings
    assetIdToVaultToTokenId[_assetId][newVault] = _maturity;

    // mint the tokens to the user
    IERC1155Receipt(assetIds[_assetId].erc1155Receipt)
        .mint(msg.sender, _maturity, _amount, "");

    emit NewMaturityCreated(_assetId, newVault, _maturity);
}

```

ISSUE-2 | Missing pause functionality

Severity	Revision found	Status
● high	59ff2b0c5841843ea6cae09927bd24d5fbb8e2a3	resolved by hourglass

Description:

Function `deployCustomTranche` in `HourglassCustodian.sol` does not implement `isPaused` check as other public functions in this contract.

Recommendation:

`deployCustomTranche` should properly implement the `isPaused` check in order to allow for effective pausing of `HourglassCustodian.sol` in case of emergency or when migrating.

```
function deployCustomTranche(
    uint256 _assetId,
    uint256[] calldata _maturities,
    uint256 _amount,
    bytes calldata _depositBytes
) public returns (address[] memory) {
    if(!assetIds[_assetId].isActive) revert AssetNotActive();

    uint256 numMaturities = _maturities.length;
    address[] memory newVaults = new address[](numMaturities);

    for (uint256 i; i < numMaturities; i++) {
        // check that the vault doesn't already exist for this maturity
        if(assetIdToMaturityToVault[_assetId][_maturities[i]] != address(0))
            revert VaultAlreadyExists();
        // clone the vault
        newVaults[i] = _createVault(_assetId, _maturities[i], _amount, _depositBytes);
    }
    return newVaults;
}
```

ISSUE-3 | DeployCustomTranche function is duplicated

Severity	Revision found	Status
● low	59ff2b0c5841843ea6cae09927bd24d5fbb8e2a3	resolved by hourglass

Description:

Function `deployCustomTranche` in `HourglassCustodian.sol` is duplicated with altered parameters set.

Recommendation:

Removed one of the duplicates for gas savings - despite the altered parameters the functionality of the duplicates is almost identical.

```
function deployCustomTranche(
    uint256 _assetId,
    uint256[] calldata _maturities,
    uint256 _amount,
    bytes calldata _depositBytes
) public returns (address[] memory) {
    if(!assetIds[_assetId].isActive) revert AssetNotActive();

    uint256 numMaturities = _maturities.length;
    address[] memory newVaults = new address[](numMaturities);

    for (uint256 i; i < numMaturities; i++) {
        // check that the vault doesn't already exist for this maturity
        if(assetIdToMaturityToVault[_assetId][_maturities[i]] != address(0))
            revert VaultAlreadyExists();
        // clone the vault
        newVaults[i] = _createVault(_assetId, _maturities[i], _amount, _depositBytes);
    }
    return newVaults;
}
```

```

function deployCustomTranche(
    uint256 _assetId,
    uint256[] calldata _maturities,
    uint256[] calldata _amounts,
    bytes calldata _depositBytes
) public returns (address[] memory) {
    if(!assetIds[_assetId].isActive) revert AssetNotActive();

    uint256 numMaturities = _maturities.length;


    // instantiate the return array
    address[] memory newVaults = new address[](numMaturities);

    // for this method of depositing, the initialization amount is the user's deposit
    for (uint256 i; i < numMaturities; i++) {
        // check that the vault doesn't already exist for this maturity
        if(assetIdToMaturityToVault[_assetId][_maturities[i]] != address(0))
            revert VaultAlreadyExists();
        // clone the vault
        newVaults[i] = _createVault(_assetId, _maturities[i], _amounts[i], _depositBytes);
    }

    return newVaults;
}

```


ISSUE-4 | transferOwnership functions are single step

Severity	Revision found	Status
 suggestion	6e6b71c34dfea993b2376e15277b796f00720c8b	Acknowledged by Hourglass

Description:

transferOwnership functions used across the project are single step, this can lead to transferring the ownership to the wrong address by mistake.

Recommendation:

We suggest using a propose & accept pending ownership two step pattern as a safer option.

```
/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 * Can only be called by the current owner.
 */
function transferOwnership(address newOwner) public onlyOwner {
    require(newOwner != address(0), "!adr(0)");
    _transferOwnership(newOwner);
}

/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 * Internal function without access restriction.
 */
function _transferOwnership(address newOwner) internal {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

ISSUE-5 | depositedAmount not validated

Severity	Revision found	Status
● suggestion	6e6b71c34dfea993b2376e15277b796f00720c8b	Fixed by Hourglass

Description:

GeneralMatureHoldingVault depositMatured function does not validate the depositedAmount with the actual transferred balanceOf.

Recommendation:

This is not an error at the moment as the function has access control and is only invoked directly after transferring the funds with the correct amount, however since depositMatured is a public function adding the validation would be the safer option.

```
/// @notice Deposits a specified amount of underlying assets into the target strategy address
/// @dev Assets must be transferred into here first
/// @param amount The amount of underlying assets to deposit
function depositMatured(uint256 amount) external onlyOwner {
    depositedAmount += amount;
}
```

ISSUE-6 | Risk of centralization in ConvexFraxVault

Severity	Revision found	Status
● suggestion	6e6b71c34dfea993b2376e15277b796f00720c8b	Acknowledged by Hourglass

Description

HourglassCustodian.sol admin can change the owner of a ConvexFraxVault via transferVaultOwnership function, ConvexFraxVault can in turn withdraw funds upon maturity without burning the erc1155 receipt simply by calling withdrawMatured on the vault.

Recommendation:

The current design would put the funds locked in ConvexFraxVault at risk in case the HourglassCustodian.sol admin key is compromised. We recognize that this is extremely unlikely, however we would like to suggest that it should not be possible for an EOA owner to withdraw the funds to an arbitrary address without burning the erc1155 receipt. We recommend validating that only HourglassCustodian.sol can call the withdrawMatured function.

ConvexFraxVault function:

```
function withdrawMatured(
    address destination,
    bool toUser,
    address user,
    uint256 userAmount
) external onlyOwner returns (uint256 totalWithdrawn) {
    // claim rewards first so it sends to reward distributor
    claimRewards();

    // withdraw from convex vault to this address
    IConvexVault(depositVault).withdrawLockedAndUnwrap(depositId[0]);

    // get the balance of the deposit token now held here
    totalWithdrawn = IERC20(depositToken).balanceOf(address(this));
    require(totalWithdrawn >= userAmount, "!userAmount");

    // if user triggered this withdrawal, send their portion to them & the rest to the mature vault
    if (toUser) {
```

```

        // send to user
        IERC20(depositToken).safeTransfer(user, userAmount);
        // send to destination mature holding vault
        IERC20(depositToken).safeTransfer(destination, totalWithdrawn - userAmount);
    } else {
        // send to destination mature holding vault
        IERC20(depositToken).safeTransfer(destination, totalWithdrawn);
    }
}

```

HourglassCustodian function:

```

/// @notice Allows the Upgrader to transfer ownership of the vaults.
/// @param _vault The vault to transfer ownership of
/// @param _newOwner The new owner of the vault
/// @dev event emitted at vault
function transferVaultOwnership(address _vault, address _newOwner) external onlyRole(UPGRADER) {
    IProxyVault(_vault).transferOwnership(_newOwner);
}

```

ISSUE-7 | Public burn functions

Severity	Revision found	Status
● low	6e6b71c34dfea993b2376e15277b796f00720c8b	Fixed by Hourglass

Description:

HourglassDepositReceipt contract contains public burn and burnBatch functions.

Recommendation:

As there is no apparent business usage for these functions and burning the receipt without withdrawing funds is not a desired outcome, we recommend removing them. We recommend removing any unused code and unnecessary public functions. Even though the mentioned functionality does not pose a threat in the current code revision, removing it helps maintain clean code inside the project and limits potential entry points for attackers in the future.

```
function burn(address account, uint256 id, uint256 value) public virtual {
    require(
        account == _msgSender() || isApprovedForAll(account, _msgSender()),
        "ERC1155: caller is not token owner or approved"
    );
    _burn(account, id, value);
}
```

```
function burnBatch(address account, uint256[] memory ids, uint256[] memory values) public virtual
{
    require(
        account == _msgSender() || isApprovedForAll(account, _msgSender()),
        "ERC1155: caller is not token owner or approved"
    );
    _burnBatch(account, ids, values);
}
```

Test coverage:

The protocol codebase contained tests located in HourglassProtocolTest.t.sol that covered protocols public functions. The test coverage was deemed acceptable, however during the audit we have identified few issues with the tests listed below:

TEST-1 | Wrong test assertion

Severity	Revision found	Status
● medium	59ff2b0c5841843ea6cae09927bd24d5fbb8e2a3	resolved by hourglass

Description:


Assertion in line 413 of HourglassCustodianTest is wrongly comparing the variable with itself yielding an always true result.

Recommendation:

The assertions should be fixed to test the actual intended outcome of the test.

```
assertEq(rwdTknsEarned[2], rwdTknsClaimed[2], "token 2 - should be the same reward token");
assertEq(rwdAmtEarned[2], rwdAmtClaimed[2], "token 2 - should be the same reward amount");
assertEq(rwdTknsEarned[1], rwdTknsClaimed[1], "token 1 - should be the same reward token");
assertEq(rwdAmtEarned[1], rwdAmtClaimed[1], "token 1 - should be the same reward amount");
assertEq(rwdTknsEarned[0], rwdTknsClaimed[0], "token 0 - should be the same reward token");
assertEq(rwdAmtEarned[0], rwdAmtClaimed[0], "token 0 - should be the same reward amount");
```

TEST-2 | Wrong assertion description

Severity	Revision found	Status
 low	59ff2b0c5841843ea6cae09927bd24d5fbb8e2a3	resolved by hourglass

Description:

Assertions in lines 310 and 318 of HourglassCustodianTest have a wrong description string compared to what they are actually checking.

Recommendation:

The assertions should be fixed so that their description is representative of what they are actually testing.

```
// check that she has the role now
assertTrue(feeManager.hasRole(role, alice), "alice should have the setter role");
// ensure that bob doesn't have the role
assertFalse(feeManager.hasRole(role, bob), "alice should not have the setter role");

vm.startPrank(alice);
vm.expectRevert();
feeManager.grantRole(role, bob);
vm.stopPrank();

// ensure bob still doesn't have the role
assertFalse(feeManager.hasRole(role, bob), "alice should still not have the setter role");
```

TEST-3 | Outdated test

Severity	Revision found	Status
● medium	59ff2b0c5841843ea6cae09927bd24d5fbb8e2a3	resolved by hourglass

Description:

Test `testCannotDeployTrancheAsNonVaultDeployer` in `HourglassCustodianTest` is passing and reverting as expected, but it's due to a `VaultAlreadyExists` error and not any error connected to the user not being a vault deployer. The test is testing a non-existent functionality.

Recommendation:

Fix this outdated test so that it actually tests the intended functionality.

```
function testCannotDeployTrancheAsNonVaultDeployer() public {
    // deploy a new vault
    vm.startPrank(alice);
    vm.expectRevert();
    maturities[0] += 1000;
    maturities[1] += 1000;
    maturities[2] += 1000;
    custodian.deployCustomTranche(assetId, maturities, amounts, "");
    console2.log("SUCCESS - Test cannot deploy vault as non-vault deployer");
    vm.stopPrank();
}
```


Static analysis

During our audit of the Hourglass smart contracts, we conducted a thorough static code analysis using a set of our proprietary rules. This analysis aimed to identify any known bugs or recurring attack vectors. The results revealed that all issues detected were false positives, indicating the absence of actual vulnerabilities or bugs. Our automated examination applied specific rules designed to detect common coding mistakes, security vulnerabilities, and attack patterns. We carefully verified each reported issue and determined that they did not pose real security risks. This static analysis provides additional assurance of the codebase's thorough examination and reinforces confidence in the security and reliability of the Hourglass smart contracts. For more detailed information, please refer to the comprehensive report provided. In conclusion, the static code analysis confirmed the absence of genuine vulnerabilities or bugs, further enhancing the security assessment's credibility.

Solidity ▾

Scanning 30 files with 40 solidity rules.

30/30 tasks 0:00:00

Results

Findings:

FALSE POSITIVE:
hourglass-platform/src/ethereum/eth-vaults/convex-frax-asset/ConvexFraxVault.sol
solidity.basic-arithmetic-underflow
Possible arithmetic underflow

112| bytes32 kekId = IConvexVault(depositVault).stakeLockedCurveLp(initAmount, (_maturityTimestamp - block.timestamp));

FALSE POSITIVE:
hourglass-platform/src/ethereum/eth-vaults/convex-frax-asset/ERC1155Receipt.sol
solidity.erc20-public-burn
Anyone can burn tokens of arbitrary accounts.

107| function burn(address account, uint256 id, uint256 value) public virtual {

Scan Summary

Some files were skipped or only partially analyzed.
Scan was limited to files tracked by git.

Ran 40 rules on 30 files: 2 findings.

Integration with Other Protocols

In the process of auditing the Hourglass contracts, we have identified that the project directly or indirectly interfaces with, or is based upon, a list of existing protocols. These protocols include Curve Finance, Convex Finance, Votium, and Frax Finance. The integration of multiple protocols within a project brings both opportunities and potential risks. This summary highlights the significance integrating with other protocols has for protocol security and emphasizes the limitations of the audit scope.

Enhancing Interoperability:

Integrating with well-established protocols enables the project to leverage existing functionalities and tap into a broader ecosystem. This interoperability provides numerous benefits, including enhanced liquidity, access to additional services, and the ability to leverage established communities. By integrating with these protocols, the project positions itself to deliver a more comprehensive and seamless experience for its users. At the same time, each integration brings another layer of complexity into the project and makes it more difficult for the team to ensure project security.

Risk Assessment:

During our audit, we diligently analyzed the Hourglass contracts for known issues or common mistakes that may arise when interacting with the aforementioned protocols. Our objective was to identify any potential vulnerabilities or risks within the contracts that could impact the integration. However, it is important to note that our audit scope was specifically limited to the Hourglass contracts and did not encompass a comprehensive verification of the integrated protocols themselves.

Ongoing Risk Management:

To mitigate potential risks associated with protocol integrations, it is imperative for the project to maintain a proactive approach to risk management. This includes staying informed about updates, upgrades, and potential vulnerabilities of the integrated protocols. By actively monitoring the security landscape and engaging in ongoing risk assessments, the project can promptly address any emerging threats or vulnerabilities, safeguarding the interests of its users and the overall integrity of the system.

The integration of the project with other protocols, including Curve Finance, Convex Finance, and Frax Finance, expands its capabilities and potential. While we have analyzed the Hourglass contracts for issues related to these integrated protocols, it is important to reiterate that our audit scope did not encompass the verification of the integrated protocols themselves. Therefore, it is crucial for project stakeholders to undertake independent audits of the integrated protocols to ensure their security and reliability. By adopting a proactive approach to risk management and ongoing assessments, the project can effectively navigate the complexities of integration and deliver a secure and robust experience for its users.