Security Review Report NM-0148 WORLDCOIN

Redeem Reserved Grants

(Nov 02, 2023)

Contents

1 Executive Summary

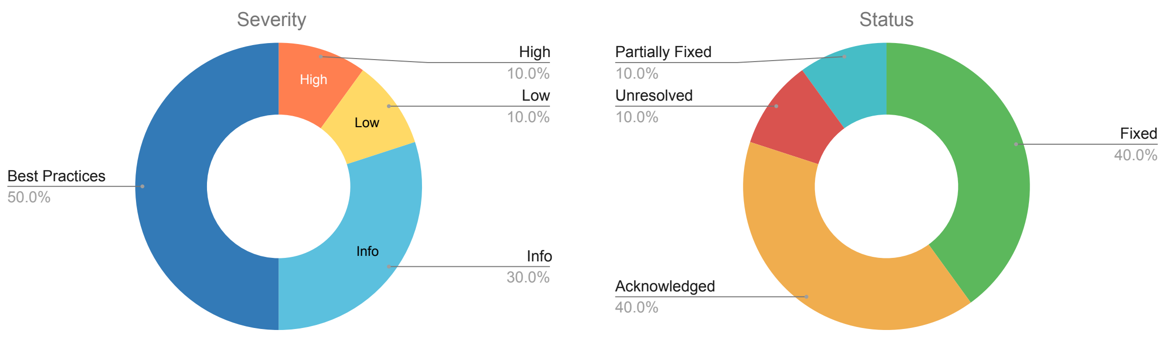
This document presents the security review performed by Nethermind on the Redeem Reserved Grants containing 359 lines of code.

In the updated version of grant contracts, the Worldcoin team has made it possible to redeem reserved grants, which are managed off-chain for cost efficiency. A backend service signs the current timestamp and the user's nullifier hash, storing it in the database during reservation. Upon redemption, this off-chain signature is submitted on-chain along with standard claim parameters.

The previously audited version did not provide public access to the nullifiers map, which posed a problem during redemption as it was difficult to determine if a grant had already been claimed. To address this issue, a try/catch was added to the `checkClaim(...)` function in the legacy contract version, which could catch errors in the nullifier check. However, this was seen as a temporary solution, leading to the creation of two contract versions. Both versions share the same logic, except for the nullifier check. To ensure they are only valid for the necessary grant periods, they are associated with specific Grant contracts that implement time restrictions.

Along the audit of this contract, we report 10 points of attention, where 1 is classified as `High`, 1 is classified as `Low`, 3-are classified as `Info` and 5 are classified as `Best Practices`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.



(a) distribution of issues

(b) status of issues

Fig. 1: Distribution of issues: Critical (0), **High** (1), **Medium** (0), **Low** (1), **Undetermined** (0), **Informational** (3), **Best Practices** (5).
Distribution of status: Fixed (4), **Acknowledged** (4), **Mitigated** (0), **Unresolved** (1), **Partially Fixed** (1)

Summary of the Audit

Audit Type	Security Review
Initial Report	Oct. 23, 2023
Response from Client	Oct. 30, 2023
Final Report	-
Methods	Manual Review, Automated Analysis
Repository	worldcoin-grants-contracts
Commit Hash (Audit)	9d22282f119f134024577031ce7fc46fd2cccb48
Documentation	README.md
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	IGrant.sol	10	13	130.0%	7	30
2	LaunchGrantLegacy.sol	30	6	20.0%	12	48
3	RecurringGrantDrop.sol	141	107	75.9%	66	314
4	RecurringGrantDropLegacy.sol	152	112	73.7%	69	333
5	WLDGrant.sol	26	4	15.4%	12	42
	Total	**359**	**242**	**67.4%**	**166**	**767**

3 Summary of Issues

	Finding	Severity	Update
1	Possible double signaling in claim function	High	Fixed
2	Centralization risk for intended behavior	Low	Acknowledged
3	Unnecessary external call for the current timestamp	Info	Fixed
4	Unnecessary external calls could be replaced by using internal calls	Info	Acknowledged
5	`allowedCallers` check can be bypassed for claiming process	Info	Acknowledged
6	External call before input validations	Best Practices	Unresolved
7	Redundant call to `grant.calculateId(...)` in the function `claimReserved(...)`	Best Practices	Acknowledged
8	Redundant check for `grantId` less than 13 in the function `checkReservationValidity(...)`	Best Practices	Fixed
9	Unused imports	Best Practices	Fixed
10	Wrong or missing NatSpec documentation	Best Practices	Partially Fixed

4 System Overview

Remarks

This audit involves five main contracts grouped as follows:
- grant drop contracts;
- grant contracts.

`RecurringGrantDrop` and `RecurringGrantDropLegacy` are grant drop contracts. They implemented a new feature allowing users to redeem reserved grants. `RecurringGrantDrop` will be used for grants with a `grantId` of 21 or higher. On the other hand, `RecurringGrantDropLegacy` is designed to work alongside the previous deployment of `RecurringGrantDrop` contract for grants with an id between 13 and 20. Each GrantDrop contract points to a Grant contract that manages and verifies grant IDs.

The different grant contracts are: `WLDGrant`, and `LaunchGrantLegacy`. They are responsible for calculating the `grantId`, verifying its validity for claims, and redeeming reservations. `WLDGrant` will be used for `grantId` of 21 or higher, while `LaunchGrantLegacy` will only work for `grantId` between 13 and 20.

Fig. 3 shows how the contracts interact within different `grantId` intervals. If `grantId` is between 13 and 20, users can interact with either the previous `RecurringGrantDrop` contract at `0xe773335550b63eed23a6e60DCC4709106A1F653c` or the `RecurringGrantDropLegacy` contract. Claims are handled at the previous deployment by calling the function `claim(...)` while redeeming a reservation is only possible on the `RecurringGrantDropLegacy` contract by calling the function `claimReserved(...)`.

The legacy contract points to the previous deployment, a call to the function `checkClaim(...)` is done before redeeming a reservation to check the `nullifierHashes` mapping stored in the old contract, and catch the `InvalidNullifier` error. Additionally, both contracts must point to the `LaunchGrantLegacy` grant contract to correctly handle intervals.

When `grantId` is 21 or higher, the previous contracts are no longer used. Only the new deployment of `RecurringGrantDrop` will be operational. Users can claim or redeem a reservation using this contract. This contract should point to the `WLDGrant` grant contract to correctly handle `grantId` intervals.

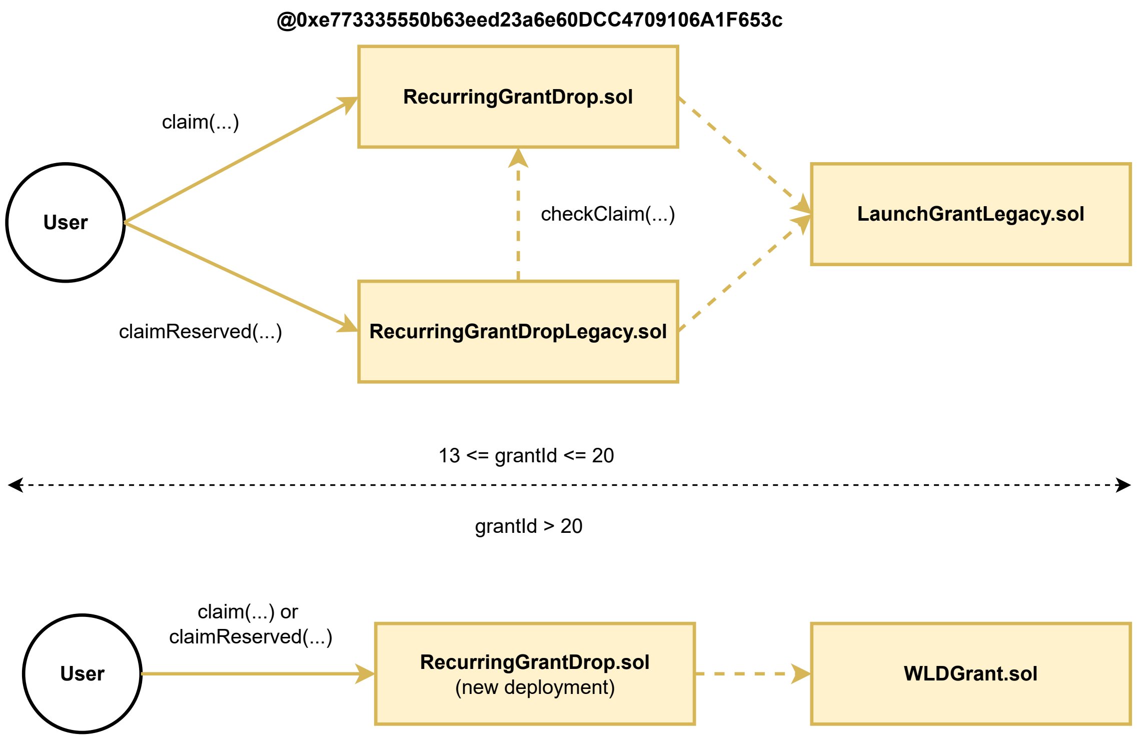

@0xe773335550b63eed23a6e60DCC4709106A1F653c
RecurringGrantDrop.sol
claim(...)
User
checkClaim(...)
LaunchGrantLegacy.sol
claimReserved(...)
RecurringGrantDropLegacy.sol
13 <= grantId <= 20
grantId > 20
claim(...) or claimReserved(...)
User
RecurringGrantDrop.sol (new deployment)
WLDGrant.sol

Fig. 3: Contracts interaction on different grantId intervals

5 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood is a measure of how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding other factors are also considered. These can include but are not limited to: Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if the finding were to be exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Findings

6.1 [High] Possible double signaling in claim function

File(s): `RecurringGrantDropLegacy.sol`, `RecurringGrantDrop.sol`

Description: When `grantId` is in the interval $[13, 20[$, users can claim either at the previous `RecurringGrantDrop` contract or the `RecurringGrantDropLegacy` contract. These two contracts manage separate instances of the `nullifierHashes` mapping. The `RecurringGrantDropLegacy` contract verifies the previously stored value in `RecurringGrantDrop` before executing any claim. However, the other way around is not handled, opening the possibility of a user making a duplicate claim using the same `nullifierHash`. To illustrate this vulnerability, we provide a diagram showing the attack flow and detail the steps involved:

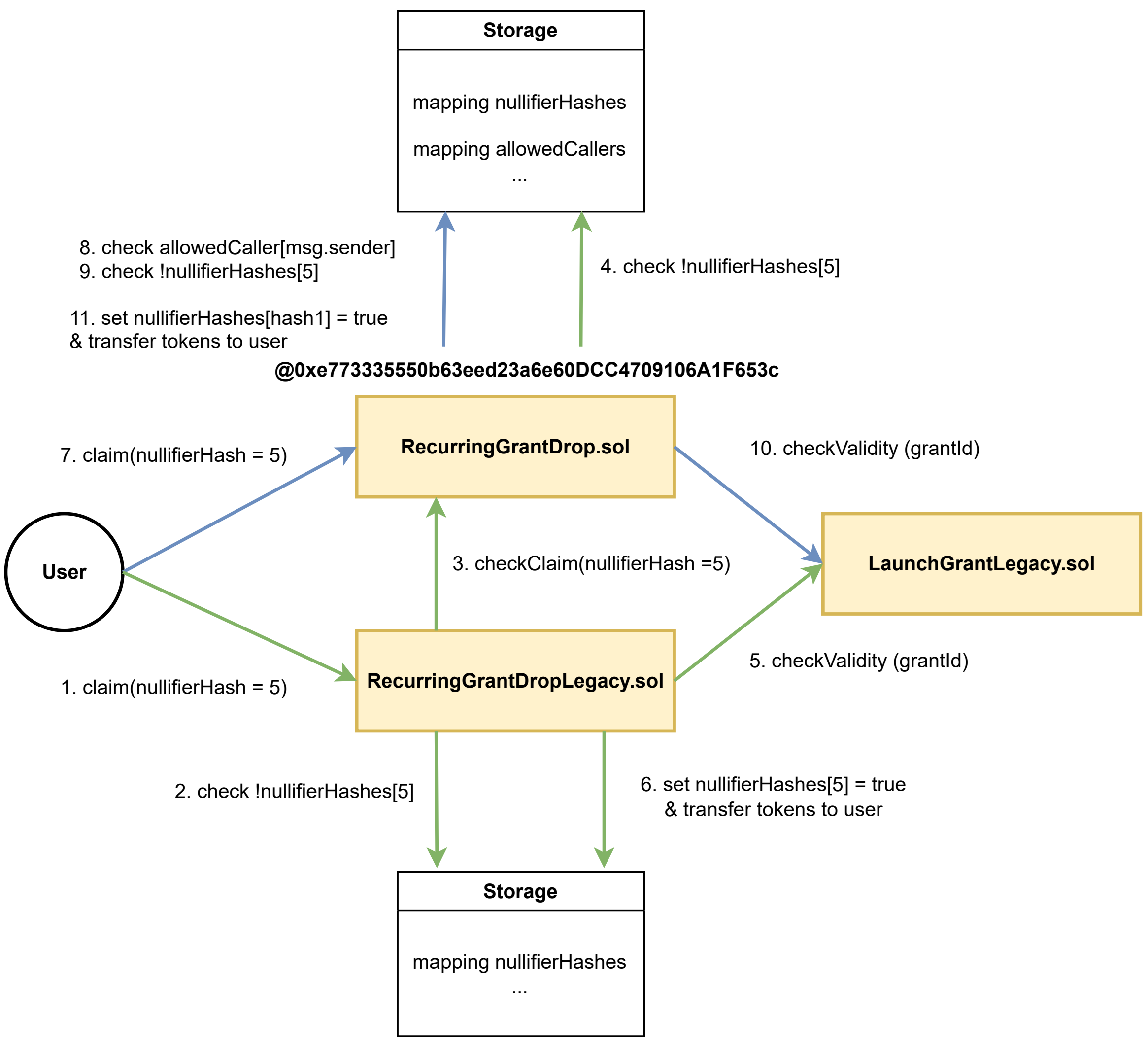

Storage
mapping nullifierHashes
mapping allowedCallers
...
8. check allowedCaller[msg.sender]
9. check !nullifierHashes[5]
11. set nullifierHashes[hash1] = true & transfer tokens to user
4. check !nullifierHashes[5]
@0xe773335550b63eed23a6e60DCC4709106A1F653c
7. claim(nullifierHash = 5)
RecurringGrantDrop.sol
10. checkValidity (grantId)
User
3. checkClaim(nullifierHash =5)
LaunchGrantLegacy.sol
1. claim(nullifierHash = 5)
RecurringGrantDropLegacy.sol
5. checkValidity (grantId)
2. check !nullifierHashes[5]
6. set nullifierHashes[5] = true & transfer tokens to user
Storage
mapping nullifierHashes
...

Fig. 3: Flow of user claiming twice using the same nullifier hash

Consider a user attempting to claim `grantId` with `nullifierHash` 5. Initially, `nullifierHashes[5]` is set to `false` in both contracts.

- (1) User initiates a call to `RecurringGrantDropLegacy.claim(...)` with `nullifierHash == 5`;
- (2,3) Check for `!nullifierHashes[5]` is performed on both contracts;
- (4,5) `checkValidity(...)` is called on the grant contract to verify that the `grantId` is within the accepted range. The contract sets `nullifierHashes[5]` to `true` and transfers tokens to the user;
- (7) User initiates another claim call to `RecurringGrantDrop.claim(...)` with `nullifierHash = 5`;
- (8,9) The contract checks the `allowedCallers` and `nullifierHashes[5]` mappings. The mapping in this contract is not updated from the previous call, resulting in the check successfully passing;

- (10,11) `checkValidity(...)` is called on the grant contract, `nullifierHashes[5]` is set to `true` in the `RecurringGrantDrop` contract, and tokens are transferred to the user;

In this scenario, the user successfully claimed the tokens using the same `nullifierHash` twice.

Recommendation(s): Managing two contracts with separate instances of the same storage data introduces complexity and increases the likelihood of errors and vulnerabilities. To mitigate this, it is advised to restrict claims within the `[13,20)` `grantId` range to be handled exclusively by the `RecurringGrantDrop` contract. This eliminates the possibility of duplicate claims and simplifies the flow.

Status: Fixed

Update from the client: Deployment procedure was now changed to the following:

- Existing contract gets updated to use `LaunchGrantLegacy` (allowing claiming only until grantId 20);
- Claiming logic got removed from `RecurringGrantDropLegacy` (only reservation redemption); This contract also has `LaunchGrantLegacy` set.
- `RecurringGrantDrop` is deployed with `WLDGrant` (allowing both redemption of reservations and normal claiming starting grantId 21);

Also updated in the previously flawed PR comment.

6.2 [Low] Centralization risk for intended behavior

File(s): `RecurringGrantDropLegacy.sol`, `RecurringGrantDrop.sol`, `WLDGrant.sol`, `LaunchGrantLegacy.sol`

Description: The intended behavior of contracts differs based on specific `grantId` intervals. Users are expected to claim or redeem grants at either the previous `RecurringGrantDrop` contract or `RecurringGrantDropLegacy` contract for grantIds $[13, 20[$, and at the new `RecurringGrantDrop` contract for grantIds $[20, \infty[$.

A centralization risk emerges because the logic handling `grantId` intervals are implemented within the Grant contracts. Both the contracts `RecurringGrantDrop` and `RecurringGrantDropLegacy` point to a specific Grant contract, and the owner can update its address through the function `setGrant(...)`. This gives the owner the possibility to disrupt the expected behavior by setting a wrong grant address.

Recommendation(s): Consider revisiting the centralization risk within the protocol and whether it is acceptable for the owner to have this level of control over the intended behavior. If necessary, design changes should be made to the solution in order to ensure a less error-prone approach.

Status: Acknowledged

Update from the client: Acknowledged. Limitation of current design. Ownership will be handed over to a governance contract in the future.

6.3 [Info] Unnecessary external call for the current timestamp

File(s): `RecurringGrantDropLegacy.sol`

Description: In the function `checkClaimReserved(...)`, a timestamp is provided as input and used to fetch the corresponding `grantId`. This `grandtId` is necessary to check the reservation validity through a call to the function `checkReservationValidity(...)`. If the provided `timestamp` indicates a future time, the function reverts. However, if the timestamp corresponds to the current block timestamp (`block.timestamp`), the `grantId` will represent the `currentGrantId` in both `LaunchGrantLegacy` and `WLDGrant` contracts. This leads to the `checkReservationValidity(...)` call always reverting.

```
function checkClaimReserved(
    uint256 timestamp,
    address receiver,
    uint256 root,
    uint256 nullifierHash,
    uint256[8] calldata proof,
    bytes calldata signature
) public {
    uint256 grantId = grant.calculateId(timestamp);

    if (receiver == address(0)) revert InvalidReceiver();
    if (timestamp > block.timestamp) revert InvalidTimestamp();
    ...

    // @audit call to `checkReservationValidity` always reverts with the current timestamp
    grant.checkReservationValidity(timestamp);
    ...
}
```

Recommendation(s): Consider updating the timestamp check to avoid an unnecessary external call. Below is the code with the suggested change:

```
if (timestamp >= block.timestamp) revert InvalidTimestamp();
```

Status: Fixed

Update from the client: Recommendation implemented.

6.4 [Info] Unnecessary external calls could be replaced by using internal calls

File(s): `LaunchGrantLegacy.sol`, `WLDGrant.sol`, `RecurringGrantDropLegacy.sol`

Description: When `this.functionName(...)` is used, the contract makes an external call to itself, causing the sender to become `address(this)`. This is useful when the target function's logic depends on the sender's address. However, in the context of `RecurringGrantDropLegacy`, `LaunchGrantLegacy`, and `WLDGrant` contracts, these are unnecessary. They can be replaced with normal internal calls to save gas while maintaining the same result. For instance, `this.calculateId(...)` can be changed to `calculateId(...)`.

```
function checkReservationValidity(uint256 timestamp) external view override {
    // @audit `this.calculateId()` is unnecessary and wasting gas
    uint256 grantId = this.calculateId(timestamp);

    // No future grants can be claimed.
    if (grantId >= this.getCurrentId()) revert InvalidGrant();

    // Only grants 20 and above can be reserved.
    if (grantId < 20) revert InvalidGrant();

    // Reservations are only valid for 12 months.
    if (block.timestamp > timestamp + 52 weeks) revert InvalidGrant();
}
```

Recommendation(s): Consider using internal calls instead to save gas.

Status: Acknowledged

Update from the client: External calls are kept because of the grant interface.

6.5 [Info] `allowedCallers` check can be bypassed for claiming process

File(s): `RecurringGrantDropLegacy.sol`, `RecurringGrantDrop.sol`

Description: Between `grantId` values 13 and 19, users can claim using either the previous `RecurringGrantDrop` contract or the newly deployed `RecurringGrantDropLegacy` contract. Both paths are expected to operate identically. However, this is not the case, as the `RecurringGrantDrop` contract enforces a caller validation check before allowing a claim. This check is absent in the `RecurringGrantDropLegacy` contract. Consequently, users can bypass this check by exclusively calling the claim function on the latter contract.

```
function claim(uint256 grantId, address receiver, uint256 root, uint256 nullifierHash, uint256[8] calldata proof)
 ↪ external {
    // @audit check is removed in the `RecurringGrantDropLegacy` contract. Users can bypass this check by calling
     ↪ `claim(...)` on the later contract instead.
    if (!allowedCallers[msg.sender]) revert Unauthorized();
    ...
}
```

Recommendation(s): Given that the `allowedCallers` check will be deprecated after `grantId` 20, it is important to verify if this is also the case for lower values when both `RecurringGrantDrop` and `RecurringGrantDropLegacy` contracts are operating. If not, we recommend updating the contract logic to ensure identical claiming behavior across the two contracts. This can be achieved by either integrating the `allowedCallers` in the `RecurringGrantDropLegacy` contract or restricting claims to be done exclusively on the previous `RecurringGrantDrop` contract.

Status: Acknowledged

Update from the client: `allowedCallers` is not relied on in production since the `MultiCall3` contract is also allowed caller. This check can be savely ignored.

6.6 [Best Practices] External call before input validations

File(s): `RecurringGrantDropLegacy.sol`

Description: The `checkClaimReserved(...)` function calls `grant.calculateId(...)` function to retrieve the `grantId`. This call is made before performing input checks, resulting in unneeded gas consumption if checks will revert.

```
function checkClaimReserved(
  uint256 timestamp,
  address receiver,
  uint256 root,
  uint256 nullifierHash,
  uint256[8] calldata proof,
  bytes calldata signature
) public {
  uint256 grantId = grant.calculateId(timestamp);
  // @audit verifications can be done before the external call, to avoid unnecessary gas consumption
  if (receiver == address(0)) revert InvalidReceiver();
  if (timestamp > block.timestamp) revert InvalidTimestamp();
  ...
}
```

Recommendation(s): Consider reordering code to perform input checks before retrieving the `grantId`.

Status: Unresolved

Update from the client: Recommendation implemented.

Update from Nethermind: The suggested optimization in `checkClaimReserved(...)` function was not implemented.

6.7 [Best Practices] Redundant call to `grant.calculateId(...)` in the function `claimReserved(...)`

File(s): `RecurringGrantDrop.sol`, `RecurringGrantDropLegacy.sol`

Description: The function `claimReserved(...)` initially calls the grant contract to calculate `grantId`. It then uses the function `checkClaimReserved(...)` to verify the claim's validity. This function, however, performs the same call to the grant contract to calculate the `grantId` again, which is not efficient.

```
function claimReserved(uint256 timestamp, address receiver, uint256 root, uint256 nullifierHash, uint256[8] calldata
↪ proof, bytes calldata signature)
    external
{
    uint256 grantId = grant.calculateId(timestamp); // @audit First call to calculate ID
    checkClaimReserved(timestamp, receiver, root, nullifierHash, proof, signature);
    ...
}

function checkClaimReserved(uint256 timestamp, address receiver, uint256 root, uint256 nullifierHash, uint256[8]
↪ calldata proof, bytes calldata signature)
    public
{
    uint256 grantId = grant.calculateId(timestamp); // @audit Second call to calculate ID
    ...
}
```

Recommendation(s): Consider passing `grantId` to the function `checkClaimReserved(...)` or have the function `checkClaimReserved(...)` return the value of `grantId` to avoid unnecessary external calls. Another possible solution is to introduce an internal function `_checkClaimReserved(...)` to avoid changing the interfaces of the two existing external functions.

Status: Acknowledged

Update from the client: Not implemented.

6.8 [Best Practices] Redundant check for `grantId` less than 13 in the function `checkReservationValidity(...)`

File(s): `LaunchGrantLegacy.sol`

Description: The function `checkReservationValidity(...)` contains a redundant check for `grantId < 13`, which is unnecessary due to the constraints set by the function `calculateId(...)`. The function `calculateId(...)` ensures that `grantId` is always greater than or equal to 13 or triggers a revert.

```
// @audit Return value at least 13
function calculateId(uint256 timestamp) external pure returns (uint256) {
    if (timestamp < launchDayTimestampInSeconds) revert InvalidGrant();

    uint weeksSinceLaunch = (timestamp - launchDayTimestampInSeconds) / 1 weeks;
    // Monday, 24 July 2023 07:00:00 until Monday, 07 August 2023 06:59:59 (2 weeks)
    if (weeksSinceLaunch < 2) return 13;
    // Monday, 07 August 2023 03:00:00 until Monday, 14 August 2023 02:59:59 (1 week)
    if (weeksSinceLaunch < 3) return 14;
    return 15 + (weeksSinceLaunch - 3) / 2;
}

function checkReservationValidity(uint256 timestamp) external view override {
    uint256 grantId = this.calculateId(timestamp);
    ...
    // @audit Check for grantId < 13 is unnecessary
    if (grantId < 13 || grantId >= 20) revert InvalidGrant();
    ...
}
```

Recommendation(s): Consider removing the redundant check in the function `checkReservationValidity(...)`. It will also help to maintain the consistency with the check in the function `checkValidity(...)`.

Status: Fixed

Update from the client: Recommendation implemented.

6.9 [Best Practices] Unused imports

File(s): RecurringGrantDropLegacy.sol, RecurringGrantDrop.sol

Description: The following imports are not being used in the RecurringGrantDrop and RecurringGrantDropLegacy contracts.

```
// @audit imports are not used by the contract
import { SafeTransferLib } from 'solmate/utils/SafeTransferLib.sol';
...
import { IWorldID } from 'world-id-contracts/interfaces/IWorldID.sol';

```

Recommendation(s): Consider removing unused imports from both contracts.

Status: Fixed

Update from the client: Recommendation implemented.

6.10 [Best Practices] Wrong or missing NatSpec documentation

File(s): `RecurringGrantDropLegacy.sol`

Description: The following issues have been identified in the `RecurringGrantDropLegacy` contract's NatSpec comments:

- Wrong title for contract:;

```
        /// @title RecurringGrantDrop
        // @audit wrong contract title
        /// @author Worldcoin
        contract RecurringGrantDropLegacy is Ownable2Step {
```

- Absence of NatSpec description for `signature` input of `claimReserved(...)` function;
- Outdated `@dev` note referring to the removed `hashToField` function in both `claim(...)` and `claimReserved(...)` functions;

```
    /// @dev hashToField function docs are in lib/world-id-contracts/src/libraries/ByteHasher.sol
    function claim(
```

```
    /// @dev hashToField function docs are in lib/world-id-contracts/src/libraries/ByteHasher.sol
    function claimReserved(
```

Recommendation(s): Consider implementing the documentation updates and fixes listed above.

Status: Partially Fixed

Update from the client: Recommendation implemented.

Update from Nethermind: We confirm that the previous documentation issues were fixed. However, we noticed a few comments that need to be corrected in the new commits. Consider fixing them.

- Comment using the previous `grantId` interval here;
- Date in this comment refers to the old timestamp;

7 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks

The documentation is presented in README.md. Additionally, we found that the insights shared in this comment within the pull request are quite valuable for grasping the contracts' design and intended functionality.

8 Test Suite Evaluation

8.1 Compilation Output

```
> forge compile
[] Compiling...
[] Compiling 46 files with 0.8.19
[] Solc 0.8.19 finished in 3.33s
Compiler run successful!
```

8.2 Tests Output: State Bridge Contracts Upgrade

```
> forge test
[] Compiling...
No files changed, compilation skipped

Running 3 tests for src/test/LaunchGrant.t.sol:MonthlyGrantTest
[PASS] testBiWeeklySwitch() (gas: 23639)
[PASS] testConsecutiveSpecialWeeks() (gas: 13644)
[PASS] testInitialLaunch2Weeks() (gas: 17927)
Test result: ok. 3 passed; 0 failed; finished in 3.89ms

Running 8 tests for src/test/RecurringGrantDrop.t.sol:RecurringGrantDropTest
[PASS] testCanClaim(uint256,uint256) (runs: 256, : 120873, ~: 120873)
[PASS] testCanClaimReservation(uint256) (runs: 256, : 142983, ~: 142983)
[PASS] testCannotClaimClaimed(uint256) (runs: 256, : 138736, ~: 138736)
[PASS] testCannotClaimFuture(uint256,uint256) (runs: 256, : 51735, ~: 51735)
[PASS] testCannotClaimPast(uint256,uint256) (runs: 256, : 51816, ~: 51816)
[PASS] testCannotDoubleClaim(uint256,uint256) (runs: 256, : 126613, ~: 126613)
[PASS] testCannotUpdateGrantIfNotManager(address) (runs: 256, : 334776, ~: 334776)
[PASS] testUpdateGrant() (gas: 337801)
Test result: ok. 8 passed; 0 failed; finished in 53.81ms
```

8.3 Code Coverage

```
> forge coverage
```

The relevant output is presented below.

```
| File                                           | % Lines         | % Statements    | % Branches     | % Funcs        |
|------------------------------------------------|-----------------|-----------------|----------------|----------------|
| src/LaunchGrantLegacy.sol                      | 60.00% (9/15)   | 51.85% (14/27)  | 45.00% (9/20)  | 60.00% (3/5)   |
| src/RecurringGrantDrop.sol                     | 57.14% (24/42)  | 49.09% (27/55)  | 37.50% (9/24)  | 25.00% (3/12)  |
| src/RecurringGrantDropLegacy.sol               | 0.00% (0/44)    | 0.00% (0/57)    | 0.00% (0/22)   | 0.00% (0/13)   |
| src/WLDGrant.sol                               | 100.00% (11/11) | 73.68% (14/19)  | 58.33% (7/12)  | 100.00% (5/5)  |
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer