Security Review Report
NM-0234 Ethereum Foundation Vault

NETHERMIND SECURITY

(May 31, 2024)

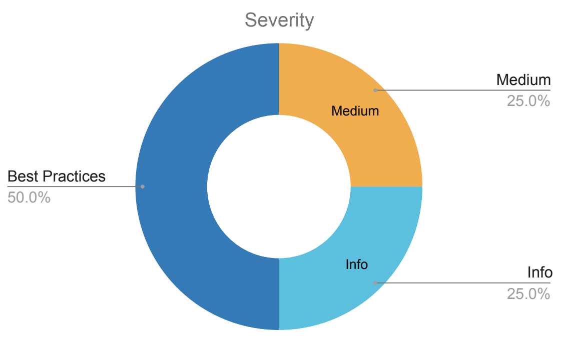
Contents

1 Executive Summary

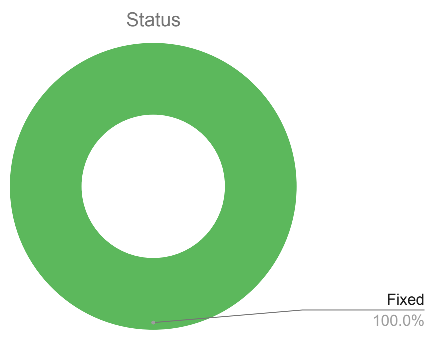
This document outlines the security review conducted by Nethermind on the Holesky Funding Vault by the Ethereum Foundation. The funding vault contract can be controlled by a grant manager, who is able to create grants to given recipients. Both the amount of Ether recieved per claim and the interval between each claim can be specified, allowing for each grant to suit the needs of the recipient. This contract is intended to be used on the Holesky testnet, as a way to reliably distribute large amounts of testnet Ether. Recipients can have multiple grants, and each grant can be individually or batch claimed. To prevent a compromised manager from draining the funding vault, grant managers have a restriction applied to the total value of the grants they can create within a given time window.

The audited code comprises 726 lines of Solidity code. **The audit was performed using**: (a) manual analysis of the codebase and (b) simulation of the smart contracts. Along this document, we report 4 points of attention, where only one is classified as `Medium`. All the other issues are informational or 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 according to the severity



(b) distribution of issues according to the status

Fig 1: (a) Distribution of issues: Critical (0), **High** (0), **Medium** (1), **Low** (0), **Undetermined** (0), **Informational** (1), **Best Practices** (2). **(b) Distribution of status: Fixed** (4), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	May 7, 2024
Response from Client	May 26, 2024
Final Report	May 30, 2024
Methods	Manual Review, Automated Analysis, Tests
Repository	Holešovice Funding Vault
Commit Hash	182d23031c590952dab6b4c6492d35ccd7dcd738
Documentation	Code comments
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/ReentrancyGuard.sol	21	43	204.8%	9	73
2	contracts/IFundingVaultToken.sol	5	1	20.0%	2	8
3	contracts/FundingVaultProxy.sol	49	11	22.4%	14	74
4	contracts/IFundingVault.sol	4	1	25.0%	1	6
5	contracts/FundingVaultToken.sol	40	13	32.5%	12	65
6	contracts/FundingVaultProxyStorage.sol	7	3	42.9%	1	11
7	contracts/FundingVaultV1.sol	362	47	13.0%	80	489
	Total	**488**	**119**	**24.4%**	**119**	**726**

3 Summary of Issues

	Finding	Severity	Update
1	Certain calls can be forced to revert with an invalid grant	Medium	Fixed
2	Manager cooldown period is not consistent	Info	Fixed
3	Missing zero address checks	Best Practices	Fixed
4	Other Best Practice Fixes	Best Practices	Fixed

4 System Overview

`Vault` allows for controlled distribution of ETH on the Holesky testnet. Detailed specification is outlined in this document. The `Vault` defines roles: Grantee, Manager, Admin. Managers create and manage grants, which allow Grantees to spend ETH up to a limit over a specified time. Admins hold special functionality like upgrading contracts, managing grants without limits, or accessing emergency functions. Grants are ERC-721 NFT tokens. Holders of a token (Grantee) can claim ETH up to a limit, but transferring the token is allowed only for Managers. To minimize the risk, Managers' actions are limited to a defined amount of ETH.

The `FundingVaultV1.sol` is an upgradeable contract that holds funds and contains the distribution logic of ETH. The contract uses the following storage variables:

```
address internal _vaultTokenAddr;
uint64 internal _grantIdCounter;
uint32 internal _claimTransferLockTime;
uint128 internal _managerLimitAmount;
uint64 internal _managerLimitInterval;
uint32 internal _managerGrantCooldown;
uint32 internal _managerGrantCooldownLock;
mapping(uint64 => Grant) internal _grants;
mapping(uint64 => uint64) internal _grantClaimLock;
mapping(address => uint64) internal _managerCooldown;
```

The contract exposes the following public or external functions:

Initialization function

```
function initialize(...) public
```

Receives ETH

```
receive(...) external payable
```

Allows admin for rescue transfer of funds

```
function rescueCall(...) public
```

Allows managers to create new grants under defined limits

```
function createGrant(...) public
```

Allows manager to increase or decrease the granted amount

```
function updateGrant(...) public
```

Allows manager to transfer grant NFT token

```
function transferGrant(...) public
```

Allows manager to remove grant

```
function removeGrant(...) public
```

Temporarily lock claiming of the funds for a particular grant

```
function lockGrant(...) public
```

Claims given amount of ETH from either specified grant or all available grants (two overfloaded functions)

```
function claim(...) public
```

Claims given amount of ETH from either specified grant or all available grants to given target (two overloaded functions)

```
function claimTo(...) public
```

Admin-only setters

```
function setPaused(...) public
function setProxyManager(...) public
function setClaimTransferLockTime(...) public
function setManagerGrantLimits(...) public
```

View functions

```
function getVaultToken(...) public view
function getGrants(...) public view
function getGrant(...) public view
function getGrantLockTime(...) public view
function getClaimableBalance(...) public view
function getClaimableBalance(...) public view
function getManagerCooldown(...) public view
```

The contract inherits the following OpenZeppelin contracts:

```
AccessControl
Pausable
```

And the following custom contracts:

```
FundingVaultProxyStorage
ReentrancyGuard
FundingVaultStorage
IFundingVault
```

The `FundingVaultToken.sol` contract is an `ERC721` token that represents a grant. The contract exposes the following non-inherited public or external functions:

Transfers received ETH to the `Vault` contract

```
receive() external payable
```

Mints, burns, or transfers the token

```
function tokenUpdate(...) public
```

Returns `Vault` address

```
function getVault(...) public view
```

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 measures how likely an attacker will uncover and exploit the finding. 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 an attacker exploits the finding. 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 Issues

6.1 [Medium] Certain calls can be forced to revert with an invalid grant

File(s): `FundingVaultV1.sol`

Description: When calling `createGrant` the argument `interval` is used to specify how often the grant recipient should be able to claim. Upon the creation of a grant the `claimTime` is set as the current timestamp minus the interval, which allows the grant recipient to claim immediately. However, it is possible to set the `interval` argument to be equal to the block timestamp, resulting in a `claimTime` of zero.

```
function createGrant(
  address addr,
  uint128 amount,
  uint64 interval
) public onlyRole(GRANT_MANAGER_ROLE) nonReentrant {
  // ...
  _grants[grantId] = Grant({
    claimTime: _getTime() - interval, // @audit Claim time can be zero
    claimInterval: interval,
    claimLimit: amount,
    dustBalance: 0
  });
  // ...
}
```

In many other functions, the existence of a grant is checked by ensuring its `claim time` is not zero. When the function determines that a grant does not exist, the function will revert. A list of all affected functions is shown below:

```
getGrant(uint64)
getGrantLockTime(uint32)
getClaimableBalance()
getClaimableBalance(uint64)
updateGrant(uint64, uint128, uint64)
transferGrant(uint64, address)
removeGrant(uint64)
lockGrant(uint64, uint64)
claim(uint256)
claim(uint64, uint256)
claimTo(uint256, address)
claimTo(uint64, uint256, address)
```

Most of these functions must take the broken grant ID as an argument to cause a revert, meaning the broken grant can be ignored, and the contract will continue to operate normally. However, the function `claim(uint256)` operates by loading and attempting to claim from all grants, meaning that any attempts to claim will always revert through this function. If a smart contract is the recipient of the given grant and the contract is written to only claim through this function, it will not be able to receive any funds, even from other unbroken grants.

Recommendation(s): Consider restricting the interval argument in `createGrant` to some maximum timeframe, such as monthly or bi-annual intervals. This restriction will prevent the interval from being high enough to create an invalid grant.

Status: Fixed

Update from the client: Great catch, fixed via 5e2e2a1f0b43. The `createGrant` function now requires the interval to be smaller than `_getTime()` to avoid the edge case where `claimTime` gets initialized with 0.

6.2 [Info] Manager cooldown period is not consistent

File(s): `FundingVaultV1.sol`

Description: The managers are limited to creating/transferring grants up to the value of 100k ETH per month. This limitation is intended to protect assets from being drained from the vault in case a grant manager wallet is compromised. Functions that check the limit of manager actions are: `createGrant(...)`, `updateGrant(...)`, and `transferGrant(...)`. The check is done in the following line:

```
// from createGrant(...)
require(_managerCooldown[_msgSender()] < _getTime() + _managerGrantCooldownLock, "manager cooldown");
```

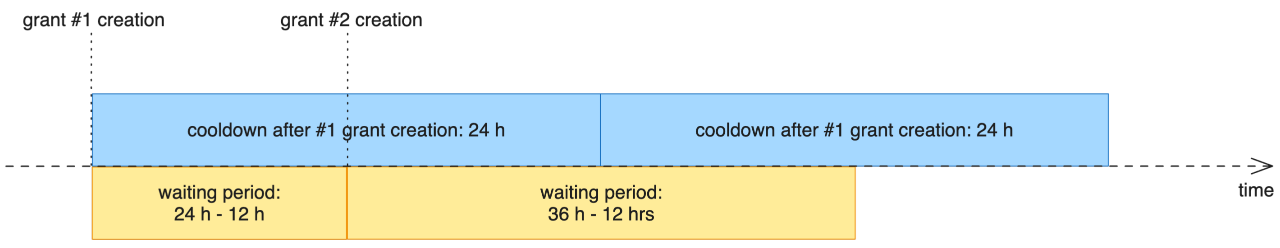
The value in `_managerCooldown` mapping is updated in the following:

```
// from createGrant(...)
_managerCooldown[_msgSender()] += uint64(_managerGrantCooldown * grantQuota / managerQuota) + 1;
```

The cooldown is increased by the number of hours that is proportional to the rato: `grantQuota / managerQuota`. If this ratio is one, `grantQuota == managerQuota`, then the cooldown is increased by `_managerGrantCooldown` which by default is 24 hours. If the grant is created with less quota, then the cooldown increases by some part of 24 hours. During grant creation, the cooldown must pass the check `_managerCooldown[_msgSender()] < _getTime() + _managerGrantCooldownLock`, which can be summarized as: `cooldown < now + 12h`. Since during the creation of a grant (with max quota), the cooldown was updated as `cooldown = now + 24h`, the next creation by this manager may be done after 12 hours. However, the difference in the lock period for managers is not always the same. Consider an example of a manager that:

- Creates a grant with a maximum quota ;
- After 12 hours, creates another grant with a maximum quota ;
- To create the next (third) grant, the manager must wait 24 hours ;

As can be seen, the waiting period is not always the same and depends on when the previous grant was created. In the given example, the cooldown is updated twice by 24 hours, but the lock period is defined as the current time minus twelve hours. That creates inconsistent waiting periods, regardless of the cooldown values set. Below, we present the described example:


grant #1 creation
grant #2 creation
cooldown after #1 grant creation: 24 h
cooldown after #1 grant creation: 24 h
waiting period: 24 h - 12 h
waiting period: 36 h - 12 hrs
time

(c) distribution of issues according to the severity

Recommendation(s): Consider removing `_managerGrantCooldownLock` from the following statement for consistent time lock periods:

```
require(_managerCooldown[_msgSender()] < _getTime() + _managerGrantCooldownLock, "manager cooldown");
```

Status: Fixed

Update from the client: This is an intended behavior of the contract to avoid immediate locking for managing multiple smaller grants in one go.

Without that "lock time threshold", managers would always get locked for a certain period of time (proportional to the grant quota) after each management action. The idea of the threshold is to allow managing certain small grants in one go, without always being locked after each interaction. With the default settings applied, a manager is able to create/update/move grants worth 50k ETH/Month before getting locked the first time.

All the manager cooldown-related settings can also be changed by the contract admin via the `setClaimTransferLockTime(...)` function.

To follow the suggestion and make the cooldown more strict, we could remove that threshold by setting `_managerGrantCooldownLock` to 0 without a code change.

I've updated the description of this behavior in the documentation as well.

6.3 [Best Practice] Missing zero address checks

File(s): FundingVaultV1.sol

Description: The functions that take an address as an argument do not validate that the address is not 0. Below, we list those functions in the FundingVault1 contract:

```
function initialize(address tokenAddr) public {
function setProxyManager(address manager) public onlyRole(DEFAULT_ADMIN_ROLE) {
function rescueCall(address addr, uint256 amount, bytes calldata data) public onlyRole(DEFAULT_ADMIN_ROLE) {
```

Recommendation(s): Consider checking that the address is not `address(0)`.

Status: Fixed

Update from the client: Fixed via a6b6aee8bb41.

6.4 [Best Practice] Other Best Practice fixes

Description: The client proactively fixed small issues discussed during the sync calls, but not reported in this document.

- Consistently use the `block.timestamp` helper function `_getTime()`: 2e58324dff2a;
- Consistently use `uint64` for grantIds: aa74fa84b776;
- Avoid unnecessary casting: 27ad0885ac14 / 6826d22704f7.

Status: Fixed

Update from the client: I've also fixed the other small issues we discussed in the meeting from Apr-30. The small fixes are detailed above (field **Description**).

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 about the documentation

The Ethereum Foundation provided a separate document that specified the desired features, roles and mechanisms in detail. Moreover, the code contained informative comments. The audit team remained in communication with the client during the audit.

8 Test Suite Evaluation

8.1 Hardhat Test Output

```
Funding Vault Tests
    Deployment
       Token vault should be proxy address (670ms)
       Proxy implementation should be vault impl address
       Vault implementations token should be token address
    Grant Management (as owner)
       Create Grant
       Update Grant amount (max unclaimed balance)
       Update Grant amount (no unclaimed balance)
       Update Grant amount (half unclaimed balance)
       Update Grant interval (max unclaimed balance)
       Update Grant interval (no unclaimed balance)
       Update Grant interval (half unclaimed balance)
       Delete Grant
       Transfer Grant
    Manager Limits
       Grant creation limits
       Grant update limits
       Grant transfer limits
    Request funds
       Request max available balance
       Request funds in multiple small claims
       Request funds after half interval
    Fuzz Testing
       Create Grant fuzzing (91ms)
```

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