QuillAudits

AUDIT REPORT

April , 2025

For

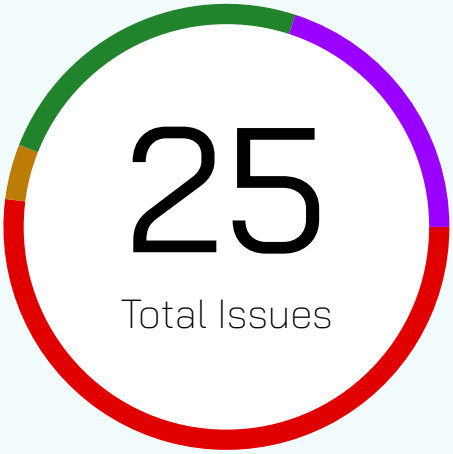
Table of Content

Executive Summary

Project name	Drakkor
Project URL	https://chaosonthechains.com/
Overview	The Presale contract manages multiple presale stages for the Drakkor token, handling whitelisting, pricing, and token purchases.
Audit Scope	The Scope of the Audit is to Analyse Security,Code Quality and Correctness of Drakkor Smart Contracts Source Code Zip File has been provided by Drakkor Team
Contracts in Scope	Presale.sol, Drakkor.sol https://drive.google.com/file/d/1CqB7innuP71ll-HcI1HI_3hqiPpV9X7bg/view?usp=drive_link
Commit Hash	NA
Language	Solidity
Blockchain	EVM
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	April1 2025 - April 7 2025
Updated Code Received	21st April 2025
Review 2	25th April 2025 to 29th April 2025

Fixed In	Zip file provided by Drakkor team
	https://drive.google.com/file/d/1rhSh0D2el1rW0ErgA5BCaF_y8rjuS-pC-/view?usp=drive_link

Number of Issues per Severity



High	13 (52.00%)
Medium	1 (4.00%)
Low	6 (24.00%)
Informational	5 (20.00%)

Severity

Issues	High	Medium	Low	Informational
Open	0	0	0	0
Resolved	**13**	**1**	**5**	**4**
Acknowledged	0	0	**1**	**1**
Partially Resolved	0	0	0	0

Checked Vulnerabilities

- ✓ Access Management
- ✓ Arbitrary write to storage
- ✓ Centralization of control
- ✓ Ether theft
- ✓ Improper or missing events
- ✓ Logical issues and flaws
- ✓ Arithmetic Computations Correctness
- ✓ Race conditions/front running
- ✓ SWC Registry
- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ Exception Disorder
- ✓ Gasless Send
- ✓ Use of tx.origin
- ✓ Malicious libraries
- ✓ Compiler version not fixed
- ✓ Address hardcoded
- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ ERC's conformance
- ✓ Dangerous strict equalities
- ✓ Tautology or contradiction
- ✓ Return values of low-level calls
- ✓ Missing Zero Address Validation
- ✓ Private modifier
- ✓ Revert/require functions
- ✓ Multiple Sends
- ✓ Using suicide
- ✓ Using delegatecall
- ✓ Upgradeable safety
- ✓ Using throw

- [x] Using inline assembly
- [x] Unsafe type inference
- [x] Style guide violation
- [x] Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools And Platforms Used For Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity statistical analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational Issues

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

Security vulnerabilities identified that must be resolved and are currently unresolved.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/ impact of the security issue, but are not completely resolved.

High Severity Issues

Unauthorized Access Control in Critical Functions

Resolved

Path

Drakkor.sol, Presale.sol

Description

In the Drakkor contract the pause() and unpause() functions lack proper access control mechanisms. These functions can be called by any address, allowing any user to halt or resume token purchases at will. The buy() function in the Presale contract explicitly checks if the Drakkor contract is paused:

```
function buy(uint64 _presaleID, uint128 _mintAmount, uint8 _token, address _factoryAddress) public
{
 require(!drakkor.isLocked(), "Drakkor is locked");
 require(!drakkor.isPaused(), "Drakkor is paused");
// ...
}
```

Any malicious actor can call pause() on the Drakkor contract at any time, which will set isPaused to true. This will cause all subsequent calls to buy() to revert because of the require(!drakkor.isPaused(), "Drakkor is paused") check.
This effectively allows any user to completely disable the token sale functionality at will, preventing legitimate users from participating in the presale. Even if the contract owner notices this and calls unpause(), a malicious actor can immediately call pause() again, creating an ongoing denial of service situation.
In Presale contract, modifyWhitelist(), modifyDiscount(), and modifyPrice(), lack the onlyOwner modifier, allowing anyone to modify presale parameters. Any address could call these functions and alter presale parameters, potentially resulting in loss of funds or manipulation of token economics.

Recommendation

1.Add access control modifiers to the pause and unpause functions to restrict their usage to authorized roles only. Since the Presale contract is the owner of the Drakkor contract.
Implement pause() and unpause() functions in the Presale with authorized roles to call the pause() and unpause() functions in the Drakkor contract respectively.

2.Add the onlyOwner modifier to the functions modifyWhitelist(), modifyDiscount(), and modifyPrice() to restrict access.

Unrestricted Access to Kill Function Enables Permanent DoS Attack

Resolved

Path

Drakkor.sol

Description

The Presale contract contains a critical vulnerability where the kill() function sets the isKilled state variable to true with no mechanism to revert this state. Once executed, this permanently blocks core contract functionality, specifically the createPresale() and unlockPresale() functions, as both contain a require(!isKilled, "Contract is killed") check.

The kill() function can be called by anyone, not just the contract owner, as it lacks access control modifiers.

The only requirement to call kill() is that the Drakkor contract must be in a locked state (drakkor.isLocked()).

Since the Drakkor contract initializes with isLocked = true in its constructor, the kill condition is immediately satisfied upon deployment.

There is no function to reset the isKilled flag back to false once it has been set.
Once triggered, the contract permanently loses the ability to create new presales or unlock existing ones, rendering the entire presale mechanism useless.

Recommendation

Modify the kill() function to include the onlyOwner modifier to ensure only the contract owner can trigger this critical function.

Missing Interface Imports

Resolved

Path

Presale.sol

Description

The buy() function references the interfaces: IUniswapV3Factory and IUniswapV3Pool. However, these interfaces are not imported in the contract, making the buy() function unusable. This will cause the contract to fail to compile.

Recommendation

Import the required interfaces at the top of the contract.

Incorrect drakkor.isLocked() Logic in Critical Functions

Resolved

Path

Presale.sol

Description

The functions kill(), modifyWhitelist(), modifyDiscount(), and modifyPrice() contain a logic flaw in their access control condition. According to the function documentation these functions are intended to be called only when the Drakkor contract is not locked However, the implementation currently uses the condition:
require(drakkor.isLocked(), "...");
This enforces that the contract must be locked, which contradicts the intended access restriction described in the comments.

Recommendation

Update the require() statements in the affected functions to properly reflect the intended logic.

Insecure Uniswap V3 Factory Address Handling in Buy Function

Resolved

Path

Presale.sol

Description

The buy() function allows users to provide any address as the _factoryAddress parameter. This parameter is used to interact with a Uniswap V3 factory contract for price calculations. Accepting an arbitrary address from user input creates a significant security vulnerability, as malicious actors could supply addresses of fake factory contracts that return manipulated price data. This could result in users underpaying or overpaying, or the protocol losing funds due to inaccurate rate calculations.

Recommendation

Hardcode or securely configure the factoryAddress (e.g., through a constructor or setter restricted to the owner or admin).

Reliance on Manipulable Uniswap V3 slot0 Price Data in buy() Function

Resolved

Path

Presale.sol

Description

The buy() function retrieves the current price (sqrtPriceX96) directly from Uniswap V3's slot0 to calculate the exchange rate. slot0 provides the instantaneous price of the pool, which is vulnerable to manipulation via MEV bots, flash loans, or sandwich attacks. This allows attackers to artificially inflate or deflate the calculated rate, leading to incorrect payment amounts for minted tokens.

Recommendation

Use the TWAP function instead of slot0 to get the value of sqrtPriceX96.

Protocol is not able to mint more than 100,000 (due to decimals misunderstanding)

Resolved

Path

Presale.sol

Description

mint() function in the Drakkor token contract enforces that _mintAmount must be between 1000 and 100000. However, since the token uses the standard 18 decimals (via OpenZeppelin's ERC20), these numeric values represent sub-token denominations, not full tokens. This effectively restricts minting to extremely small amounts — less than a fraction of a single token, which is likely unintended.

```
function mint(uint128 _mintAmount, address _buyer) external onlyOwner {
 require(_mintAmount > 1000 && _mintAmount <= 100000, "Cannot mint more than 100000 and less then 1000 tokens at once");
...
}
```

The ERC20 contract uses 18 decimals, meaning that 1 full token = 1 * 10^18 units. The current check allows minting only if _mintAmount is between 1000 and 100000: 1000 = 0.000000000000001 tokens 100000 = 0.0000000000001 tokens.

Recommendation

Improve the decimals handling.

Decimal Mismatch Vulnerability in buy Function

Resolved

Path

Presale.sol

Description

The buy function supports purchasing with WETH (18 decimals), USDC (6 decimals), and USDT (6 decimals). However, it does not adjust for decimal mismatches between the presale token price and the payment token. If price is set in 18 decimals (WETH-style), then users paying with USDC/USDT (6 decimals) will be overcharged by a factor of 1e12. Conversely, if price is set in 6 decimals (for USDC/USDT-style), then users paying with WETH (18 decimals) will be undercharged by 1e12.

IERC20(tokens[_token]).transferFrom(msg.sender, reserveAddress, priceDiscounted * rate);

There is no scaling logic here to normalize between token decimals, leading to incorrect transfer amounts. A proper fix would involve adjusting the final transfer amount based on the token's actual decimal places relative to the price unit.

Recommendation

Handle decimals accordingly.

Inconsistent Exchange Rate Calculation for USDC and USDT in buy Function

Resolved

Path

Presale.sol

Description

In the buy function, the contract uses different logic to calculate the exchange rate for USDC and USDT, even though both tokens are dollar-pegged stablecoins. While the theoretical price of USDC and USDT should be the same, the contract performs different mathematical operations for each, including bit-shifting and inversion, which can result in inconsistent rate values. This could cause users to pay different amounts for the same token purchase depending on whether they use USDC or USDT.

Recommendation

Add single logic for stable coin

Incorrect Rate Calculation for USDT in buy() Function Results in Excessive Token Transfer

Resolved

Path

Presale.sol

Description

The buy() function incorrectly calculates the conversion rate when _token == 1 (USDT), leading to an unreasonably high token amount being required for minting drakkor tokens. The issue lies in the way sqrtPriceX96 is processed to compute the rate, resulting in a value that's orders of magnitude larger than expected.
The current logic:

rate = ((sqrtPriceX96 * 1e18)>>96) * ((sqrtPriceX96 * 1e18)>>96);
rate = rate * 10**12 / 10**36;

For sqrtPriceX96 = 2001948226849368737922989573497318 (WETH/USDC pool), the computed rate becomes:

638478850898924304780 (≈ 638e18) https://etherscan.io/address/0x88e6a0c2ddd26feeb64f039a2c41296fcb3f5640#readContract
This leads to users needing to send 638478850898924.304780 USDT for _mintAmount of drakkor token, which is clearly incorrect. This creates loss of user's funds.

Recommendation

Replace the flawed calculation with a correct formula for converting sqrtPriceX96 to a token rate.
https://etherscan.io/address/0x88e6a0c2ddd26feeb64f039a2c41296fcb3f5640

Mint Amount Not Accounted in Payment Calculation.

Resolved

Path

Presale.sol

Description

The buy() function does not multiply the discounted price by _mintAmount before transferring tokens from the user. As a result, a malicious user can pass a very high _mintAmount and only pay for one token at the discounted rate, effectively minting multiple tokens for the price of one.

Impact

1. Users can mint unlimited tokens for the price of one, causing economic loss and supply inflation.
2. Could lead to a complete breakdown of token economics in the presale.

Recommendation

Include _mintAmount in the price calculation before performing the token transfer:

IERC20(tokens[_token]).transferFrom(msg.sender, reserveAddress, uint256(priceDiscounted) * _mintAmount * rate);

Incorrect Discount Calculation in buy Function

Resolved

Path

Presale.sol

Description

In the buy function of the Presale contract, the calculation for the discounted price is implemented incorrectly. The current logic computes the discount amount, rather than subtracting it from the base price to derive the discounted price.

priceDiscounted = presales[_presaleID].price * uint128(discount) / uint128(100);

For example, if the base price is 100 and the discount is 5%, the above code sets the price to 5, rather than the correct discounted price of 95. This results in users being charged only the discount portion, instead of the correct discounted total. this would cause significant underpayment by users, leading to financial loss or token mispricing.

Recommendation

Update the discount calculation logic in the buy function to correctly reflect the discounted price:

priceDiscounted = presales[_presaleID].price * (100 - uint128(discount)) / 100;

This ensures that users pay the correct amount after the discount is applied.

Incorrect Stablecoin Payment Calculation

Resolved

Path

Presale.sol

Description

The buy() function contains a critical mathematical error in its payment calculation that systematically undercharges users paying with stablecoins (USDT/USDC). Due to improper decimal handling:
WETH: 18 decimals (correctly handled via /1e18)
USDT/USDC: 6 decimals (incorrectly uses same /1e18 division)

Recommendation

priceDiscounted * rate * _mintAmount) / 10**18 — this is okay.
For the rest of the tokens:
(priceDiscounted * rate * _mintAmount) / dividen

Medium Severity Issues

Unsafe ERC20 Token Transfers in buy() Function

Resolved

Path

Presale.sol

Description

The buy() function uses IERC20.transferFrom() for token operations. Tokens that do not comply with the ERC20 specification could return false from the transfer function call to indicate the transfer fails, while the calling contract would not notice the failure if the return value is not checked. Checking the return value is a requirement, as written in the EIP-20 specification: Callers MUST handle false from returns (bool success). Callers MUST NOT assume that false is never returned! Some tokens do not return a bool (e.g. USDT) on ERC20 methods. This will make the call break, making impossible to use these tokens.

Recommendation

Use SafeTransferLib or SafeERC20, replace transferFrom for safeTransferFrom when transferring ERC20 tokens.

Low Severity Issues

Unused imported interfaces

Resolved

Path

Drakkor.sol

Description

The Drakkor contract inherits the following interfaces but not used in the Drakkor contract.

```
import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
import "@uniswap/v3-periphery/contracts/interfaces/ISwapRouter.sol";
import "@uniswap/v3-periphery/contracts/libraries/TransferHelper.sol";
import "@uniswap/v3-core/contracts/interfaces/IUniswapV3Factory.sol";
import "@uniswap/v3-core/contracts/interfaces/IUniswapV3Pool.sol";
```

At the same time remove these following imports as well since these are only used for testing purpose.

```
import { Weth } from "./Mocks/WethMock.sol";
import "hardhat/console.sol";
```

Recommendation

Make sure to delete unused imports and import them in the required contract that is Presale.sol.

Contract ownership can transferred to the wrong owner

Resolved

Path

Presale.sol

Description

Access control is a critical part of smart contract security. Within this contract, access control is handled by the Ownable library which allows the owner to relinquish their rights to a new owner in one function call, transferOwnership(). This is risky if the new owner address passed in is malicious or is a victim of an address poisoning attack where some bytes of the address are changed when copied to the clipboard.

Recommendation

Implement two-step ownership that includes confirmation of ownership transfer from the new owner before it is accepted.

Ownership Renouncement Can Break Critical Functions

Resolved

Path

Presale.sol

Description

The Ownable contract includes a renounceOwnership() function that allows the contract owner to relinquish ownership. If this function is called on the Presale contract, it will leave the contract without an owner, potentially disrupting critical functions that rely on the onlyOwner modifier.

Recommendation

Override the renounceOwnership() function to disable its functionality.

Multiple Unnecessary Calls to drakkor.lock() Due to Incorrect Loop Logic

Resolved

Path

Presale.sol

Description

In the lockPresale function, there is a logic error in the loop structure that causes the drakkor.lock() function to be called multiple times unnecessarily. The current implementation places the drakkor.lock() call inside the loop, which means it will be executed for each iteration where the if condition is false.Each unnecessary call to drakkor.lock() consumes gas, resulting in higher transaction costs.

Recommendation

Move the drakkor.lock() call outside the loop to ensure it's only called once after confirming that no presale stage is active.

Missing Validation Checks for _presaleID and _token Parameters

Acknowledged

Path

Presale.sol

Description

The buy function lacks proper validation for two input parameters. The function accesses presales[_presaleID] without first verifying that this presaleID exists or is valid. This creates a risk of accessing uninitialized or invalid data, which could lead to unexpected behavior or reverts.The function accepts a parameter _token but doesn't validate that its value is within the expected range (0, 1, 2). This parameter is used to determine which token is used for payment, but without validation, invalid token types could be passed.

Recommendation

Implement explicit validation checks to ensure inputs adhere to expected constraints

Incorrect view Modifier in renounceOwnership()

Resolved

Path

Presale.sol

Description

The renounceOwnership() function incorrectly includes a view modifier when overriding OpenZeppelin's Ownable2Step implementation. While the function always reverts, this modifier:

Violates Inheritance Rules

OpenZeppelin's Ownable2Step.renounceOwnership() is not view.

Overriding with view breaks the standard interface, potentially causing integration issues.

Misleads Tooling/Proxies

Proxy patterns (Transparent/UUPS) might handle delegate calls incorrectly.

No Functional Risk

Since the function reverts, there's no direct exploit.

Recommendation

```
function renounceOwnership() public override onlyOwner {
 revert("renounceOwnership is disabled");
}
```

Informational Severity Issues

Remove all console.log for production readiness

Resolved

Path

Drakkor.sol, Presale.sol

Description

-

Recommendation

Consider removing console.log in the Drakkor contract and Presale contract before deploying contracts.

Incorrect totalSupply implementation deviates from ERC20 Standards

Resolved

Path

Drakkor.sol

Description

The Drakkor contract overrides the totalSupply() function to return a hardcoded value (total_supply = 5000 million tokens), even though only 4500 million tokens are initially minted.
This violates the ERC20 standard, where totalSupply() must reflect the actual number of tokens minted and in circulation.
The remaining 500 million tokens are minted gradually via the mint() function, but totalSupply() inaccurately reports the total as 5000 million from deployment.

Recommendation

Use OpenZeppelin's ERC20Capped contract to properly track the total supply and enforce the maximum cap

Remove stageRunning field in the PresaleData struct- for production readiness

Resolved

Path

Presale.sol

Description

-

Recommendation

Consider removing stageRunning field in the PresaleData struct for production readiness

Missing Events

Acknowledged

Path

Presale.sol, Drakkor.sol

Description

Throughout the contracts in scope, there are instances where administrative functions change contract state by modifying core state variables. However, these changes are not reflected in any event emission.

Recommendation

Emit events for all state changes.

Unused elements

Resolved

Path

Presale.sol, Drakkor.sol

Description

No References: Not used anywhere in the contract.
Originally for Uniswap Math: Likely a leftover from fixed-point arithmetic, but this contract uses OracleLibrary instead.
Gas Waste: Increases contract size unnecessarily.

Recommendation

remove it

Closing Summary

In this report, we have considered the security of Drakkor. We performed our audit according to the procedure described above.

Some issues of High, Medium and Low severity were found. Some suggestions, gas optimizations and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem

7+ Years of Expertise	**1M+** Lines of Code Audited
$30B+ Secured in Digital Assets	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

April , 2025

For

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com