Blaize.Security

September 27th, 2023 / V. 1.0

UNIGREED

UNIGREED
SMART CONTRACT AUDIT

TABLE OF CONTENTS

AUDIT RATING

SCORE **9.8**/10

The scope of the project includes Unigrid smart contracts:

- casinoERC20.sol
- lottery.sol

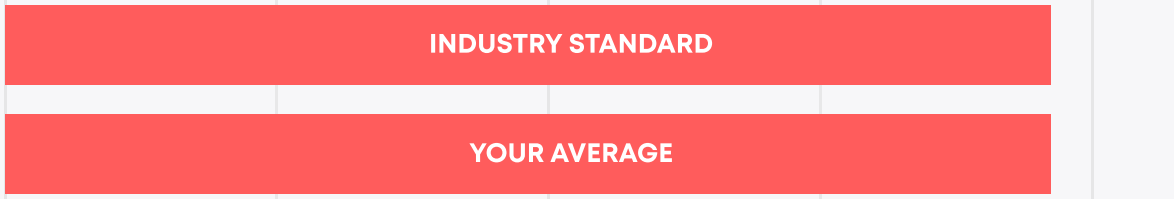
Contracts were delivered in a form of separate contracts.
SHA256 checksum of final versions:

- casinoERC20.sol:
  3d790b470a1713ba5c492989d076d47f68a998cb70d2a0e8fc546d81b5fe9edb
- lottery.sol:
  afc30f30179dc1bcca02955fdbf0565d44255934c19e94d594989eb1109bc8a4

TECHNICAL SUMMARY

During the audit, we examined the security of smart contracts for the **Unigreed** protocol. Our task was to find and describe any security issues in the smart contracts of the platform. This report presents the findings of the security audit of the **Unigreed** smart contracts conducted between **September 19th, 2023** and **September 27th, 2023.**

Testable code



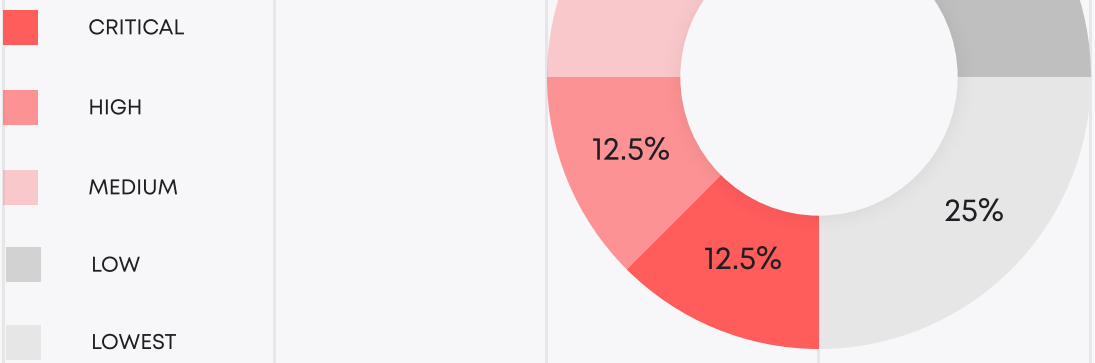
Auditors approved code as testable within the industry standard.

The audit scope includes all tests and scripts, documentation, and requirements presented by the **Unigreed** team. The coverage is calculated based on the set of Hardhat framework tests and scripts from additional testing strategies, and includes testable code from manual and exploratory rounds.

However, to ensure the security of the contract, the **Blaize.Security** team suggests that the **Unigreed** team follow post-audit steps:

1. launch **active protection** over the deployed contracts to have a system of early detection and alerts for malicious activity. We recommend the AI-powered threat prevention platform **VigiLens**, by the **CyVers** team.
2. launch a **bug bounty program** to encourage further active analysis of the smart contracts.

THE GRAPH OF VULNERABILITIES DISTRIBUTION:



The table below shows the number of the detected issues and their severity. A total of 8 problems were found. 8 issues were fixed or verified by the Unigreed team.

	FOUND	FIXED/VERIFIED
Critical	1	1
High	1	1
Medium	2	2
Low	2	2
Lowest	2	2

SEVERITY DEFINITION

Critical

The system contains several issues ranked as very serious and dangerous for users and the secure work of the system. Requires immediate fixes and a further check.

High

The system contains a couple of serious issues, which lead to unreliable work of the system and migh cause a huge data or financial leak. Requires immediate fixes and a further check.

Medium

The system contains issues that may lead to medium financial loss or users' private information leak. Requires immediate fixes and a further check.

Low

The system contains several risks ranked as relatively small with the low impact on the users' information and financial security. Requires fixes.

Lowest

The system does not contain any issues critical to the secure work of the system, yet is relevant for best practices

AUDITING STRATEGY AND TECHNIQUES APPLIED/PROCEDURE

Blaize.Security auditors start the audit by developing an **auditing strategy** - an individual plan where the team plans methods, techniques, approaches for the audited components. That includes a list of activities:

Manual audit stage

- Manual line-by-line code by at least 2 security auditors with crosschecks and validation from the security lead;
- Protocol decomposition and components analysis with building an interaction scheme, depicting internal flows between the components and sequence diagrams;
- Business logic inspection for potential loopholes, deadlocks, backdoors;
- Math operations and calculations analysis, formula modeling;
- Access control review, roles structure, analysis of user and admin capabilities and behavior;
- Review of dependencies, 3rd parties, and integrations;
- Review with automated tools and static analysis;
- Vulnerabilities analysis against several checklists, including internal Blaize.Security checklist;
- Storage usage review;
- Gas (or tx weight or cross-contract calls or another analog) optimization;
- Code quality, documentation, and consistency review.

For advanced components:

- Cryptographical elements and keys storage/usage audit (if applicable);
- Review against OWASP recommendations (if applicable);
- Blockchain interacting components and transactions flow (if applicable);
- Review against CCSSA (C4) checklist and recommendations (if applicable);

Testing stage:

- Development of edge cases based on manual stage results for false positives validation;
- Integration tests for checking connections with 3rd parties;
- Manual exploratory tests over the locally deployed protocol;
- Checking the existing set of tests and performing additional unit testing;
- Fuzzy and mutation tests (by request or necessity);
- End-to-end testing of complex systems;

In case of any issues found during audit activities, the team provides detailed recommendations for all findings.

EXECUTIVE SUMMARY

The security team conducted the audit for the Unigreed lottery. Audited smart contracts represent a simple lottery that consists of the ticket token and the Lottery itself. A ticket token is an ERC20 token with several restrictions: it's minted just once, its liquidity is added to UniswapV2 just once, and after that, no more liquidity can be added, and the token can only be purchased from the pair. Therefore, the LP tokens form the lottery reward. Each token purchased from the pool gives the user lottery tickets. Once the pool is full (the limit requirement is fulfilled), the Lottery stops, and the contract requests a random seed from Chainlink oracle and generates 21 lucky numbers - from 0 to the number of purchased tickets. Winning numbers allow users with that number to get a part of LP rewards. The Lottery can be re-launched until the Uni pair contains no ticket tokens.

The security team checked the protocol against several potential attack vectors, including dust attacks and flashloans. Auditors also reviewed the correctness of access control - that the protocol owner is not overpowered and has no backdoors to exploit. Auditors also checked the correctness of rewards distributions, usage of LP rewards, winners' choice, and bonus structure. The Unigreed team fixed all found issues.

It's also worth mentioning that during the audit, the Unigreed team provided several major upgrades following auditors' recommendations: the protocol integrated Chainlink oracle, switched the price calculation from ETH to USDT, and made the Lottery re-launchable several times. Thus, the contract has a safe randomness mechanism, reusability, and protection from flashloans.

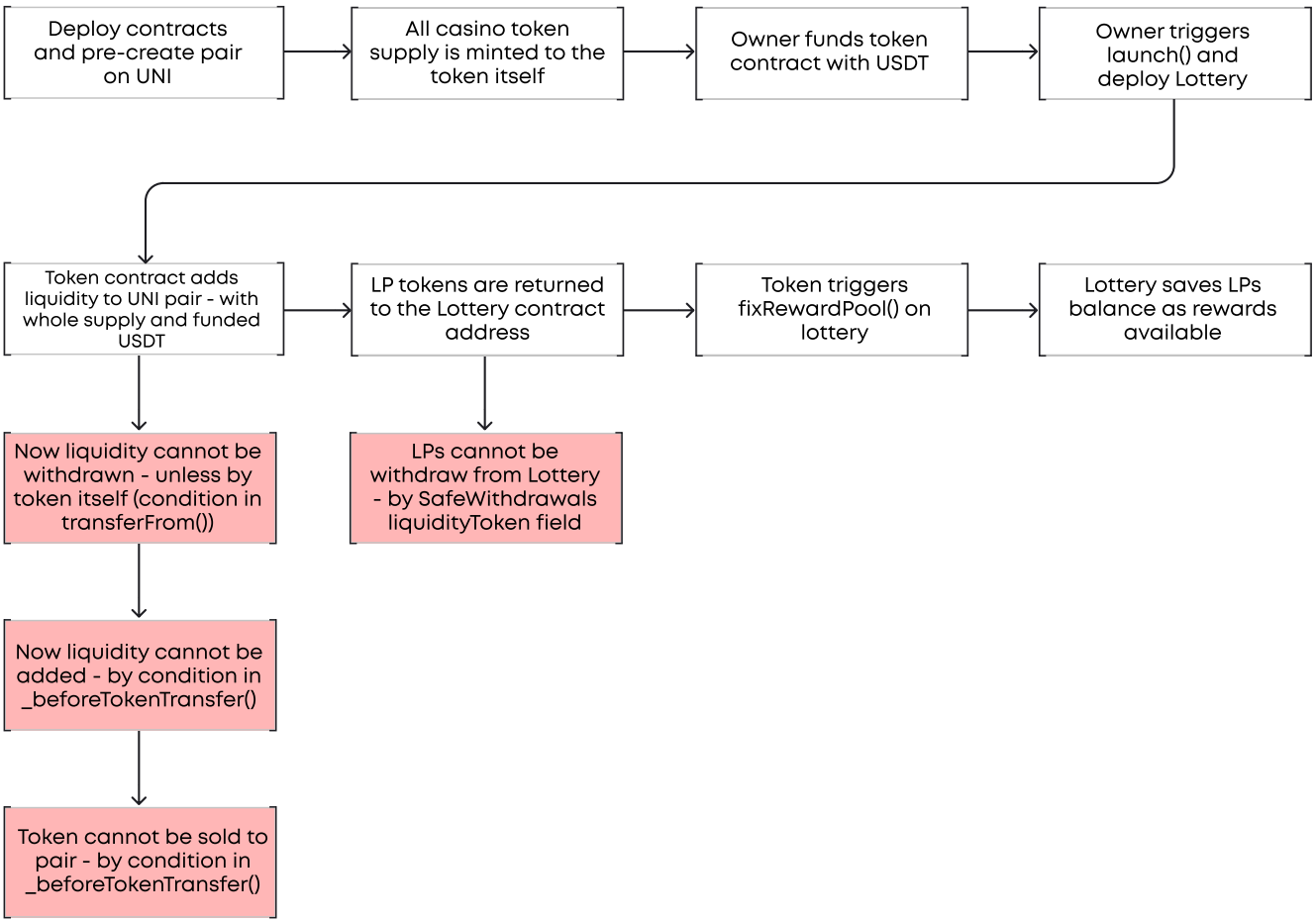
Despite several places for improvement (e.g., tangled implementation of bonus calculations), contracts have a high security level.

	RATING
Security	10
Logic optimization	9.6
Code quality	9.6
Test coverage**	10
Total	9.8

** Unigreed team performed manual testing, therefore the mark is based on tests performed by Blaize Security team.

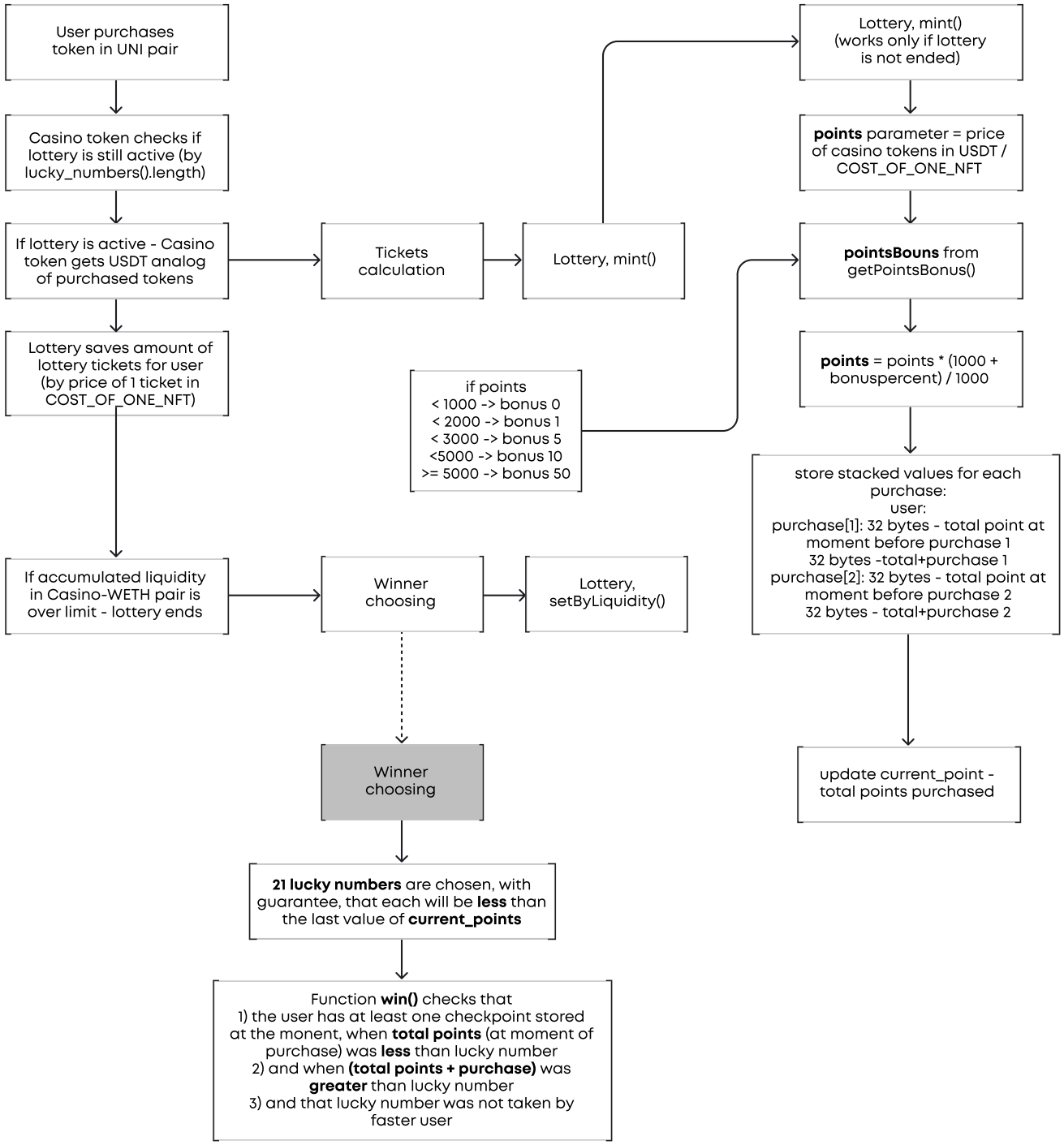
UNIGREED

Initializing flow



UNIGREED

Token purchase flow



COMPLETE ANALYSIS

CRITICAL-1	✔ Resolved

Chainlink feed is not connected in all cases.

The lottery can be finished after the last casino token purchase to fill the pool limit (in casinoERC20, _beforeTokenTransfer()). However, the randomness is set via the pseudo-random based on the hash of the previous block number, which can be easily precalculated. And so - the attacker can easily manipulate the last purchases to fit into the necessary slot of the lucky number.

Recommendation:

Set lucky numbers with Chainlink in all the cases or consider a more secure way to achieve on-chain randomness. For instance, we recommend the usage of a future randao value. In this solution, the setting of lucky numbers can be split into two transactions. The first transaction determines the future block number (for example, block.number + 128), and the second one will perform the setting using the block.prevrandao for a truly random value after a determined earlier block.
Note! This solution works only for Ethereum or forks of Ethereum, which migrated to PoS. In other chains, you may use a hash of a future block specified block (block.number + 128). It is more difficult to manipulate future blockchash than blockhash of the previous block. Note: function `blockchash`has access only to the last 256 blocks; thus, you should be on time to be able to set lucky numbers using the hash of the specified block.

Post-audit. Randomness is achieved with Chainlink in all cases now.

HIGH-1	✔ Resolved

Incorrect querying of array's length.

casinoERC20Final.sol: _beforeTokenTransfer(), line 384.
The length of array `lucky_numbers` is got using `ILottery(_lottery).lucky_numbers().length == 0`. However, the storage array doesn't generate a getter to return the whole array. Instead, it generates a getter for the elements. Thus, the following way of getting the array's length will revert during the execution of a transaction.

Recommendation:

Add a getter in Lottery smart contract which will return the length of array and use it instead.

Post-audit. Getter was added which checks if lucky numbers are set.

MEDIUM-1	✔ Resolved

DDoS possible.

lottery.sol, mint()
lottery.sol, win()
Each user gets the array that stores earned points - lottery tickets. Each purchase is added separately. Also, there are several factors:

- win() function has a triple cycle, which includes a cycle over the user points records
- mint() allows minting zero points (which will be added to the array)

Therefore, there is a possibility to provide a DDoS attack with quite low expanses by consequent purchases of fractions of casino tokens. That will effectively create storage long enough to force the win() function to run out of gas, effectively blocking funds in the pair (since removing liquidity is forbidden)

The function is marked as a medium since the win() function receives an arbitrary array of addresses to check as a parameter.

Recommendation:

Disallow minting 0 lottery tickets AND consider splitting win() function logic to check one lucky number at a time. E.g., add a parameter for the lucky number to check against - so the function will check some users against 1st lucky number (or 3rd or 4th - which is passed as a parameter).

Post-audit. Minting of zero shares was disabled. A new parameter was added in the win() function where a certain range to be checked per user is specified.

MEDIUM-2	✔ Resolved

Owner can withdraw developer share multiple times.

lottery.sol: claimDevelopersReward().
The function can be called several times, claiming the developer shares each time. Thus, the function can be used to drain users' rewards from the lottery. The issue is marked as a medium since only the owner can execute the function; thus, it requires the loss of the owner's private key to perform the malicious action.

Recommendation:

Restrict the function so that it can be called only once.

Post-audit. Shares can be claimed only once now.

LOW-1	✔ Verified

There is a flow for users to not receive tickets.

CasinoERC20.sol, _beforeTokenTransfer(), calculation of tickets to be minted.
There is a loss of accuracy in amountOfUsdt / COST_OF_ONE_NFT statement. While it may not be necessary if the amount paid covers more than an integer number of tickets, the user still may buy fewer casino tokens than the price of 1 lottery ticket - and lose money. Therefore, it is better to revert in case amountOfUsdt / COST_OF_ONE_NFT equals zero.

Recommendation:

Revert for minting of 0 tickets (in lottery contract) AND make NFT price constant public for users to plan the purchase.

Post-audit: the logic verified to be an intended part of the lottery.

LOW-2	✔ Resolved

Opened pool limit conditions.

CasinoERC20.sol, _beforeTokenTransfer().
Lottery end condition is checked as
liquidity+amountOfUsdt>TOTAL_POOL_LIMIT

Though it is better to use the condition to include the pool limit itself.

Recommendation:

Use >= sign.

LOWEST-1	✔ Verified

Check USDT decimals before the deployment

CasinoERC20.sol. The contract contains a constant for determining the USDT decimals, which is set to 18. It is ok to have it for the testnet, though the mainnet version of the USDT has 6 decimals.

Recommendation:

Use mainnet settings even in test environments. Also, it is recommended to pull the decimals number directly from the USDT contract (e.g., during the construction) - especially since the contract contains a constant for USDT address.

Post-audit:

The team verified the testing setup, and will prepare the update of stablecoin settings for the mainnet version.

LOWEST-2	✔ Verified

Check casino settings before the mainnet launch

CasinoERC20.sol. Constants TOTAL_POOL_LIMIT and COST_OF_ONE_NFT are set to 10 and 1 USDTs respectively. These are testnet values, therefore it should be reviewed before the deployment. Also, the comment states that the pool limit should be set to 1000 USDT, which contradicts the test value.

Recommendation:

Provide necessary values as arguments in the constructor to avoid confusion and errors in production and add those values to the deployment script.

Post-audit:

The team verified the testing setup, and will prepare the update of settings for the mainnet version.

STANDARD CHECKLIST

	casinoERC20.sol lottery.sol
✔ Re-entrancy	Pass
✔ Access Management Hierarchy	Pass
✔ Arithmetic Over/Under Flows	Pass
✔ Delegatecall Unexpected Ether	Pass
✔ Default Public Visibility	Pass
✔ Hidden Malicious Code	Pass
✔ Entropy Illusion (Lack of Randomness)	Pass
✔ External Contract Referencing	Pass
✔ Short Address/Parameter Attack	Pass
✔ Unchecked CALL Return Values	Pass
✔ Race Conditions/Front Running	Pass
✔ General Denial Of Service (DOS)	Pass
✔ Uninitialized Storage Pointers	Pass
✔ Floating Points and Precision	Pass
✔ Tx.Origin Authentication	Pass
✔ Signatures Replay	Pass
✔ Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES, PREPARED BY BLAIZE SECURITY TEAM

Contract: Lottery

- ✓ Should correctly calculate bonus points (2920ms)
- ✓ Should determine winners (6420ms)
- ✓ Can't claim twice (1052ms)
- ✓ Should not allow flashloan attack (1052ms)

Contract: Casino

- ✓ Deployment
- ✓ Should launch casino (2766ms)
- ✓ Should not let launch again (95ms)
- ✓ Should let transfer Casino tokens (604ms)
- ✓ Should purchase tickets (238ms)
- ✓ Should not let add liquidity (176ms)

Note:

The team also conducted an additional round of manual exploratory testing over the locally deployed contracts and verified the robustness against several attack vectors, including flashloans, correct funds distribution for the winners, and correct lottery re-launch.

DISCLAIMER

The information presented in this report is an intellectual property of the customer, including all the presented documentation, code databases, labels, titles, ways of usage, as well as the information about potential vulnerabilities and methods of their exploitation. This audit report does not give any warranties on the absolute security of the code. Blaize.Security is not responsible for how you use this product and does not constitute any investment advice.

Blaize.Security does not provide any warranty that the working product will be compatible with any software, system, protocol or service and operate without interruption. We do not claim the investigated product is able to meet your or anyone else's requirements and be fully secure, complete, accurate, and free of any errors and code inconsistency.

We are not responsible for all subsequent changes, deletions, and relocations of the code within the contracts that are the subjects of this report.

You should perceive Blaize.Security as a tool, which helps to investigate and detect the weaknesses and vulnerable parts that may accelerate the technology improvements and faster error elimination.