Blaize.Security

February 24th, 2023 / V. 1.0

TOKENDEAL

NEW MINTING LOGIC

TOKENDEAL

SMART CONTRACT AUDIT

TABLE OF CONTENTS

AUDIT RATING

TokenDeal contract's source code was taken from the archive provided by the TokenDeal team.

SCORE **9.9** /10

The scope of the project is **TokenDeal** set of contracts during 1st audit iteration:

1/ TokenDeal.sol

Code delivered as archive
Initial code:
Archive SHA256:
b4e556481c6af5f39e55946534b3fbc7eaac26f82111e51851c7ed7d85608693

TokenDeal.sol SHA256:
c7148ada58ea459a0d62f420960b6866c09e128d5add7d22a1ef7946bb7634c5

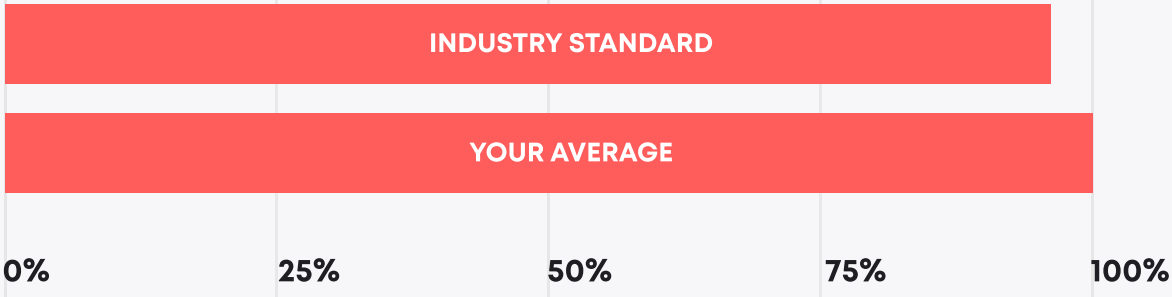
Final code:
Archive SHA256:
4b23675a3c17d0ce21d5cccf070534c186780f330944e4e0703cb3343d14702e

TokenDeal.sol SHA256:
ede40bf4f0a6ef8e1120f42cdb4048f22e6816f84faac96639834f3cbb986c15

TECHNICAL SUMMARY

In this report, we consider the security of the contracts for TokenDeal protocol. Our task is to find and describe security issues in the smart contracts of the platform. This report presents the findings of the security audit of **TokenDeal** smart contracts conducted between **February 23rd, 2023 - February 24th, 2023**

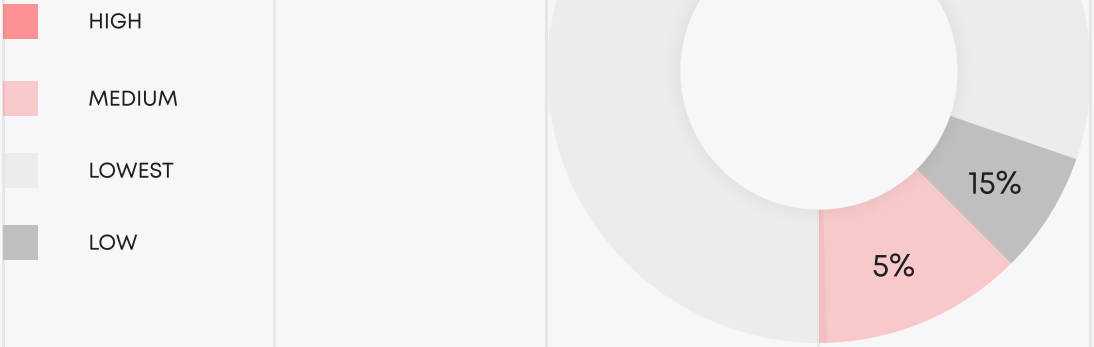
Testable code



The testable code is 100%, which above the industry standard of 95%.

The scope of the audit includes the unit test coverage, that bases on the smart contracts code, documentation and requirements presented by the TokenDeal team. Coverage is calculated based on the set of Hardhat framework tests and scripts from additional testing strategies. Though, in order to ensure a security of the contract Blaize.Security team recommends the TokenDeal team put in place a bug bounty program to encourage further and active analysis of the smart contracts.

THE GRAPH OF VULNERABILITIES DISTRIBUTION:



The table below shows the number of found issues and their severity. A total of 12 problems were found. 12 issues were fixed or verified by the TokenDeal team.

	FOUND	FIXED/VERIFIED
Critical	0	0
High	0	0
Medium	1	1
Low	3	3
Lowest	8	8

SEVERITY DEFINITION

Critical

A system contains several issues ranked as very serious and dangerous for users and the secure work of the system. Needs immediate improvements and further checking.

High

A system contains a couple of serious issues, which lead to unreliable work of the system and migh cause a huge information or financial leak. Needs immediate improvements and further checking.

Medium

A system contains issues which may lead to mediumfinancial loss or users' private information leak. Needs immediate improvements and further checking.

Low

A system contains several risks ranked as relatively small with the low impact on the users' information and financial security. Needs improvements.

Lowest

A system does not contain any issue critical to the secure work of the system, yet is relevant for best

AUDITING STRATEGY AND TECHNIQUES APPLIED \ PROCEDURE

We have scanned this smart contract for commonly known and more specific vulnerabilities:

- Unsafe type inference;
- Timestamp Dependence;
- Reentrancy;
- Implicit visibility level;
- Gas Limit and Loops;
- Transaction-Ordering Dependence;
- Unchecked external call - Unchecked math;
- DoS with Block Gas Limit;
- DoS with (unexpected) Throw;
- Byte array vulnerabilities;
- Malicious libraries;
- Style guide violation;
- ERC20 API violation;
- Uninitialized state/storage/ local variables;
- Compile version not fixed.

Procedure

In our report we checked the contract with the following parameters:

- Whether the contract is secure;
- Whether the contract corresponds to the documentation;
- Whether the contract meets best practices in efficient use of gas, code readability;

Automated analysis:

Scanning contract by several public available automated analysis tools such as Mythril, Solhint, Slither and Smartdec. Manual verification of all the issues found with tools.

Manual audit:

Manual analysis of smart contracts for security vulnerabilities. Checking smart contract logic and comparing it with the one described in the documentation.

EXECUTIVE SUMMARY

The audited set of contracts represents the protocol for the NFT sale. It handles funds collection, NFT minting, and distribution of funds between the manager and the owner.

The auditor's team checked the contract against common vulnerabilities and its own security checklist, checked the funds flow, user operations, correct roles assignment, and the overall business logic (in search of loopholes, backdoors, and potential disruptions of the contract workflow).

Also, the team performed several testing rounds against the whole NFT sale process.

The audit found no critical issues. But the team has prepared a description and recommendation for the row of low-risk issues, including best practices violations, accuracy loss, and several questions connected to the business logic of the protocol. There are several unclear edgecases for the refund operations, NFT minting, and lock time for the contract. Though, TokenDeal team resolved or verified all the issues.

Also, the Blaize Security team needs to notice that the contract depends on 3rd party contract - the actual NFT which will be sold. For now, the NFT contract is not written; therefore, there is no way to check the whole process of future NFT minting. Nevertheless, the DealToken team prepared the interface for the future NFT, compatible with the sale logic.

Contracts are well documented with natspec comments and have good gas optimization. Despite current recommendations for additional checks for the lock change and token address during the minting, the overall security is high enough to comply the security standard.

	RATING
Security	9.8
Gas usage and logic optimization	9.8
Code quality	9.9
Test coverage**	10
Total	9.9

**Contract has a native coverage, prepared by TokenDeal team, though, Blaize Security has prepared their own set of unit tests and additional scenarios to cover the whole code. The mark shows the final testable code

PROTOCOL OVERVIEW

TokenDeal is a protocol for NFT sales. The contract represents the main sale phases: funds collection during purchase, purchased NFT mint after the sale, and collected funds withdrawal.

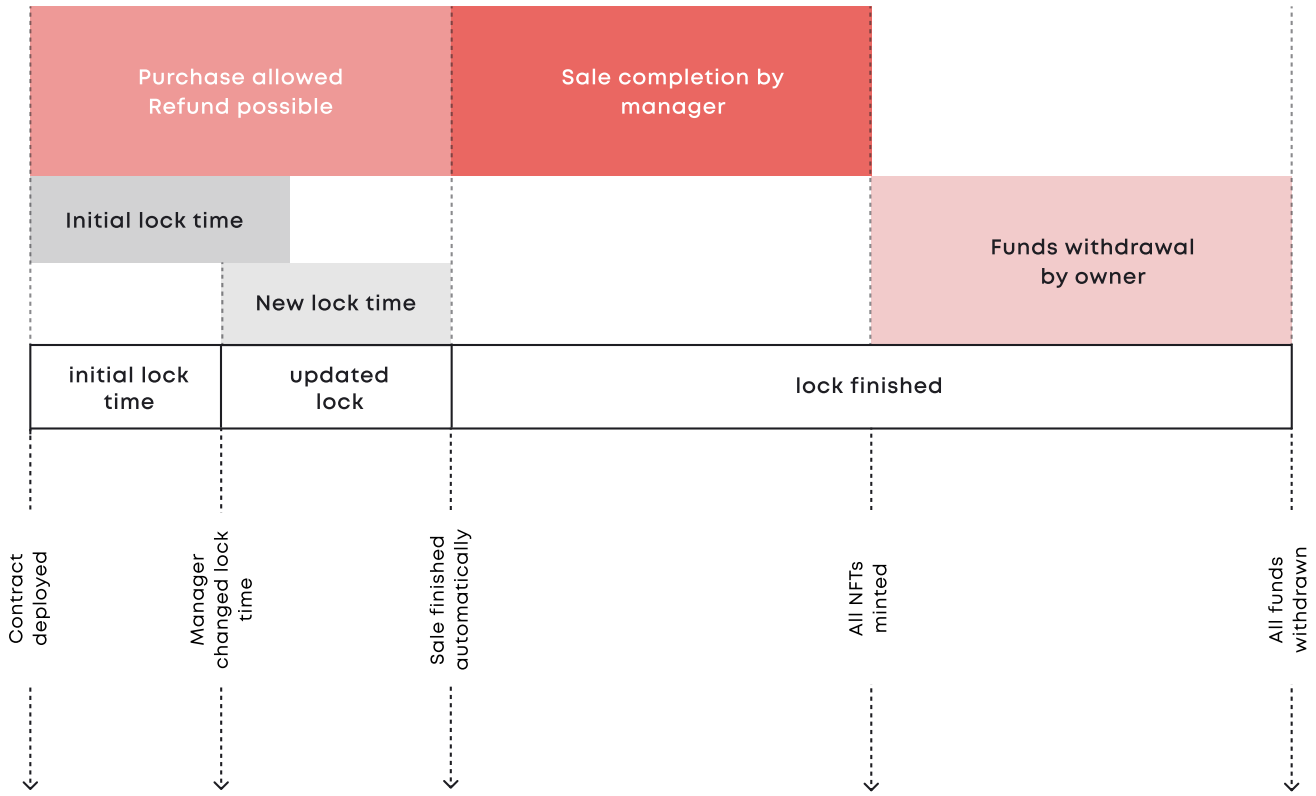
1. Funds collection
The sale starts immediately after the contract deployment. The lock time parameter regulates its duration: until the contract is locked, it allows NFT purchasing.
* the manager can prolong lock time
* once lock time is over, the contract stops receiving money from users
* there is no ability to get ETH directly by receiving function. Users are obeyed to use purchase() function. Users will lose all funds sent directly.
* user can refund ETH after the purchase
* contract saves each user who purchased NFTs
* there is a limit for NFTs to be purchased (set in the constructor)
* NFT price is set in the constructor

2. NFT mint
After the sale is over (after the lock timestamp), the manager completes it. completeSale() function goes through all saved users and mints NFTs until enough gas exists.

3. Funds withdrawal
After all, NFTs are minted, owner withdraws collected funds.
* manager gets a commission by FEE percent (10% by current setting)
* all funds collected (contract balance) go to the owner

4. Global refund
Function allowRefund() stops any current process (sale, mint, or withdrawal) and turns on "refund-only" mode.
* the only allowed operation is refund by the user
* there is no difference if some NFTs are minter, or neither NFTs are minted

TOKENDEAL. NFT SALE FLOW



COMPLETE ANALYSIS

MEDIUM-1	✔ Resolved

Accuracy loss

TokenDeal.sol: _calculateFee()
Function violates mul-div pattern: in order to calculate the fraction, you need to multiply by the percent first and divide by the accuracy divisor after. The issue is classified as Medium since it violates standard Solidity rules against vulnerabilities, leading to a loss of accuracy and important values.

Recommendation:

Change operations order to: value * FEE / 100000

LOW-1	✔ Verified

Max lock timestamp step should be introduced

TokenDeal.sol: changeLockTimestamp()
The contract has limitations against setting timestamps in the past, but there are no limitations for setting timestamps in the future. It creates a wide possibility for a human mistake with an incorrect value for the new lock timestamp. Providing timestamps far in the future and creating a deadlock for the funds is possible, so it will be impossible to claim funds. Also it is impossible to re-set the timestamp into the past. The issue is classified as Low, because the function can be called only by the owner. Though it creates a risk for funds block, it is recommended to take care of it.
General recommendation is a maximum step for the timelock update and add the check into the function,against that step (e.g. newTimestamp - lockTimestamp <= MAX_TIMESTAMP_STEP).

Recommendation:

Consider adding the step for timestamp change.

Post-audit:

TokenDeal team will not add the check to keep gas consumption low and verified that uint32 for timestamp will handle this case. Though auditors recommend to provide additional measures on the dApp part for prevention human error with timestamp.

LOW-2	✔ Verified

Same user can be added to the userAddress list twice

TokenDeal.sol: purchase()
The function does not distinguish if the user is already added to the list. Since there are no restrictions for users to participate in the sale several times, any user can call purchase() function for several times. Therefore, the user will be added to the userAddress list several times. This will influence gas spendings during the completeSale() call.

Recommendation:

Add condition to add user to the userAddress list only if he purchases for the first time OR add limitation to the purchase() calls for the user.

Post-audit:

Verified as false-positive. Auditors double confirmed with appropriate tests.

LOW-3	✔ Verified

Refund may not be possible in case if all NFTs are minted

Consider scenario:
- finish sale with all NFTs sold (nftLimit == nftSold)
- complete the sale (mint all NFTs)
- allow refund (call allowRefund function)

After this action, the only possible functionality for the contract is to provide a refund for users. Though, at this point, the owner is allowed to withdraw all funds (with the fee paid to the manager). The issue is marked as Low, since it requires actions from admins. But to verify the behavior's correctness and exclude dishonest owner/manager, auditors added issue to the report.

Recommendation:

Verify that owner can withdraw funds in case of refund-only mode after the sale completion OR restrict such behavior.

Post-audit:

Verified by TokenDeal team as expected behavior.

LOWEST-1	✔ Verified

Token deal manager constant

TokenDeal.sol: TOKEN_DEAL_MANAGER

Constant contains the address of the operator role (token deal manager). However, it has hardcoded value for the testnet, which will mostly be changed to the mainnet one. The comments around the constant partly confirm this. So, it will lead to contract code changes after the audit, which may require additional checks. Therefore it is recommended to use the immutable variable for the address set up in the constructor. Also, having the necessary manager addresses in the deployment scripts is recommended.

Recommendation:

Change the constant address to the immutable variable and set it up in the constructor. Add the necessary address to the deployment script.
OR verify that the address will be the same for the mainnet and change the comment to indicate that.

Post-audit:

TokenDeal team is acknowledged and decided to leave hardcoded address. The team will update it to mainnet one before the release. Contracts will be deployed by users from the frontend. So the team left hardcoded address to avoid possible attempts of replacing the address with the wrong one. Auditors recommend to verify the code before the deployment with the security team, since there will be changes in the code.

LOWEST-2	✔ Resolved

Variables may be declared as immutable

TokenDeal.sol: nftPrice, nftLimit

These variables get value just once during the deployment of the contract. Therefore it is recommended to mark them as immutable.

Recommendation:

Set variables as immutable.

LOWEST-3	✔ Verified

Potential backdoor with token address

TokenDeal.sol: completeSale()

The function gets a token address as a parameter. This token is the one which will be minted for the sale participants. Since only the admin can call the function, and since the function is designed to be called several times with potentially different gas, the admin can use different tokens for different users. Or the admin can use another token at all. Therefore, such an approach can be classified as a controllable backdoor in the sale logic, which can be disturbing for users.

Mark the token address transparently and set it just once: either in the constructor (if possible) or in the separate setter with the event emitted.

Recommendation:

Set a token address for NFT minting just once.

Post-audit:

By the TokenDeal team, the owner does not know the address of the token when the contract is deployed. Separate method will cost gas, which is unacceptable in TokenDeal case. TokenDeal team verified, that since only TokenDealManager can call this method, a backdoor is unlikely. Nevertheless, auditors recommend to provide additional layer of checks before the mint start.

LOWEST-4	✔ Resolved

Magic number used

TokenDeal.sol: _calculateFee()

The function contains the accuracy divisor 100000 set as a “magic number”. It is recommended to provide an appropriate constant for better code readability.

Recommendation:

Provide constant.

LOWEST-5	✔ Resolved

User can purchase 0 tokens and cause DoS

TokenDeal.sol: purchase()
The user can purchase 0 tokens, which will be a legal transaction that will add the user to the list. Despite that it is unprofitable for the user (for tx fees), it allows the general DoS attack to abuse the storage in the userAddress list (since the user will be added to the list). And that will create problems for the sale completion.
The issue is marked as Info, since the completion function has gas control mechanics.

Recommendation:

Restrict the purchase of 0 amount.

LOWEST-6	✔ Resolved

Balance for tx refund is included twice for the manager

TokenDeal.sol: withdrawOwnerFunds()
txBalance is meant to be a minting tx fee refund for the manager. This balance and the fee collected from the sale create full compensation for him. Though the function includes txBalance into the manager fee calculation. So the current formula is
manager fee = full amount * 10% + tx balance

rather than
manager fee = (full amount - tx balance) * 10% + tx balance

The issue is marked as info as it is related to the business logic and correctness of fee calculation, so it needs verification from the TokenDeal team.

Recommendation:

Verify the correctness of the chosen fee calculator formula.

LOWEST-7	✔ Verified

Batch mint recommended

TokenDeal.sol: completeSale()
Function mints NFTs one by one, which is not gas efficient. It is recommended to consider batch mint functionality for the target NFT, since ERC721 supports it.

Recommendation:

Consider batch minting functionality for the NFT.

Post-audit:

TokenDeal team verified that 1 by 1 minting complies with the protocol needs.

LOWEST-8	✔ Resolved

Provide initialization of the variable out of the cycle

TokenDeal.sol: completeSale()
j counter is defined within the cycle, which may lead to incorrect initialization and memory re-usage. Consider defining j out of the cycle and provide a strict 0 assignment.
The issue is marked as info, since there is extremely low chance of its reproducing. Though there is a possibility of incorrect memory distribution, which may lead to the memory not being cleaned for j before the next cycle turn.

Recommendation:

Consider j definition out of the cycle.

	TokenDeal.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 TOKENDEAL TEAM

TokenDeal::AllowRefund

Purchase

- ✓ Success: purchase 2 tokens by user1
- ✓ Success: purchase 1 more token by user1
- ✓ Fail: purchase 2 tokens with wrong price by user

Allow Refund activate

- ✓ Fail: activate allowRefund by not manager

1677151548

- ✓ Success: activate allowRefund by manager
- ✓ Fail: try to purchase after allowRefund was activated

0.009999999999934464

- ✓ Success: withdraw by user after allowRefund was activated
- ✓ Fail: try to compileSale after allowRefund was activated
- ✓ Fail: try to withdrawOwnerFunds after allowRefund was activated

Error: value out-of-bounds

- ✓ Fail: try to changeLockTimestamp after allowRefund was activated (value out-of-bounds)
- ✓ Fail: try to changeLockTimestamp after allowRefund was activated (TimeMustBeHigherThanPrevious)
- ✓ Fail: check getAvailableAmount
- ✓ Fail: try to purchase after allowRefund was activated

0.019999999999868928

- ✓ Success: withdraw by user after allowRefund was activated

TokenDeal

Gas Used (deploy TokenDeal): 2442022

Purchase

- ✓ Success: purchase 2 tokens by user1

Gas Used purchase 1: 47684

- ✓ Success: purchase 1 more token by user1
- ✓ Fail: purchase 2 tokens with wrong price by user
- ✓ Fail: purchase more than limit tokens by user
- ✓ Success: purchase all tokens by users (41ms)

WithdrawFunds by user

✓ Fail: withdraw more than available tokens by user

Gas Used withdraw 1: 50236

0.009999999999934464

✓ Success: withdraw locked tokens by user

✓ Success: purchase token by user

Gas Used withdraw 1: 50236

0.009999999999934464

✓ Success: withdraw locked tokens by user

Change time lock

✓ Fail: change to less time

✓ Fail: change time by not tokenDealManager

✓ Success: change time by tokenDealManager

Try to run methods before expired time and compiled sale

✓ Fail: completeSale by owner before expired time

✓ Fail: withdraw by owner before expired time and compiled sale

View funcs

✓ Success: get user addresses length

✓ Success: get available amount after complete sale

30 passing (557ms)

TEST COVERAGE RESULTS

TOKENDEAL TEAM

FILE	% STMTS	% BRANCH	% FUNCS
TokenDeal.sol	57.89	58.7	91.67

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

TokenDeal

purchase

- ✓ can purchase NFT
- ✓ cannot purchase NFT after lock time ends
- ✓ cannot purchase nft when msg.value is not sufficient [1 NFT]
- ✓ cannot purchase nft when msg.value is not sufficient [3 NFT]
- ✓ cannot purchase nft after all nfts are sold

withdrawFunds

- ✓ user can withdrawFunds
- ✓ user can withdrawFunds fund for 1 out of 2 NFT
- ✓ user cannot withdrawFunds more than he has paid
- ✓ user cannot withdrawFunds zero amount
- ✓ user cannot withdrawFunds after lock

allowRefund

- ✓ only manager can call allowRefund
- ✓ allowRefund block ability to buy nft
- ✓ allowRefund doesn't block ability to withdrawFunds

changeLockTimestamp

- ✓ only manager can changeLockTimestamp to a new value
- ✓ lock timestamp cannot be changed if it is expired
- ✓ new lock timestamp cannot be before previous lock timestamp

completeSale

- ✓ only manager can completeSale
- ✓ cannot completeSale while time lock is not expired
- ✓ completeSale (58ms)
- ✓ completeSale in two TX

withdrawOwnerFunds

- ✓ only owner can call withdrawOwnerFunds
- ✓ cannot withdraw funds when there is no funds
- ✓ cannot withdraw funds when nft are not minted
- ✓ withdrawOwnerFunds

24 passing (521ms)

TEST COVERAGE RESULTS

BLAIZE SECURITY TEAM

FILE	% STMTS	% BRANCH	% FUNCS
TokenDeal.sol	100	86.96	100

** Additionally Blaize Security team provided an exploratory testing of the edgecases for the sale period (sale of the whole limit, several purchases by the same user, unexpected end of sale, purchases of different numbers of NFTs) and for the completeSale() function inm particular.

DISCLAIMER

The information presented in this report is an intellectual property of the customer including all 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 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.