Blaize.Security

February 18th, 2022 / V. 2.0

NEMUS
SMART CONTRACT AUDIT

TABLE OF CONTENTS

AUDIT RATING

Nemus contract's source code was taken from the repository provided by the Nemus team.

SCORE **9.75** /10

The scope of the project is **Nemus** set of contracts**:**

1/ AbstractMintVoucherFactory

2/ NeaMintTicketFactory

Repository:
https://github.com/Nemus-Team/nemus-contracts
Initial comit:

- 791840944dffc0346b840941a1d5e6de4a0b66f2

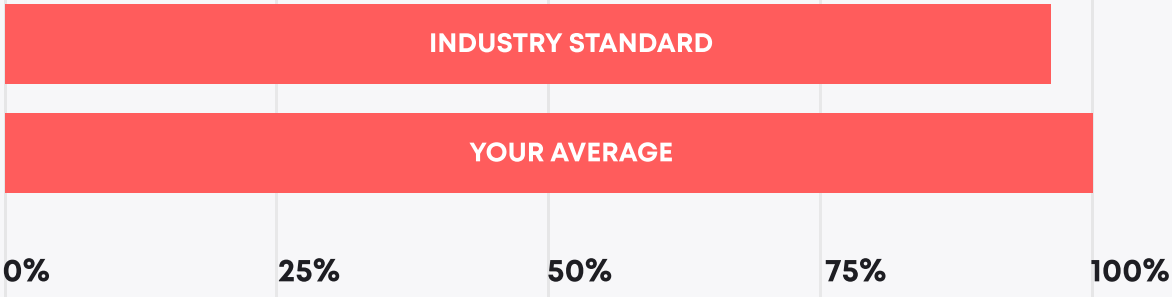
Last audited commit:

- 18dc0e24c2737dc1cdb626754f2e7c83dedb40a3

TECHNICAL SUMMARY

In this report, we consider the security of the contracts for Nemus 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 **Nemus** smart contracts conducted between **January 20th, 2022 - February 08th, 2022.**
Audit update was performed on **February 18, 2022**

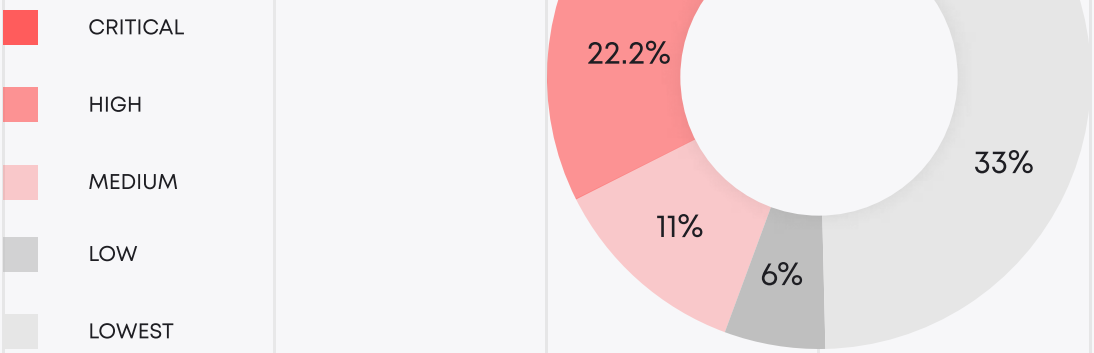
Testable code



The testable code is 99.46%, which is 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 Nemus team. Coverage is calculated based on the set of Truffle framework tests and scripts from additional testing strategies. Though, in order to ensure a security of the contract Blaize.Security team recommends the Nemus 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 18 problems were found. 14 issues were fixed or verified by the Nemus team.

	FOUND	FIXED/VERIFIED
Critical	5	5
High	4	4
Medium	2	2
Low	1	0
Lowest	6	3

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 STRATEGYAND 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 contract contained several critical issues which did not allow correct NFT minting for the most of user's scenarios. Also several high risk issues from the standard auditors list were found - reentrancy and problem with ETH handling in particular. Though, the team has fixed all the issues.

All other issues were connected to missed checks, which may block the contract, and code quality. Nevertheless, all security risk issues were fixed by the team.

The overall code quality and readability are high enough.

	RATING
Security	9.2
Gas usage and logic optimization	9.8
Code quality	10
Test coverage**	10
Total	9.75

** There was very few initial tests presented by Nemus team, the whole unit tests system was written by Blaize.Security engineers.

COMPLETE ANALYSIS

NeaMintTicketFactory.sol

CRITICAL ✔ Resolved

Contracts don't compile.

Contract compilation fails with error "CompileError: CompilerError: Stack too deep when compiling inline assembly: Variable headStart is 1 slot(s) too deep inside the stack". This is caused due to the amount of local variables in the function editMintTicket().

Recommendation:

Reduce the amount of variables by packing them into struct.

CRITICAL ✔ Resolved

Early access claimed amount is not updated.

Function claimMultipleEarlyAccess(). Variable from mapping 'earlyAccessMintedCounts' is not updated which lets early accessor to claim NFTs without limitations.

Recommendation:

Update the variable.

Post-audit.

After discussion of possible fixes with the Nemus team in spite of recommended fix, also added value from mapping earlyAccessMintedCounts (Line 279) to 'userMintedAmount' only once before the loop.

CRITICAL	✔ Resolved

Buyer of NFTs is able to buy more tokens than limited

function claimMultiple(), claimMultipleEarlyAccess(). It is possible to pass an index of a ticket multiple times in an array, and thus perform purchasing in one ticket several times during one transaction. This way buyer is able to get around the validation in function isValidClaim() (lines 300, 302, 303) and function sValidEarlyAccessClaim() (line 321) and mint more NFTs than mintTickets[mtIndexes].maxSupply, mintTickets[mtIndexes].maxPerWallet or mintTickets[mtIndexes].maxMintPerTxn restrict to.

For example a ticket with index 1 has only one nft left. User passes index 1 multiple times and this way he can get around a validation, since total supply doesn't change after each validation. After validation mintBatch will mint nft from index 1 several times, exceeding the limitation.

Recommendation:

Either verify that the user can't pass the same index multiple times or change storage(mint NFTs, add claimedNFTs to user) after each validation, thus providing updated storage before the next validation.

Post-audit.

After the discussion with the Nemus team, it was discovered, that It was still possible to avoid validations (for max supply and max per tx), since NFTs are minted after the validations and max per tx is validating separately for each passed index. Another set of checks was added to the contract to avoid this issue.

CRITICAL ✔ Resolved

Buyer of NFTs is able to pay not for all tokens

function claimMultiple(), claimMultipleEarlyAccess(). It's verified that the user has sent enough ETH for NFTs from each ticket separately and this way one is able to pay not for all tokens he is buying.

Example: there are two tickets where each NFT costs one ETH per token. User buys one nft from each ticket and sends 1 ether. isValidClaim() and isValidEarlyAccessClaim() check that there is enough ETH sent for each ticket separately in line 300 and 319. Buyer was supposed to have sent 2 ETH, but instead he sent 1 and the validation still won't revert the transaction, letting the user pay less than he had to.

Recommendation:

Calculate total cost for purchasing all the NFTs and check that Buyer has sent enough ETH.

CRITICAL ✔ Resolved

The amount of NFTs bought is not verified.

Line 262, function claimMultipleEarlyAccess(). The amount of early access minted NFTs for user is calculated in variable 'userMintedAmount', however the variable is not verified like on line 237 in function claimEarlyAccess().

Recommendation:

Validate that minted amount doesn't exceed limitation.

HIGH ✔ Resolved

Possible re-entrancy attack.

All the updates to the storages should be performed before external calls(Returning of exceeded Ether).
In case, sender of transaction is using re-entrancy, he would be able to avoid validations on max supply of NFT or maximum amount, an early accessor can claim.

Recommendation:

Either use a Non-reentrant modifier from OpenZeppelin library on the following function: claim(), claimMultiple(), claimEarlyAccess(), claimMultipleEarlyAccess(), or consider making the following changes to these functions:
claim(). Actions on lines 200-203 should be before returning Ether.
claimMultiple(). Actions on line 234 should be before returning Ether.
claimEarlyAccess(). Actions on lines 252-254, 257, 260 should be before returning Ether.
claimMultipleEarlyAccess(). Actions on lines 293, 296 should be before returning Ether.

HIGH ✔ Resolved

Deprecated send() call for Gnosis Safe wallet.

withdrawFunds() utilizes sned() method for transfering Ether to the treasury and Gnosis Safe wallet addresses. Since send() utilizes 2300 gas for a call and does not forward gas further - it will fail on the Eth sending to the multisig wallet.

Recommendation:

Eliminate send() usage and use call() for both cases within withdrawFunds()

HIGH ✔ Resolved

Early access NFTs can be minted even after public sale.

isValidEarlyAccessClaim() validates that early access is in progress by calling the function isEarlyAccessOpen(). This function only validates that early access has started but it doesn't check if it has ended and early access tokens can be minted even after public sale.

Recommendation:

Check the window for early access sale.

HIGH ✔ Resolved

Send exceeded ETH.

function claimMultiple(). Exceeded ETH should be sent back to the message caller like in function claim().

Recommendation:

Transfer exceeded ETH to the message caller.

MEDIUM ✔ Resolved

Validate that the provided index exists.

Function editMintTicket(). It is possible to pass non-existent '_mtIndex' (greater than mtCounter.current()) and write data to non-existent tickets, thus corrupting the storage.

Recommendation:

Verify that the provided index exists.

MEDIUM

✔ **Resolved**

Validate timestamp dependent variables.

Functions addMintTicket() and editMintTicket(). Variables '_earlyAccessOpens' and '_publicSaleOpens' should be validated to be greater than block.timestamp to verify that sale periods are happening at the actual time period.

Recommendation:

Verify that '_earlyAccessOpens' and '_publicSaleOpens' are greater than block.timestamp.

LOW

✔ **Unresolved**

Sending 'dust' values back to the message caller consumes more gas than will actually be sent.

Lines 184, 229. Sending an exceeded value each time can cause unnecessary gas spending(for example if the exceeded value is too low and transferring it back will cost more gas than the actual or exceeded amount).

Recommendation:

Do not send funds if transfer takes more gas than will actually be sent.

LOWEST	✔ Resolved

Unnecessary usage of SafeMath library.

Starting from Solidity version 0.8 usage of SafeMath library is unnecessary since Solidity has built-in checks for over/underflow. SafeMath only increases gas spending during function calls.

Recommendation:

Replace all SafeMath functions with arithmetic operators.

LOWEST	✔ Resolved

Validate function parameters.

Line 96-97, 111. 144. Constructor parameters '_treasuryAddress' and '_nemusAddress' should be validated not to be zero address. Parameter '_redeemableContract' in functions addMintTicket() and editMintTicket() should be validated as well.

Recommendation:

Add 'requires' to validate that address parameters are not zero addresses.

LOWEST	✔ Unresolved

Unnecessary usage of 'require'

Line 180, 209. Using 'require' is unnecessary and only increases gas spendings since function isValidClaim() doesn't return false. It either reverts or returns true.

Recommendation:

Remove 'require' and just call the function isValidClaim() instead.

LOWEST	✔ Unresolved

Use the Address library.

Lines 185-186, 230-231, 382-383, 385-386.. ETH should be sent with Address.sendValue. This function performs all the necessary security checks.

Recommendation:

Use the Address library instead.

Post-audit.

Usage of 'call' already can cause reentrancy, however all the changes of storage variables in the code happens after the call. The team can also consider using a Non-reentrant library to increase protection of the code.
Example of Address library usage:
Address.sendValue(_msgSender(), excessPayment)

LOWEST	✔ Unresolved

'Require' statement will never revert

Line 324. This 'require' will never revert since in case index doesn't exist, it will revert on line 321.
Lines 228, 286. These 'requires' will never revert, because of subtractions on lines 224, 282 which will revert sooner, than 'requires

Recommendation:

Remove unnecessary 'require'

LOWEST	✔ Resolved

Add more setters.

Currently, in order to change one parameter, the admin has to call the function editMintTicket() which changes all the parameters of sale.

Recommendation:

Add more setters.

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

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Contract: NeaMintTicketFactory

Early access claim function

- ✓ Should not let claim during early acces if zero provided (435ms)
- ✓ Should not let not ealy accesser claim during early access (485ms)
- ✓ Shoult not let claim early access tokens if early access period finished (439ms)
- ✓ Should not let claim during early access if not enough Ether sent (498ms)
- ✓ Should not let claim during early access if amount exceeds max supply (983ms)
- ✓ Should not let claim if amount exceeds max amount per early accesser (530ms)
- ✓ Should claim early access tokens (1144ms)
- ✓ Should return exceed Ether during early access claim (820ms)

Claim function

- ✓ Should not let claim if paused (737ms)
- ✓ Should not let claim if index doesn't exist (405ms)
- ✓ Should not let claim if sale is paused (1459ms)
- ✓ Should not let claim if sale is not live (410ms)
- ✓ Should not let claim if not enough Ether sent (785ms)
- ✓ Should not let claim more than allowed for one wallet (2160ms)
- ✓ Should not let claim more than allowed for one tx (505ms)
- ✓ Should not claim more than max supply (2998ms)
- ✓ Should claim (1420ms)

- ✓ Should return exceeded Ether during claim (570ms)
- ✓ Should revert if sale is paused (854ms)

Multiple early claim

- ✓ Should claim multiple early access (3695ms)
- ✓ Should not let claim multiple more, than allowed for early claim (1459ms)
- ✓ Should not let early claim multiple if paused (877ms)
- ✓ Should not let early claim multiple if one or more claim is invalid (1247ms)
- ✓ Should not let early claim multiple if not enough Ether sent (695ms)
- ✓ Should return exceeded Ether during early claim (1110ms)
- ✓ Should not let pass repeatable indexes (723ms)

Multiple claim function

- ✓ Should claim multiple (2933ms)
- ✓ Should not let claim multiple when paused (1016ms)
- ✓ Should not let claim if one or more claims are invalid (1364ms)
- ✓ Should not let claim if not enough Ether sent (976ms)
- ✓ Should return exceeded Ether (1277ms)
- ✓ Should not let put repeatable indexes in array (736ms)

Test admin and view functions

- ✓ Should turn sale off and on (1142ms)
- ✓ Should add and remove from early access list (1036ms)
- ✓ Should return ticket size id (137ms)
- ✓ Should mint (641ms)
- ✓ Should mint batch (1676ms)
- ✓ Should withdraw funds (1068ms)

- ✓ Should not change percentages if their summation is greater than 100 (338ms)
- ✓ Should change percentages (1564ms)
- ✓ Should return uri (738ms)
- ✓ Should let admin unpause (3795ms)
- ✓ Should return name and symbol (298ms)
- ✓ Should set new URI (1540ms)
- ✓ Should set new owner (2925ms)
- ✓ Should return support interface id (149ms)

Test add and edit ticket

- ✓ Should revert if early sale open is after public sale open (949ms)
- ✓ Should revert if public sale open is after public sale close (1904ms)
- ✓ Cannot set open or close sale to zero (1075ms)
- ✓ Cannot set open sale below current timestamp (599ms)
- ✓ Cannot set redeemable contract to zero address (1003ms)
- ✓ Should not edit params to non-existant ticket (624ms)
- ✓ Should set early access start to existing ticket (1396ms)
- ✓ Should set public access start to existing ticket (1771ms)
- ✓ Should set public sale end to existing ticket (1218ms)
- ✓ Should set new token price to existing ticket (993ms)
- ✓ Should set new total supply to existing ticket (1328ms)
- ✓ Should set new max mint per tx to existing ticket (1262ms)
- ✓ Should set new max mint per wallet (1143ms)

- ✓ Should set new size id to existing ticket (1028ms)
- ✓ Should set new metadata id to existing ticket (995ms)
- ✓ Should set new redeemable contract to existing ticket (1624ms)

Test burn

- ✓ Should burn from redeem (1355ms)
- ✓ Only redeemable contract can burn (1243ms)

Test burn

- ✓ Should burn from redeem (1355ms)
- ✓ Only redeemable contract can burn (1243ms)

64 passing (2m)

TEST COVERAGE RESULTS

FILE	% STMTS	% BRANCH	% FUNCS
AbstractMintVoucherFactory	91.67	100	91.67
NeaMintTicketFactory	100	91.91	100
All files	**99.46**	**91.91**	**97.92**

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.