SMART CONTRACT

Security Audit Report

Customer: WnDGameTG ERC721

Platform: Ethereum

Language: Solidity

Contents

Commission

Audited Project	**WnDGameTG**
Contract Owner	**0xc7defa20ec54917669f29e15d1acb7c121b4780c**
Smart Contract	**0xeE231dca742BcD784dc07BA5eC935aBB98E468d7**
Blockchain	**Ethereum Mainnet**

Arbitech Solutions was commissioned by **WnDGameTG** smart contract owners to perform an audit of their main smart contract. The purpose of the audit was to achieve the following:

- Ensure that the smart contract functions as intended.
- Identify potential security issues with the smart contract.

The information in this report should be used to understand the risk exposure of the smart contract, and as a guide to improve the security posture of the smart contract by remediating the issues that were identified.

Disclaimer

This is a limited report on our finding based on our analysis, in accordance with good industry practice as at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below please make sure to read it in full.

DISCLAIMER: By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis, and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and Arbitech Solutions and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers and other representatives) (Arbitech Solutions) owe no duty of care towards you or any other person, nor does Arbitech Solutions make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties or other terms of any kind except as set out in this disclaimer, and Arbitech Solutions hereby excludes all representations, warranties, conditions and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, Arbitech Solutions hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against Arbitech Solutions, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and wh in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (wh innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security.

WNDGAMETG Properties

Contract name	WnDGameTG
Contract address	0xeE231dca742BcD784dc07BA5eC935aBB98E468d7
wndNFT address	0x999e88075692bcee3dbc07e7e64cd32f39a1d3ab
Training Ground address	0xeea43b892d4593e7c44530db1f75b0519db0bd50
Traits address	0xa895a9b7515452073d690dbe1f809536c83d1a35
Contract deployer address	0xC7dEFA20Ec54917669f29e15D1ACB7c121b4780c
Contract’s current owner address	0xc7defa20ec54917669f29e15d1acb7c121b4780c

Contract Functions

View

i. function getPendingTrainingCommits(address addr) external view returns (uint16)
ii. function getPendingMintCommits(address addr) external view returns (uint16)
iii. function hasStaleMintCommit() external view returns (bool)
iv. function hasStaleTrainingCommit() external view returns (bool)
v. function isTokenPendingReveal(uint256 tokenId) external view returns (bool)
vi. function mintCost(uint256 tokenId, uint256 maxTokens) public view returns (uint256)
vii. function paused() public view virtual returns (bool)

Executables

i. function addToTower(uint16[] calldata tokenIds) external whenNotPaused
ii. function addToTrainingCommit(uint16[] calldata tokenIds) external whenNotPaused
iii. function claimTrainingsCommit(uint16[] calldata tokenIds, bool isUnstaking, bool isTraining) external whenNotPaused
iv. function makeTreasureChests(uint16 qty) external whenNotPaused
v. function mintCommit(uint256 amount, bool stake) external whenNotPaused nonReentrant
vi. function sellTreasureChests(uint16 qty) external whenNotPaused
vii. function revealOldestMint() external whenNotPaused
viii. function revealOldestTraining() external whenNotPaused

Owner Executables

i. function renounceOwnership() public virtual onlyOwner
ii. function setAllowCommits(bool allowed) external onlyOwner
iii. function setContracts(address _gp, address _traits, address _wnd, address _alter, address _trainingGrounds) external onlyOwner
iv. function setMaxGpCost(uint256 _amount) external requireContractsSet onlyOwner
v. function setPaused(bool _paused) external requireContractsSet onlyOwner
vi. function setPendingMintAmt(uint256 pendingAmt) external onlyOwner
vii. function setRevealRewardAmt(uint256 rewardAmt) external onlyOwner
viii. function skipOldestMint() external onlyOwner
ix. function skipOldestTraining() external onlyOwner
x. function transferOwnership(address newOwner) public virtual onlyOwner
xi. function withdraw() external onlyOwner

Checklist

Compiler errors.	Passed
Possible delays in data delivery.	Passed
Timestamp dependence.	Low Severity
Integer Overflow and Underflow.	Passed
Race Conditions and Reentrancy.	Passed
DoS with Revert.	Passed
DoS with gas limit.	Passed
Methods execution permissions.	Passed
Economy model of the contract.	Passed
Private user data leaks.	Passed
Malicious Events Log.	Passed
Scoping and Declarations.	Passed
Uninitialized storage pointers.	Passed
Arithmetic accuracy.	Passed
Design Logic.	Passed
Impact of the exchange rate.	Passed
Oracle Calls.	Passed
Cross-function race conditions.	Passed
Fallback function security.	Passed
Front Running.	Passed
Safe Open Zeppelin contracts and implementation usage.	Passed
Whitepaper-Website-Contract correlation.	Passed

WNDGAMETG Contract

This function will pass array of tokenID's to training ground contract. Note: token id has no pending commit. If the length of tokenID's array increase the gas fee will also increase. Gas Cost is increasing exponentially with each itteration of loop.

```
function addToTower(uint16[] calldata tokenIds) external whenNotPaused {
  require(_msgSender() == tx.origin, "Only EOA");
  for (uint256 i = 0; i < tokenIds.length; i++) {
    require(!tokenHasPendingCommit[tokenIds[i]], "token has pending commit");
  }
  trainingGrounds.addManyToTowerAndFlight(tx.origin, tokenIds);
}
```

Check if current commit batch is past the threshold for time and increment commitId if so increment commitId to start a new batch. Loop through the number of tokens being added but if number of tokenId's increase the length of array increase which will consume more gas. Add N number of commits to the queue. This is so people reveal the same number of commits as they added.

```
function addToTrainingCommit(uint16[] calldata tokenIds) external whenNotPaused {
  require(allowCommits, "adding commits disallowed");
  require(tx.origin == _msgSender(), "Only EOA");
  if(commitIdStartTimeTraining[_commitIdCurTraining] == 0) {
    commitIdStartTimeTraining[_commitIdCurTraining] = block.timestamp;
  }

  // Check if current commit batch is past the threshold for time and increment commitId if so
  if(commitIdStartTimeTraining[_commitIdCurTraining] < block.timestamp - timePerCommitBatch) {
    // increment commitId to start a new batch
    _commitIdCurTraining += 1;
    commitIdStartTimeTraining[_commitIdCurTraining] = block.timestamp;
  }
  // Loop through the amount of tokens being added
  uint16 numDragons;
  for (uint i = 0; i < tokenIds.length; i++) {
    require(address(trainingGrounds) != wndNFT.ownerOf(tokenIds[i]), "token already staked");
    require(!tokenHasPendingCommit[tokenIds[i]], "token has pending commit");
    require(_msgSender() == wndNFT.ownerOf(tokenIds[i]), "not owner of token");
    if(!wndNFT.isWizard(tokenIds[i])) {
      numDragons += 1;
    }
```

```
    tokenHasPendingCommit[tokenIds[i]] = true;
    // Add N number of commits to the queue. This is so people reveal the same number of commits as they added.
    commitQueueTraining[_commitIdCurTraining].push(TrainingCommit(_msgSender(), tokenIds[i], true, false, true));
  }
  gpToken.burn(_msgSender(), stakingCost * (tokenIds.length - numDragons)); // Dragons are free to stake
  gpToken.updateOriginAccess();
  pendingTrainingCommitsForAddr[_msgSender()] += uint16(tokenIds.length);
  tryRevealTraining(tokenIds.length);
}
```

Check if current commit batch is past the threshold for time and increment commitId if so increment commitId to start a new batch. Loop through the number of tokens being added but if number of tokenId's increase the length of array increase which will consume more gas. Add N number of commits to the queue. This is so people reveal the same number of commits as they added. Check to see if the wizard has earned enough to withdraw. If emissions run out, allow them to attempt to withdraw anyways. Add N number of commits to the queue. This is so people reveal the same number of commits as they added.

```
function claimTrainingsCommit(uint16[] calldata tokenIds, bool isUnstaking, bool isTraining) external whenNotPaused {
  require(allowCommits, "adding commits disallowed");
  require(tx.origin == _msgSender(), "Only EOA");
  if(commitIdStartTimeTraining[_commitIdCurTraining] == 0) {
    commitIdStartTimeTraining[_commitIdCurTraining] = block.timestamp;
  }

  // Check if current commit batch is past the threshold for time and increment commitId if so
  if(commitIdStartTimeTraining[_commitIdCurTraining] < block.timestamp - timePerCommitBatch) {
    // increment commitId to start a new batch
    _commitIdCurTraining += 1;
    commitIdStartTimeTraining[_commitIdCurTraining] = block.timestamp;
  }
  uint16 numDragons;
  // Loop through the amount of tokens being added
  for (uint i = 0; i < tokenIds.length; i++) {
    require(!tokenHasPendingCommit[tokenIds[i]], "token has pending commit");
    require(trainingGrounds.isTokenStaked(tokenIds[i], isTraining) && trainingGrounds.ownsToken(tokenIds[i])
    , "Token not in staking pool");
    if(isUnstaking) {
      if(wndNFT.isWizard(tokenIds[i])) {
```

```
      // Check to see if the wizard has earned enough to withdraw.
      // If emissions run out, allow them to attempt to withdraw anyways.
      if(isTraining) {
        require(trainingGrounds.curWhipsEmitted() >= 16000
          || trainingGrounds.calculateErcEmissionRewards(tokenIds[i]) > 0, "can't unstake wizard yet");
      }
      else {
        require(trainingGrounds.totalGPEarned() > 500000000 ether - 4000 ether
          || trainingGrounds.calculateGpRewards(tokenIds[i]) >= 4000 ether, "can't unstake wizard yet");
      }
    }
    else {
      numDragons += 1;
    }
  }
  tokenHasPendingCommit[tokenIds[i]] = true;
  // Add N number of commits to the queue. This is so people reveal the same number of commits as they added.
  commitQueueTraining[_commitIdCurTraining].push(TrainingCommit(_msgSender(), tokenIds[i], false, isUnstaking, isTraining));
}
if(isUnstaking) {
    gpToken.burn(_msgSender(), stakingCost * (tokenIds.length - numDragons)); // Dragons are free to stake
    gpToken.updateOriginAccess();
  }
  pendingTrainingCommitsForAddr[_msgSender()] += uint16(tokenIds.length);
  tryRevealTraining(tokenIds.length);
}
```

$GP exchange amount handled within alter contract .Will fail if sender doesn't have enough $GP .Transfer does not need approved, as there is established trust between this contract and the alter contract

```
function makeTreasureChests(uint16 qty) external whenNotPaused {
  require(tx.origin == _msgSender(), "Only EOA");
  // $GP exchange amount handled within alter contract
  // Will fail if sender doesn't have enough $GP
  // Transfer does not need approved,
  //  as there is established trust between this contract and the alter contract
  alter.mint(TREASURE_CHEST, qty, _msgSender());
}
```

Initiate the start of a mint. This action burns $GP, as the intent of committing is that you cannot back out once you've started. This will add users into the pending queue, to be revealed after a random seed is generated and assigned to the commit id this commit was added to. if current commit batch is past the threshold for time and increment commitId if so increment commitId to start a new batch. Add this mint request to the commit queue for the current commitId. Loop through the amount of. Add N number of commits to the queue. This is so people reveal the same number of commits as they added. if the revealable commitId has anything to commit and increment it until it does, or is the same as the current commitId. Only iterate if the commit pending is empty and behind the current id. This is to prevent it from being in front of the current id and missing commits. if there is a commit in a revealable batch and pop/reveal it. If the pending batch is old enough to be revealed and has stuff in

it, mine the number that was added to the queue. First iteration is guaranteed to have 1 commit to mine, so we can always retroactively check that we can continue to reveal after. To see if we are able to continue mining commits. If there are no more commits to reveal, exit.

```
function mintCommit(uint256 amount, bool stake) external whenNotPaused nonReentrant {
  require(allowCommits, "adding commits disallowed");
  require(tx.origin == _msgSender(), "Only EOA");
  uint16 minted = wndNFT.minted();
  uint256 maxTokens = wndNFT.getMaxTokens();
  require(minted + pendingMintAmt + amount <= maxTokens, "All tokens minted");
  require(amount > 0 && amount <= 10, "Invalid mint amount");
  if(commitIdStartTimeMints[_commitIdCurMints] == 0) {
    commitIdStartTimeMints[_commitIdCurMints] = block.timestamp;
  }

  // Check if current commit batch is past the threshold for time and increment commitId if so
  if(commitIdStartTimeMints[_commitIdCurMints] < block.timestamp - timePerCommitBatch) {
    // increment commitId to start a new batch
    _commitIdCurMints += 1;
    commitIdStartTimeMints[_commitIdCurMints] = block.timestamp;
  }

  // Add this mint request to the commit queue for the current commitId
  uint256 totalGpCost = 0;
  // Loop through the amount of
  for (uint i = 1; i <= amount; i++) {
    // Add N number of commits to the queue. This is so people reveal the same number of commits as they added.
    commitQueueMints[_commitIdCurMints].push(MintCommit(_msgSender(), stake, 1));
    totalGpCost += mintCost(minted + pendingMintAmt + i, maxTokens);
  }
```

```
if (totalGpCost > 0) {
  gpToken.burn(_msgSender(), totalGpCost);
  gpToken.updateOriginAccess();
}
uint16 amt = uint16(amount);
pendingMintCommitsForAddr[_msgSender()] += amt;
pendingMintAmt += amt;

// Check if the revealable commitId has anything to commit and increment it until it does, or is the same as the current commitId
while(commitQueueMints[_commitIdPendingMints].length == 0 && _commitIdPendingMints < _commitIdCurMints) {
  // Only iterate if the commit pending is empty and behind the current id.
  // This is to prevent it from being in front of the current id and missing commits.
  _commitIdPendingMints += 1;
}
// Check if there is a commit in a revealable batch and pop/reveal it
if(commitIdStartTimeMints[_commitIdPendingMints] < block.timestamp - timePerCommitBatch && commitQueueMints[_commitIdPendingMints].length > 0)
  // If the pending batch is old enough to be revealed and has stuff in it, mine the number that was added to the queue.
  for (uint256 i = 0; i < amount; i++) {
    // First iteration is guaranteed to have 1 commit to mine, so we can always retroactively check that we can continue to reveal after
    MintCommit memory commit = commitQueueMints[_commitIdPendingMints][commitQueueMints[_commitIdPendingMints].length - 1];
    commitQueueMints[_commitIdPendingMints].pop();
    revealMint(commit);
    // Check to see if we are able to continue mining commits
    if(commitQueueMints[_commitIdPendingMints].length == 0 && _commitIdPendingMints < _commitIdCurMints) {
```

$GP exchange amount handled within alter contract. “qty” is passed to the alter contract’s function burn where this will burn.

```
function sellTreasureChests(uint16 qty) external whenNotPaused {
  require(tx.origin == _msgSender(), "Only EOA");
  // $GP exchange amount handled within alter contract
  alter.burn(TREASURE_CHEST, qty, _msgSender());
}
```

Allow users to reveal the oldest commit for GP. Mints commits must be stale to be able to be revealed this way. if the revealable commitId has anything to commit and increment it until it does, or is the same as the current commitId. Only iterate if the commit pending is empty and behind the current id. This is to prevent it from being in front of the current id and missing commits. if there is a commit in a revealable batch and pop/reveal it. If the pending batch is old enough to be revealed and has stuff in it, mine one.

```
function revealOldestMint() external whenNotPaused {
  require(tx.origin == _msgSender(), "Only EOA");

  /* Check if the revealable commitId has anything to commit and increment it until it does, or is the same
  as the current commitId */
  while(commitQueueMints[_commitIdPendingMints].length == 0 && _commitIdPendingMints < _commitIdCurMints) {
    // Only iterate if the commit pending is empty and behind the current id.
    // This is to prevent it from being in front of the current id and missing commits.
    _commitIdPendingMints += 1;
  }
  // Check if there is a commit in a revealable batch and pop/reveal it
  require(commitIdStartTimeMints[_commitIdPendingMints] < block.timestamp - timeToAllowArb &&
   commitQueueMints[_commitIdPendingMints].length > 0, "No stale commits to reveal");
  // If the pending batch is old enough to be revealed and has stuff in it, mine one.
  MintCommit memory commit = commitQueueMints[_commitIdPendingMints][commitQueueMints[_commitIdPendingMints].length - 1];
  commitQueueMints[_commitIdPendingMints].pop();
  revealMint(commit);
  gpToken.mint(_msgSender(), revealRewardAmt * commit.amount);
}
```

if the revealable commitId has anything to commit and increment it until it does, or is the same as the current commitId. Only iterate if the commit pending is empty and behind the current id. This is to prevent it from being in front of the current id and missing commits. Check if there is a commit in a revealable batch and pop/reveal it. If the pending batch is old enough to be revealed and has stuff in it, mine one.

```
function revealOldestTraining() external whenNotPaused {
  require(tx.origin == _msgSender(), "Only EOA");

  /* Check if the revealable commitId has anything to commit and increment it until it does, or is the same as the
   current commitId */
  while(commitQueueTraining[_commitIdPendingTraining].length == 0 && _commitIdPendingTraining < _commitIdCurTraining) {
    // Only iterate if the commit pending is empty and behind the current id.
    // This is to prevent it from being in front of the current id and missing commits.
    _commitIdPendingTraining += 1;
  }
  // Check if there is a commit in a revealable batch and pop/reveal it
  require(commitIdStartTimeTraining[_commitIdPendingTraining] < block.timestamp - timeToAllowArb &&
  commitQueueTraining[_commitIdPendingTraining].length > 0, "No stale commits to reveal");
  // If the pending batch is old enough to be revealed and has stuff in it, mine one.
  TrainingCommit memory commit = commitQueueTraining[_commitIdPendingTraining]
  [commitQueueTraining[_commitIdPendingTraining].length - 1];
  commitQueueTraining[_commitIdPendingTraining].pop();
  revealTraining(commit);
  gpToken.mint(_msgSender(), revealRewardAmt);
}
```

Leaves the contract without owner. It will not be possible to call `onlyOwner` functions anymore. Can only be called by the current owner. Renouncing ownership will leave the contract without an owner, thereby removing any functionality that is only available to the owner.

```
function renounceOwnership() public virtual onlyOwner {
    _setOwner(address(0));
}
```

Owner call this function to allow commits.

```
function setAllowCommits(bool allowed) external onlyOwner {
  allowCommits = allowed;
}
```

Owner call this function to update the gpToken contract address, traits contract address, wndNFT contract address, alter contract address and trainingGrounds contract address.

```
function setContracts(address _gp, address _traits, address _wnd, address _alter, address _trainingGrounds)
 external onlyOwner {
  gpToken = IGP(_gp);
  traits = ITraits(_traits);
  wndNFT = IWnD(_wnd);
  alter = ISacrificialAlter(_alter);
  trainingGrounds = ITrainingGrounds(_trainingGrounds);
}
```

Owner call this function to set new “maxGpCost”.

```
function setMaxGpCost(uint256 _amount) external requireContractsSet onlyOwner {
  maxGpCost = _amount;
}
```

Owner call this function to pause / unpause contract.

```
function setPaused(bool _paused) external requireContractsSet onlyOwner {
  if (_paused) _pause();
  else _unpause();
}
```

Allow the contract owner to set the pending mint amount. This allows any long-standing pending commits to be overwritten, say for instance if the max supply has been reached but there are many stale pending commits, it could be used to free up those spaces if needed/desired by the community. This function should not be called lightly, this will have negative consequences on the game.

```
function setPendingMintAmt(uint256 pendingAmt) external onlyOwner {
  pendingMintAmt = uint16(pendingAmt);
}
```

Owner call this function to set new revealRewardamt.

```
function setRevealRewardAmt(uint256 rewardAmt) external onlyOwner {
  revealRewardAmt = rewardAmt;
}
```

Allow users to reveal the oldest commit for GP. Mints commits must be stale to be able to be revealed this way. But onlyOwner can call this function. if the revealable commitId has anything to commit and increment it until it does, or is the same as the current commitId. Only iterate if the commit pending is empty and behind the current id. This is to prevent it from being in front of the current id and missing commits. if there is a commit in a revealable batch and pop/reveal it. If the pending batch is old enough to be revealed and has stuff in it, mine one. Do not reveal the commit, only pop it from the queue and move on.

```
function skipOldestMint() external onlyOwner {
  /* Check if the revealable commitId has anything to commit and increment it until it does, or is
  the same as the current commitId*/
  while(commitQueueMints[_commitIdPendingMints].length == 0 && _commitIdPendingMints < _commitIdCurMints) {
    // Only iterate if the commit pending is empty and behind the current id.
    // This is to prevent it from being in front of the current id and missing commits.
    _commitIdPendingMints += 1;
  }
  // Check if there is a commit in a revealable batch and pop/reveal it
  require(commitQueueMints[_commitIdPendingMints].length > 0, "No stale commits to reveal");
  // If the pending batch is old enough to be revealed and has stuff in it, mine one.
  commitQueueMints[_commitIdPendingMints].pop();
  // Do not reveal the commit, only pop it from the queue and move on.
  // revealMint(commit);
}
```

Transfers ownership of the contract to a new account (`newOwner`). Can only be called by the current owner.

```
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    _setOwner(newOwner);
}
```

allows owner to withdraw funds from minting.

```
function withdraw() external onlyOwner {
  payable(owner()).transfer(address(this).balance);
}
```

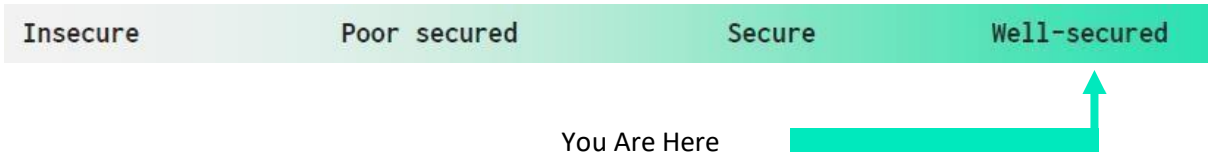
Quick Stats:

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version too old	Passed
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	N/A
	Fallback function misuse	N/A
	Race condition	Passed
	Logical vulnerability	Passed
	Other programming issues	Passed
Code Specification	Visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Other code specification issues	Passed
Gas Optimization	Assert () misuse	Passed
	High consumption ‘for/while’ loop	Passed
	High consumption ‘storage’ storage	Passed
	“Out of Gas” Attack	Passed
Business Risk	The maximum limit for mintage not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: Passed

Executive Summary

According to the standard audit assessment, Customer`s solidity smart contract is Well-Secure. Again, it is recommended to perform an Extensive audit assessment to bring a more assured conclusion.



We used various tools like Mythril, Slither and Remix IDE. At the same time this finding is based on critical analysis of the manual audit.

All issues found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the Quick Stat section.

We found 0 critical, 0 high, 0 medium and 1 low level issues.

Code Quality

The WNDGAMETG ERC721 Token protocol consists of one smart contract. It has other inherited contracts like IWnDGame, Ownable, ReentrancyGuard, Pausable. These are compact and well written contracts. Libraries used in WNDGAMETG ERC721 Token are part of its logical algorithm. They are smart contracts which contain reusable code. Once deployed on the Blockchain (only once), it is assigned a specific address and its properties / methods can be reused many times by other contracts in protocol. The ARBITECH SOLUTIONS team has **not** provided scenario and unit test scripts, which would help to determine the integrity of the code in an automated way.

Overall, the code is not commented. Commenting can provide rich documentation for functions, return variables and more.

Documentation

As mentioned above, it's recommended to write comments in the smart contract code, so anyone can quickly understand the programming flow as well as complex code logic. We were given a WNDGAMETG contract code in the form of File.

Use of Dependencies

As per our observation, the libraries are used in this smart contract infrastructure that are based on well-known industry standard open-source projects. And even core code is written well and systematically. This smart contract does not interact with other external smart contracts.

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to token loss etc.
High	High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. public access to crucial functions
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens lose
Low	Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can't have significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations and info statements can't affect smart contract execution and can be ignored.

Audit Findings

Critical

No Critical severity vulnerabilities were found.

High

No high severity vulnerabilities were found.

Medium

No Medium severity vulnerabilities were found.

Low

(1) Compiler version can be upgraded.

```
pragma solidity ^0.8.0;
```

Although this does not raise any security vulnerability, using the latest compiler version can help to prevent any compiler level bugs.

Solution: This issue is acknowledged.

Conclusion

The Smart Contract code passed the audit successfully on the Ethereum Mainnet with some considerations to take. There were one low severity warnings raised meaning that they should be taken into consideration. The last change is advisable in order to provide more security to new holders. Nonetheless this is not necessary if the holders and/or investors feel confident with the contract owners. We were given a contract code. And we have used all possible tests based on given objects as files. So, it is good to go for production.
Since possible test cases can be unlimited for such extensive smart contract protocol, hence we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover maximum possible test cases to scan everything. Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart Contract's high-level description of functionality was presented in Quick Stat section of the report.

Audit report contains all found security vulnerabilities and other issues in the reviewed code.

Security state of the reviewed contract is "Well- Secured".

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review:

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high-level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Disclaimers

Privacy Arbitech Solutions Disclaimer

Arbitech Solutions team has analyzed this smart contract in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment and functionality (performing the intended functions).

Due to the fact that the total number of test cases are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug free status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Technical Disclaimer

Smart contracts are deployed and executed on the Blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks.

Thus, the audit can’t guarantee explicit security of the audited smart contracts.