

# BetDeEx Smart Contract Audit

## Overview

This Audit Report highlights the overall security of BetDeEx Smart Contracts. Ginete Technologies performed a security review of the BetDeEx Smart Contracts, at the request of the EraSwap team. The version used for this report is commit:

```
25c1dada8a8f89e96fa6e57f226be6b7dbaea36e
```

## Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the contract's intended purpose by reading the available documentation.
2. Automated scanning of the contract with static code analysis tools for security vulnerabilities and use of best practice guidelines.
3. Manual line by line analysis of the contract's source code for security vulnerabilities and use of best practice guidelines, including but not limited to: re-entrancy analysis, race condition analysis, front-running issues and transaction order dependencies, timestamp dependencies, under- / overflow issues, function visibility issues, possible denial of service attacks, and storage layout vulnerabilities.
4. Report preparation.

## Security Level References

Every issue in this report was assigned a severity level from the following:

1. **High Severity** issues will probably bring problems and should be fixed.
2. **Medium Severity** issues could potentially bring problems and should eventually be fixed.
3. **Low Severity** issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

# Scope of this Audit

The scope of the audit, conducted by Ginete Technologies, was restricted to:

1. The git repository located at:  
<https://github.com/zemse/betdeex/>
2. **BetDeEx.sol** Smart contract at commit  
`25c1dada8a8f89e96fa6e57f226be6b7dbaea36e`

# Automatic Auditing Tool Outputs

## 1. Slither

Slither, is another static analysis tool from Trail of Bits. It includes aids for contract summaries, which can be helpful for making a mental model of the contract and rechecking assumptions.

```

Administrator C:\Windows\System32\cmd.exe
Compiling your contracts...
=====
> Compiling \contracts\BetDeEx.sol
> Compiling \contracts\ESContract.sol
> Compiling \contracts\migrations.sol
> Compiling \contracts\SSContract.sol
> Artifacts written to C:\Users\anudit\Documents\Github\betdeex\build\contracts
> Compilation successful by using:
- solc: 0.5.0+commit.1d4f565a.Emscripten.clang

INFO:Detectors:
pragmaERC20_name (ESContract.sol#398) shadows:
- detailERC20_name (ESContract.sol#92)
pragmaERC20_symbol (ESContract.sol#400) shadows:
- detailERC20_symbol (ESContract.sol#93)
pragmaERC20_decimal (ESContract.sol#404) shadows:
- detailERC20_decimal (ESContract.sol#94)
pragmaERC20_cap (ESContract.sol#401) shadows:
- CappedToken_cap (ESContract.sol#370)
Reference: https://github.com/cryptic/slither/wiki/Detector-documentation#state-variable-shadowing
INFO:Detectors:
arrayIndex in bet_enterBet (BetDeEx.sol#205) is a local variable never initialized
Reference: https://github.com/cryptic/slither/wiki/Detector-documentation#initialized-local-variables
INFO:Detectors:
loc_endless (BetDeEx.sol#239-277) ignores return value by external calls "betDeEx.sendTokensToAddress(_winnerBettors[i],bettorAddress,_winnerBettors[i],betAmountInExts.mul(totalPrize).div(totalBetTokensInExtsByChoice[_choice]))" (BetDeEx.sol#259-262)
loc_endless (BetDeEx.sol#239-277) ignores return value by external calls "betDeEx.sendTokensToAddress(BetDeEx.superManager(),_platformFee)" (BetDeEx.sol#272)
Reference: https://github.com/cryptic/slither/wiki/Detector-documentation#unused-return
INFO:Detectors:
bet_amount_in_external_calls_inside_a_loop: "betDeEx.sendTokensToAddress(_winnerBettors[i],bettorAddress,_winnerBettors[i],betAmountInExts.mul(totalPrize).div(totalBetTokensInExtsByChoice[_choice]))" (BetDeEx.sol#259-262)
Reference: https://github.com/cryptic/slither/wiki/Detector-documentation#calls-inside-a-loop
INFO:Detectors:
reentrancy in BetDeEx.collectBettorTokens (BetDeEx.sol#131-137):
External call:
- require(bool)(esTokenContract.transferFrom(_from,address(this),_betTokensInExts)) (BetDeEx.sol#134)
State variables written after the call(s):
- betBalanceInExts (BetDeEx.sol#133)
Reentrancy in bet_sendBet (BetDeEx.sol#239-277):
External call:
- betDeEx.sendTokensToAddress(_winnerBettors[i],bettorAddress,_winnerBettors[i],betAmountInExts.mul(totalPrize).div(totalBetTokensInExtsByChoice[_choice])) (BetDeEx.sol#259-262)
- _platformFee = betBalanceInExts(0) (BetDeEx.sol#267)
State variables written after the call(s):
- emsibly (BetDeEx.sol#169)
Reentrancy in bet_sendBet (BetDeEx.sol#239-277):
External call:
- totalPrize = betBalanceInExts(0).mul(_prizePerContractThousand).div(1000) (BetDeEx.sol#254)
- betDeEx.sendTokensToAddress(_winnerBettors[i],bettorAddress,_winnerBettors[i],betAmountInExts.mul(totalPrize).div(totalBetTokensInExtsByChoice[_choice])) (BetDeEx.sol#259-262)
- _platformFee = betBalanceInExts(0) (BetDeEx.sol#267)
- betDeEx.sendTokensToAddress(BetDeEx.superManager(),_platformFee) (BetDeEx.sol#272)
- emit tokenMint (betDeEx.sol#274)
- emit tokenMint (betDeEx.sol#274)
Reentrancy in bet_sendBet (BetDeEx.sol#239-277):
External call:
- require(bool)(betDeEx.getBettorBalance(msg.sender) >= _betTokensInExts) (BetDeEx.sol#204)
State variables written after the call(s):
- @_bettors (BetDeEx.sol#223-224)
- @_bettors (BetDeEx.sol#209-212)
- totalBetTokensInExtsByChoice (BetDeEx.sol#207)
- totalBetTokensInExtsByChoice (BetDeEx.sol#214)
- totalBetTokensInExtsByChoice (BetDeEx.sol#214)
- _winner (BetDeEx.sol#216-219)
Reentrancy in BetDeEx.sendTokensToAddress (BetDeEx.sol#140-143):
External call:
- require(bool)(esTokenContract.transfer(_to,_tokensInExts)) (BetDeEx.sol#141)
State variables written after the call(s):
- betBalanceInExts (BetDeEx.sol#142)
Reference: https://github.com/cryptic/slither/wiki/Detector-documentation#reentrancy-vulnerabilities-2
INFO:Detectors:
pragmaERC20_name should be constant (ESContract.sol#401)
pragmaERC20_decimal should be constant (ESContract.sol#400)
pragmaERC20_name should be constant (ESContract.sol#398)

```



### 3. Truffle Compilation Verification

```
Administrator: C:\Windows\System32\cmd.exe - truffle develop

truffle(develop)> compile

Compiling your contracts...
=====
> Compiling .\contracts\BetDeEx.sol
> Compiling .\contracts\ESContract.sol
> Compiling .\contracts\Migrations.sol
> Compiling .\contracts\ESContract.sol
> Artifacts written to .\build\contracts
> Compiled successfully using:
  - solc: 0.5.0+commit.1d4f565a.Emscripten.clang

truffle(develop)>
```

### 4. Truffle Migration and Gas Estimation

```
Administrator: C:\Windows\System32\cmd.exe - truffle develop

Starting migrations...
=====
> Network name: 'develop'
> Network id: 377
> Block gas limit: 6721975

1_initial_migration.js
=====
Deploying 'Migrations'
-----
> transaction hash: 0x8d9f95a8c2aef917a1c03d52e4df32f2fc36511a40a40117ce0701dca867b
> Blocks: 0
> Seconds: 0
> contract address: 0xb848a082724aa21b65272c2a106789880826e907
> account: 0x97a2a803bc200f4e8e7ec048c3660b1054511a
> balance: 99.59830164
> gas used: 284908
> gas price: 20 gwei
> value sent: 0 ETH
> total cost: 0.00569816 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost: 0.00569816 ETH

2_deploy_contracts.js
=====
Deploying 'EraswapToken'
-----
> transaction hash: 0xfcc52916700d573ae3777983bd134dfa3957da875bced7348f0f5db401b7b6
> Blocks: 0
> Seconds: 0
> contract address: 0x0c8f8e294844641d07c9a07985455b0140ffa099
> account: 0x97a2a803bc200f4e8e7ec048c3660b1054511a
> balance: 99.60836678
> gas used: 534619
> gas price: 20 gwei
> value sent: 0 ETH
> total cost: 0.10709238 ETH

Deploying 'BetDeEx'
-----
> transaction hash: 0x8e47d4215b83ab17519096fb60b900c3e5675216f86fa0e9b164aa0d5c2827e
> Blocks: 0
> Seconds: 0
> contract address: 0xb5a4e6e440e3b3aacc97a7d07307ff1f08c22f26
> account: 0x97a2a803bc200f4e8e7ec048c3660b1054511a
> balance: 99.62093668
> gas used: 327105
> gas price: 20 gwei
> value sent: 0 ETH
> total cost: 0.0654321 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost: 0.17252448 ETH

Summary
-----
> Total deployments: 3
> Final cost: 0.1722264 ETH

truffle(develop)>
```

## Contract Deployment Testing

The contracts have been deployed and tested at the following addresses on the Ropsten Testnet:

- **ESContract.sol** at `0xd7bf79aed5d06a7282b02805f15667aa6a1eaa6`
- **BetDeEx.sol** at `0x98ff29bccc1d89d648e990cd4e80c5050aa5cf50`

## High Severity

All the instances of high severity instances have been resolved and now the code doesn't seem to have any critical severity bug.

## Medium Severity

All the instances of medium severity instances has been resolved and now the code doesn't seems to have any bug.

## Low Severity

All the instances of low severity instances, warnings, best practices have been resolved and now the code doesn't seem to have the warnings, bugs.

## Possible Issues

### 1. Racing Condition

All transactions in Ethereum are run serially. Just one after another. Everything your transaction executes, including calling from one contract to another, happens within the context of your transaction and nothing else runs until your contract is done.

### 2. Changing of values inside a function

The require conditions in conjunction with the modifiers setup with the function definition prevent unintended access so that the function cannot be accessed by someone else.

### **3. Cross-Racing Condition**

Race conditions should not be a concern. You can call `balanceOf()` on another contract, put the result in a local variable, and use it with no worries that the balance in the other contract will change before you're done.

### **4. Serial Function Calls**

Continuous function calls one after the other do not affect the state as the Ethereum Virtual Machine keeps track of all the changes and verifies the validity of the transactions happening.

### **5. Conditional Access using require**

The `require` conditions in conjunction with the modifiers setup with the function definition prevent unintended access so that the function cannot be accessed by someone else.

### **6. Ether Lock**

It has been verified that the transaction is reverted if any ether is sent to contract unintentionally and the modifiers prevent any ethers being transferred inside the contract from anyone else.

## Summary of the Audit

The contract seems to have implemented the best security practices. It is good to use the OpenZeppelin framework wherever required and the contract is using it very efficiently where required. The contract stores the funds safely and transacts safely wherever needed.

The contracts are written keeping in mind the best security practices of Solidity and it is using the **pull** mechanism of the funds which is the best way to avoid any attacks and the misuse of the funds by the attackers. The contract is also using the **SafeMath** library of OpenZeppelin which avoids the underflows and overflows. All the vulnerabilities found in the previous version of the audit have been fixed by the team.

Since the contract is free from any security vulnerability and external attacks, the contract is **Good to Deploy** over mainnet.