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/

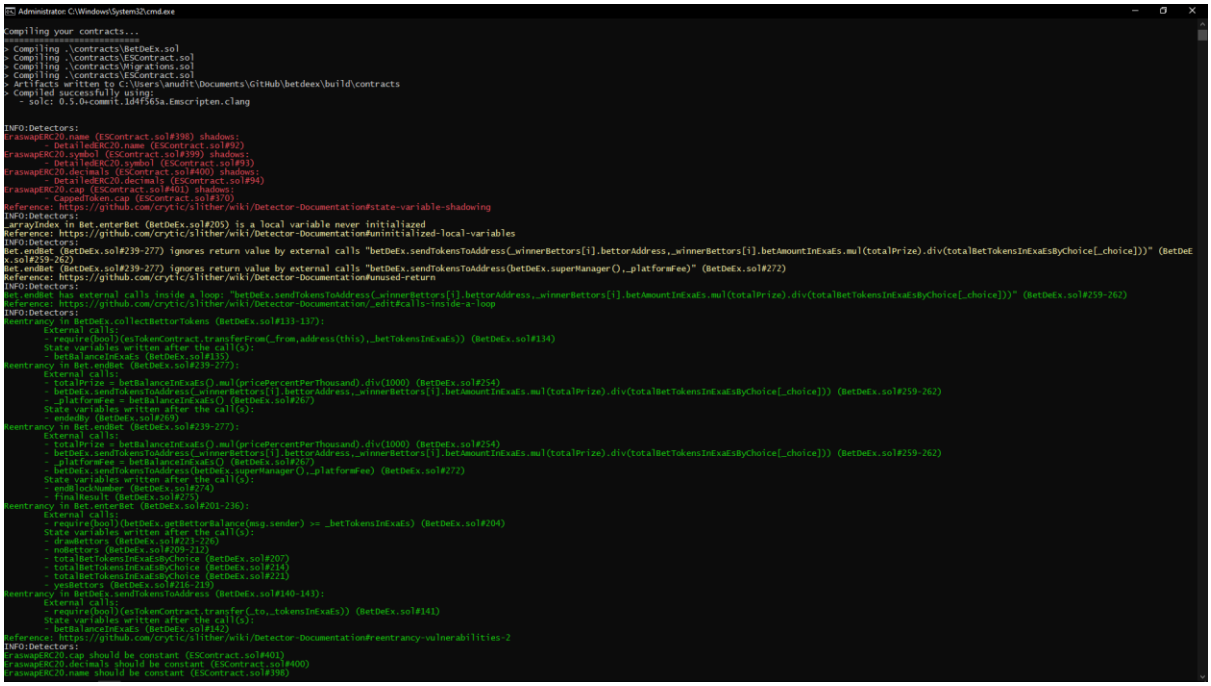
2. **BetDeEx.sol** Smart contract at commit

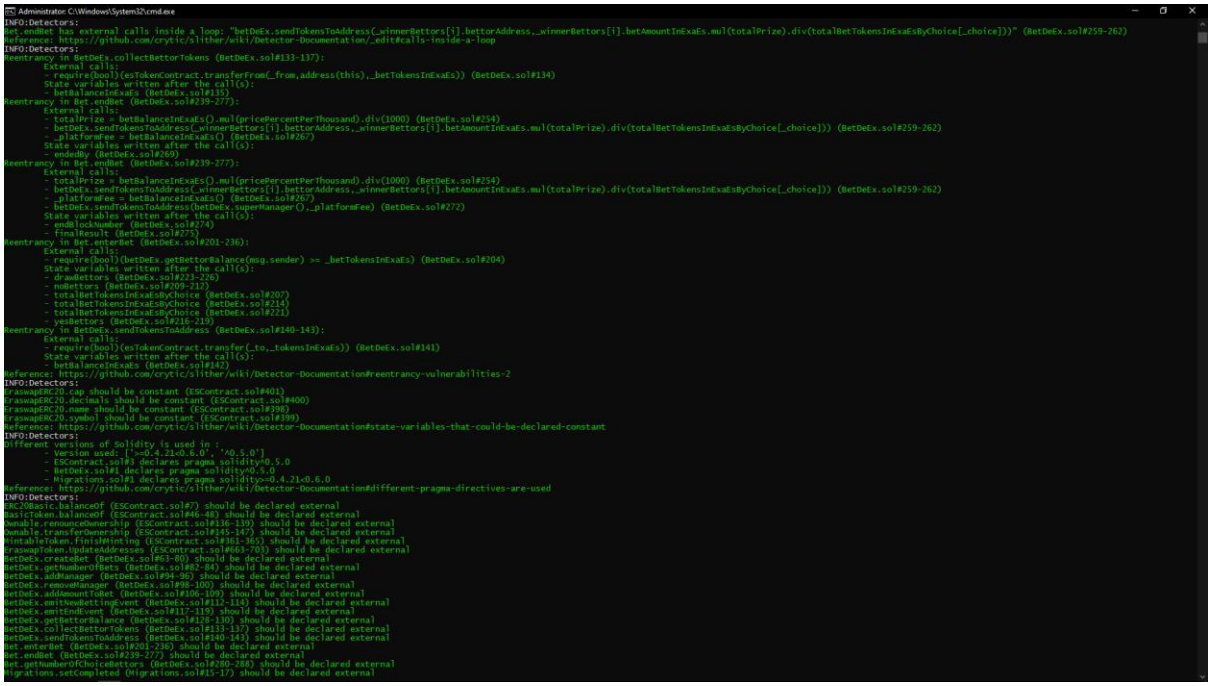
   `25c1dada8a8f89e96fa6e57f226be6b7dbaea36e`

Automatic Auditing Tool Outputs

1. Slither

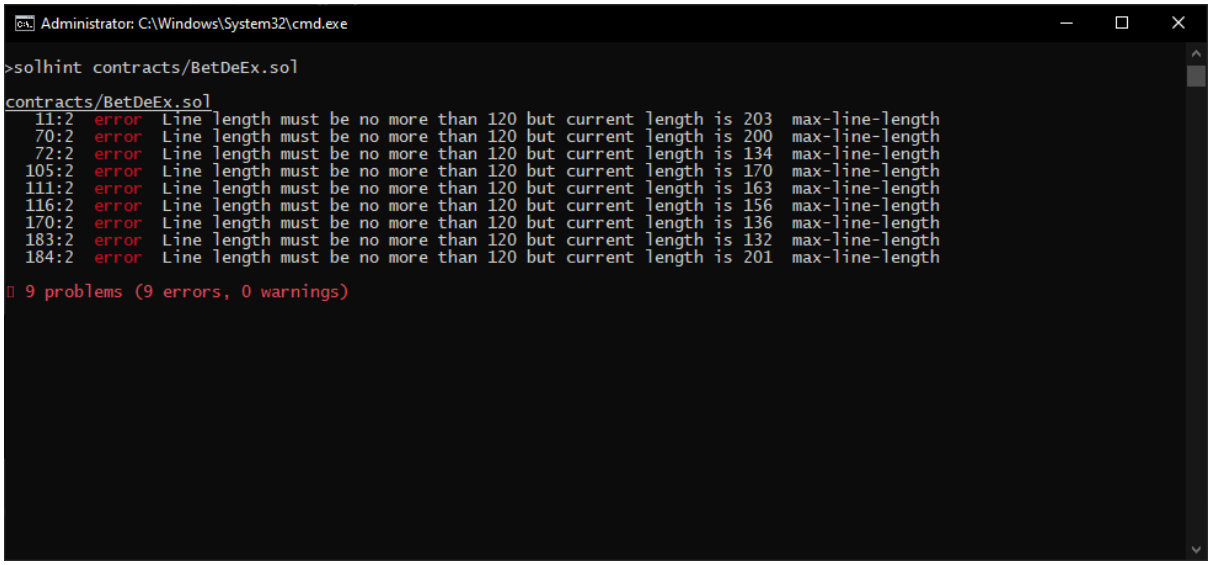
> Slither, is another static analysis tool from Trail of Bits. It includes aids for contract summaries, which can be helpful for making a mental model of the contract and rechecking assumptions.



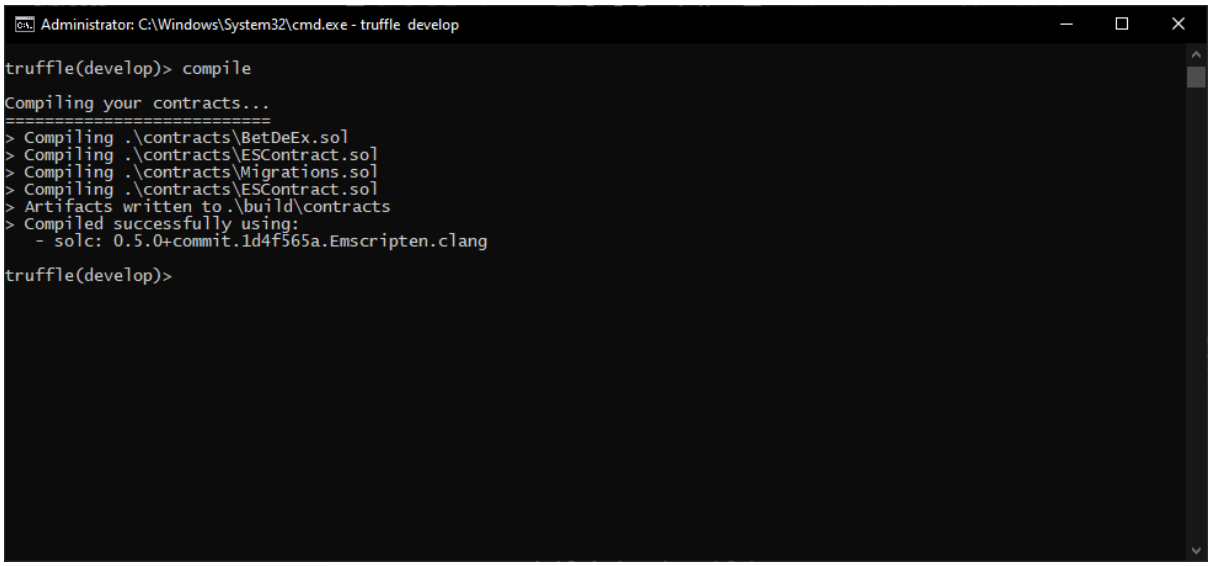


2. Solhint

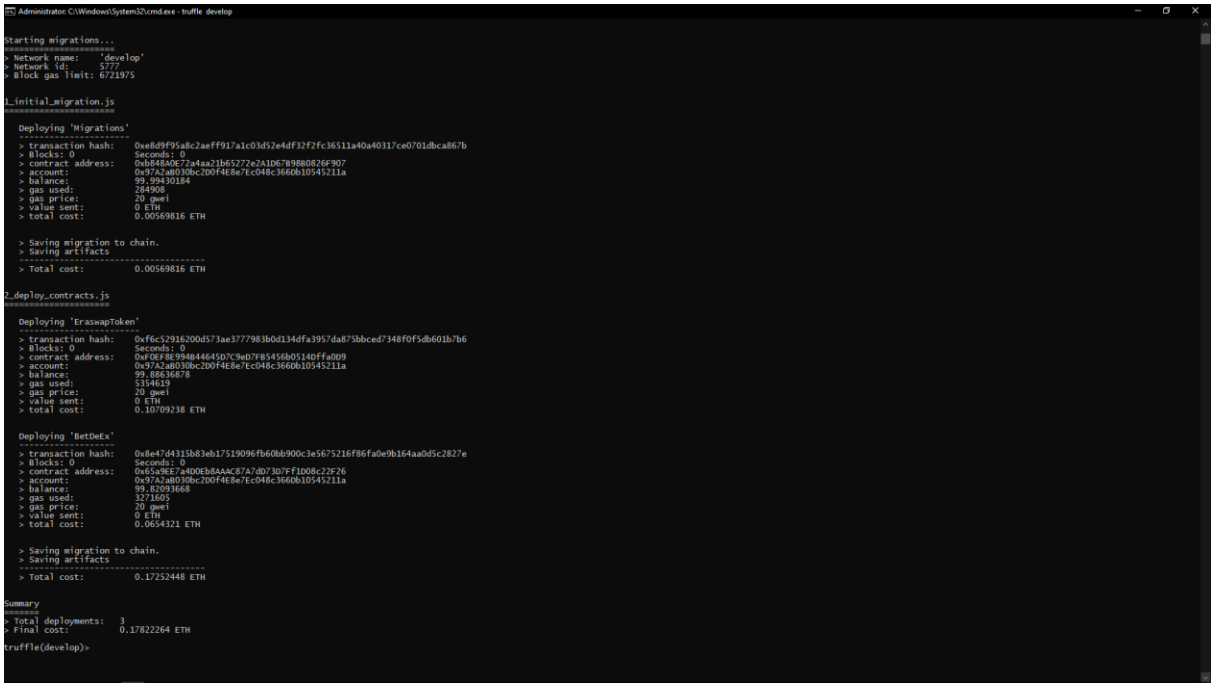
This is an open source tool for linting solidity code. This tool provides both **security** and **style guide** validations.



3. Truffle Compilation Verification



4. Truffle Migration and Gas Estimation



Contract Deployment Testing

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

- **ESContract.sol at** `0xd7bf79aed5d06a7282b02805f15667aa6a1eaaf6`
- **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 implement the best security practices. It is good to use to OpenZeppelin framework wherever required and the contract is using it very efficiently where required. The contract stores the funds safely and transact safely wherever needed.

The contracts are written keeping in mind the best security practices of the 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 has 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.