robert.monks@sentetechnologies.com
www.sentetech.com

EZYTRX

Smart Contract Audit

Prepared by: Sente Technologies LLC

Prepared on: October 24, 2020

robert.monks@sentetechnologies.com
www.sentetech.com

Table of Contents

robert.monks@sentetechnologies.com
www.sentetech.com

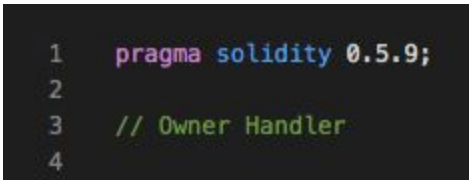
Overview

The contract and source code for EZYTRX on the Tron blockchain can be found at TronScan. The source code for EZYTRX is all contained in one file of Solidity code totaling around 350 lines once formatted. Through thorough examination, it has been determined that there is very minimal risk of vulnerabilities.

Good Aspects

Solidity Version

The first line of the contract states the desired Solidity Version to be used with the smart contract.



Because the language is constantly evolving and new changes are being implemented, targeting a specific release version is usually recommended so that newer versions don't cause breaking changes which this contract does. Additionally, while the newest version is 0.7.3, there is no problem with using earlier versions if they are more stable or if the code was designed for that version.

robert.monks@sentetechnologies.com
www.sentetech.com

Comments

It is unanimously best practice to include comments with your code. They increase readability and provide documentation in the source code of the intended functionality. With that in mind, the EZYTRX does a good job providing ample comments in the code.

```
5   contract ownerShip {    // Auction Contract Owner and OwherShip change
6       //Global storage declaration
7       address payable public ownerWallet;
8       address payable public newOwner;
9
10      //Event defined for ownership transfered
11      event OwnershipTransferredEv(address indexed previousOwner, address indexed newOwner);
12
13      //Sets owner only on first run
14      constructor() public {
15          //Set contract owner
16          ownerWallet = msg.sender;
17      }
```

These comments are extremely helpful and will help everybody who ventures into the EZYTRX source code.

Input and Variable Validation

At the beginning of new functions, especially public facing functions, it is critical to perform validations to ensure that the inputs are allowed and expected so that using them later in the code won't break anything. The EZYTRX contract does a very good job validating variables throughout the entire codebase.

```
146     function buyLevel(uint _level) public payable {
147         require(userInfos[msg.sender].joined, 'User not exist');
148         require(_level > 0 && _level <= 10, 'Incorrect level');
149         if(_level == 1) {
150             require(msg.value == priceOfLevel[1], 'Incorrect Value');
151             userInfos[msg.sender].levelBuyCheck[1] = 1;
152         } else {
153             require(msg.value == priceOfLevel[_level], 'Incorrect Value');
154             for(uint l =_level - 1; l > 0; l--) require(userInfos[msg.sender].levelBuyCheck[l] == 1 , 'Buy the previous level');
155             if(userInfos[msg.sender].levelBuyCheck[_level] == 0) userInfos[msg.sender].levelBuyCheck[_level] = 1;
156             else userInfos[msg.sender].levelBuyCheck[_level] = 1;
157         }
```

robert.monks@sentetechnologies.com
www.sentetech.com

This function to perform the purchase of the next level performs a lot of validation. First, it checks that the user and the level are valid, then it continues making similar validations with the final check confirming that all previous levels have been purchased. Additionally, the error messages from any failed validation provide meaningful explanations.

Attacks and Vulnerabilities

The Solidity documentation provides a list of security considerations when developing smart contracts. These outline some of the most problematic and prevalent edge cases to be cautious of. To ensure there are no bugs, each security concern will be addressed.

Overflow and Underflow

In binary, there is a minimum and maximum value representable by a series of bits. For unsigned integers using n bits, the minimum is 0 and the maximum is $2^n - 1$. For example, an unsigned integer using 8 bits will have a minimum of 0 and a maximum of 255. Whenever mathematical operations would cause the value to be outside of the safe range, a variety of exceptions can occur. In Solidity, adding 1 to the maximum value will wrap the integer back to 0 and subtracting 1 from the minimum wraps to the maximum value, so these edge cases can be breaking in the wrong cases.

In the EZYTRX code, each unsigned integer isn't 8 bits, but 256. That means that the maximum value assignable into these variables is about $1.15 * 10^{77}$. While Solidity recommends using the SafeMath just for these edge cases, there are no places in the contract where the unsigned integer would cause an overflow or underflow and there are validations to ensure this never happens.

Using tx.origin

tx.origin is a value in the smart contract that will be assigned to the origin of the transaction and it is generally recommended not to use tx.origin because it's behavior

can sometimes be the source of bugs. Instead, the recommendation is to use msg.sender for the safety it provides and the bugs that it protects from.

In EZYTRX, tx.origin is never used. The much more recommended msg.sender is used throughout the contract so there should be no problems with tx.origin.

Callstack Depth

Functions in programming languages contain a list of instructions to perform. Many times, one of those instructions is to call another function, or synonymously, to execute the instructions in that function. In Solidity, the maximum depth of this function chain, the callstack, is 1024. If it exceeds the maximum callstack depth, then an exception will be thrown in most cases.

After a thorough review of the EZYTRX source code, there is no recursion or instances where the callstack would increase dramatically. Therefore, there is no reason to believe that the callstack depth limitation should present a problem.

Gas Limit and Loops

Due to the block gas limit, transactions can only consume a certain amount of gas. Therefore, loops that do not have a finite number of iterations can cause the complete contract to be stalled at a certain point. In other words, loops that don't have an upper limit of iterations or can potentially run indefinitely can churn through the available gas and stall the contract.

EZYTRX contains eight for-loops and zero while-loops. In each of these eight for-loops, the maximum number of iterations is clearly defined as part of the loop declaration. Therefore, it is clear that EZYTRX does not contain any indefinitely running for loops or for loops with extremely large numbers of iterations.

Re-Entrancy

As explained by the Solidity documentation, "Any interaction from a contract (A) with another contract (B) and any transfer of Ether hands over control to that contract (B). This makes it possible for B to call back into A before this interaction is completed." While the currency is different in EZYTRX than the quote calls out, the ideas remain the same.

In short, re-entrancy allows other contracts to execute their code and potentially perform malicious activity if not guarded against.

To avoid re-entrancy, the Solidity documentation advises using a pattern called Checks-Effects-Interactions. This pattern will ensure that re-entrancy cannot be performed. After thorough investigation of the EZYTRX abides by this pattern and protects against re-entrancy.

Sending and Receiving Tron

The Solidity documentation recommends using a “withdraw” pattern instead of a “send” pattern, even though the “send” pattern is most intuitive. This is because of three potential pitfalls when using the straightforward transfer function. The first is that if the recipient is a contract, the recipient’s fallback function will be called which can call back into the original contract. This is similar to the re-entrancy loophole. The second is that the callstack can reach over the limit of 1024. Finally, the third is that the transfer can fail if it runs out of gas which can block progress in the sending contract if there is a check of successful transfer.

The EZYTRX contract uses the “send” pattern instead of the “withdraw” so the three pitfalls need to be accounted for. Re-entrancy and callstack depth were mentioned earlier and were determined not to be issues with this contract. Finally, out of gas exceptions will not block progress of EZYTRX and so the contract abides by all of the best practices when using the “send” paradigm.

Summary

After reviewing the Solidity documentation regarding security considerations and checking that against the source code for EZYTRX, it has been determined that the risks are minimal or nonexistent for each category. There is room for improvement regarding using the SafeMath library or using the “withdraw” pattern instead of the “send” pattern for transfering funds, but NO security vulnerabilities were found and the contract should function as intended.

robert.monks@sentetechnologies.com
www.sentetech.com

Conclusion

Despite multiple forms of attacks and edge case testing performed, the contract showed no signs of reacting negatively or breaking. EZYTRX's smart contract is resilient to a variety of tests and provides robust validation of inputs to ensure smooth operation. Additionally, the code is secure by following Solidity security best practices. Overall, the code is resilient, robust, secure, and performant.

robert.monks@sentetechnologies.com
www.sentetech.com

SMART CONTRACT AUDIT CERTIFICATE OF EZYTRX.IO

Sente Technologies has researched the EZYTRX.IO smart contract.

The SMART CONTRACT was deployed to Tron Mainnet:
https://tronscan.org/#/contract/TGZfB3J9NvHJ3BXnPA19FzBT2uNsrgenKw/code

SUMMARY OF THE AUDIT

Comments:

1. There is a method in the code:

```
function() payable external {
  // function code here
}
```

After funding their TRX wallet, users will be able to interact with the contract to purchase levels.

2. When the function is called, the balance is checked and the transaction will immediately and directly be sent to the correct users' TRX Wallet Addresses.

3. There are no administration commissions in the contract.

4. Best practices regarding the use of assert, require etc., is already being done by EZYTRX.IO

5. There were no serious or critical security vulnerabilities found in EZYTRX.IO

IN THE FINAL CONTRACT WAS NOT FOUND

- Backdoors for investor funds withdrawal by anyone
- Bug allowing to steal money from the contract