Smart Contract Code Smell Summarization

Demographic && Coding Proficiency

This survey has two pages. In this page, we will ask 7 questions which are related with Demographic and Coding Proficiency

1. Are you a professional smart contract developer?

- ◯ Yes
- ◯ No

2. Are you involved in open source software development efforts?

- ◯ Yes
- ◯ No

3. Please describe your main role in developing smart contract

- ◯ Testing
- ◯ Development
- ◯ Management
- ◯ Other (please specify)

4. How many years of experience do you have in smart contract development/testing/project management(decimals ok)

5. What is your current country of residence

6. What is your highest educational qualification?

- ◯ Less than high school
- ◯ Graduated high school
- ◯ Trade/technical school
- ◯ Some college, no degree
- ◯ Other (please specify)
- ◯ Associate degree
- ◯ Bachelor's degree
- ◯ Advanced degree(Master's, Ph.d., M.D.)

7. Professional Skills
Please rate the level of importance of these factors to determine one's coding proficiency:

	Very Unimportant	Unimportant	Neutral	Important	Very Important	Prefer not to answer/ I don't understand
Able to write smart contracts efficiently(i.e., clear coding task in a short amount of time)	◯	◯	◯	◯	◯	◯
Able to write efficient smart contracts, e.g., the contracts can run very fast, use less memory, use less gas, etc.	◯	◯	◯	◯	◯	◯
Able to implement functionality avoiding design anti-patterns(e.g., god class, brain class, feature envy, etc.).	◯	◯	◯	◯	◯	◯
Able to refactor code by identifying and eliminating code and architecture smells.	◯	◯	◯	◯	◯	◯
Able to reuse code created internally rather than reinventing the wheel.	◯	◯	◯	◯	◯	◯
Able to master several blockchain infrastructure(e.g., Bitcoin, Ethereum, etc.)	◯	◯	◯	◯	◯	◯

Smart Contract Code Smell Summarization

Read Me: Code smells are bad patterns in the source code that indicates deeper problems. The detection of code smell is a typical method to avoid potential bugs and improve the design of existing code. In this page, we summarize 20 kinds of code smells for smart contracts. We *assume that a smart contract contains these code smells.* So whether removing these code smells can improve the quantity of code design, safety or readability?

8. **Unchecked External Call:** To transfer Ethers or call functions of other smart contracts, Solidity provides a series of external call functions for raw addresses, i.e., *address.send(), address.call(), address.delegatecall()*. Unfortunately, these methods may be failed due to the network errors or out-of-gas error. When errors happen, these methods will return a Boolean value (False), but never throw an exception. If the callers do not check the return values of the external calls, they cannot ensure whether the logic of the following code snippets are correct.

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

9. **DoS under External Influence:** When an exception is detected, the smart contract will rollback the transaction. In other words, the function will not be executed because of the exception. Therefore, if the errors which lead to the exceptions cannot be fixed, the function will denial of service forever.

For example, *members* is an array which stores many addresses. However, one of these addresses is an attacker contract and the *transfer* function can trigger an out-of-gas exception due to the 2300 gas limitation. Then, the contract state will rollback. Since the code cannot be modified, the contract can not remove the attack address from members list, which means that if the attacker does not stop attacking, the following function cannot work anymore.

```
for(var i = 0;i < members.length; i++){
   if(this.balance > 0.1 ether)
      members[i].transfer(0.1 ether);
}
```

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

10. **Strict Balance Equality:** Attackers can send Ethers to any contracts forcibly by utilizing selfdestruct*()*. This method will not trigger fallback function, which means the victim contract cannot reject the Ethers. Therefore, the equations will fail to work due to the unexpected ethers send by attackers.

For example, the *doingSomething()* function can only be triggered when the balance strict equal to 10 eth. However, the attacker can send 1 wei to the contract to make the balance never equal to 10 eth.

```
function Demo(){
   if(this.balance == 10 eth)
      doingSomething();
}
```

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

11. **Unmatched type assignment:** Solidity supports different types of integer (e.g., *uint8, uint16, uint256*) and type deduction var. The default type of integer is uint256 which supports a range from 0 to 2^256. uint8 takes little memory, but only supports from 0 to 2^8. Solidity will not throw an exception when the value exceeds its maximum value. A progressive increase is a common operation in programming, without checking the maximum value may lead to overflow.

For example, *for(var i = 0; i < member.length;i++)*, the variable *i* is assigned to uint8. If the members.length is larger than 255, the value of i after 255 is 0. Thus, the loop will not stop until running out of the gas

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

12. **Transaction State Dependency:** Contracts need to check whether the caller has the permission in some permission sensitive function. The failure of the permission check can cause serious consequence. tx.origin can get the original address that kicked off the transaction, but this method is not reliable since the address returned by this method depends on transaction state. Therefore, do not use tx.origin to check whether the caller has permission to execute functions.

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

13. **Block Info Dependency:** Ethereum provides a set of APIs(e.g. block.blockhash, block.timestamp) to help smart contracts obtain block information. Many contracts use these block information to execute some operations(i.e. generate a random number). However, miners can influence block information, for example, miners can vary block time stamp by roughly 900 seconds, while still having other miners accept the block. In other words, block info dependency operation can be controlled by miners to some extent.

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

14. **Re-entrancy:** Concurrency is an important feature for traditional software. However, Solidity does not support it and functions of a smart contract can be interrupted while it's running. Solidity allows parallel external invocations by using call method. If the callee contract does not correctly manage the global state, the callee contract will be attacked, which is called the re-entrancy attack.

The following code shows an example of re-entrancy. The attacker contract invokes Victim contract's withDraw() function. However, Victim contract sends Ethers to attacker contract before resetting the balance. *msg.sender.call.value(amount)()* will invoke the fallback function of attacker contract and lead to repeated invocation.

```
contract Victim {
   mapping(address => uint) public userBalannce;
   ...
   function withDraw(){
      uint amount = userBalannce[msg.sender];
      if(amount > 0){
         msg.sender.call.value(amount)();
         userBalannce[msg.sender] = 0;
      }
   }
   ...
}
contract Attacker{
   ...
   function() payable{
       Victim(msg.sender).withDraw();
   }
   function reentrancy(address addr){
      Victim(addr).withDraw();
   }
   ...
}
```

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

15. **Hard Code Address:** Since we cannot modify smart contracts after deploying them to the blockchain, hard code address can lead to vulnerability in some situation.

There are two main kinds of errors that this code smell can lead to. The first one is illegal address. Ethereum uses the mixed-case address checksum to verify whether an address is legal or not. The rule is defined in EIP-55. Some addresses may not legal. The second on is Suicide Address. selfdestruct function can remove code from the blockchain, but it is potentially dangerous, as if someone sends Ether to removed contracts, the Ether will forever lose.

For example, addr is a contract address who performed selfdestruct function before. Since the smart contract cannot be modified, the following function cannot be used anymore.

```
function withDraw(uint amount){
   address addr = 0xfff.fff;
   addr.call.value(0xfff...ffff);
}
```

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

16. **Nest Call:** Instruction *CALL* is very expensive (9000 gas paid for a non-zero value transfer as part of the *CALL* operation). If a loop body contains CALL operation, the total gas cost would have a high probability to exceed the gas limitation.

For example, if we do not limit the length of *member[]*, attackers can increase its size to cause out-of-gas error.

```
for(uint i = 0; i < member.length; i++){
   member[i].send(1 wei);
}
```

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

17. **Deprecated APIs**: Some instruction will be modified or discarded after a hard fork. Besides, Solidity is a young and rapid iterative programming language, some APIs/instructions will be discarded or updated in the future versions. Do not use these deprecated APIs since these APIs are not beneficial for code reuse.

For example, CALLCODE operation will be discarded in the future. *throw, suicide, sha3* are replaced by revert, selfdestruct, *keccak256* respectively in the recent version. The deprecated APIs are not beneficial for code reuse.

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

18. **Unfixed Compiler Version:** Different versions of Solidity may contain different APIs/instructions. In Solidity programming, indicating the compiler version is benificial for code reuse.

For example, *pragma solidity ^0.4.25* means that this contract supports compile version 0.4.25 and above (except for v0.5.0) while *pragma solidity 0.4.25* means that the contract only supports compile version 0.4.25. Since it is hard to foresee the language constructions in the future version, it is recommended to indicate a specific compile version to avoid unnecessary bugs. respectively in the recent version. The deprecated APIs are not beneficial for code reuse.

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

19. **Misleading Data Location:** In traditional programming languages like Java or C, variables created inside a function are local variables. The data of these variables are stored in memory and the memory will be released after function finished. But in Solidity, the data of struct, mapping, arrays are stored in storage even they are created inside a function. However, since storage in solidity is not dynamically allocated, storage variables created inside a function will point to the storage slot0 by default.

For example, Function *reAssignArray* creates a local variable tmp. The default data location of tmp is *storage*, but EVM cannot allocate *storage* dynamically. There is no space for tmp, but instead, it will point to the storage slot 0 (*uint variable*). For the result, once function *reAssignArray* is called, the *variable* will add 1, which can cause bugs for the contract.

```
contract Demo{
   uint variable;
   uint[] investList;
   function reAssignArray(){
      uint[] tmp;
      tmp.push(0);
      investList = tmp;
   }
}
```

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

20. **Unused Statement:** If function parameters or local variables do not affect any contract statements nor return value, it is better to remove these code snippets in order to improve code readability.

For example, *uint newValue* is useless. So, remove this variable to increase code readability

```
contract Demo{
   bool state = false;
   function changeState(bool newState, uint value){
      uint newValue = value;
      state = newState;
   }
}
```

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

21. **Unmatched ERC-20 standard:** ERC-20TokenStandard is a technical standard on Ethereum for implementing tokens of cryptocurrencies. It defines a common list of rules for Ethereum tokens to follow within the larger Ethereum ecosystem, allowing developers to accurately predict interaction between tokens.
ERC20 defines 9 different functions and 2 events to ensure the tokens based on ERC20 can easily be exchanged with other ERC20 tokens. However, we find that many smart contracts miss return values or miss some functions.

For example, *transfer* and *transferFrom* are two functions defined by ERC20. They are used to transfer tokens from one account to another. ERC20 defines that these two functions have to return a boolean value, but many smart contracts miss the return value, which may lead to vulnerability when transfer tokens.

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

22. **Missing Return statement:** There are some functions who denote the type of return value but do not return anything. For these functions, EVM will add a default return value when compiling the code to ByteCode. Since the callers may not know the source code of the callee contract, they may use the return value to handle the code execution and lead to unpredictable bugs.

For example, the following function declares the return type bool, but the function does not return true or false. Then, EVM will assign a default return value as false. If developers call this function, the return value will always be the false and some functions in the caller contracts may never be executed.

```
function test(address addr) returns(bool){
   addr.transfer(0.1 ether);
}
```

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

23. **Missing Interrupter:** Bugs are hard to inevitable. When bugs are found by attackers, the attackers can attack the contracts and steal their Ethers. We can not modify contracts after deploying them to the blockchain, but if the contracts contain breaker or other backdoor mechanisms, the owner of the victim contract can reduce their losses. The easiest breaker is adding a selfdestruct function, Ethers on the contracts can be withdrawn and the contracts will be destroyed.

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

24. **Missing Reminder:** Smart contracts can be called by other programs through the Contract Application Binary Interface (ABI). However, developers can only call the function through ABI rather than know the detail information about the function. Add a reminder or throw an event to remind caller whether the function is successfully executed can reduce unnecessary errors Bugs are hard to inevitable. When bugs are found by attackers, the attackers can attack the contracts and steal their Ethers. We can not modify contracts after deploying them to the blockchain, but if the contracts contain breaker or other backdoor mechanisms, the owner of the victim contract can reduce their losses. The easiest breaker is adding a selfdestruct function, Ethers on the contracts can be withdrawn and the contracts will be destroyed.

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

25. **Greedy Contract:** A contract can withdraw Ethers by sending Ethers to other address or using selfdesturct function. Without these withdraw-related functions, Ethers in the contracts can never be withdrawn and the Ethers will be locked forever. We define a contract is a greedy contract if the contract can receive ethers(contains payable fallback function) but there is no way to withdraw the Ethers.

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

26. **High Gas Consumption Function Type:** For public functions, Solidity immediately copies function arguments(Arrays) to memory, while external functions can read directly from EVM stack. Memory allocation is expensive, whereas reading from stack is cheap. To lower gas consumption, if there are no internal functions call this function and the function parameters contain array, it is recommended to use external instead of public to save gas.

For example, function *highGas* and function *lowGas* have the same capabilities. The only difference is that *highGas* is modified by *public* which can be called by external and internal functions. *lowGas* is modified by *external* which can only be called by external. Calling function *highGas* in costs 496 gas while calling lowGas only costs 261 gas.

```
function highGas(uint[20] a) public returns (uint){
   return a[10]*2;
}
function lowGas(uint[20] a) external returns (uint){
   return a[10]*2;
}
```

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

27. **High Gas Consumption Data Type:** *bytes* is dynamically sized byte array, *byte[]* is similar with bytes, but *bytes* cost less gas than *byte[]*, because it is packed tightly in calldata. EVM operates on 32 bytes at each time, *byte[]* always occupy multiples of 32 bytes which means a great number of space is wasted but not for *bytes*. Therefore, *bytes* takes less storage slot and cost less gas. To lower the gas consumption, it is recommended to use *bytes* instead of *byte[].*

- ◯ Very important
- ◯ Important
- ◯ Neutral
- ◯ Unimportant
- ◯ Very Unimportant
- ◯ I don't understand

Do you have any comments or tell us the reason why you choose it? (Optional)

28. Do you have any other comments, questions, or concerns?

29. If you want to enter the raffle, please enter your email, we will give out 50 USD Amazon vouchers to two randomly selected participants

PREV DONE