Sufficient condition to judge bug

Feng Xiao

May 2020

1 introduction

This is an explanatory document accompanying the *A Framework and Data Set for Bugs in Ethereum Smart Contracts* paper. This document describes the sufficient conditions for determining all kinds of bugs in *A Framework and Data Set for Bugs in Ethereum Smart Contracts*. Please refer to *A Framework and Data Set for Bugs in Ethereum Smart Contracts* for the specific situation of these kinds of bugs.

2 Sufficient conditions to determine the existence of various bugs

1. *A-a-ID*: The result of dividing two integers (variables or constants) is a decimal.
2. *A-a-IO*: The result of the integer operation exceeds the boundary value of the saved result variable.
3. *A-a-IS*: Forcibly convert an *int* variable with a negative value to a *uint* variable.
4. *A-a-IT*: Forcibly convert a long *int/uint* variable into a short *int/int* variable that cannot accurately save the original value.
5. *A-a-W*: Use the *Solidity* version before 0.5.0 to write smart contracts and the =+ or =- operator exists in the contract.
6. *A-b-HB*: The name of the developerdefined variable is the same as the built-in symbol of the programming language.
7. *A-b-HS*: The subclass contract and the base class contract have the same named state variable, and the function of the subclass contract will be affected by the hidden state variable in the base class contract.

8. *A-b-I*: When multiple inheritance occurs, two (or more) base-class contracts contain structures that can lead to overwriting each other's variables, functions, and everything that can be overwritten, and the inheritance order of base class contracts close to subclasses (in the inheritance tree) is specified after the base class contracts of remote ion classes.

9. *A-c-UL*: : The developers do not initialize the local/state variables in the contract.

10. *A-c-US*: The developers do not initialize the storage variables in the contract.

11. *B-a-R*: : The source code of the contract (including comments) contains U+202E characters.

12. *C-a-D*: In the source code, there are some operations to delete the array elements by using the *delete-statement*, and there is no statement to move subsequent elements.

13. *C-a-U*: Developers use *Solidity* before version 0.5.0 to write smart contracts, and the following situation exists in the source code: The *continue-statement* and *do-while-statement* are used, and the *continue-statement* exists in the *do-while-statement*, and the *continue-statement* is reachable.

14. *D-a-R*: When there are the following four characteristics in the contract, it will cause the *reentrancy bug*:

    (a) The *call-statement* is used to send ethers.

    (b) The amount of *gas* to be carried is not specified.

    (c) No callee's response function is specified.

    (d) Ethers are transferred first and callee's balance is deduced later.

15. *D-a-U*: The source code (without comments) contains the low-level call statements, and does not receive the return value or receives the return value but does not check the return value.

16. *D-b-F*: The function of the contract is related to whether the balance of the contract is at a certain value.

17. *D-b-L*: At least one function in the contract is declared *payable*, but there is no statement in the contract to transfer out ethers, or the statement to transfer out ethers is unreachable.

18. *D-b-P*: The function of the contract is related to whether the balance of the contract is zero.

19. *E-a-C*: The parameters (address or data) of the *call/delegatecall-statement* are given by the external users and the contract does not check these parameters.

20. *E-a-H*: Use *abi.encodePacked()* to pack the parameters of the hash function, and the parameters contain multiple arrays of the same type.

21. *E-a-SA*: The contract uses a function to transfer ether or tokens, and the amount and payment address are given by external users, and there is no operation to check the length of the payment address in the function.

22. *E-a-SW*: There is an operation in the contract that uses the *ecrecover()* to calculate the public key address, and it does not deal with the case where the *ecrecover* returns 0.

23. *E-b-T*: Developers do not follow the provisions of the token standard (eg., *ERC20*) for variables, functions and events to develop token contracts.

24. *F-a-R*: Use assembly statement return values in the constructor.

25. *F-a-S*: Use the *mstore* instruction of the assembly code and the function variables in the parameters. Or use assignment operations in assembly code to assign values to function variables.

26. *F-b-DBC*: Executing the fallback function of the contract consumes more than 2300 gas.

27. *F-b-DBG*: There is an infinite loop in the contract.

28. *F-b-DBN*: The contract function depends on the return value of the external contract. Or the contract uses the *transfer-statement* in the loop to transfer out ethers to the external address.

29. *F-c-R*: Special variables related to block attributes or mining attributes are used in the contract, and the function of the contract is related to the value of these special variables.

30. *F-c-T*: There is the *approve* function specified in the *ERC20 token standard* in the contract, and in the *approve* function, the quota of the approved address is set from one nonzero value to another nonzero value.

31. *F-d-S*: When modifying storage variables or writing new values to storage, the contract does not authenticate the operator.

32. *G-a-B*: Use *byte[]* type variables in the contract.

33. *G-a-II*: There are some invariants in the loop (including the condition judgment part).

34. *G-a-IS*: There are constants in the state variable that will not be modified, but these constants are not declared as *constant*.

35. *G-a-U*: There are *public* functions in the contract that are not called by other contract functions.

36. *H-a-R*: Signature verification is performed in the contract, but the contract does not store the signature value that has been processed.

37. *H-a-S*: There is a *selfdestruct* instruction in the contract, and there is a lack of authority control over the *selfdestruct* instruction.

38. *H-a-U*: The contract uses *tx.origin* for authentication (eg., *tx.origin == owner*), but this statement (*tx.origin == msg.sender*) does not belong to this type of situation.

39. *H-a-WC*: Users can withdraw ethers from the contract, and the contract does not perform identity verification or balance check on this operation.

40. *H-a-WCN*: There is no function declared as a *constructor* in the contract, but there is a function in the contract whose name is similar to the contract name.

41. *H-b-N*: State variables whose visibility is *private* or *internal* in the contract can be accessed by functions declared as *public* or *external*.

42. *H-b-P*: There are state variables with *private* visibility in the contract.

43. *I-a-I*: There are state variables or functions in the contract that do not explicitly indicate visibility.

44. *I-a-N*: The naming of contracts, libraries, structures, events, local/state variables, functions, function parameters, modifiers, enumerations, and constants does not comply with the naming conventions specified in the *Solidity* documentation `https://solidity.readthedocs.io/en/v0.6.2/style-gu\ide.html#naming-conventions`.

45. *I-a-T*: In the source code (without comments), five (or more) digits appear consecutively.

46. *I-a-UC*: The version pragma statement contains the $\wedge$ symbol. Or the version pragma statement contains the $>=$ symbol but not the $<$ symbol.

47. *I-a-UD*: Developers use *Solidity* after version 0.5.0 to write smart contracts, and use built-in symbols that have been deprecated.

48. *I-b-F*: There are functions declared as *view* or *constant* in the contract, and there are statements in these functions that can modify the state of the contract.

49. *I-b-I*: The following three situations can cause this kind of bug:

    (a) The *require-statement*, *assert-statement* or *revert-statement* causes the contract to be DOS under any circumstances.

    (b) Use the *require-statement* to handle internal errors or check invariants.

    (c) Use the *assert-statement* to check the external input or return value.