1 | RESEARCH RESULTS

After literature review and research, this paper has sorted out a total of 30 HF chaincode vulnerabilities, which are divided into five categories. This section describes each vulnerability type in detail. Among them, Chaincode Sandbox, Read-write Conflict, Transaction Sequence Dependency, Secure Single Point of Failure and Log Injection vulnerabilities are all vulnerabilities caused by the imperfect HF transaction mechanism. With improvements of HF, these vulnerabilities will be fixed, while cannot be prevented and avoided in chaincode level for the time being.

Language Instruction Non-determinism: Chaincode is executed in distributed and independent peers, therefore, the determinism of chaincode business logic is a major concern [5, 60], otherwise it will lead to inconsistent endorsement results from different nodes, which violates the consensus rules of HF, causing the failure of transaction in the endorsement phase [72]. Therefore, it is necessary to find non-deterministic instructions in chaincode.

(1) Random Number Generation: Random number generation is one of the typical non-deterministic codes. In the endorsement phase, the endorsing nodes simulate and execute chaincode independently in different environments, and it is difficult to ensure that each node has the same result when generating random numbers. Substituting true random numbers with predictable data generation methods also has corresponding risks. A secure random number generator (RNG) should be unpredictable, while the result of calling chaincode must be deterministic. These conflicting properties make implementing a secure RNG in chaincode a challenging problem.

(2) Reified Object Addresses: Developers can manipulate the value of variables through pointers. A pointer is an address in memory, and the address depends on the environment. If the value of a pointer is forcibly modified, the pointer will point to a specific address. Then in different nodes, the chaincode will have different address due to different environments, resulting in different variable values.

(3) System Timestamp: Like random number generation, the timestamp function may be called at different times in each endorsing node. Therefore, it is difficult to ensure that different endorsement nodes generate consistent transaction results.

(4) Global Variable: Global variables are common sources of non-determinism since they can be accessed in any method. Depending on the endorsement strategy, not every node emulates the same transaction, which could make the value of global variables different in every node, leading to non-determinism. In addition, global variables are only global to a single node. If a node fails, global variables may no longer be consistent across all nodes, therefore putting global variables into ledger can lead to inconsistent endorsement results.

(5) Concurrency of Program: Go language provides rich support for program concurrency by using lightweight thread Goroutine. If concurrent programs are not handled properly, race condition problems can easily arise, making the execution order of the chaincode non-determinism.

(6) Map Structure Iteration: Due to the specification of Go language, the order of key-value pairs is not unique when developers use iteration with map structures. Therefore, the use of iterative results may cause non-determinism, which can lead to endorsement failures. Unlike random number generation and system timestamp, this behavior is a hidden implementation detail of Go language.

(7) Field Declaration: When developers implement Chaincode structure, they need to implement the Init method and the Invoke method to satisfy the chaincode interface. When these methods are implemented as methods of a struct, the developer can define the fields of the struct. These fields can be accessed by all methods in the struct. However, since peer nodes do not execute every transaction, these fields will be the same as global variables and cannot be guaranteed to maintain the same value across different nodes.

External Access Non-determinism: Like language instruction non-determinism, external access non-determinism is also related to uncertainty, while it mainly emphasizes the causes come from external access. Because external calls may have different execution logic or indeterminate data sources on different nodes, there is no guarantee that the results will be consistent. When encountering non-deterministic external calls, developers need to clearly understand the calling result.

(1) External File Accessing: External file access does not guarantee that the same command results will be observed across different endorsing peers. Therefore, developers need to be aware of the consequences of chaincode accessing external files in each independent environment.

(2) External Library Calling: Developers usually use third-party libraries to reduce the workload of developing software, however, the potential defects of third-party libraries are difficult to control. Developers need to pay special attention to the behavior of the third-party libraries during using them.

(3) Web Service: In the context of chaincode, when a business logic needs information from outside the blockchain, the developer needs to access the Oracle node, which provides the chaincode with data outside the blockchain. If the Oracle node returns different results to different peer nodes, it will lead to inconsistent endorsement results.

(4) System Command Execution: Go language can execute external commands through the os/exec package. This feature is very useful, while developers need to be very careful when using external commands in the chaincode development environment. External commands cannot guarantee that the same command results will be returned in different endorsement nodes. In addition, the exception of external commands is also difficult to guarantee.

HF Platform: Risks about HF platforms depend on the norms or mechanisms of HF. This type of vulnerability mainly exists in the operation of the HF platform such as the operation of the HF state database and the transaction mechanism.

(1) Range Query Risks: HF provides some range query functions to access the state database, such as GetQueryResult. These functions are executed during the endorsement phase, while not re-executed during the verification phase. This means that the chaincode will not detect phantom reads. A typical case is using CouchDB as state database to perform range queries on assets stored in the ledger. The default access controls provided by CouchDB are weak, and malicious actors on the network can easily access a node's CouchDB instance to directly modify and control the state stored by the node without invoking any ledger transactions. Since there is no change record, it is difficult for a node to know how its state has changed. Therefore, if an attacker modifies the state database of a sufficient number of endorsing nodes to pass consensus, these illegal changes may propagate to the ledger. This means that invalid transactions can be issued in this way.

(2) Read Your Write: In HF, the operation of writing transaction data into the ledger is performed after the transaction is completed and verified. For the write statement to take effect, the transaction must first be submitted. Therefore, the write operation performed on the variable in the chaincode does not be actually written to the ledger, which means the variable value does not exist in the ledger or the old value of the variable exists. If the variable is read in the same process at this time, there is a potential risk of data inconsistency.

(3) Chaincode Sandboxing: HF chaincode is executed in a closed Docker container with absolute permissions of the Docker container. However, closed is not equal to secure, and Docker can be exploited by malicious attackers.

(4) Read-write Conflict: During the endorsement process of a transaction, a read-write set is generated. The read set consists of a unique key obtained from the state database and a list of versions submitted by the transaction. The write set consists of the unique keys written to the state database and the list of values produced after the transaction was committed. At the same time, the version of the key in the write set will also be updated. For a transaction, when the transaction result is stored in the ledger, there is a time interval between the endorsement phase and the submission phase. If the version of the read set key changes within this time interval, the transaction is interrupted. The risk of read and write conflicts may invalidate some transactions, making the outcome of the transaction uncertain.

(5) Transaction Order Dependency: After the transaction is submitted, HF will process the transaction information. Efficient transaction processing can change the ledger state. However, if users submit multiple transactions to the HF network at the same time or within a short time interval, the final order in which the transactions are submitted is indeterminate. On the one hand, it is because of network latency and execution time differences. On the other hand, the wrong ordering node may disrupt the order in which transactions are received. Therefore, the ledger state of the blockchain is also inconsistent. Disordered transaction order can also affect the execution of individual transactions.

(6) Security Single Points of Failure: In blockchain, the nodes participating in the transaction may be faulty. If the system uses a single orderer and compromises, the attacker can either dismiss the transaction or endorse the malicious transaction.

(7) Log Injection: Any corruption of chaincode log messages could prevent the automatic execution of blockchain and allow an attacker to view the processed logs. An attacker can bug the system by injecting malicious logs.

Common Practices: Common practice risk refers to the risk of chaincode crash due to irregular software development during the chaincode development stage.

(1) Unhandled Errors: In Go language, developers can skip indexes or values by assigning them to "_". Even if the value is an error type, the developer can skip it. However, it is easy to ignore the occurrence of errors, and errors will lead to unimaginable problems if they are not properly handled.

(2) Unchecked Input Arguments: This is a common situation in any programming language. An error occurs if the input parameters are not checked and the program allows access to non-existing elements.

(3) Infinite Loops: The execution of chaincodes requires support of environment. Infinite loop will cause problems such as a significant increase in the number of memory addresses. Not only chaincodes will not work properly, but also it will be expensive to run chaincodes.

(4) Uninitialized Storage Pointer: In Go, local struct, array, and map struct type variables are linked to store zero address by default, and uninitialized objects with these types will overwrite anything in zero address. Inside the method, the map struct type variable must be initialized before compilation. Similarly, if array and struct type variables are not initialized nor specified storage location, then it will default to storage pointer.

(5) Write to an Arbitrary Storage Location: Chaincode can store some data, and wrong variable assignment can break it. For example, the index exceeds the length of the array and the chaincode fails.

(6) Using Inherited Functions and Variables: Chaincode can use inheritance in a chaincode language using an object-oriented paradigm. Go language implements multiple inheritance through structures. If several super classes have methods or variables with the same name, their behavior in subclasses depends on inheritance order, which may affect already defined values or methods, causing undesired results.

(7) Using Deprecated Functions: Programming languages are constantly updated and iterative, and in the process, many deprecated functions will appear. The deprecation of functions explains the imperfection of these functions, therefore it is not recommended to use these functions to ensure the normal operation of the chaincode.

(8) Using Built-in Functions: Built-in functions are functions that can be used without introducing a package, however, built-in functions can be easily exploited by attackers.

(9) Comment Headers Insufficient for Checking Implementation and Usage: If the rules of interaction between each method and its caller are documented, then this will reduce the difficulty for a developer to read and understand codes. While it is not a vulnerability that can be exploited by malicious users, it is a security risk if the commenting irregularity reduces the efficiency of the code review process.

Privacy Data Security: Privacy data security risk refers to the risk of transaction failure due to operational permission issues or sensitive data leakage due to lack of security measures when simulating chaincode.

(1) Cross Channel Chaincode Invocation: HF provides cross-channel chaincode calling function. A chaincode can call other chaincodes on the same channel to access or modify the state database, while cannot call other chaincodes on different channels to create new transactions and only obtaining what a method returns. Therefore, when using cross-channel invocation methods, developers should avoid calling chaincode on another channel to create new transactions.

(2) Unencrypted Sensitive Data: If there is unencrypted sensitive data in chaincode, the plaintext of the sensitive transaction data will be stored in the ledger, which can lead to transaction data leakage.

(3) Unused Privacy Data Mechanism: To protect sensitive data, HF also provides a private data mechanism to enhance the security of transaction data. If a chaincode does not use the privacy data mechanism, the security of transaction data may be weakened.