CONTRACT WOLF

Blockchain Security Assessment

MetChain

Verified on 11/09/2023

SUMMARY

Project	CHAIN	METHODOLOGY
MetChain	MetChain	Manual & Automatic Analysis

FILES	DELIVERY	TYPE
Single	11/09/2023	Blockchain Audit

0	5	0	0	0	0	0
Total Findings	Resolved	Critical	Major	Medium	Minor	Informational

🟥 0 Critical	An exposure that can affect the contract functions in several events that can risk and disrupt the contract
🟧 0 Major	An exposure that can affect the outcome when using the contract that can serve as an opening in manipulating the contract in an unwanted manner
🟧 0 Medium	An opening that could affect the outcome in executing the contract in a specific situation
⬜ 0 Minor	An opening but doesn't have an impact on the functionality of the contract
🟦 0 Informational	An opening that consists information but will not risk or affect the contract

STATUS ✔️**AUDIT PASSED**

TABLE OF CONTENTS | MetChain

DISCLAIMER | MetChain

ContractWolf audits and reports should not be considered as a form of project's "Advertisement" and does not cover any interaction and assessment from "Project Contract" to "External Contracts" such as PancakeSwap, UniSwap, SushiSwap or similar.

ContractWolf does not provide any warranty on its released report and should not be used as a decision to invest into audited projects.

ContractWolf provides a transparent report to all its "Clients" and to its "Clients Participants" and will not claim any guarantee of bug-free code within its Client Files.

ContractWolf's presence is to analyze, audit and assess the Client's Files to find any underlying risk and to eliminate any logic and flow errors within its code.

Each company or project should be liable to its security flaws and functionalities.

SCOPE OF WORK | MetChain

MetChain team has agreed and provided us with the files that need to be tested (*Github, BSCscan, Etherscan, Local files etc*). The scope of audit is the main blockchain files.

The goal of this engagement is to identify if there is a possibility of security flaws in the implementation of its blockchain and its systems.

ContractWolf will be focusing on issues and functionalities along with the project claims from its blockchain files to their website, whitepaper, repository which has been provided by **MetChain**.

AUDITING APPROACH | MetChain

Every line of code along with its functionalities will undergo manual review to check for security issues, quality of logic and contract scope of inheritance. The manual review will be done by our team that will document any issues that they discovered.

METHODOLOGY

The auditing process follows a routine series of steps :

1. Code review that includes the following :
   - Review of the specifications, sources and instructions provided to ContractWolf to make sure we understand the size, scope and functionality of the Blockchain.
   - Manual review of code. Our team will have a process of reading the code line-by-line with the intention of identifying potential vulnerabilities, underlying and hidden security flaws.

2. Testing and automated analysis that includes :
   - Testing the function with common test cases and scenarios to ensure that it returns the expected results.

3. Best practices and ethical review. The team will review the contract with the aim to improve efficiency, effectiveness, clarifications, maintainability, security and control within the blockchain.

4. Recommendations to help the project take steps to eliminate or minimize threats and secure the blockchain.

TOKEN DETAILS | MetChain

Metchain aims to provide the next generation metaverse and gaming ecosystem built on Metchains three layered blockchain.

SOURCE

Source	*https://github.com/Metchain/*

FINDINGS | MetChain

0	5	0	0	0	0	0
Total Findings	Resolved	Critical	Major	Medium	Minor	Informational

This report has been prepared to state the issues and vulnerabilities for MetChain through this audit. The goal of this report findings is to identify specifically and fix any underlying issues and errors

Title	File & Line #	Severity	Status
MET Allowance	stake.go	Major	Resolved
Unsecured Modification of Transaction	serverapi.go	Major	Resolved
Potential SQL Injection	amounts.go	Medium	Resolved
Potential SQL Injection	blockchain.go	Medium	Resolved
Redundant statements	verifytransactionsignature.go	Informational	Resolved

FIXES & RECOMMENDATIONS

Security Analysis | amount.go

Security vulnerabilities:

The `CalculateTotalAmount()` function could be vulnerable to SQL injection attacks. The blockchainAddress parameter is passed directly to the database without any sanitization. This could allow an attacker to inject malicious SQL code into the database.

Potential improvements:

The `CalculateTotalAmount()` function could be made more efficient by caching the total amount for each blockchain address.

Recommendation

To fix the SQL injection vulnerability in the `CalculateTotalAmount()` function, you should sanitize the blockchainAddress parameter before passing it to the database. You can use a database library like sqlx to do this.

To make the `CalculateTotalAmount()` function more manageable, you should break it down into smaller functions. For example, you could create a function to get the total amount for a given blockchain address and a function to calculate the total amount for a given block.

Transaction fee address : *c04c08740cf8ce03b8ad842c63d2750c2a0aca23cb36cad623bdda558a736b47*

Security vulnerabilities:

The `Start()` function does not check for errors when calling the `LastMiniBlockRPC()` function. This could allow an attacker to inject malicious code into the database.

Potential improvements:

The `Start()` function could be made more efficient by caching the last block height, hash, and time.

The Blockchain struct could be made more reusable by extracting the common fields into a separate struct.

Recommendations

To fix the security vulnerability in the `Start()` function, you should check for errors when calling the `LastMiniBlockRPC()` function. If there is an error, you should return the error to the caller.

Security Analysis | blockverify.go

No security vulnerabilities found

Specific Code Analysis

ValidChain : The `ValidChain` function efficiently verifies the validity of a chain of **miniblocks**. It iterates through the chain, checking the hash of each block against the previous block's hash, ensuring the chain's integrity.

ConvertBlockToDomainBlock : The `ConvertBlockToDomainBlock` function successfully converts a miniblock from its domain representation to a protocol representation (protobuf message). It extracts the relevant fields from the miniblock and creates the corresponding protobuf message.

ConvertGroupToSecureDomainInfo : The `ConvertGroupToSecureDomainInfo` function assembles a protocol message (*P2PBlockWithTrustedDataResponseMessage*) containing a group of blocks and a nil P2PMessage.

ConvertBlockToDomainGroupBlock : The `ConvertBlockToDomainGroupBlock` function converts a single block from its protocol representation to a domain block representation (protobuf message). It creates a slice containing the block and returns it.

P2PBlockUngroup : The `P2PBlockUngroup` function extracts the first block from a protocol message (P2PBlockWithTrustedDataRequestMessage) containing a group of blocks. It assumes the group contains only one block and returns it.

MatchDomainBlockToP2PBlock : The `MatchDomainBlockToP2PBlock` function compares a domain block with a P2P block, ensuring their consistency. It checks the block height, block hash, previous hash, megablock, metblock, and current hash.

MatchDomainToP2PBytes : The `MatchDomainToP2PBytes` function compares two byte arrays, ensuring they are of equal length and all elements are equal. It returns an error if the arrays don't match.

MatchBlockHeight : The `MatchBlockHeight` function checks whether two block heights are equal. It returns an error if the heights differ.

BlacklistDomain : The `BlacklistDomain` function is a placeholder for blacklisting a domain IP address. Its implementation is not provided.

Recommendations

Error handling : Consider adding error handling to functions that interact with external systems,

such as the `BlacklistDomain` function, to gracefully handle potential errors and provide informative error messages.

Security Analysis | config.go

No security vulnerabilities found

Security Analysis | conflicts.go

No security vulnerabilities found

The provided code defines a method named `ResolveConflicts()` within the `Blockchain` struct. This method aims to resolve conflicts among different blockchain versions maintained by connected nodes. It identifies the longest valid chain among the connected nodes and replaces the local chain if a longer valid chain is found.

While the code's functionality is clear, there are a few potential improvements that could enhance its robustness and efficiency:

1. **Error Handling**

   Implement proper error handling mechanisms to gracefully handle potential issues during the process, such as *network errors* or *invalid JSON* responses. This could involve using `defer` statements to close resources and propagating errors appropriately.

2. **Concurrency**

   Consider using a more concurrent approach to fetching chains from neighboring nodes. Instead of making sequential HTTP requests, consider using goroutines or asynchronous programming techniques to improve performance and responsiveness.

3. **Validation Efficiency**

   Optimize the chain validation process to avoid unnecessary computations. Instead of validating the entire chain for each received chain, consider caching validation results or using incremental validation techniques.

4. **Conflict Resolution Criteria**

   Consider refining the conflict resolution criteria. The current approach simply chooses the longest chain, but it might not always be the most up-to-date chain. Additional factors such as block timestamps or block timestamps could be considered to ensure the most relevant chain is chosen.

5. **Logging Enhancement**

   Improve the logging messages to provide more informative details about the conflict resolution process. For instance, logging the lengths of the compared chains and the reasons for choosing or rejecting a chain could provide valuable insights.

By implementing these improvements, the `ResolveConflicts()` method can become more

robust, efficient, and informative, ensuring that the local blockchain remains consistent with the longest valid chain among connected nodes.

Security Analysis | consensus.go

No security vulnerabilities found

Specific Findings :

- The `GenesisGenrate()` function could be made more efficient by using a struct to store the genesis block data instead of using separate variables. This would make the code more concise and easier to read.
- The `VerifyGenesis()` function could be improved by breaking it down into smaller, more manageable functions. This would make the code easier to follow and debug.
- The `GensisCompile()` function could be made more efficient by using a JSON marshaler instead of manually converting the genesis block data to a JSON string.
- The `MarshalJSON()` method of the Txtransaction struct could be made more concise by using a struct literal instead of a map.
- The `GenesisUncompile()` function could be made more efficient by using a JSON unmarshaler instead of manually parsing the JSON string into a struct.

Recommendations

Use a struct to store the genesis block data in the `GenesisGenrate()` function.

Break down the `VerifyGenesis()` function into smaller functions.

Use a JSON marshaler in the `GensisCompile()` function.

Use a struct literal in the `MarshalJSON()` method of the Txtransaction struct.

Use a JSON unmarshaler in the `GenesisUncompile()` function.

Overall, the code is well-written and easy to understand. The recommendations above are minor suggestions that could improve the code's efficiency and readability.

Security Analysis | deletefile.go

No security vulnerabilities found

Security Analysis | miniblock.go

No security vulnerabilities found

It effectively handles block creation, storage, and retrieval. The use of comments and descriptive variable names makes the code more readable and maintainable. The code is also well-organized, with functions and structs appropriately named and grouped.

Specific Findings

- `CreateMiniBlock()` function efficiently handles block creation, including timestamp validation, difficulty adjustment, and transaction inclusion.
- `BlockToDB()` function effectively stores block data and associated transactions in the database.
- `Hash()` function correctly calculates the block hash using the provided parameters.
- `MarshalJSON()` method properly converts block data into a JSON format.
- `LastMiniBlock()` function retrieves the last block from the database and returns its details.
- `LastMiniBlock()` function retrieves the last block from the database and returns its height, hash, and timestamp.
- `StringTo32Byte()` function converts a string representation of a byte array into a [32]byte array.

Overall, the code is well-written and performs its intended functions correctly. The provided comments and descriptive variable names make the code easy to understand and maintain.

Security Analysis | mining.go

No security vulnerabilities found

Mining fee address : 56b06b1530a4c26dab1fa0476af7edfa87c15edfbc5a3190345833e7bea4134b

The code is well-structured and easy to understand. It effectively handles NFT management, proof-of-work generation, and transaction validation. The use of comments and descriptive variable names makes the code more readable and maintainable. The code is also well-organized, with functions and structs appropriately named and grouped.

Specific Findings

- `GetNFT()` function efficiently retrieves and returns the NFT data structure.
- `ValidProof()` function correctly verifies the proof-of-work for a given block.
- `ProofOfWork()` function efficiently generates a valid proof-of-work for the next block.

Security Analysis | neighbors.go

No security vulnerabilities found

Security Analysis | rpc_messages.go

No security vulnerabilities found

Specific Findings :

The `GetBlockTemplateBC()` function retrieves the latest mini-block using the LastMiniBlock() function.

It constructs a new DomainBlock struct and copies the relevant data from the mini-block.

It sets the `Nonce` field of the domain block to a specific value of *569658475*, indicating that the block is intended for a specific purpose.

Security Analysis | serverapi.go

Potential security vulnerability:

The `AddTransaction` function takes a *s *utils.Signature* parameter, which is a pointer to a *utils.Signature* struct. This means that the caller of the `CreateTransaction` function can modify the utils.Signature struct after the function has been called. This could be used to create a fraudulent transaction.

Suggestion:

To prevent this vulnerability, you should make a copy of the utils.Signature struct before passing it to the `AddTransaction` function.

In addition to these issues, we also have some suggestions for improvement:

The `CreateTransaction` function returns a boolean value indicating whether the transaction was successful. This is useful for error handling, but it would also be helpful to return the transaction itself.

Suggestion: You should modify the `CreateTransaction` function to return both the boolean value and the transaction itself.

The `CreateTransaction` function takes a `recipient` parameter of type string. This is not a good practice, as it means that the recipient can only be a string.

Suggestion:

You should modify the `CreateTransaction` function to take a recipient parameter of type interface{}. This will allow the recipient to be any type of value.

Security Analysis | stake.go

Potential security vulnerability

The `StakeNFTMet` function does not verify that the *sender* has enough MET to stake. This could allow a user to stake more MET than they have, which could lead to a variety of problems, such as the blockchain becoming unstable.

Suggestion

You should add a check to the `StakeNFTMet` function to verify that the sender has enough MET to stake. This could be done by querying the sender's balance and comparing it to the amount of MET they are trying to stake.

Overview

`StakeNFTMet` function

- The function does not verify that the sender has enough MET to stake.
- The function does not verify that the NFT is valid.
- The function does not check for errors when adding the staking transaction to the pool.

`StakedNFT` function

- The function is very long and complex.
- The function does not handle errors properly.
- The function does not use the correct type for the UnlockTime and LockTime fields.

`CheckNFTRewards` function:

- The function does not handle errors properly.
- The function does not use the correct type for the UnlockTime and LockTime fields.
- The function does not calculate the correct reward for staked NFTs.

`calculatereward` function:

- The function does not handle errors properly.
- The function does not use the correct type for the lt parameter.

`ClearStake` function

- The function does not handle errors properly.
- The function does not use the correct type for the ls parameter.

Security Analysis | sync.go

No security vulnerabilities found

The provided code defines three functions related to node synchronization in a blockchain :

1. `SyncNodes`: This function locks the bc.muxNeighbors mutex, updates the list of neighboring nodes using the SetNeighbors function, and then unlocks the mutex.

2. `StartSyncNodes`: This function calls the SyncNodes function and then schedules itself to be called again after a specified interval (BLOCKCHAIN_NEIGHBOR_SYNC_TIME_SEC seconds).

3. `Run`: This function simply calls the StartSyncNodes function, which effectively initiates the periodic node synchronization process.

The `SyncNodes` function ensures thread-safe access to the *bc.muxNeighbors mutex*, preventing conflicts while updating the neighbor list. The StartSyncNodes function schedules the synchronization process to run periodically, maintaining an up-to-date list of neighboring nodes.

Security Analysis | testblock.go

No security vulnerabilities found

The code defines two functions related to the creation and deserialization of MBlock objects :

1. `NewLastMiniBlock`: This function retrieves the last block from the Metchain domain using the `mc.LastBlock()` function and creates a new MBlock object using the retrieved data. The `n.UnmarshalJSON(lbv)` call deserializes the block data into the newly created MBlock object.

2. `UnmarshalJSON`: This function takes a JSON-encoded byte array as input and deserializes it into an MBlock object. It uses a temporary struct v to hold the deserialized values and then copies those values into the *MBlock* object b.

It effectively handles the creation and deserialization of MBlock objects, ensuring that the objects are populated with the correct data from both JSON and Metchain domain sources.

Security Analysis | transactionpool.go

No security vulnerabilities found

The code defines two functions related to the management of transaction pools:

1. `TransactionPool`: This function simply returns the current transaction pool maintained by the Blockchain object.

2. `ClearTransactionPool`: This function clears the transaction pool and the NFT pool by slicing them to length zero. The [:0] slice operation effectively empties the respective pool arrays.

Security Analysis | transactionrequest.go

No security vulnerabilities found

TransactionRequest Struct

The `TransactionRequest` struct represents a transaction request and holds the following fields:

`SenderBlockchainAddress`: The sender's blockchain address
`RecipientBlockchainAddress`: The recipient's blockchain address
`SenderPublicKey`: The sender's public key
`Value`: The transaction value
`Signature`: The transaction signature

Validate Method

The Validate method checks whether the required fields of the TransactionRequest struct are present and non-nil. It returns true if all required fields are valid and false otherwise.

Here's a breakdown of the Validate method's logic as it checks if :

- `SenderPublicKey` field is not nil.
- `SenderBlockchainAddress` field is not nil.
- `Value` field is not nil.
- `Signature` field is not nil.
- If all four checks pass, it returns ***true***; otherwise, it returns ***false***.
- The Validate method ensures that the transaction request contains the necessary information for *processing* and *validation*.

Security Analysis | unmarshal.go

No security vulnerabilities found

UnmarshalJSON for MiniBlock

The `UnmarshalJSON` method for the `MiniBlock` struct takes a JSON-encoded byte array (`data`) as input and deserializes it into the `MiniBlock` object (b). It utilizes a temporary struct **v** to hold the deserialized values and then copies the **bits** field from **v** to **b**. The json.Unmarshal function handles the actual deserialization process.

UnmarshalJSON for Transaction

The `UnmarshalJSON` method for the Transaction struct takes a JSON-encoded byte array (`data`) as input and deserializes it into the Transaction object (t). It utilizes a temporary struct v to hold the deserialized values and then copies the *senderBlockchainAddress*, *recipientBlockchainAddress*, *value*, *txhash*, and timestamp fields from **v** to **t**. The json.Unmarshal function handles the actual deserialization process.

Security Analysis | verifytransactionsignature.go

Potential Issues:

Duplicated return statement : The function `VerifyTransactionSignature` has two identical return statements, which is redundant and unnecessary. Remove one of the return statements.

Unnecessary os.Exit statement : The ***os.Exit(555)*** statement is not needed and should be removed. If there is an error during the transaction verification process, it's more appropriate to log the error and continue execution.

Recommendations

Add error handling : Currently, the function doesn't handle errors properly. If there's an error during the JSON marshaling or signature verification process, it's essential to handle it appropriately. For example, you could return an error message or log the error and continue execution.

Consider using a dedicated signature verification function : Instead of directly calling the ecdsa & `Verify` function, consider creating a separate signature verification function that takes the transaction data, the sender's public key, and the signature as parameters. This would improve code modularity and make the function more reusable with no underlying threats.

Security Analysis | wallet.go

No security vulnerabilities found

Specific Code Analysis

AddWallet : The `AddWallet` function successfully adds a new wallet to the Wallets slice. It logs the wallet address to the console and returns true.

WalletCreated struct and MarshalJSON method: The WalletCreated struct defines the fields associated with a created wallet. The **MarshalJSON** method correctly marshals the struct into a JSON-encoded byte array, including the wallet address, block hash, and lock hash.

WalletToDB : The `WalletToDB` function efficiently stores wallet information in a *leveldb* database. It generates a unique key for each wallet, calculates the lock hash, marshals the wallet data into a *protobuf message*, and writes the data to the database. It also updates the last wallet information and clears the Wallets slice.

LastWallet : The `LastWallet` function retrieves the last wallet information from the *leveldb* database. It iterates over the wallet keys in reverse order, retrieves the value for the last wallet key, and returns the key and value.

UpdateWalletList : The UpdateWalletList function updates wallet balances and NFT holdings based on transactions in a mini-block. It iterates over the transactions, identifies the sender and receiver addresses, retrieves the sender's and receiver's wallet information, and updates their balances and NFT holdings based on the transaction type.

VerifyAddressPrefix : The VerifyAddressPrefix function ensures that wallet addresses have the correct prefix, adding it if necessary.

AddBig & SubBig : The AddBig and SubBig functions perform addition and subtraction operations on large numerical values using the big.Float library.

Containsint : The containsint function checks whether a specific integer value exists in a slice of integers.

AUDIT COMMENTS | MetChain

- Hotfix commit ID : *65bb621042243ec11e3a1be94aff4eb8f4607834* (nov 26, 2023)
- This **Blockchain Audit** is exclusively for the project **Metchain**
- Metchain may or may not follow the recommendations with the findings.
- The files from */blockchain* folder over the repository may be updated upon deployment and ContractWolf holds no warranty over the changes that will be made in the future.

CONTRACTWOLF

Blockchain Security - Smart Contract Audits