SecurityInnovation®
THE SOFTWARE SECURITY COMPANY

Metacash
Metacash V2
Smart Contract Audit

Document Name:	Smart Contract Audit
Date:	May 17, 2019
Customer Contact:	Nour Haridy <nour@lamarkaz.com>
Author:	Mick Ayzenberg <mayzenberg@securityinnovation.com> Ben Stewart <bstewart@securityinnovation.com>
Project Manager:	Garrett Jaynes <gjaynes@securityinnovation.com>

SecurityInnovation®
THE SOFTWARE SECURITY COMPANY
BOSTON | SEATTLE | PUNE
187 Ballardvale St., Suite A195 ● Wilmington, MA 01887 ● Ph: +1.978.694.1008
getsecure@securityinnovation.com ● www.securityinnovation.com

Introduction

Security Innovation performed a smart contract audit of Metacash V2 on behalf of Metacash. Security Innovation conducted this review from the following commit over the course of two engineering weeks.

https://github.com/Lamarkaz/Metacash-contracts/commit/9032fef98c662d4ff108c29c06df490b5d0027a9

This report summarizes the issues that were uncovered.

Metacash V2 is the second release of Metacash which is a gasless, non-custodial mobile DAI wallet that allows you to send DAI or other ERC20 tokens and pay transaction fees in that token instead of Ether. This is accomplished through an alternative wallet architecture combining meta transactions, smart contract wallets, and the new CREATE2 opcode released as part of Ethereum Constantinople upgrade.

The first version is fully-functional on the Ethereum mainnet and the client is available as an Android application. The second release that was reviewed as part of this report, adds functionality to the smart wallets including the ability to execute call, delegatecall, create, and create2 operations directly from the wallet. Additionally, inline documentation saw a sizeable update.

Testing focused on the entire suite of contracts composing Metacash V2 (Factory, Ownable, Proxy, RelayRegistry, SmartWallet) and review was largely a manual effort with usage of a handful of automated tools providing static or security analysis. Conversations were held with the development team around functionality, applicable use-cases, and future direction.

Testing was time-boxed to two engineers, five days each.

Overall, there were few major security concerns identified in Metacash V2. There were a couple of high severity issues identified that could lead to locked or stolen funds for a targeted user if not remediated. In addition, numerous lower risk issues were identified and outlined in further detail in the Problem Reports section. Also, a few minor observations were made that can impact the overall security of the system.

It is recommended that the development team study these issues carefully and verify if any of the findings have the potential to be more severe. The author(s) of this report retain no responsibility for any unidentified vulnerabilities, known or unknown, in the target application.

Major observations are as follows:

- A total of 10 security issues were identified:
  - PR1 - Fund Theft Through Signature Phishing
  - PR2 - Fund Theft Through Transaction Collisions
  - PR3 - Double Spend After Relayer Message Withholding
  - PR4 - Non-Compliant Tokens are Not Supported
  - PR5 - Failure to Validate Return Value of ecrecover
  - PR6 - Floating Pragma Version
  - PR7 - Functions Not Declared Payable
  - PR8 - Re-Entrancy Within execCall Function

    - PR9 - User-Activated Self-Destruct Enables Replay Attacks
    - PR10 - Potential to Upgrade Away Fees
- If the issues outlined in this report are not remediated, a malicious DApp or Relay may be able to steal or lock funds of a targeted Metacash user.

Contract Description

Metacash is comprised of 5 individual contracts:

Factory

This contract is deployed once and is responsible for creating individual instances of Metacash for end-users. It uses the create2 opcode so that the Metacash addresses are deterministic and users can preload their wallet with funds before deployment.

External functions are either designated as manual or relayer-only. Manual functions are initiated by the wallet owner, whereas relayer functions are executed by a select group of addresses that submit signed messages from the owner. In exchange for submitting these messages and paying the gas fee, these relayers collect a fee in a token designated by the wallet owner.

Factory defines the following functions:

deployCreate2(address owner)

- Internal function that deploys a Metacash instance for a user

deployWallet(uint fee, address token, uint8 v, bytes32 r, bytes32 s)

- Relayers submit a signed message to deploy a wallet for a user

deployWallet(uint fee, address token, address to, uint value, uint8 v, bytes32 r, bytes32 s)

- Relayers submit a signed message to deploy wallet and make an initial token payment

deployWallet()

- Manual function for deploying a new wallet

deployWallet(address token, address to, uint value)

- Manual function for deploying a wallet and making an initial payment

deployWalletExecCall(address contractAddress, bytes memory data, uint msgValue)

- Manual function for deploying a wallet and making an initial call to a separate contract

deployWalletExecCall(address contractAddress, bytes memory data, uint msgValue, uint fee, address token, uint8 v, bytes32 r, bytes32 s)

- Relayers submit a signed message to deploy wallet and make an initial call to a separate contract

deployWalletExecDelegatecall(address contractAddress, bytes memory data)

- Manual function for deploying a contract and making an initial delegatecall to a

Manual function for deploying a contract and making an initial delegatecall to a separate contract

deployWalletExecDelegatecall(address contractAddress, bytes memory data, uint fee, address token, uint8 v, bytes32 r, bytes32 s)

- Relayers submit a signed message to deploy wallet and make an initial delegatecall to a separate contract

deployWalletExecCreate(bytes memory data)

- Manual function for deploying a wallet and creating a new contract with the Metacash wallet as the creator

deployWalletExecCreate(bytes memory data, uint fee, address token, uint8 v, bytes32 r, bytes32 s)

- Relayers submit a signed message to deploy wallet and create a new contract with the Metacash wallet as the creator

deployWalletExecCreate2(bytes memory data, uint salt)

- Manual function for deploying a wallet and creating a new contract with the create2 opcode and Metacash wallet as the creator

deployWalletExecCreate2(bytes memory data, uint salt, uint fee, address token, uint8 v, bytes32 r, bytes32 s)

- Relayers submit a signed message to deploy wallet and create a new contract with the create2 opcode and Metacash wallet as the creator

getCreate2Address(address owner)

- View function that returns a Metacash wallet address for a given user

getCreate2Address()

- View function that returns a Metacash wallet address for msg.sender

canDeploy(address owner)

- View function that returns whether a Metacash wallet for a user already has been deployed

canDeploy()

- View function that returns whether a Metacash wallet for msg.sender has already has been deployed

recover(bytes32 messageHash, uint8 v, bytes32 r, bytes32 s)

- Returns the signer of a message if the signature provided is correct, else it returns address(0)

Proxy

The Proxy function is the contract that is deployed by the Factory that represents a wallet for individual users. This contract is lightweight and uses delegatecall in its fallback function to execute all functionality. It defines a mapping for state storage called `store` that encodes all persistent data with `abi.encode`. The constructor stores the hard coded fallback address of the SmartWallet library as well as the Factory's address.

`fallback` - SmartWallet implementation library

`factory` - Creator contract of this wallet

SmartWallet

This contract is deployed once and is to be used as a library by individual Proxies through delegatecall. It contains all the functionality of the wallet and stores two additional fields in the storage array:

`owner` - the user of the wallet

`nonce` - a incrementing uint used for replay protection

SmartWallet defines the following functions:

initiate(address owner)

- Called only by Factory to initialize owner and nonce

initiate(address owner, address relay, uint fee, address token)

- Initializes owner and nonce and pays a fee to a relayer

pay(address to, uint value, uint fee, address tokenContract, uint8 v, bytes32 r, bytes32 s)

- Relayer submitted message to transfer an ERC20 token

pay(address to, uint value, address tokenContract)

- Manual transaction to transfer an ERC20 token

pay(address[] memory to, uint[] memory value, address[] memory tokenContract)

- Manual transaction to batch transfer tokens

_execCall(address contractAddress, bytes memory data, uint256 msgValue)

- Internal function to execute a call against a contract

_execDelegatecall(address contractAddress, bytes memory data)

- Internal function to execute a delegatecall against an arbitrary contract

_execCreate(bytes memory data)

- Internal function to create a new contract

_execCreate2(bytes memory data, uint256 salt)

- Internal function to create a new contract with the create2 opcode

execCall(address contractAddress, bytes memory data, uint256 msgValue)

- Manual function to execute a call against another contract from the wallet

execCall(address contractAddress, bytes memory data, uint256 msgValue, uint fee, address tokenContract, uint8 v, bytes32 r, bytes32 s)

- Relayer submitted message to execute a call against another contract from the wallet

execDelegatecall(address contractAddress, bytes memory data)

- Manual function to execute a delegatecall against another contract from the wallet

execDelegatecall(address contractAddress, bytes memory data, uint fee, address tokenContract, uint8 v, bytes32 r, bytes32 s)

- Relayer submitted message to execute a delegatecall against another contract from the wallet

execCreate(bytes memory data)

- Manual function that executes _execCreate(...)

execCreate(bytes memory data, uint fee, address tokenContract, uint8 v, bytes32 r, bytes32 s)

- Relayer submitted message that executes _execCreate(...)

execCreate2(bytes memory data, uint salt)

- Manual function that executes _execCreate2(...)

execCreate2(bytes memory data, uint salt, uint fee, address tokenContract, uint8 v, bytes32 r, bytes32 s)

- Relayer submitted message that executes _execCreate2(...)

depositEth()

- Payable empty function to receive Ether

withdrawEth()

- Manual function that withdraws all available to Ether and transfers to the owner

upgrade(address implementation, uint fee, address feeContract, uint8 v, bytes32 r,

bytes32 s)

- Relayer submitted function that upgrades the SmartWallet implementation by overwriting store["fallback"]

upgrade(address implementation)

- Manual function that upgrades the SmartWallet implementation by overwriting store["fallback"]

recover(bytes32 messageHash, uint8 v, bytes32 r, bytes32 s)

- Identical to function in Factory

RelayRegistry

A simple registry containing a mapping of addresses to a boolean indicating whether they are a relayer or not. Relayers can only be added or removed by the owner (an administrative account belonging to Metacash).

It defines one function:

triggerRelay(address relay, bool value)

- Adds or removes a given relayer from the registry and emits an event

Ownable

Based on the OpenZepplin contract, used by Relay Registry for authorization.

Attack Surface Analysis

This section describes the results of our attack surface analysis of Metacash. This attack surface analysis was one of the factors used to guide the smart contract review process.

While investigating, the following assumptions were made:

- Trusted relayers are added by Metacash for now, but this will eventually be replaced with a decentralized organization
- Relayer scripts, server code, and APIs are out of scope
- Android and iOS wallet applications are out of scope

The following characteristics impact attack surface:

- Users can deploy only one instance of a Metacash wallet from a given Factory
- All wallet actions can be performed either manually or through a signed message sent to a relayer
- Individual wallet owners are the only role capable of authorizing a wallet upgrade
- Users are expected to store a significant amount of value in their wallets

The following assets should be protected:

- Wallet Assets: Ether or ERC20 tokens
- Gas: Fuel used when executing an ethereum transaction
- Metacash Message: A signed message authorizing a relayer to execute an action on the user's wallet
- Private Keys: Held within client wallet software, used to sign Ethereum transactions and Metacash messages

Based on the attack surface and the assets, the following top risks were identified:

- An anonymous party can steal or lock user funds in the wallet
- A malicious DApp can steal or lock user funds in the wallet
- A malicious relayer can steal or lock funds in a user's wallet
- A malicious user can waste gas of a relayer

Problem Report Summary

A total of **10** issues were identified. This section describes, at a high level, each of the problems discovered. See the Problem Summaries section for a table of each problem discovered, its severity, description and consequences.

Problem reports are sorted by severity:

- **Critical**: funds lost for all users from an untrusted attacker
- **High**: funds lost for targeted users from an untrusted attacker
- **Medium**: funds lost for targeted users from a trusted attacker
- **Low**: unlikely edge case, defense in depth
- **Observation**: best practices and gas optimizations

The problem report summaries are sorted by problem report ID. The format of the problem report table is as follows:

- The Problem Report ID
- The component in which the issue was discovered.
- The severity of the issue
- The issue title

PR #	Component	Severity	Title
1	Signature	High	Fund Theft Through Signature Phishing
2	Signature	High	Fund Theft Through Transaction Collisions
3	SmartWallet	Medium	Double Spend After Relayer Message Withholding
4	SmartWallet	Medium	Non-Compliant Tokens are Not Supported
5	SmartWallet	Low	Failure to Validate Return Value of ecrecover
6	Metacash.sol	Low	Floating Pragma Version
7	SmartWallet	Low	Functions Not Declared Payable
8	SmartWallet	Low	Re-entrancy Within execCall Function
9	Upgrades	Low	User-Activated Self-Destruct Enables Replay Attacks
10	Upgrades	Low	Potential to Upgrade Away Fees

Problem Reports

Below are the details for each of the Problem Reports.

Problem Report 1 - Fund Theft Through Signature Phishing

Severity	Target	Line Number
High	Signature Verification	382

Description

Transactions that are committed by relayers are constructed in the standard format produced by the Web3 "eth_sign" method. For Metacash, transactions are a signed message where the payload is of the following form:

```
"\x19Ethereum Signed Message:\n32"+keccak256(payload)
```

When asked to sign messages with eth_sign, client wallets such as Metamask do not use a standard way of structuring the message payload for the user to verify. Because of this, there is no default UX that user's can expect for verifying where the signed message is intended for.

An attacker can abuse this lack of UX clarity to construct a phishing attack that may move funds from their Metacash wallet without their knowledge.

Consider the following attack:

A malicious DApp (cryptokitties2) has their users sign a message as part of the functionality, such as a user registration or login.

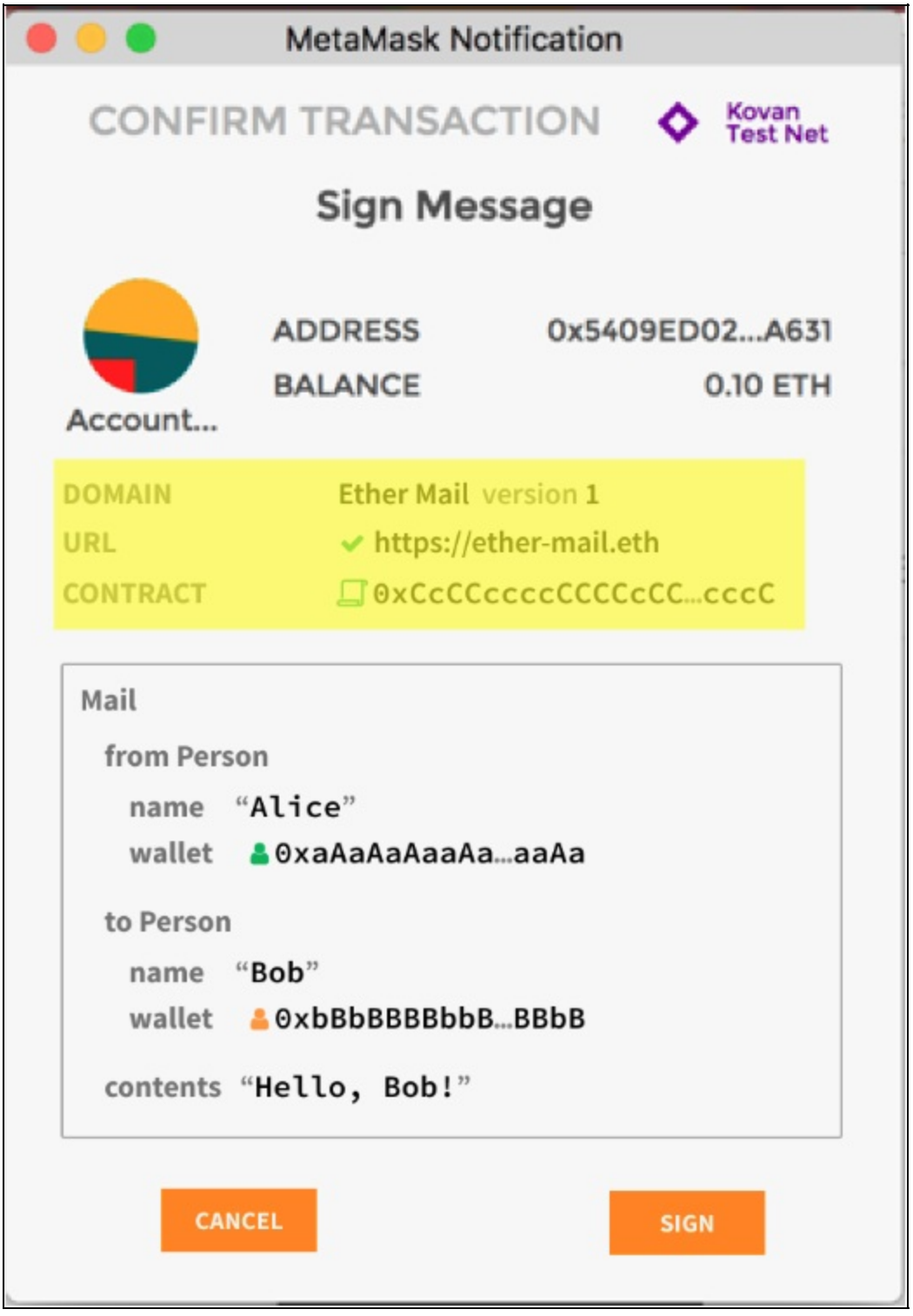
CryptoKitties2 makes the user sign a malicious message with eth_sign that happens to be identical to a request to their Metacash wallet to transfer out their funds. Since the eth_sign method does not present any details to the structure of the payload, the message will appear only as hex data to the user. Once the user signs the message, the malicious DApp then replays that signed message to a MetaCash relayer, draining that user's funds.

Remediation

Consider Implementing EIP712 - EIP712 is a standard that defines a readable structure to signed messages for enabling user verification of the message from the client UX. EIP712 messages are signed with a yet to be implemented "eth_signTypedData" RPC call as opposed to eth_sign.

Part of this standard defines a *Domain Separator* that allows DApps to clearly verify the intended recipient as part of the signed data.

EIP712 is still in draft format, however, popular wallets like Metamask are beginning to support displaying messages in this format with the domain clearly presented to the user.



It is recommended that despite EIP712 not being finalized, that Metacash implement signature verification under this standard.

More information on EIP712 is available at the following URLs:

https://github.com/ethereum/EIPs/blob/master/EIPS/eip-712.md

https://weijiekoh.github.io/eip712-signing-demo/index.html

Example Solidity and Javascript for implementing EIP712 is available at the following URLs:

https://github.com/ethereum/EIPs/blob/master/assets/eip-712/Example.sol

https://github.com/ethereum/EIPs/blob/master/assets/eip-712/Example.js

Warn Users Not to Sign Messages Outside of Metacash - Since the current version of Metacash is intended to be used by a custom mobile client and not a generic Web3 client, it may be sufficient to instruct users through a clear warning that signing a message for another DApp with their Metacash private key may put their funds at risk of theft. This may be a sufficient path forward until EIP712 is finalized, though it will only be temporary if users are later expected to interact with their wallet through other clients.

Metacash Response

Because EIP712 has not yet been finalized, we think it is not yet sufficiently stable to be used in our Solidity codebase. Instead, we resorted to protect our users against potential phishing attacks by replacing the standard Ethereum RPC signature prefix by the unique prefix `\x19Metacash Signed Message:\n32`.

Problem Report 2 - Fund Theft Through Transaction Collisions

Severity	Target	Line Number
High	Message Signature	382

Description

Several functions in Metacash verify signed message hashes of unstructured data that is encoded with `abi.encode(...)`. The following table describes what data is encoded in Metacash for accepted messages. Each row includes the function name along with an ordered argument to the encoded signature payload.

Note that all fields with the label `data` are byte arrays and are variable in length. All other fields are fixed in size depending on their data type. Due to the way abi.encode(a,b,c,...) encodes data, variable length arrays will be directly appended into the result. If the variable length array is empty, no data will be written in that location.

Function	1	2	3	4	5	6	7	8	9
dW (deployWallet)	factory	relayer	token	gasprice	fee				
dW	factory	relayer	token	to	gasprice	fee	value		
dWExecCall	factory	relayer	token	contract	data	msgValue	gasprice	fee	
dWExecDelegateCall	factory	relayer	token	contract	data	gasprice	fee		
dWExecCreate	factory	relayer	token	data	gasprice	fee			
dWExecCreate2	factory	relayer	token	data	gasprice	salt	fee		
execCall	relayer	contract	token	factory	data	msgValue	fee	gasprice	nonce
execDelegatecall	relayer	contract	token	factory	data	fee	gasprice	nonce	
pay	relayer	to	token	factory	value	fee	gasprice	nonce	

Function	1	2	3	4	5	6	7	8	9
upgrade	relayer	impl	token	factory	fee	gasprice	nonce		
execCreate	relayer	token	factory	data	fee	gasprice	nonce		
execCreate2	relayer	token	factory	data	salt	fee	gasprice	nonce	

Observe that these message payloads do not include the target function they are meant for. This means that a malicious relayer can collect a valid signed message and construct a request to the Metacash wallet that accomplishes a different task than what the owner intended. Some consequences of this include the following:

execCall Message used in Pay - If a wallet owner signs a message to execCall that does not include any *data* field, the empty byte array will take up 0 bytes when abi.encoded and the message will be identical in size and structure to a pay message that is transferring tokens to the target contract. A malicious relayer can create a DApp that accepts Ether with a fallback function, for example a token sale contract, and rather than completing that transaction, they can use that signed message to transfer DAI or whatever ERC20 token is used for fees to that contract instead.

execDelegateCall used in Upgrade - If a user creates an execDelegateCall message with an empty *data* field, the message can be used when calling upgrade which will most likely brick the wallet and lock the users funds.

execDelegateCall used in Pay - If the *data* field of an execDelegateCall message is the size of a uint256, then the message can be interpreted as a token transfer with *data* tokens being transferred to the target contract.

execCall used in ExecDelegateCall - If the *data* field is empty in a execCall message, the format will match an execDelegateCall message where the message value is interpreted as the data.

Note: Due to the scope of testing, we did not verify that these malicious payloads could be created from the Metacash DApp.

Remediation

Verify the Function in the Signed Message - Include the function identifier as part of the signed message and verify it within the smart contract on execution.

Remove execDelegateCall - This function exposes Metacash to unnecessary risk without providing substantial benefit to the end-user.

Metacash Response

We have added a string containing the function identifier to the beginning of each Metacash signed message. We have also removed execDelegateCall from SmartWallet and Factory contracts.

Problem Report 3 - Double Spend After Relayer Message Withholding

Severity	Target	Line Number
Medium	SmartWallet	161

Description

Metacash uses an incrementing nonce to ensure that signed messages cannot be replayed. The contract only uses this nonce when executing signed messages. For manual transactions sent by the owner, nonces are not used.

If all relayers go offline or decide to censor a user's message, the user may grow impatient and perform a manual transaction instead. Once that transaction has been executed, the relayers can then submit the previously signed message, since the nonce in the signed message is still valid. If the relayer partners with the intended recipient of the fund, they can execute this attack in order to collect double the payment from the owner.

Remediation

Increment the Nonce for All Transactions - This will prevent any previously valid messages from being used after a manual transaction.

Note: As mentioned during discussions by Metacash, this can create the possibility of a malicious user front-running a relayer to waste the relayer's gas, which can be a profitable attack if the malicious user is a miner.

Include a Maximum Timeout for All Signed Messages - Verify this timeout in the contract against a future block number rather than a timestamp to minimize miner variability.

Metacash Response

We have included an additional uint256 deadline to all Metacash signed messages that must be lower or equal to the block number where the transaction is included. However, we chose not to increment nonces for non-signed message transactions because it would open the door to spam frontrunning attacks by users against relayers.

Problem Report 4 - Non-Compliant Tokens are Not Supported

Severity	Target	Line Number
Medium	SmartWallet	163

Description

The ERC20 token standard describes a transfer function as returning a boolean True value if the transfer is successful. Unfortunately, many popular tokens do not correctly implement this standard and instead do not return any value. Two such tokens include OMG and BNB. More information on the topic is available in the following post:

https://medium.com/coinmonks/missing-return-value-bug-at-least-130-tokens-affected-d67bf08521ca

The Metacash wallet assumes that all tokens that are transferred strictly follow this standard and wraps all token transfers in a `require(...)` to validate the returned value is true. This means that if a token is sent to this contract that does not correctly implement IERC20 and does not return a boolean, all calls to transfer that token will fail.

Note that it is still possible to retrieve these tokens by constructing a call for execCall(...), though this is non-intuitive for an average user.

Remediation

Provide Support for Non-Compliant Tokens - Provide a separate payNoncompliant(...) function that does not expect a boolean return in the transfer so that typical wallet owners have a way of moving non-compliant tokens out of their wallet.

Metacash Response

Instead of adding a special function for non-compliant tokens, we have removed the requirement for external ERC20 token transfers to return true. This should be compatible with tokens that do return true as well as those that do not.

Problem Report 5 - Failure to Validate Return Value of ecrecover

Severity	Target	Line Number
Low	SmartWallet	385, 690

Description

If the parameters passed to an `ecrecover(...)` are incorrect, such as when a signature is invalid, the result will be 0x0000000000000000000000000000000000000000. The `recover(...)` functions within Metacash do not check for this case and as such will return the null value as if it is a valid address.

With this, the following scenario was suspected to be possible:

1. Assume the instance of the SmartWallet library contract is deployed at 0xAABBCCDD...
2. In this attack, a malicious relayer sends an execDelegateCall(...) transaction to that contract address (0xAABBCC..., not an individual Proxy). The signature data will be invalid, which will mean ecrecover returns 0. Because the value of `store["owner"]` will also be 0 by default since the contract was never initialized, the signature check will succeed.
3. The delegatecall can then be made to a contract with a self-destruct, destroying the SmartWallet library contract and locking all instances of Proxy, similar to what occurred with the second Parity multi-sig attack.

This attack was not exploitable in practice, due to that fact that abi.decoding an uninitialized `bytes` array will revert rather than return zero:

```
require(abi.decode(store["owner"], (address)) ==
recover(keccak256(abi.encodePacked(msg.sender, contractAddress, tokenContract,
abi.decode(store["factory"], (address)), data, fee, tx.gasprice, currentNonce)), v, r,
s));
```

Despite no practical exploit existing, it is best practice to validate the return value from all low-level functions such as ecrecover. The behavior of `abi.decode` reverting is currently not documented in Solidity docs and could change in the future.

Remediation

Validate the Output of ecrecover - This can be done by storing the output as an address in memory and requiring it not be equal to address(0).

Remove execDelegateCall - This function exposes Metacash to unnecessary risk without

providing substantial benefit to the end-user.

Consider Moving the Upgrade Functionality to Proxy - This will increase the deployment size of each wallet, but it will allow a way for owners to recover their wallets in the event that the SmartWallet library contract is destroyed.

Metacash Response

Since we are fixing the contracts source to pragma solidity v0.5.8, if the behaviour of Solidity changes in the future, it will not have any effect on our existing code. Therefore, we chose not to validate the output of ecrecover.

Also as a response to Problem Report 2, we have decided to remove execDelegateCall from our contracts.

We decided to keep the upgrade functionality at the SmartWallet contract in order not to increase the gas cost of deploying user proxy contracts.

Problem Report 6 - Floating Pragma Version

Severity	Target	Line Number
Low	Metacash.sol	1

Description

To lower the potential risk of undiscovered bugs, smart contracts should be deployed with the compiler version that has already been thoroughly tested. By locking in the pragma, it helps to ensure that contracts do not get deployed with a different version which could potentially increase the chances for bugs or other unexpected consequences that the original authors may not have originally considered.

The current version is specified at a minimum of v5.0.3 and is not locked in:

`pragma solidity ^0.5.3;`

Remediation

Lock Pragmas to Specific Compiler Version - Locking smart contracts to a specific version helps guarantee deployment occurs consistently with the original authors intentions and limits the risk of unknown bugs in latter versions.

myContract.sol

```
// less secure
pragma solidity ^0.5.3;

// more secure
pragma solidity 0.5.3;
```

Use Latest Compiler Build - It is recommended that Metacash V2 use the latest available update of the solidity compiler, which is version 0.5.8. Older versions such as 0.5.5 and 0.5.6 have known vulnerabilities and must be avoided.

It is also important to note that upgrading SmartWallet via delegatecall to a contract compiled

with a different Solidity version is dangerous and can cause unexpected changes to the storage layout that may put user funds at risk. For wallets that have been deployed with a previous version of solidity, it is recommended that Metacash advise its users to only upgrade to a contract compiled with the same version of Solidity.

Metacash Response

We have fixed the Solidity version to pragma solidity v0.5.8.

Problem Report 7 - Functions Not Declared Payable

Severity	Target	Line Number
Low	SmartWallet	414, 547

Description

The Proxy contract's fallback function is not declared `payable`. Since all calls to the SmartWallet library are transacted through the Proxy's fallback, this means that Ether can not be sent to any of the functions defined in SmartWallet. Since some of the functionality defined in SmartWallet expects Ether to be received and stored in the wallet, such as `depositEth(...)` and `execCall(...)`, this prevents some of the intended functionality of the wallet.

Additionally, `deployWalletExecCall(...)` in Factory expects to send a callvalue, but can only send 0 Ether since it is not payable.

Remediation

Mark All Necessary Functions as Payable - This includes the fallback defined in Proxy and `deployWalletExecCall(...)` in factory. Currently, functions that invoke a `_execCreate(...)` or `_execCreate2(...)` will only accept contract creations that provide no value. If this changes in the future, functions that invoke these calls may need to be marked payable as well.

Do Not Redundantly Accept msgValue as a Function Parameter - The function `deployWalletExecCall(...)` accepts a parameter *msgValue* that can only be less than or equal to the msg.value of the transaction. Since there is no use case for this parameter aside from matching msg.value, it is recommended this parameter be removed and msg.value be used instead.

Remove the Fallback in SmartWallet - The fallback defined in Line 86 will still require Proxy's fallback to be payable in order to accept Ether. Further, if the fallback is called by a smart contract in response to an `execCall(...)` function where the smart contract used `msg.sender.send(...)` or `msg.sender.transfer(...)`, the entire transaction is likely to fail. This is because the fallback will be limited to the 2300 gas stipend, which is not enough to both abi.decode the owner and send the msg.value. Instead, it is recommended that this fallback be removed entirely. The owner will still be capable of withdrawing Ether manually with the `withdrawEth()` function.

Add Unit Tests - This issue is a functional bug that could have been caught through comprehensive unit testing of the contracts' features.

Metacash Response

We have marked the said functions as payable. We have removed the redundant msgValue

function parameter. We have removed the fallback function from SmartWallet contract.

Problem Report 8 - Re-entrancy Within `execCall` Function

Severity	Target	Line Number
Low	SmartWallet	249

Description

The following `execCall` function allows a relayer to execute a call to any contract on behalf of the owner. As we can see from the below parameters, there is a uint fee amount parameter passed into this function and this fee is paid to the relayer for executing the call after the call has been made. Depending on the contract that call is made to, this may pose a potential re-entrancy risk that could allow a malicious relayer to collect multiple fees.

While unlikely, the following scenario could present a double fee collection based on re-entrancy:

1. A malicious relayer exists as an attack contract `X`.
2. The relayer convinces a user to sign tx `T` that executes a call to honeypot `Y`.
3. The relayer then writes `T` to their contract `X` as stored data, `X` then submits `T` to the Metacash wallet which execCalls `Y`.
4. On receipt of the call, `Y` calls `X` and `X` replays `T` once more.
5. Because the nonce will not have incremented yet, the same signed message will succeed, meaning the relayer can collect fees for `T` more than once.

Note that this attack is dependent on a user agreeing to make an execCall to a malicious contract, which is dangerous on its own.

Metacash.sol

```
246 function execCall(address contractAddress, bytes memory data,  uint256 msgValue, uint
fee, address tokenContract, uint8 v, bytes32 r, bytes32 s) onlyRelay public returns (bool)
{
247   uint currentNonce = abi.decode(store["nonce"], (uint));
248   require(abi.decode(store["owner"], (address)) ==
recover(keccak256(abi.encodePacked(msg.sender, contractAddress, tokenContract,
abi.decode(store["factory"], (address)), data, msgValue, fee, tx.gasprice, currentNonce)),
v, r, s));
249   require(_execCall(contractAddress, data, msgValue));
250   IERC20 token = IERC20(tokenContract);
251   store["nonce"] = abi.encode(currentNonce+1);
252   require(token.transfer(msg.sender, fee));
253   return true;
254 }
```

Metacash.sol

```
184 function _execCall(address contractAddress, bytes memory data, uint256 msgValue)
internal returns (bool result) {
185    assembly {
186        result := call(gas, contractAddress, msgValue, add(data, 0x20), mload(data), 0,
0)
187    }
```

```
188  }
```

Remediation

Transaction Ordering While the exploit imagined is somewhat impractical, it is best practice that the external call be moved to the end of the function, after the nonce has been incremented.

Add Additional Comments Provide additional inline comments about all potential functions that pose a re-entrancy risk.

Metacash Response

We have moved the external call to the end of the execCall function body to avoid the potential re-entrancy attack. Also, we have added a comment at the internal _execCall function to warm readers of potential re-entrancy risk.

Problem Report 9 - User-Activated Self-Destruct Enables Replay Attacks

Severity	Target	Line Number
Low	Upgrade Mechanism	372

Description

Currently, a signed message will always be valid until that nonce has been used. Transactions sent to Factory do not use a nonce since they can only be applied once if the wallet has not yet been deployed.

If a user is later able to destroy their wallet with a self-destruct operation, through `execDelegateCall(...)` or by upgrading to a contract that supports self-destruct, a malicious relayer will be able to replay all previous messages and potentially recreate the destroyed wallet in order to steal token balances and collect fees.

Remediation

Remove execDelegateCall - This function exposes Metacash to unnecessary risk without providing substantial benefit to the end-user.

Include a Maximum Timeout for All Signed Messages - Verify this timeout in the contract against a future block number rather than a timestamp.

Do Not Introduce Self-Destruct in A Future Upgrade - This functionality will increase the likelihood of previous transactions being replayed.

Metacash Response

We have removed execDelegateCall from our contracts.

We have added a block inclusion deadline for all Metacash signed messages.

Problem Report 10 - Potential to Upgrade Away Fees

Severity	Target	Line Number
Low	Upgrade Mechanism	372

Severity	Target	Line Number

Description

Wallet owners have the ability to upgrade their Metacash wallet to any implementation they choose. This means they can upgrade to a bootleg contract that removes fee payments for relayers. If a relayer fails to sufficiently validate what version of the SmartWallet contract is running, they can perform unintended actions and miss out on fee collection.

Remediation

Relayers Must Validate SmartWallet Code - It is not enough to verify the address of a wallet came from Factory. Instead, before submitting any transactions, relayers must validate that the SmartWallet implementation is one that was created by Metacash. One way to do this would be by checking the hash of the library's code against a known list.

Metacash Response

In the existing and future relayer designs, we will protect the relayer against fee-avoidance attacks by fetching for past Upgrade events emitted by the user’s smart wallet before admitting each signed message. Relayers will not accept signed messages that are directed to smart wallets that have been previously upgraded to an unknown implementation.

Observations

Over the course of testing, there were a total of 10 problems identified. In addition, there were several other observations made about Metacash V2. While not severe enough to warrant problem reports, Security Innovation recommends that Metacash investigate these in addition to the security-related issues that were found.

Observation 1: Nonces Must be Serial

A Metacash wallet will only accept one valid transaction at a given moment, since the nonce must be equal to the next nonce sequentially and nonces can not be reused. This prevents a power user from creating many transactions at the same time, such as with a large air drop, without limiting it to one single transaction that fits within the maximum block gas limit. While this architecture is simpler and arguably more secure, it does limit the capabilities of the user. If support for parallel transactions is later added, it is recommended that the client generate random nonces for each transaction it creates and marks them as used in a mapping. This is demonstrated in the following article:

https://programtheblockchain.com/posts/2018/02/17/signing-and-verifying-messages-in-ethereum/

Observation 2: Redundant Return True

Several functions were found to return True even if there was no condition where False could be returned. Rather, if a failure occurred the transaction would revert. Since there is no condition where False can be returned, this return value is redundant. It is recommended that all returns that do not offer useful information or are not part of a standard be removed so that no value is returned.

Observation 3: Inconsistent Return Style

While reviewing the Metacash, it was noticed that there were multiple functions specifying the return type parameter but provided no return statement. This does not pose a direct security concern, however, it is inconsistent and can be easily confused or mistaken by developers leveraging this smart contract code.

Below is an example :

Metacash.sol

```
184 function _execCall(address contractAddress, bytes memory data, uint256 msgValue)
internal returns (bool result) {
185    assembly {
186        result := call(gas, contractAddress, msgValue, add(data, 0x20), mload(data), 0,
0)
187    }
188  }
```

In this example, result is declared but there is no return statement inside the function. In addition, to the above function, this was observed on L184 and L669. It is recommended that for consistency all functions that return a value use the explicit return statement.

Observation 4: Event Reduction for Gas Optimization

While reviewing the Metacash smart contracts, it was identified that the `RelayRegistry` makes use of two separate events. The first event is for when a new relayer gets added to the registry and the second is for when an existing relayer gets removed from the registry. The event that is output is then determined based on a conditional if statement.

Metacash.sol

```
48  event AddedRelay(address relay);
49  event RemovedRelay(address relay);
```

By reducing the events to only one that could work for either an addition or a removal to the registry, a small gas savings can be achieved.

Observation 5: Code Consistency and Quality

There are a few development inconsistencies that do not pose a direct security concern but deviate slightly from other representations in the Metacash source code. These subtle differences could open the door to developer confusion or potential difficulties maintaining the code. The below table references these inconsistencies specifically:

Target	Line Number	Inconsistency	Description
Metacash.sol	343	Parens Usage	The `onlyOwner` modifier inconsistently uses parentheses in the modifier
Metacash.sol	462,672	iszero Check	One function uses the `iszero` assembly code whereas another uses `eq(...)`
Metacash.sol	115	Exception Behavior	The `initate(...)` function returns false on failure whereas all other functions throw.

Observation 6: Use of Explicit Sized uint

The contracts use `uint` in place of the more explicit `uint256`. As a best practice, it is recommended that the size of data types be explicit, ensuring the future readability of the contract.

Observation 7: Function Visibility

During the review of the Metacash smart contracts, it was noticed that the visibility on multiple functions were set to *public* when they did not require that level of visibility. Although this does not provide a direct security concern, restricting these functions could provide a more thorough defense in depth approach that could potentially limit the attack surface for future bugs and attacks. Below is a list of results returned from the Slither static text analyzer:

- RelayRegistry.triggerRelay (Metacash.sol#61-69) should be declared external
- SmartWallet.initiate (Metacash.sol#129-136) should be declared external
- SmartWallet.pay (Metacash.sol#145-153) should be declared external
- SmartWallet.pay (Metacash.sol#161-165) should be declared external
- SmartWallet.pay (Metacash.sol#170-176) should be declared external
- SmartWallet.execCall (Metacash.sol#233-236) should be declared external
- SmartWallet.execCall (Metacash.sol#246-254) should be declared external

- SmartWallet.execDelegatecall (Metacash.sol#261-264) should be declared external
- SmartWallet.execDelegatecall (Metacash.sol#273-281) should be declared external
- SmartWallet.execCreate (Metacash.sol#287-290) should be declared external
- SmartWallet.execCreate (Metacash.sol#298-306) should be declared external
- SmartWallet.execCreate2 (Metacash.sol#313-316) should be declared external
- SmartWallet.execCreate2 (Metacash.sol#325-333) should be declared external
- SmartWallet.depositEth (Metacash.sol#338) should be declared external
- SmartWallet.withdrawEth (Metacash.sol#343-346) should be declared external
- SmartWallet.upgrade (Metacash.sol#354-366) should be declared external
- SmartWallet.upgrade (Metacash.sol#372-376) should be declared external
- Factory.deployWallet (Metacash.sol#472-479) should be declared external
- Factory.deployWallet (Metacash.sol#488-495) should be declared external
- Factory.deployWallet (Metacash.sol#500-506) should be declared external
- Factory.deployWallet (Metacash.sol#514-521) should be declared external
- Factory.deployWalletExecCall (Metacash.sol#530-537) should be declared external
- Factory.deployWalletExecCall (Metacash.sol#547-554) should be declared external
- Factory.deployWalletExecDelegatecall (Metacash.sol#561-568) should be declared external
- Factory.deployWalletExecDelegatecall (Metacash.sol#577-584) should be declared external
- Factory.deployWalletExecCreate (Metacash.sol#590-597) should be declared external
- Factory.deployWalletExecCreate (Metacash.sol#605-612) should be declared external
- Factory.deployWalletExecCreate2 (Metacash.sol#619-626) should be declared external
- Factory.deployWalletExecCreate2 (Metacash.sol#635-642) should be declared external
- Factory.getCreate2Address (Metacash.sol#661-663) should be declared external
- Factory.canDeploy (Metacash.sol#679-681) should be declared external

Please note that this list might not include every occurrence of the aforementioned observation. Upon remediation, it is advised that the development team review each occurrence in depth.

Observation 8: Missing Return value

The `_execCall()` and `_execDelegatecall()` functions are not equipped to return the resulting data from these calls to the user. Depending on the use case, it may be necessary for users to receive this data in order to perform their desired action.

Observation 9: Missing Revert Messages

As of Solidity v0.4.24, all reverts can include a reason string. It is recommended that these strings be included in all `require(...)` statements to add clarity when encountering an error.

Observation 10: Unclear Function Naming

There are four functions with the name *deployWallet*. Two of these functions offer the manual and relayed version of deploying a new wallet, whereas the other two offer the manual and

relayed version of deploying a wallet and making a payment. This naming collision can be confusing to readers of the source and ABI. It is recommended that the functions that create and pay be renamed to *deployWalletAndPay*.

Observation 11: Incorrect Commenting

Line 204 and Line 216 have errors in the comments that prevent Solidity from compiling:

```
Error: Documented parameter "bytecode" not found in the parameter list of the function.
Error: Documented parameter "bytecode" not found in the parameter list of the function.
```

To fix this, insert the string *data* between *@param* and *bytecode*.

Metacash.sol

```
202  /*
203   * @dev Internal function that creates any contract
204   * @param data bytecode of the new contract
205   */
206  function _execCreate(bytes memory data) internal returns (bool result) {
```

Metacash.sol

```
214  /*
215   * @dev Internal function that creates any contract using create2
216   * @param data bytecode of the new contract
217   * @param salt Create2 salt parameter
218   */
219  function _execCreate2(bytes memory data, uint256 salt) internal returns (bool result)
{
```

Observation 12: Batch Pay Assertions

The batch pay function makes the assumption that the lengths all the parameter arrays are equal. If a read is made outside of the array bounds, the transaction will revert. Despite this, it is best practice that these inputs be checked beforehand and explicitly throw when they are not equal in length.

Metacash.sol

```
167  /*
168   * @dev Same as above but allows batched transfers in multiple tokens
169   */
170  function pay(address[] memory to, uint[] memory value, address[] memory tokenContract)
onlyOwner public returns (bool) {
171      for (uint i; i < to.length; i++) {
172          IERC20 token = IERC20(tokenContract[i]);
173          require(token.transfer(to[i], value[i]));
174      }
175      return true;
176  }
```

Observation 13: Inconsistent Order of Initiate and Pay

The Factory contract typically will run the initiate wallet function last. This is to allow a case where a previous action (such as call or create) funds the wallet for the fees required in initiate.

In one function on L493 the pay occurs after initiate. It is not clear why this is the case and is recommended for consistency sake that this function follow the style of the others and run initiate last.

Additionally, a comment on L525 describes the collect-call use case described above to justify putting initiate last, but this comment precedes a manual function where no fee is collected. It is recommended this justification move to a relayer function instead.

Observation 14: Relayed Pay Message Must Transfer A Supported Token

The pay(...) functionality requires that the ERC20 token being transferred is the same token that is collected by the relayer in fee. This means that if a user wants to transfer a token that is less common or uninteresting to the relayer, they will need to do a manual transaction. It is recommended that the functionality be expanded such that the token for fee payment can be a different token than the one being transferred in a pay message.

Observation 15: NFTs Unsupported

The Metacash wallet does not currently support NFTs. This means that a user who sends an NFT to their Metacash wallet will be unable to move it and it will be locked unless they can correctly use the execCall function. It is recommended that Metacash add an nftWithdrawl(...) function so that users can manually transfer out their ERC721 tokens.

Executed Test Cases

The following table shows the breakdown of executed test cases, including any problem reports relevant to that item, and gives a brief summary of the methodology used to check that item and any other observations.

Column descriptions are as follows:

- ID - An identifier for quick test case reference
- Title - A title describing the test case
- Description - A short description of the test case and why it was performed
- Outcome - Either 'Pass' or a reference to the Problem Report Number

ID	Title	Description	Outcome
1	Assert Violation	(SWC-110) As a best practice, assertion failures should never occur on live code.	**PASS**
2	Audited Dependencies	Verify that all dependencies have been audited, such as OpenZeppelin's contracts.	**PASS**
3	BlockHash for Current/Future Block	Check that the blockhash(block.now) or greater is not used since it returns 0.	**PASS**
4	BlockHash for Old Block	Check that the blockhash of a block 256+ blocks ago is not used since it returns 0.	**PASS**
5	Comments	Confirm dangerous areas of code are commented as such.	**PR8**
6	Compiler	(SWC-102) Check that the contract was compiled with the most recent solidity compiler.	**PR6**
7	Constructor Declaration	(SWC-118) Check that contracts written in Solidity v0.4.22 or greater use the constructor declaration, and that older ones do not contain a typo in the function name.	**PASS**
8	Contract Detection Bypass	Confirm that when checking if a source is a contract, the following is used: require(msg.sender==tx.origin).	**PASS**
9	Delegatecall Storage Slot Matching	(SWC-124) Check that implementations that utilize delegatecall correctly maintain storage slot ordering.	**PASS**
10	Denial of Service 1 - Loops	(SWC-128) Check that loops are used only as needed to avoid reaching gas limits and are not on arbitrarily sized maps/arrays.	**PASS**
11	Denial of Service 2 - Owner	Check that an operation from an owner is not necessary for users to withdraw and leave.	**PASS**
12	Denial of Service 3 - External Contracts	(SWC-113) Check that an operation does not depend on successfully calling another contract.	**PR4**

ID	Title	Description	Outcome
13	External Call Within a Contract	Confirm that any external calls within the same contract (i.e. using this.a(...) rather than just a(...)) do not incorrectly use msg.sender, which would be the contract's address and not the original msg.sender.	**PASS**
14	Floating Point	Confirm that floating points are never used unless there is a good reason.	**PASS**
15	For-Loop Endless Loops	(SWC-101) Check that the conditional in a For-Loop will eventually terminate (i.e. uint8 < 256).	**PASS**
16	Function Visibility	(SWC-100) Check that all functions are appropriately labeled as Public, Private, External, or Internal.	**Observation 7**
17	Inheritance Name Collision	(SWC-119) Confirm that a contract does not unintentionally overwrite a variable from an inherited contract.	**PASS**
18	Inheritance Ordering	(SWC-125) Confirm that the order in which multiple contracts are inherited is correct (Diamond Problem resolution).	**PASS**
19	Inherited Function Signature Mismatch	Confirm that functions that are intended to overwrite inherited functions match signatures correctly.	**PASS**
20	Insufficient Signature Validation	(SWC-122) If an off-chain signature scheme is used, ensure that msg.sender is not assumed to be the source (for cases where the source might be a contract instead).	**PASS**
21	Integer Overflow and Underflow	(SWC-101) Check that the safe math library is used and that arithmetic operations are done with .sub, .add, .mul, and .div.	**PASS**
22	Malicious Owner	Check that a malicious owner can not perform harm to their contract's users, such as stealing balances or changing an external contract.	**PASS**
23	Matching Versions	(SWC-103) Check that the pragma for all contracts is consistent and is locked to tested versions.	**PR6**
24	Miner Front-Running	(SWC-114) Check that miners do not have an economically worthwhile advantage to copy and mine their own transactions.	**PR4**
25	Missing Modifiers	Confirm that modifiers are still in place after inheritance.	**PASS**
26	Missing Revert Messages	Confirm that all reverts, requires, asserts, etc include string explanations, as enabled in solidity 0.4.22.	**Observation 9**
27	Optimizations	Check that the contract is sufficiently optimized to not waste gas.	**Observation 4**

ID	Title	Description	Outcome
28	Randomness	(SWC-120) Check that a source of randomness is not predictable.	**PASS**
29	Re-entrency	(SWC-107) Check that .call() is avoided when possible, and that value sends happen at the end of the function.	**PR8**
30	Revert Early and Often	(SWC-123) Check that a contract reverts as soon as it detects a violation.	**PASS**
31	Safemath Calculation Stored	Ensure that safemath operations store the result in a variable.	**PASS**
32	Sensitive Data in Private Variables	Check that all data in private variables are not sensitive, since their value can be determined in a chain explorer.	**PASS**
33	Signature Malleability	(SWC-117) Confirm that any signatures that are manually checked do not incorrectly depend on a constant hash.	**PASS**
34	Signature Replay Attacks	(SWC-121) Confirm that any signed messages that are interpreted by the contract can not be replayed with the same hash and signature.	**PR1, PR2**
35	Signed vs Unsigned Casting	(SWC-101) Check that there is not implicit casting of signed to unsigned.	**PASS**
36	Storage Overwrite via Uninitialized Pointer	(SWC-109) Check that all storage pointers are initialized (pre Solidity 0.5).	**PASS**
37	Storage Overwrite via array.length	(SWC-124) Check that an array length is never manually set.	**PASS**
38	Timestamp Operations	(SWC-116) Check that block.timestamp is not checked for precision in any trusted operations.	**PASS**
39	Timing Attacks in ERC20 - Approve	(SWC-114) Confirm that a contract does not unsafely set the approve value of an ERC20 token such that it can be double spent.	**PASS**
40	Token Decimals	Confirm that the correct decimal is taken into account when performing arithmetic on tokens.	**PASS**
41	Tx.origin Authentication	(SWC-115) Check that tx.origin is not used for authorization in a contract.	**PASS**
42	Unary Operators	(SWC-129) Confirm that when an int is negated with a minus sign, the case of 0x8000.. is considered.	**PASS**
43	Unchecked Return Values	(SWC-104) Check that the return values of .call(...), .callcode(...), .send(...), and .delegetecall(...) are always validated.	**PR5**
44	Unexpected Balance	Check that a contract with a balance that skips the fallback (selfdestruct or precomputed address) does not affect the business logic.	**PASS**

ID	Title	Description	Outcome
45	Unexpected Consequence of Uninitialized Memory	Confirm that an uninitialized variable defaulting to 0 does not mean anything (such as a 0 index in an array).	**PASS**
46	Unit Tests	Check that a sufficient number of unit tests have been created.	**PR7**
47	Unprotected Self Destruct	(SWC-106) Confirm that selfdestruct cannot be called by an unauthorized user.	**PR9**
48	Unsafe Assembly	(SWC-127) Verify that all uses of direct assembly are safe.	**PASS**
49	Unsafe Delegatecall	(SWC-112) Check that delegatecall is only used against trusted contracts.	**PR10**
50	Upgradability	Confirm that if upgrading the contract is desired, then data is stored in a separate contract from instructions.	**PASS**
51	Use of Deprecated Functions	(SWC-111) Confirm that no deprecated functions are in use, including: suicide(...), block.blockhash(...), sha3(...), callcode(...), throw, msg.gas, and constant.	**PASS**
52	User Initiated Withdrawal	(SWC-113) Confirm that withdrawals are user initiated as a best practice.	**PR7**
53	Variable Visibility	(SWC-108) Check that all contract variables are appropriately labeled as Public, Private, External, or Internal.	**PASS**
54	View Functions used Internally	Check that inefficient view functions are not called by internal functions.	**PASS**
55	Noncompliant Tokens	Verify that tokens that do not follow ERC standards can still be recovered.	**PR3**
56	Unspecified Integer Size	Verify that all integer values specify an exact integer size to limit ambiguity.	**Observation 6**
57	Redundant Return Value	Verify that return values avoid redundancy.	**Observation 2**

Tools

While testing Metacash V2, the following tools were employed:

Tool	Description	Link
Remix	Solidity IDE	https://remix.ethereum.org
Mythril Classic	Security analysis tool for Ethereum smart contracts	https://github.com/ConsenSys/mythril-classic
Slither	Static Analyzer for Solidity	https://github.com/crytic/slither

Next Steps

This section contains our recommendations for areas that may benefit from additional testing. For each section, we describe why it is important to test these sections, either more thoroughly or for the first time.

Retest After Remediation - A retest of the Metacash V2 smart contracts is recommended to be performed when the problems found as a result of this test have been remediated. This validates the remediation put in place and ensures that other vulnerabilities have not been introduced in the course of remediation. Additionally, as new features are added to the smart contract, additional auditing is recommended to ensure the new functionality does not expose users to additional risk.

Other Integration Points - Numerous other applications are used by the user to interact with the contract. Each one of these interfacing applications should undergo a security review to make sure they are not putting user funds at risk. Every integration point is another opportunity for an attacker to get in or take control of a system, so each should receive the same level of scrutiny. Specifically the recently released Android application and the soon to be released iOS application should be reviewed, along with any server code or APIs used by relayers.

The author(s) of this report retain no responsibility for any unidentified vulnerabilities, known or unknown, in the target application. Inconsistencies, errors, and reproducibility problems associated with this report should be directed through the contact person to the testers indicated at the beginning of this report.