ABDK CONSULTING

SMART CONTRACT AUDIT

Brink

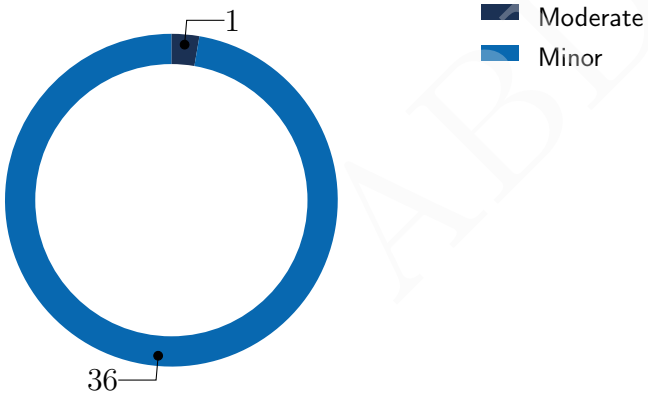
Solidity

abdk.consulting

SMART CONTRACT AUDIT CONCLUSION

by Mikhail Vladimirov and Dmitry Khovratovich
26th October 2021

We've been asked to review the 12 files in a github repo. We found 1 moderate and a few less important issues.



Findings

ID	Severity	Category	Status
CVF-1	Minor	Procedural	Fixed
CVF-2	Minor	Procedural	Fixed
CVF-3	Minor	Procedural	Fixed
CVF-4	Minor	Procedural	Info
CVF-5	Minor	Flaw	Fixed
CVF-6	Minor	Documentation	Info
CVF-7	Minor	Bad datatype	Fixed
CVF-8	Moderate	Flaw	Info
CVF-9	Minor	Suboptimal	Info
CVF-10	Minor	Readability	Fixed
CVF-11	Minor	Suboptimal	Info
CVF-12	Minor	Bad naming	Fixed
CVF-13	Minor	Suboptimal	Fixed
CVF-14	Minor	Suboptimal	Fixed
CVF-15	Minor	Flaw	Fixed
CVF-16	Minor	Bad naming	Fixed
CVF-17	Minor	Documentation	Info
CVF-18	Minor	Documentation	Info
CVF-19	Minor	Suboptimal	Info
CVF-20	Minor	Procedural	Fixed
CVF-21	Minor	Suboptimal	Fixed
CVF-22	Minor	Suboptimal	Fixed
CVF-23	Minor	Readability	Fixed
CVF-24	Minor	Flaw	Fixed
CVF-25	Minor	Suboptimal	Info
CVF-26	Minor	Flaw	Info
CVF-27	Minor	Procedural	Info

ID	Severity	Category	Status
CVF-28	Minor	Flaw	Info
CVF-29	Minor	Flaw	Fixed
CVF-30	Minor	Suboptimal	Info
CVF-31	Minor	Bad naming	Info
CVF-32	Minor	Bad naming	Fixed
CVF-33	Minor	Suboptimal	Info
CVF-34	Minor	Documentation	Fixed
CVF-35	Minor	Suboptimal	Info
CVF-36	Minor	Flaw	Info
CVF-37	Minor	Suboptimal	Info

Contents

1 Document properties

Version

Version	Date	Author	Description
0.1	October 25, 2021	D. Khovratovich	Initial Draft
0.2	October 25, 2021	D. Khovratovich	Minor revision
1.0	October 26, 2021	D. Khovratovich	Release

Contact

D. Khovratovich

khovratovich@gmail.com

2 Introduction

The following document provides the result of the audit performed by ABDK Consulting at the customer request. The audit goal is a general review of the smart contracts structure, critical/major bugs detection and issuing the general recommendations. We have reviewed the contracts at REPO:

- DeployAndExecute.sol
- ISingletonFactory.sol
- EIP1271Validator.sol
- EIP712SignerRecovery.sol
- ReplayBits.sol
- LimitSwapVerifier.sol
- Account.sol
- CallExecutor.sol
- ProxyStorage.sol
- CancelVerifier.sol
- Proxy.sol
- TransferHelper.sol

2.1 About ABDK

ABDK Consulting, established in 2016, is a leading service provider in the space of blockchain development and audit. It has contributed to numerous blockchain projects, and co-authored some widely known blockchain primitives like Poseidon hash function. The ABDK Audit Team, led by Mikhail Vladimirov and Dmitry Khovratovich, has conducted over 40 audits of blockchain projects in Solidity, Rust, Circom, C++, JavaScript, and other languages.

2.2 Disclaimer

Note that the performed audit represents current best practices and smart contract standards which are relevant at the date of publication. After fixing the indicated issues the smart contracts should be re-audited.

2.3 Methodology

The methodology is not a strict formal procedure, but rather a collection of methods and tactics that combined differently and tuned for every particular project, depending on the project structure and and used technologies, as well as on what the client is expecting from the audit. In current audit we use:

- **General Code Assessment**. The code is reviewed for clarity, consistency, style, and for whether it follows code best practices applicable to the particular programming language used. We check indentation, naming convention, commented code blocks, code duplication, confusing names, confusing, irrelevant, or missing comments etc. At this phase we also understand overall code structure.

- **Entity Usage Analysis**. Usages of various entities defined in the code are analysed. This includes both: internal usages from other parts of the code as well as potential external usages. We check that entities are defined in proper places and that their visibility scopes and access levels are relevant. At this phase we understand overall system architecture and how different parts of the code are related to each other.

- **Access Control Analysis**. For those entities, that could be accessed externally, access control measures are analysed. We check that access control is relevant and is done properly. At this phase we understand user roles and permissions, as well as what assets the system ought to protect.

- **Code Logic Analysis**. The code logic of particular functions is analysed for correctness and efficiency. We check that code actually does what it is supposed to do, that algorithms are optimal and correct, and that proper data types are used. We also check that external libraries used in the code are up to date and relevant to the tasks they solve in the code. At this phase we also understand data structures used and the purposes they are used for.

3 Detailed Results

3.1 CVF-1

- **Severity** Minor
- **Status** Fixed
- **Category** Procedural
- **Source** DeployAndExecute.sol

Description Should be "ô.7.0" or "ô.7.6" according to a common best practice. Also relevant for the next files: ISingletonFactory.sol, EIP1271Validator.sol, EIP712SignerRecovery.sol, IERC1271.sol, ReplayBits.sol, LimitSwapVerifier.sol, Account.sol, CallExecutor.sol, CallExecutor.sol, TransferVerifier.sol, ProxyStorage.sol, ProxySettable.sol, CancelVerifier.sol, Proxy.sol, ProxyGettable.sol, TransferHelper.sol.

Listing 1:

```
2 solidity >=0.7.6;
```

3.2 CVF-2

- **Severity** Minor
- **Status** Fixed
- **Category** Procedural
- **Source** DeployAndExecute.sol

Recommendation This variable should be declared as immutable.

Listing 2:

```
10 ISingletonFactory singletonFactory;
```

3.3 CVF-3

- **Severity** Minor
- **Status** Fixed
- **Category** Procedural
- **Source** DeployAndExecute.sol

Description There is not access modifier for this variable, so it will be internal by default.
Recommendation Consider specifying access level explicitly.

Listing 3:

```
10 ISingletonFactory singletonFactory;
```

3.4 CVF-4

- **Severity** Minor
- **Status** Info
- **Category** Procedural
- **Source** DeployAndExecute.sol

Recommendation It would be more convenient and less error-prone is the "deploy" function would revert on unsuccessful deployment attempt, rather than just return zero.
Client Comment We're using this canonical deployment of SingletonFactory https://etherscan.io/address/0xce0042B868300000d44A59004Da54A005ffdcf9f#code from https://eips.ethereum.org/EIPS/eip-2470

Listing 4:

```
26 address createdContract = singletonFactory.deploy(initCode, salt
   ↪ );

29 require(createdContract != address(0), "DeployAndExecute:
   ↪ contract not deployed");
```

3.5 CVF-5

- **Severity** Minor
- **Status** Fixed
- **Category** Flaw
- **Source** DeployAndExecute.sol

Description The returned data is propagated to the caller on unsuccessful call, but not on a successful one.
Recommendation Consider propagating in any case.

Listing 5:

```
35 returndatacopy(0, 0, returndatasize())
```

3.6 CVF-6

- **Severity** Minor
- **Status** Info
- **Category** Documentation
- **Source** ISingletonFactory.sol

Description This function returns zero address on unsuccessful deployment attempt, which is not obvious.
Recommendation Consider adding a documentation comment explaining this. Alternatively, change the function semantics to revert on unsuccessful deployment.
Client Comment We're using this canonical deployment of SingletonFactory https://etherscan.io/address/0xce0042B868300000d44A59004Da54A005ffdcf9f#code from https://eips.ethereum.org/EIPS/eip-2470

Listing 6:

```
5 function deploy(bytes memory _initCode, bytes32 _salt) external
    ↪ returns (address payable createdContract);
```

3.7 CVF-7

- **Severity** Minor
- **Status** Fixed
- **Category** Bad datatype
- **Source** EIP1271Validator.sol

Recommendation This immutable variable should be turned into a compile-time constant and moved to the "IERC1271" interface.

Listing 7:

```
11 bytes4 constant internal MAGICVALUE = 0x1626ba7e;
```

3.8 CVF-8

- **Severity** Moderate
- **Status** Info
- **Category** Flaw
- **Source** EIP712SignerRecovery.sol

Description Using an immutable chain ID passed as a constructor argument doesn't guarantee that the chain ID used by the contract will always be equal to the real chain ID of the blockchain where the transactions are mined. Even if at the deployment time the correct chain ID will be passed, the blockchain chain ID could change in the future after some hard fork.
Recommendation Consider obtaining the real chain ID via "chainid" opcode every time it is needed.
Client Comment We're using the same pattern that DAI uses for EIP712 permit. See https://etherscan.io/address/0x6b175474e89094c44da98b954eedeac495271d0f#code#L110 The value of chainId that we're using is actually arbitrary, it doesn't have to match the real chainId. It's really just used to make sure that a message signed for one chain cannot be replayed on another. It's up to us to deploy to each chain with a different chainId.
If we used the actual chainId value here, and the chainId was changed by a hardfork, it would invalidate all existing signed messages

Listing 8:

```
10 uint256 internal immutable CHAIN_ID;
```

3.9 CVF-9

- **Severity** Minor
- **Status** Info
- **Category** Suboptimal
- **Source** EIP712SignerRecovery.sol

Description This hash doesn't depend on anything that could change after the deployment, so it could be calculated once in the constructor and stored in and immutable variable. Even in case chain ID will be obtain every time, it still worth storing this hash in an immutable variable together with the chain ID used to calculate it, and reuse the stored hash in case the chain ID didn't change.
Client Comment It actually does depend on one thing that is variable - https://github.com/brinktrade/brink-core/blob/39e726d507e6d47f9e700223d6f9b057b408ee80/contracts/Account/EIP712SignerRecovery.sol#L29 the address of the contract. Because this code is used by Brink proxy accounts that delegatecall to Account.sol, the address in this context will be the address of the Brink proxy account.

Listing 9:

```
23 // hash the EIP712 domain separator
   keccak256(abi.encode(
     keccak256("EIP712Domain(string name,string version,uint256
       ↪ chainId,address verifyingContract)"),
     keccak256("BrinkAccount"),
     keccak256("1"),
     CHAIN_ID,
     address(this)
30 )),
```

3.10 CVF-10

- **Severity** Minor
- **Status** Fixed
- **Category** Readability
- **Source** EIP712SignerRecovery.sol

Description This line looks like a plain assignment, while it actually returns a value from the function.
Recommendation Consider using "return" statement.
Client Comment .

Listing 10:

```
35 signer = ECDSA.recover(messageHash, signature);
```

3.11 CVF-11

- **Severity** Minor
- **Status** Info
- **Category** Suboptimal
- **Source** ReplayBits.sol

Description This library implements a replay protection bitmap shared among all the verifiers, but each verifier is responsible for using it properly. This approach is inflexible and error-prone, as at one hand all verifiers has to use the same pattern, while different verifiers may have different requirements; while an the other hand each verifier has to implement this pattern on its own and incorrect implementation in one verifier may affect other verifiers.
Recommendation Consider, for example, a limit order verifier that allows an order to be partially filled multiple times until either expired or fully filled. Such verifier would need to store a number for each order, not just a single bit, so such verifier would need to allocate space in the storage shared with other verifiers, potentially colliding with other verifiers. Consider giving each verifier its own dedicated storage space (i.e. based on the address of the verifier contract), where each verifier may store arbitrary data without affecting other verifiers. This could be implemented as a pair of functions: getVerifierData (address verifier, bytes32 ptr) returns (bytes32) setVerifierData (address verifier; bytes32 ptr, bytes32 data) A verifier may store its own address in an immutable variable at deployment time and use it later when calling the functions mentioned above.
Client Comment There is potential for storage collisions across verifiers, but there is also potential for collisions within the same verifier because an account owner can sign many messages with the same verifier contract and function call. For example, multiple ethToToken calls can be signed to the LimitSwapVerifier, with the same bit. These bit collisions can actually be desirable, since they allow the account owner to create "one cancels the other" orders (OCO), where successful execution of one order causes other orders to become unexecutable.
A developer in the future implementing partial-fill limit orders would probably opt to use a different method of storage (other than what is provided by https://github.com/brinktrade/brink-verifiers/blob/ae7f98d8650fe357feea21a764e1a5ac47f3ba39/contracts/Libraries/Bit.sol).
A verifier has the ability to manipulate any storage on an account that allows a delegatecall to that verifier. Adding verifier storage scoping functions to Account.sol would not restrict verifiers from modifying storage in other ways. Account.sol also has no knowledge of verifiers, as it only handles signature verification and proxying of calldata.

Listing 11:

```
5 @notice Handles storage and loads for replay protection bits
```

3.12 CVF-12

- **Severity** Minor
- **Status** Fixed
- **Category** Bad naming
- **Source** ReplayBits.sol

Recommendation The library has no functionality specific for the replay protection, so just 'Bit' or 'Bitmask' would be a better name.

Listing 12:

```
8 ReplayBits {
```

3.13 CVF-13

- **Severity** Minor
- **Status** Fixed
- **Category** Suboptimal
- **Source** ReplayBits.sol

Description Here the bitmap is loaded from the storage twice.
Recommendation Consider caching the loaded value for the future use.

Listing 13:

```
31 require(!bitUsed(bitmapIndex, bit), "BIT_USED");

33 uint256 updatedBitmap = loadUint(ptr) | bit;
```

3.14 CVF-14

- **Severity** Minor
- **Status** Fixed
- **Category** Suboptimal
- **Source** ReplayBits.sol

Description The expression "bitmapPtr(bitmapIndex}" is calculated twice: once inside the "bitUsed" call, and another time after this call.
Recommendation Consider refactoring the code to calculate this expression only once.

Listing 14:

```
31 require(!bitUsed(bitmapIndex, bit), "BIT_USED");
   bytes32 ptr = bitmapPtr(bitmapIndex);
```

3.15 CVF-15

- **Severity** Minor
- **Status** Fixed
- **Category** Flaw
- **Source** ReplayBits.sol

Description The bitmap at given index is loaded from the storage twice: once inside the "bitUsed" call and another time after the call.
Recommendation Consider refactoring the code to load this value only once. For example like this: require (validBit (bit)); uint bitmapPtr = bitmapPtr (bitmapIndex); uint oldBitmap = loadUint (bitmapPtr); uint newBitmap = oldBitmap | bit; require (newBitmap != oldBitmap); assembly { sstore (bitmapPtr, newBitmap) }

Listing 15:

```
31 require(!bitUsed(bitmapIndex, bit), "BIT_USED");
   bytes32 ptr = bitmapPtr(bitmapIndex);
   uint256 updatedBitmap = loadUint(ptr) | bit;
```

3.16 CVF-16

- **Severity** Minor
- **Status** Fixed
- **Category** Bad naming
- **Source** LimitSwapVerifier.sol

Recommendation The function name should be 'metaDelegateCall_EIP1271'.

Listing 16:

```
11 @notice These functions should be executed by
   ↪ metaPartialSignedDelegateCall() on Brink account proxy
   ↪ contracts
```

3.17 CVF-17

- **Severity** Minor
- **Status** Info
- **Category** Documentation
- **Source** LimitSwapVerifier.sol

Description The caller code does not check the length of the signed data, so it is not guaranteed that exactly these parameters are signed. It may happen that some other are signed, or some last ones are unsigned.
Recommendation Consider adding this to the documentation.
Client Comment This is documented in the verifier as the recommended way the verifier should be used. There are no restrictions to what an account owner is allowed to sign.

Listing 17:

```
22 /// @notice This should be executed by
     ↪ metaPartialSignedDelegateCall() with the following signed
     ↪ and unsigned params

51 /// @notice This should be executed by
     ↪ metaPartialSignedDelegateCall() with the following signed
     ↪ and unsigned params

79 /// @notice This should be executed by
     ↪ metaPartialSignedDelegateCall() with the following signed
     ↪ and unsigned params
```

3.18 CVF-18

- **Severity** Minor
- **Status** Info
- **Category** Documentation
- **Source** LimitSwapVerifier.sol

Description It seems to be possible to have ‘tokenIn==tokenOut‘ i.e. just provide a flash loan of the token. In this case ‘tokenOutAmount‘ will be not (at least) the full amount to be received back, but rather the premium.
Recommendation Consider adding a comment on how ‘tokenOutAmount‘ should be set in this usecase.
Client Comment It’s a really interesting use case that we hadn’t considered. We’ll consider adding it to more future docs and developer guides

Listing 18:

```
28 /// @param tokenOutAmount Amount of tokenOut required to be
     ↪ received [signed]
```

3.19 CVF-19

- **Severity** Minor
- **Status** Info
- **Category** Suboptimal
- **Source** LimitSwapVerifier.sol

Recommendation It would be cheaper to pass a single bit index argument instead of bitmap index and bit arguments.
Client Comment We think the complexity associated with combining these args would not be worth the gas savings.

Listing 19:

```
33 uint256 bitmapIndex, uint256 bit, IERC20 tokenIn, IERC20
     ↪ tokenOut, uint256 tokenInAmount, uint256 tokenOutAmount,

61 uint256 bitmapIndex, uint256 bit, IERC20 token, uint256
     ↪ ethAmount, uint256 tokenAmount, uint256 expiryBlock,

89 uint256 bitmapIndex, uint256 bit, IERC20 token, uint256
     ↪ tokenAmount, uint256 ethAmount, uint256 expiryBlock,
```

3.20 CVF-20

- **Severity** Minor
- **Status** Fixed
- **Category** Procedural
- **Source** LimitSwapVerifier.sol

Description There is no such check in the token swap call.
Recommendation Consider having the same checks everywhere for consistency.

Listing 20:

```
67 require(address(this).balance >= ethAmount, "NOT_ENOUGH_ETH");
```

3.21 CVF-21

- **Severity** Minor
- **Status** Fixed
- **Category** Suboptimal
- **Source** LimitSwapVerifier.sol

Description This check is redundant, as the attempt to transfer ether along with call with anyway fail in case of insufficient balance.

Listing 21:

```
67 require(address(this).balance >= ethAmount, "NOT_ENOUGH_ETH");
```

3.22 CVF-22

- **Severity** Minor
- **Status** Fixed
- **Category** Suboptimal
- **Source** Account.sol

Recommendation Consider using the hash calculating expression instead of the hardcoded value, as Solidity is smart enough to calculate hashes at compile time.

Listing 22:

```
12 /// @dev keccak256("MetaDelegateCall(address to,bytes data)")
   bytes32 internal constant META_DELEGATE_CALL_TYPEHASH =
     0
       ↪ x023ce5d01636bb12b4ffde3c4f5a66fb1044aa0dbc251394e60f0a26f1591043
       ↪ ;

17 /// @dev keccak256("MetaDelegateCall_EIP1271(address to,bytes
     ↪ data)")
   bytes32 internal constant META_DELEGATE_CALL_EIP1271_TYPEHASH =
     0
       ↪ x1d3b50d88adeb95016e86033ab418b64b7ecd66b70783b0dca7b0afc8bfb8a1e
       ↪ ;
```

3.23 CVF-23

- **Severity** Minor
- **Status** Fixed
- **Category** Readability
- **Source** Account.sol

Recommendation It is a good practice to put a comment into an empty block to explain why the block is empty.

Listing 23:

```
22 constructor(uint256 chainId_) EIP712SignerRecovery(chainId_) { }
```

3.24 CVF-24

- **Severity** Minor
- **Status** Fixed
- **Category** Flaw
- **Source** Account.sol

Description The returned data is propagated to the caller on unsuccessful call, but not on a successful one.
Recommendation Consider propagating the returned data in any case.

Listing 24:

```
40    returndatacopy(0, 0, returndatasize())

54    returndatacopy(0, 0, returndatasize())

81  let result := delegatecall(gas(), to, add(callData, 0x20), mload
      ↪ (callData), 0, 0)
```

3.25 CVF-25

- **Severity** Minor
- **Status** Info
- **Category** Suboptimal
- **Source** Account.sol

Description These functions have a common part.
Recommendation Consider extracting it into a utility function.
Client Comment Increases gas cost - will keep as is for gas optimization.

Listing 25:

```
49  function delegateCall(address to, bytes memory data) external {

69  function metaDelegateCall(

98  function metaDelegateCall_EIP1271(
```

3.26 CVF-26

- **Severity** Minor
- **Status** Info
- **Category** Flaw
- **Source** Account.sol

Description These functions are very dangerous as they could be used to self destruct the master contract effectively breaking up all the proxies delegating to it. Currently an "Account.sol" master contract is protected by the fact the its proxy owner storage variable is zero and it is impossible to change the owner. However, this protection is implicit.
Recommendation Consider explicitly forbidding calling the master contract directly, i.e. not through a delegate call.
Client Comment Increases gas cost - will keep as is for gas optimization
We're choosing to accept the implicit protections (proxyOwner checks) as sufficiently secure. It's impossible for this ECDS.recover() to return a zero address https://github.com/brinktrade/brink-core/blob/c5b831d9e0239ff119034403bb93cae17ddd4590/contracts/Account/EIP712SignerRecovery.sol#L3
Also impossible for EIP1271 validation call to succeed if 'signer' is zero address https://github.com/brinktrade/brink-core/blob/c5b831d9e0239ff119034403bb93cae17ddd4590/contracts/Account/EIP1271Validator.sol#L19
. This makes it impossible to pass validation checks and execute arbitrary calldata on the canonical Account.sol deployment.

Listing 26:

```
49 function delegateCall(address to, bytes memory data) external {

69 function metaDelegateCall(

98 function metaDelegateCall_EIP1271(
```

3.27 CVF-27

- **Severity** Minor
- **Status** Info
- **Category** Procedural
- **Source** Account.sol

Description It is unclear how much trust the signer should put into the ‘to‘ contract.
Recommendation Consider providing explicit conditions to check.
Client Comment The signer needs to fully trust the ‘to‘ contract, just as an EOA needs to fully trust the ‘to‘ contract of a tx it signs and submits to the Ethereum network. In both cases, a malicious ‘to‘ contract can steal the signer's funds. There is no way to explicitly check for this programmatically.

Listing 27:

```
66 /// @notice WARNING: The ‘to‘ contract is responsible for secure
   ↪ handling of the call provided in the encoded

95 /// @notice WARNING: The ‘to‘ contract is responsible for secure
   ↪ handling of the call provided in the encoded
```

3.28 CVF-28

- **Severity** Minor
- **Status** Info
- **Category** Flaw
- **Source** CallExecutor.sol

Description This is not actually enforced in the code being audited.
Client Comment Using CallExecutor is the recommended way for verifiers to execute arbitrary calldata (see https://github.com/brinktrade/brink-verifiers/blob/ae7f98d8650fe357feea21a764e1a5ac47f3ba39/contracts/Verifiers/LimitSwapVerifier.sol#L45 as an example). This can't be enforced at the Account level.

Listing 28:

```
17 * but also makes other malicious calls that rely on msg.sender.
   ↪ Forcing all
   * unsigned data execution to be done through a CallExecutor
   ↪ ensures that an
   * attacker cannot impersonate the users's account.
```

3.29 CVF-29

- **Severity** Minor
- **Status** Fixed
- **Category** Flaw
- **Source** CallExecutor.sol

Description The returned data is propagated to the caller on unsuccessful delegation attempts, but not on the successful ones.
Recommendation Consider propagating the returned data in any case.

Listing 29:

```
35 returndatacopy(0, 0, returndatasize())

53 returndatacopy(0, 0, returndatasize())
```

3.30 CVF-30

- **Severity** Minor
- **Status** Info
- **Category** Suboptimal
- **Source** ProxyStorage.sol

Recommendation Making these variables public would avoid the need for their excplicit getters.
Client Comment Increases gas cost - will keep as is for gas optimization

Listing 30:

```
8 address internal _implementation;
  address internal _owner;
```

3.31 CVF-31

- **Severity** Minor
- **Status** Info
- **Category** Bad naming
- **Source** CancelVerifier.sol

Description The semantics of ‘cancelling‘ and ‘use‘ is quite unclear for this contract. Its functionality has nothing to do with cancellation but rather with setting certain bits.
Recommendation Consider renaming the contract to represent its functionality better.
Client Comment The primary use case for this verifier is cancelling messages/orders, so we would like to keep the naming as such

Listing 31:

```
6 @title Verifier for cancel of messages signed with a bitmapIndex
    ↪ and bit
  @notice Uses the ReplayBits library to use the bit, which
    ↪ invalidates messages signed with the same bit
```

3.32 CVF-32

- **Severity** Minor
- **Status** Fixed
- **Category** Bad naming
- **Source** CancelVerifier.sol

Recommendation Events are usually named via nouns, such as "Cancel".

Listing 32:

```
9 event Cancelled (uint256 bitmapIndex, uint256 bit);
```

3.33 CVF-33

- **Severity** Minor
- **Status** Info
- **Category** Suboptimal
- **Source** CancelVerifier.sol

Recommendation In case the bitmapIndex fits into 248 bits, these two arguments could be combined into one to make calls cheaper.
Client Comment We think the complexity associated with combining these args would not be worth the gas savings.

Listing 33:

```
14 function cancel(uint256 bitmapIndex, uint256 bit) external {
```

3.34 CVF-34

- **Severity** Minor
- **Status** Fixed
- **Category** Documentation
- **Source** Proxy.sol

Recommendation It is a good practice to put a comment into an empty block to explain why the block is empty.

Listing 34:

```
47 receive() external payable { }
```

3.35 CVF-35

- **Severity** Minor
- **Status** Info
- **Category** Suboptimal
- **Source** TransferHelper.sol

Recommendation Consider using hash expressions instead of hardcoded function selectors like this: bytes4(keccak256("transferFrom(address,address,uint256)"))
Client Comment Increases gas cost - will keep as is for gas optimization.

Listing 35:

```
12 (bool success, bytes memory data) = token.call(abi.
    ↪ encodeWithSelector(0x095ea7b3, to, value));

22 (bool success, bytes memory data) = token.call(abi.
    ↪ encodeWithSelector(0xa9059cbb, to, value));

33 (bool success, bytes memory data) = token.call(abi.
    ↪ encodeWithSelector(0x23b872dd, from, to, value));
```

3.36 CVF-36

- **Severity** Minor
- **Status** Info
- **Category** Flaw
- **Source** TransferHelper.sol

Description This check will pass even if the callee does not return any value, which does not seem to be compliant with EIP-20.
Client Comment This library was originally developed by Uniswap to deal with non EIP-20 compliant tokens https://github.com/Uniswap/solidity-lib/blob/master/contracts/libraries/TransferHelper.sol

Listing 36:

```
13 require(success && (data.length == 0 || abi.decode(data, (bool))
    ↪ ), 'APPROVE_FAILED');

23 require(success && (data.length == 0 || abi.decode(data, (bool))
    ↪ ), 'TRANSFER_FAILED');

34 require(success && (data.length == 0 || abi.decode(data, (bool))
    ↪ ), 'TRANSFER_FROM_FAILED');
```

3.37 CVF-37

- **Severity** Minor
- **Status** Info
- **Category** Suboptimal
- **Source** TransferHelper.sol

Description These statements hide original error message from the called contract in case it reverted the transaction.
Recommendation Consider rethrowing the returned data as is in case success is false.
Client Comment This library was originally developed by Uniswap to deal with non EIP-20 compliant tokens https://github.com/Uniswap/solidity-lib/blob/master/contracts/libraries/TransferHelper.sol

Listing 37:

```
13 require(success && (data.length == 0 || abi.decode(data, (bool))
   ↪ ), 'APPROVE_FAILED');

23 require(success && (data.length == 0 || abi.decode(data, (bool))
   ↪ ), 'TRANSFER_FAILED');

34 require(success && (data.length == 0 || abi.decode(data, (bool))
   ↪ ), 'TRANSFER_FROM_FAILED');

39 require(success, 'ETH_TRANSFER_FAILED');
```