Brink Security Review

Engagement I

Reviewers
Gerard, Lead
Hari, Lead
Max, Researcher

December 24, 2021

1 Executive Summary

Over the course of a week (a total of 3 engineering weeks), Brink engaged with Spearbit to review Brink-core and Brink-verifiers. We found a total of 25 issues with Brink. The most critical issue is 4.1.1. All issues requiring a fix has already been fixed by Brink, or falls under the risk assumptions for the protocol. Overall, Spearbit found the codebase to be of very high quality.

Repository	**Commit**
Brink-core	c5b831d9e0239ff119034403bb93cae17ddd4590
Brink-verifiers	e3966482c42eb045794a790b6ca1f9d0bd914a9a

Summary

Type of Project	Automation, DeFi
Timeline	November 7th, 2021 - November 17th, 2021
Methods	Manual Review, Computer Aided Verification
Documentation	High
Testing Coverage	High

Total Issues

High Risk	1
Medium Risk	3
Low Risk	1
Gas Optimizations and Informational	20

Contents

2 Spearbit

Spearbit is a decentralized network of expert Web3 security engineers. Together, we help secure the Web3 ecosystem. We offer security reviews and related services to Web3 projects. Our network has experience at every part of the stack, including protocol design, smart contracts, and the Solidity compiler itself. Spearbit brings in untapped security talent: expert freelance auditors want flexibility to work on interesting projects together. Learn more about us at spearbit.com.

3 Introduction

The Brink protocol is designed for automating conditional orders on EVM compatible chains. For example, one can sign an order for swapping a certain amount of ETH for DAI if the price of ETH is above a certain predefined value in DAI. This would then get executed automatically when the condition is met.

The core architecture is designed in the following way:

- `Proxy.sol`: a smart contract wallet `Proxy.sol` that delegate-calls the implementation `Account.sol`.
- `Account.sol`: The implementation contract, that is always meant to be delegatecalled from the proxy. The most critical function here is metaDelegateCall, which allows external parties (executors) to execute transactions.

The focus of the security review was on the following:

1. Could the deployment of `Account` contract be made inaccessible using `selfdestruct`, for example, by calling `metaDelegateCall`? This would lead to lockup of funds.
2. Bugs related to overwriting storage slots corresponding to `_implementation` and `_owner`?
3. Can the signed transactions be replayed on the same chain? Can they be replayed on the other chains?
4. Gas optimizations.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of brink according to the specific commit by a three person

team. Any modifications to the code will require a new security review.

4 Findings

4.1 High Risk

4.1.1 The storage slots corresponding to `_implementation` and `_owner` could be accidentally overwritten

Severity: High Risk

Context: `ProxyStorage.sol#L7-L10`

```
contract ProxyStorage {
 address internal _implementation;
 address internal _owner;
}
```

The state variables `_implementation` and `_owner` are at slots `0` and `1`. The protocol architecture relies on executors calling `metaDelegateCall` to verifier contracts. Therefore, the storage slots are shared with the storage space of the verifiers.

Risk: The first 2 state variables declared in a verifier will overlap with `_implementation` and `_owner`. Accidentally changing these variables will result in changing the implementation and changing the owner. Funds could be stolen or made inaccessible (accidentally or on purpose).

Recommendations:

1. Store the variables at a quasi random memory location determined by a Keccak-256 hash. This pattern is used in `Bit.sol#L33`.
2. Let the verifiers inherit from `ProxyStorage.sol` so that the variables `_implementation` and `_owner` are mapped in storage and will not be overwritten accidentally. It is recommended to check the storage layout using `solc --storage-layout` or the equivalent standard-json flag to verify that this is indeed the case; we recommend building a small tool for doing this.
3. Store the `_implementation` and the `_owner` as constants or immutable. This allows for a smaller proxy that saves gas. However, this requires some architectural changes.

Brink: We chose to fix, but took a slightly different approach than any of the recommended fixes. Our fix is closest to recommendation 3.While recommendations 1 and 2 would have been simpler to implement, and may prevent an accidental overwrite of the `_implementation` and `_owner` values, fixing in this way would not prevent an attacker from overwriting these values by tricking an account owner into signing permissions for the overwrite.

We wanted to make the values fully immutable after `Proxy` deployment. We updated `Proxy.sol` to include two constants: `ACCOUNT_IMPLEMENTATION`, which is the deterministic address for the `Account.sol` deployment and will be consistent across all chains, and `OWNER`, which is set as a placeholder address `0xfefeFEFeFEFEFEFEFeFefefefefefEfEfefefEfe`. We created `AccountFactory.sol` which dynamically creates `Proxy` init code, inserting the actual owner address at the same location as the `OWNER` placeholder. When `Proxy` makes a `delegatecall` to `Account`, the value of `OWNER` is read using `extcodecopy` (`ProxyGettable.sol`). Constant and immutable storage read from a contract executed via delegate will be read from the implementation contract (`Account.sol`), not the calling contract (`Proxy`). Using `extcodecopy` lets us read a constant value from the Proxy deployed bytecode.

Note: In order for us to compile `Proxy.sol` with the `OWNER` placeholder included in the deployed bytecode, we had to include a reference to it in the contract (outside of the constructor). We found `Proxy.sol#L40` to be the most gas efficient way to accomplish this. We believe it has no security impact on the Proxy, and only increases the gas for incoming ETH transfers by a negligible amount.

Brink Update: We updated to use Minimal Proxy with the owner address appended at the end of the deployed bytecode commit 0ed725b.

Spearbit: We welcome this change, and performed a follow up review focussed on the modification on Minimal Proxy contract. Overall, we agree that the issues are resolved. Detailed comments can be found in the follow up report.

4.2 Medium Risk

4.2.1 Risk of replay attacks across chains

Severity: Medium Risk, Gas Optimization

Context: `EIP712SignerRecovery.sol#L12-L14`

Currently, for deploying on EVM compatible chains, the unique identifier `_chainId` is specified by Brink. The users need to trust the deployer, i.e. Brink

to have unique values of `_chainId` for different chains. If the deployer violates this condition, there is a risk of replay attacks, where signed messages on one chain may be replayed on another chain.

Recommendation: Read `_chainId` on chain directly, using `block.chainid` (available from Solidity 0.8.0) or using `chainid()` in inline assembly for older solidity versions, rather than as a constructor parameter. It is also a gas optimization (saves 1 gas).

```
constructor(uint256_chainId) EIP7125SignerRecovery(chainId_){ ... }
```

Brink: Fixed in commit feef7d9. There is potential risk that a chain could hardfork a change to the `chainID` value, which would invalidate all signed messages. The main benefit and reason for the fix was so the `Account.sol` address can be consistent across all chains. Since `Proxy.sol` now sets this address to the `ACCOUNT_IMPLEMENTATION` constant, this keeps all Proxy addresses consistent across all chains as well.

Spearbit: Resolved. We think that a hardfork / protocol-upgrade that changes the value of `chainid` is extremely unlikely on almost all big chains, therefore this risk is acceptable.

4.2.2 Selfdestruct risks in `delegateCall()`

Severity: Medium Risk

Context: `Account.sol#L48-L57`

```
function delegateCall(address to, bytes memory data) external {
    require(proxyOwner() == msg.sender, "NOT_OWNER");
    assembly {
      let result := delegatecall(gas(), to, add(data, 0x20), mload(data), 0, 0)
      if eq(result, 0) {
        returndatacopy(0, 0, returndatasize())
        revert(0, returndatasize())
      }
    }
  }
```

The address where `Account.sol` gets deployed can be directly called. There is the risk of a potential `selfdestruct`, which would result in user wallets getting bricked. This risk depends on the access control of the functions `delegateCall`, `metaDelegateCall`, and `metaDelegateCall_EIP1271`. However, we couldn't find a hole in the access control. We would still recommend the following changes:

1. Explicitly enforce that these functions are only delegatecalled. The following contract demonstrates how this can be achieved.

```
abstract contract OnlyDelegateCallable {
    address immutable deploymentAddress = address(this);
    modifier onlyDelegateCallable() {
        require(address(this) != deploymentAddress);
        _;
    }
}
```

2. Note that the Solidity compiler enforces call protection for libraries, i.e., the compiler automatically adds an equivalent of the above `onlyDelegateCallable` modifier to state modifying functions. Changing the contract to a library would.

3. Deploy `Account.sol` via `CREATE2` so it can be redeployed if necessary.

Note: Assuming that the current access control can be broken, and that the address corresponding to the `Account.sol` contract holds funds, then an attacker can steal this funds from this address. However, this is not the most significant risk.

Brink: Fixed in commit 3afeaf1.

Spearbit: Resolved.

4.2.3 Check for non-zero data

Severity: Medium Risk

Context: `Account.sol#L76-L78`

```
bytes memory callData = abi.encodePacked(data, unsignedData);
```

The function `metaDelegateCall()` does not currently check if `data` is non-empty.

Risk: If `data.length` is equal to `0`, it should call the `fallback` or `receive` function, however, a malicious verifier can redirect this call to another function by crafting `unsignedData`. For example, assume that the user signed a meta transaction to the following contract's `fallback` function:

```
contract C {
    fallback() external {
        // do something useful
    }
    function f() external {
        // steal funds
    }
}
```

The executor can append the selector of `f` to `unsignedData`, thereby executing the call to `f` instead of to the `fallback` function.

Note: similarly, if the `data` (signed) is less than 4 bytes in length, i.e., length of a function selector, executors could potentially redirect the actual call to another function in the same contract, and potentially steal funds.

Recommendation: Check `data.length` is greater than or equal to 4 bytes.

```
require(data.length > 4);
```

Brink: We chose not to implement a fix. Our reasoning is that there are many ways for a malicious verifier to attack a user's account, but all of these attacks require tricking the user into signing a message with the malicious verifier contract set as the `to` address. Even if we prevent valid signing of empty call data, users could still sign malicious messages with valid call data. Reverting on empty call data doesn't reduce the potential for this type of attack. These attacks need to be mitigated off-chain.

Spearbit: As this falls under the trust model for the protocol, i.e., trusting verifier contracts, we accept Brink's approach. See the section on verifiers that talks about the risk model.

4.3 Low Risk

4.3.1 Implement SafeERC20 to avoid non-zero to non-zero approvals

Severity: Low Risk

Context: `TransferHelper.sol#L12-L13`

```
(bool success, bytes memory data) =
↪  token.call(abi.encodeWithSelector(0x095ea7b3, to, value));
require(success && (data.length == 0 || abi.decode(data, (bool))),
↪  'APPROVE_FAILED');
```

Current function allows for non-zero to non-zero approvals, however, some ERC20 tokens would revert in such cases. Reference: approval race protections

Recommendations:

1. Implement the SafeERC20 from OpenZeppelin, in particular `safeApprove`, if it is used.
2. Replace the hex codes with `.selector`, i.e.

```
abi.encodeWithSelector(token.approve.selector, to, value);
```

3. The function `safeApprove` is currently not used. We recommend removing `safeApprove()`.

Brink: We removed `TransferHelper` from `LimitSwapVerifiers` commit 7e6df62, because transfer checks were not needed here. It is still used in `TransferVerifier`, but `safeApprove()` is not used. We will consider implementing this fix for future verifiers if `safeApprove()` is used.

Spearbit: Resolved.

4.4 Gas Optimizations and Informational

4.4.1 The function `storageLoad()` is not required

Severity: Gas Optimization

Context: `Account.sol#L25-L27`

The function `storageLoad` function is unnecessary.

Recommendation: This information can be obtained off-chain using `getStorageAt` JSON RPC method. This will reduce overall gas since the function `storageLoad` will end up in the function dispatch, making other function calls more expensive.

For additional info, here is the part where storageLoad is used from the sdk: Account.js#L269

Brink: Fixed in commit 743eea1.

Spearbit: Resolved.

4.4.2 Use `_owner` directly instead of the function `proxyOwner`

Severity: Gas Optimization

The `proxyOwner()` gettable is called several times in `Account.sol`.

Recommendation: Use `_owner` directly. If possible, consider making it immutable and query the owner using a callback to the proxy.

1. `Account.sol#L35`
2. `Account.sol#L53`
3. `Account.sol#L83`

```
require(proxyOwner() == signer, "NOT_OWNER");
```

Note: this optimization may be redundant if a newer version of solidity is used (>= 0.8.2). It has a bytecode level inliner that can inline `proxyOwner`.

Brink: No longer relevant after the changes in proxy: commit d2df98f.

Spearbit: Resolved.

4.4.3 Copy `_delegate` code to be inline in the `fallback()` function

Severity: Gas Optimization

Context: `Proxy.sol#L27-L29`

```
fallback() external payable {
   _delegate(_implementation);
}
```

Currently, the function `_delegate` is declared outside the `fallback()` function. However, inlining this function would save some gas.

Recommendation:

1. Use the code inside the `_delegate` function inside the `fallback()` function. Save around 20-30 gas.
2. Remove the `_delegate` function.

Note: the improvements for the inliner in Solidity 0.8.2 is not relevant here, i.e., this function needs to be manually inlined.

Brink: Fixed in commit 8ec3c96.

Spearbit: Resolved.

4.4.4 The functions `metaDelegateCall()` and `externalCall` can be made payable

Severity: Gas Optimization

Context:

1. `Account.sol#L34`
2. `Account.sol#L76`

Recommendation: Make the `metaDelegateCall()` function payable. This saves gas by omitting the check for the absence of ETH attached to the call.

Brink: Fixed in commit 5014986.

Spearbit: Resolved.

4.4.5 Change `memory` to `calldata`

Severity: Gas Optimization

For external function parameters, it is often more optimal to have the reference location to be `calldata` instead of `memory`. Details can be found in the appendix.

Examples:

1. The variable `signature` in the function `metaDelegateCall_EIP1271()`. Context: `Account.sol#L109-L116`.

```
function metaDelegateCall_EIP1271(
address to, bytes memory data, bytes memory signature, bytes memory
 ↪  unsignedData
) external {
  require(_isValidSignature(
    proxyOwner(),
    keccak256(abi.encode(META_DELEGATE_CALL_EIP1271_TYPEHASH, to,
 ↪  keccak256(data))),
    signature
  ), , "INVALID_SIGNATURE");
```

2. The variable `initCode` in the function `deployAndExecute`. Context: `DeployAndExecute.sol#L23`.

```
function deployAndExecute(bytes memory initCode, bytes32 salt, bytes memory
↪  execData) external {
```

Note: this is assuming that the last suggestion about having the code inline is not implemented. Note: the gas for the deployAndExecute's unit test decreased by around 1000 after this change.

3. The variable `data` in the functions `tokenToToken`, `ethToToken`, and `tokenToEth`. Context: `LimitSwapVerifier.sol#L33-L34`.

```
  function tokenToToken(
    uint256 bitmapIndex, uint256 bit, IERC20 tokenIn, IERC20 tokenOut, uint256
↪  tokenInAmount, uint256 tokenOutAmount,
    uint256 expiryBlock, address to, bytes memory data
  )
```

4. The variable `signature` in the function `_recoverSigner`. Context: `EIP712SignerRecovery.sol#L19`.

```
function _recoverSigner(bytes32 dataHash, bytes memory signature) internal view
↪  returns (address) {
```

Brink:

1. Fixed in commit 2831884 and commit 68c68ef.
2. `DeployAndExecute.sol` was replaced with `DeployAndCall.sol`. With the new `AccountFactory.sol` there is no `initCode` parameter.
3. Fixed in commit 1a0345f.
4. Fixed in commit 2831884.

Sperabit: Resolved.

4.4.6 `initCode` could be hard coded if it is always the same

Severity: Gas Optimization

Context: `DeployAndExecute.sol#L23`

```
function deployAndExecute(bytes memory initCode, bytes32 salt, bytes memory
↪  execData) external {
```

Recommendation: Hard code initCode into the contract. This saves a significant amount of gas, as well as guarantee that the right init code is used.

```
new Proxy{salt: salt}(implementation, proxyOwner);
```

If the code for the deployed contract comes from `calldata` (as is the case here), it has to go through an inefficient copy to `memory` (via a for-loop that reads 32-byte chunks of `calldata` using `calldataload`, and storing them in `memory` via `mstore`). On the other hand, if the `initCode` is available along with the runtime code of the contract and is called via `new Proxy{salt: ...}(...)`, (it does not matter if it is a `create2` or `create`), it is more efficient, because it uses `codecopy` to place the underlying `initcode` in `memory` rather than the inefficient for loop.

Note: this assumes that the `initCode` remains consistent, which is likely the case for Brink.

Brink: We created `AccountFactory.sol` to deploy Proxy account contracts with constant account implementation addresses and owners, as part of the fix. Proxy `initCode` is now dynamically created on deploy.

Spearbit: Resolved.

4.4.7 The function `proxyCall` can be removed

Severity: Informational, Gas Optimization

Context: `CallExecutor.sol#L52`

```
function proxyPayableCall(address to, bytes memory data) public
payable {
```

There isn't a need for `proxyCall` as `proxyPaybleCall` is more general. The function selector of the function `proxyCall(...)` is smaller than the selector of `proxyPayableCall`. Therefore, in the function dispatch, `proxyCall` will appear first (Note: this is not always true).

Recommendation: We recommend getting rid of `proxyCall` and just use `proxyPayableCall`, after, perhaps renaming it.

Brink: We chose not to fix because of the low impact, but may fix in future verifier contracts.

4.4.8 Gas optimization for `keccak256()`

Severity: Gas Optimization

Context: `Bit.sol#L33`

```
return keccak256(abi.encodePacked("bmp", bitmapIndex));
```

Currently, Brink.trade is using `keccak256()` every time on line 33.

Recommendation: It may be worth computing the keccak hash as a constant, and the pointer is computed by incrementing this constant. This avoids computing the Keccak-256 hash at runtime. For example, define a

```
bytes32 initialPtr = keccak256(“bmp”);
```

Then the actual pointer can be computed by:

```
uint256 ptr = initialPtr + n;
```

Note that it's critical that value of `ptr` is not the slot `0` or `1`. Otherwise, the storage slots will collide with the owner and implementation for proxy. One way to achieve this is by limiting the type of `n` to be a short unsigned integer type, for example `uint16`.

Brink: Fixed in commit 1fdf2ea.

Spearbit: Resolved.

4.4.9 Use inline assembly to avoid short-circuiting

Severity: Gas Optimization

Context: `Bit.sol#L26-L28`.

```
function validBit(uint256 bit) internal pure returns (bool) {
   return bit > 0 && bit & bit-1 == 0;
}
```

Solidity has short-circuiting for boolean operations. The way it's implemented is by using `if` (i.e., `jumpi` instruction) for each sub expression. This can be unnecessary in some cases, especially, when the sub expressions are side-effect free.

Recommendation: Implement it using inline assembly to avoid the short-circuiting:

```
isValid := and(
    iszero(iszero(bit)),
    iszero(and(bit, sub(bit, 1)))
)
```

Note: this expression is parsed as `((bit > 0) && (bit & (bit - 1))) == 0`. If this is the intended, consider adding braces to make it more explicit.

Brink: Fixed in commit cb078d3, and commit 5f487e8.

Spearbit: Resolved.

4.4.10 Use nonces for `Bit.sol`

Severity: Gas Optimization, Informational

Context: `Bit.sol#L8`

```
library Bit { ... }
```

Currently, for replay protection, random storage slots are reused up to 256 times, using 1 bit per transaction. With a large number of transactions more and more storage slots are used.

Recommendation: Use nonces for storing transactions that are currently in flight. For example, assuming that there can only be at most 10 in-flight transactions, consider a state variable `uint256[10] nonces`. For each in-flight transaction, one of the nonces can be used, then incremented during the transaction. An advantage is that each storage slot can be used up to `2**256` times (in contrast, currently, it can be used at most `256` times), leading to some gas savings.

Also if more simultaneous transactions in flight are expected, then a dynamic array should be used to allow for more nonces. Note: If you can define an upper limit to the transactions in flight, you can use fixed storage slots (0 - 10) in the lower part of the storage, which might be cheaper in the future with the proposed move to stateless ethereum.

Note: this suggestion assumes that `_implementation` and `_owner` variables are moved to `immutable` variables; otherwise, using the slots `0` and `1` would lead to storage slot clash and potential lockup of proxied account.

Brink: We chose not to fix because of the complexity of implementation compared to the relatively small gas savings. We'll consider this implementation for future verifier contracts.

Spearbit: While, this is reasonable, it's important that all verifiers have compatible mechanisms to prevent replay prevention. This avoids using a nonce that was scheduled by another verifier, by accident.

4.4.11 Upgrade to the latest compiler version: `0.8.10`

Severity: Informational, Gas optimization

Context: All contracts.

Currently, brink.trade is using the `0.7.6` version of the compiler.

Recommendation: Upgrade the compiler to `0.8.10` and lock it for that specific version. Advantages to switching include external calls requiring less gas (extcodesize is no longer done on functions that have return parameters). Example of an external call that will be 100 gas cheaper: `EIP1271Validator.sol#L19`

```
pragma solidity =0.8.10
```

Brink: Fixed.

- commit 9fb87e4 and
- commit 864aef4.

Spearbit: Resolved.

4.4.12 Custom errors from version `0.8.4` are more gas efficient

Severity: Gas Optimization

Context: All uses of revert strings, for example `DeployAndExecute.sol#L29`

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

Custom errors from Solidity `0.8.4` are more gas efficient than revert strings, when the revert condition is met, and also decreases deploy time costs.

Recommendation: Use 0.8.4's custom errors instead to save on gas.

For more information: Solidity blog.

Brink: Fixed commit 1260ad0 and commit d1bf851

Spearbit: Resolved.

4.4.13 Add call protection to `LimitSwapVerifier`

Severity: Informational

The contract `LimitSwapVerifier` is only meant to be delegatecalled. This can be explicitly enforced by adding call protection.

Recommendation: Add call protection or, alternatively, use a library instead of a contract, as the solidity compiler can automatically add this check. Example of call protection:

```
abstract contract OnlyDelegateCallable {
    address immutable deploymentAddress = address(this);
    modifier onlyDelegateCallable() {
        require(address(this) != deploymentAddress);
        _;
    }
}
```

Brink: We chose not to implement the fix because there is no risk of `selfdestruct`, or any other consequence of calling `LimitSwapVerifier` directly.

4.4.14 Not following solidity memory model in inline assembly usage

Severity: Informational

Context:

1. `Account.sol#L36`

```
assembly {
    let result := call(gas(), to, value, add(data, 0x20), mload(data), 0, 0)
    returndatacopy(0, 0, returndatasize())
    switch result
    case 0 {
      revert(0, returndatasize())
    }
    default {
      return(0, returndatasize())
    }
}
```

2. `DeployAndExecute.sol#L32-L42`

These inline assembly snippets does not follow Solidity's memory model. The `returndatacopy` of calls are stored at memory location starting from 0. However, this may lead to overwriting the free memory pointer as well as writing beyond 0-64.

Risk:

1. This may cause issues in the future versions, if you need to use stack to memory mover. This is true for almost all inline assembly usage. Reference: (`https://twitter.com/ethchris/status/1439879409675259907`)
2. The free memory pointer may be overwritten by the return data. If a future refactoring of the code passes the control flow to high-level solidity, i.e., outside the inline assembly block, this would lead to undefined behaviour.

Recommendation: This currently will not cause any issues. Be aware of upgrading to a different version of the compiler while using stack to memory mover or a refactoring that passes control flow outside the inline assembly block that overwrites the free memory pointer.

The idiomatic approach would be to store dynamic data starting from `mload(64)` (the free memory pointer) rather than from `0`. However, making this change is not strictly necessary right now.

Brink: No fix implemented. The current implementation does not adversely affect functionality, and will not have an impact on any of the call data execution that happens before `returndatacopy` is executed.

4.4.15 Improving documentation regarding delegatecalls.

Severity: Informational

Context: `Account.sol#L9`

There is no explicit documentation stating that the `Account` contract is supposed to only be delegatecalled.

Recommendation: Add a NatSpec comment that the contract is only supposed to be delegatecalled.

Brink: Fixed commit c92b533

Spearbit: Resolved.

4.4.16 Variables can be defined inline

Severity: Informational

Context: `Account.sol#L12-L21`

Currently, the constants `META_DELEGATE_CALL_TYPEHASH` and `META_DELEGATE_CALL_EIP1271_TYPEHASH` are not defined inline. They could be defined inline to

improve readability.

Recommendation: Declare these variables inline.

```
bytes32 constant META_DELEGATE_CALL_TYPEHASH =
↪  keccak256("MetaDelegateCall(address to,bytes data)");
bytes32 constant META_DELEGATE_CALL_EIP1271_TYPEHASH =
↪  keccak256("MetaDelegateCall_EIP1271(address to,bytes data)");
```

Brink: Chose not to implement because impact is very low.

4.4.17 Add events to other functions beyond `cancel`

Severity: Informational

Context: `CancelVerifier.sol#L16`

```
emit Cancel(bitmapIndex, bit);
```

There is an `emit` in `cancel()`, but not other functions.

Recommendation: Retrieving the execution state is important for `cancel()` as well as for other functions. If you are doing an `emit`, it is probably better to do it from `Access.sol`. This way it is always there and opens up the opportunity to create a graph or subgraph for it.

Brink: We have generally avoided emitting events with data that is already included in functional params. We may actually remove this `Cancel` event in a future version of `CancelVerifier`.

4.4.18 The function `proxyCall` can be made external

Severity: Informational

Context: `CallExecutor.sol#L30`

```
function proxyCall(address to, bytes memory data) public {
```

Function currently marked `public`, however, it is never called internally.

Recommendation: Mark function `external`. It is generally a good practice to apply the most constrictive visibility possible.

Brink: We chose not to fix because of the low impact, but may fix in future verifier contracts.

4.4.19 Contracts that could be made abstract

Severity: Informational

It is a good practice to mark contracts that are not meant to be instantiated directly as `abstract contracts`. This is an idiomatic way to ensure that a contract is not meant to be deployed by itself.

The following contracts can be made abstract:

1. EIP712SignerRecovery
2. EIP712Validator
3. ProxySettable
4. ProxyGettable
5. ProxyStorage

Brink: Fixed commit 3b6d6dd, and `ProxySettable.sol` was removed, as it was no longer relevant.

Spearbit: Resolved.

4.4.20 Floating pragma is set

Severity: Informational

Context: All contracts.

The current pragma Solidity directive is `^0.7.6`. It is recommended to specify a specific compiler version to ensure that the byte code produced does not vary between builds. Contracts should be deployed using the same compiler version/flags with which they have been tested. Locking the pragma (for e.g. by not using ^ in `pragma solidity 0.8.10`) ensures that contracts do not accidentally get deployed using an older compiler version with known compiler bugs.

Recommendation: Lock the compiler to a specific version.

```
pragma solidity =0.7.6;
```

Brink: Fixed

- commit 9fb87e4 and
- commit 864aef4.

Spearbit: Resolved.

5 A note on verifiers

The Brink protocol relies on verifiers, i.e., smart contracts that perform a specific automation. This is achieved by signing a meta delegatecall to these smart contracts. Because of that, it is very important that these verifier contracts are well vetted. For example, the risk of a malicious verifier directly impacts the user potentially losing all their funds by signing a message to a malicious verifier.

There are several ways in which a verifier can hide their malicious behaviour. Some examples are given below:

1. Verifier Contract is upgradable: If a verifier contract is upgradable, and a user signs a message to this contract, the verifier can upgrade the contract just before the execution, to make the signed data malicious.
    - Since a typical upgradable contract works by delegatecalling its implementation, where the implementation is stored in storage, these kinds of contracts are inconsistent with Brink.trade. This is because the meta transactions delegatecalls into the verifier, which means that the verifier cannot use its own storage.
    - A non-typical upgradable pattern involves `CREATE2` deploy + `selfdestruct` + `CREATE2` redeploy, where the second redeploy will deploy a different runtime code. There are some tricks involved getting the same address, but having different runtime code: example. If the signer also signs the extcodehash for the verifier contract, such attacks can be prevented. If during execution, the extcodehash is different from what was part of the signature, the execution reverts. However, this might not cover all possible cases.
2. If anyone can add arbitrary verifier contracts, you could trick a user in the following way.
    - The verifier contract has the innocent looking function with signature: `cancel(uint248 bitmapIndex, uint248 bit)` external.
    - The verifier contract has a function named `safeguardfunds_<nonce>`, where `nonce` is engineered to collide with the selector of the function `cancel(uint256,uint256)`. Such collisions can be easily found in a few minutes or one can also lookup 4byte.directory.

- The user may be tricked to sign in to our cancel verifier, however, in turn it executes `safeguardfunds_<nonce>`, which steals funds.

Recommendations:

1. Thoroughly vet all verifier contracts, including the function selectors.
2. Avoid verifier contracts that are upgradable.
3. Consider enforcing that all verifiers should be libraries with external functions. The solidity compiler enforces that libraries do not have state variables and automatically adds call protection to state modifying functions.
4. Design modular verifiers doing only a single task.

6 A note on executors

The Brink protocol relies on executors running the transaction when certain conditions are met. For example, assume that a user tries to do a limit swap of 1 ETH to 5000 DAI, an executor is required to watch the ETH to DAI price, and perform a swap when the price is greater than 5000 DAI. The executor profits any difference in price.

It is therefore imperative that the protocol relies on more than one executor. This is because a single executor can decide to prolong the execution, for example, by waiting for the ETH price to be 5500 DAI, making a net profit of 500 DAI.

Note: This strategy involves the risk of losing out on potential rewards, by waiting for the price to move. Regardless, having multiple independent executors would make the protocol more efficient by making the margins low.

We recommend Brink to onboard new executors to the platform. The MEV Job Board for example would be a good place to advertise for bots.

A second issue is related to how the executors perform the execution. Brink mentioned that currently these transactions are sent to the public mempool (via Alchemy). Assuming permissionless contracts are used all throughout, this leads to front-running risk, as anyone can copy the calldata for the transaction and front-run the transaction. The original executor's transaction will still end up in the chain, except that the transaction would revert, leading to the executor paying for gas and getting nothing in return. Although front-running does not directly add security risk to the protocol, it indirectly affects it, as being an executor becomes less profitable. Note that regardless of the front-running issue,

the problem of executors paying the gas for a failed transaction can still happen in practice. If there are multiple executors trying to concurrently send the same transaction to the mempool, only one of them can profit from the opportunity. One way to solve this issue is to use a service like flashbots. This offers two advantages:

1. The transaction will remain private in the mempool, i.e., font-running can be avoided.
2. You can configure to not include a transaction in the block if it reverts (this is the default configuration). This would prevent executors from paying gas for reverting transactions.

Note: an alternative for (2) along with (1) is to use ethermine's MEV relay. However, it has a lower hash rate than Flashbots. Note: an alternative for (1) is to use taichi's private transaction.

7 Appendix

7.1 Gas optimization: Use `calldata` instead of `memory` for function parameters

In some cases, having function arguments in `calldata` instead of `memory` is more optimal. Consider the following generic example:

```
contract C {
    function add(uint[] memory arr) external returns (uint sum) {
        uint length = arr.length;
        for (uint i = 0; i < length; i++) {
            sum += arr[i];
        }
    }
}
```

In the above example, the dynamic array `arr` has the storage location `memory`. When the function gets called externally, the array values are kept in `calldata` and copied to `memory` during ABI decoding (using the opcode `calldataload` and `mstore`). And during the for loop, `arr[i]` accesses the value in memory using a `mload`. However, for the above example this is inefficient. Consider the following snippet instead:

```
contract C {
    function add(uint[] calldata arr) external returns (uint sum) {
        uint length = arr.length;
        for (uint i = 0; i < length; i++) {
            sum += arr[i];
        }
    }
}
```

In the above snippet, instead of going via `memory`, the value is directly read from `calldata` using calldataload. That is, there are no intermediate memory operations that carries this value. Gas savings: In the former example, the ABI decoding begins with copying value from `calldata` to `memory` in a for loop. Each iteration would cost at least 60 gas. In the latter example, this can be completely avoided. This will also reduce the number of instructions and therefore reduces the deploy time cost of the contract. In short, use calldata instead of memory if the function argument is only read. Note that in older Solidity versions, changing some function arguments from `memory` to `calldata` may cause "unimplemented feature error". This can be avoided by using a newer (0.8.*) Solidity compiler.