Brink Security Review

Engagement II

SPEARBIT

Reviewers
Hari
Alex
Gerard

December 28, 2021

1 Executive Summary

Over the course of 2 days in total, Brink engaged with Spearbit to review Brink-core. This security review is a follow-up of the 3 engineer week engagement (November 7 - November 17) between Brink and Spearbit.

More specifically, this specialized review focussed on a core architectural change since the last engagement in the following contracts:

- `AccountFactory.sol`
- `Account.sol`

We found a total of 13 issues with Brink. Only the two low severity issues (4.1.1 and 4.1.2) require changes to the codebase. Both of these have been fixed by Brink. Overall, Spearbit found the codebase to be of very high quality.

Repository	**Commit**
Brink-core	db0027533b228a6994acdbcb06713b5a3a3ecb38

Summary

Type of Project	Automation, DeFi
Timeline	November 29th, 2021 - December 3rd, 2021
Methods	Manual Review
Documentation	High
Testing Coverage	High

Total Issues

High Risk	0
Medium Risk	0
Low Risk	2
Gas Optimizations and Informational	11

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 `https://spearbit.com`.

3 Introduction

The Brink protocol is designed for automating conditional orders on EVM compatible chains. For an introduction to the basic mechanics, one can consult the report from Spearbit's first security review of Brink. This follow up specialized review by Spearbit focussed on a unique extension of "EIP-1167: Minimal Proxy Contract".

The Brink protocol designed a gas efficient proxy implementation, based on EIP-1167 that also stores the address of the owner in the proxy. Details can be found in the section: "custom proxy code". In short, the address of the owner is appended at the end of the runtime code, and read using a `extcodecopy` of the relevant bytes in code; let's call this the *data section*.

The focus of the security review was on the following:

1. The correctness of the EIP-1167 extension, i.e., the account proxy, as well as the `NotOwner` check.
2. Can the data section of the code be executable? That is, can the execution ever reach anywhere in the final `20-byte-address`? Depending on the address, this possibility can make proxy accounts vulnerable.
3. Do any of the proposed EVM changes make the data section executable?
4. Do any of the proposed EVM changes create issues with the account proxies?
5. 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 Low Severity

4.1.1 Inline assembly leaves dirty higher order bits for `_proxyOwner` variable

Severity: Low, Gas optimization

Context: `Account.sol#L160-L165`

```
function proxyOwner() internal view returns (address _proxyOwner) {
  assembly {
    extcodecopy(address(), mload(0x40), 0x2d, 0x14)
    _proxyOwner := mload(sub(mload(0x40), 0x0c))
  }
}
```

The expression `mload(sub(mload(0x40), 0x0c))` reads from memory location that was not directly used in the context of `proxyOwner`; more specifically, the memory location, starting from `mload(0x40) - 0x0c` until `mload(0x40)`. Since the contents of this location cannot be predicted (Solidity does not clean up memory after use), the `proxyOwner` variable will have dirty higher order bits, i.e., the most significant 12 bytes (32 - 20) need not be zero.

However, the compiler would clean up this value towards the end of the function, by doing `_proxyOwner := and(2**160 - 1, _proxyOwner)`.

Recommendation:

Internally, the Solidity compiler tries to be careful about not leaving dirty higher order bits. Many critical compiler bugs are often the result of the compiler missing such cleanups. We recommend using right shift (`shr`) to properly clean the value read from memory.

```
  extcodecopy(address(), mload(0x40), 0x2d, 0x14)
  _proxyOwner := shr(0x60, mload(mload(0x40)))
```

This also saves some gas.

Note: The `SHR` (EIP-145) instruction was introduced the same time as `CREATE2`, and since the project relies on `CREATE2` (EIP-1014), this change should not cause any issues, while deploying on other EVM compatible chains. These changes were introduced in the Petersburg hardfork. For deploying on EVM-compatible chains, we recommend documenting this requirement.

A proof of concept is provided in the appendix.

Brink: Fixed in PR #43.

Spearbit: Resolved.

4.1.2 The function `deployAndCall` may silently fail during account creation

Severity: Low

Context:

1. `AccountFactory.sol#L23`.
2. `DeployAndCall.sol#L17`

The function `deployAndCall` relies on `AccountFactory.deployAccount` to create a new account and to return its address. This function currently just passes through the return value of the underlying `CREATE2` instruction. This instruction will return the value `0` in certain failure cases.

The function `deployAndCall` will then call this newly created account if `callData.length > 0`. In case of a `CREATE2` failure this will result in calling `address(0)`, successfully.

One quirky failure condition of `CREATE2` resulting in `0` is when it runs out of call depth, i.e. it is the 1024th call in the transaction frame. This is fairly easy to accumplish by a malicious actor. In this case `deployAndCall` will silently fail.

Recommendation:

A suggested remedy is to have a `require(account != 0);` statement in `AccountFactory.deployAccount`.

Reference: yellowpaper.

Brink: Fixed in PR #47.

Spearbit: Resolved.

4.2 Gas Optimizations and Informational

4.2.1 Hardcode `mload(initCode)` to save 3 gas

Severity: Gas Optimization

Context: `AccountFactory.sol#L23`.

```
@@ -20,7 +20,7 @@ contract AccountFactory {
       owner
     );
     assembly {
-      account := create2(0, add(initCode, 0x20), mload(initCode), SALT)
+      account := create2(0, add(initCode, 0x20), 0x4B, SALT)
     }
   }
 }
```

4.2.2 Use scratch space in `proxyOwner` to save gas

Severity: Gas Optimization

Context: `Account.sol#L163`

Recap the suggested code from the first issue above:

```
   extcodecopy(address(), mload(0x40), 0x2d, 0x14)
   _proxyOwner := shr(0x60, mload(mload(0x40)))
```

This currently relies on the free memory pointer, which is the adviseable way to use memory. However, for short memory requirements, it is possible to use the so called *scratch space*. From the `Layout in Memory` section of the Solidity documentation:

> Scratch space can be used between statements (i.e. within inline assembly).

A possible code making use of this is the following:

```
   extcodecopy(address(), 0, 0x2d, 0x14)
   _proxyOwner := shr(0x60, mload(0))
```

Note that this saves two `MLOAD`s.

Note: While Solidity has not changed the memory layout and these reserved slots for a very long time, it cannot be guaranteed that this will be the case in the future. For this reason we also recommend to leave a comment in the code should this change be enacted.

Brink: Fixed in PR #47.

Spearbit: Resolved.

4.2.3 Saving 1 byte off the constructor code

Severity: Informational

Context: `AccountFactory.sol#L19`

The following is the current constructor for the account:

PC	**Opcode**	**Stack**
00000	RETURNDATASIZE	[0]
00001	PUSH1 0x41	[0x41, 0]
00003	DUP1	[0x41, 0x41, 0]
00004	PUSH1 0x0a	[0x0a, 0x41, 0x41, 0]
00006	RETURNDATASIZE	[0, 0x0a, 0x41, 0x41, 0]
00007	CODECOPY	[0x41, 0]
00008	DUP2	[0, 0x41, 0]
00009	RETURN	[0]

However, the `dup2` before the `return` indicates a possible optimization by re-arranging the stack. Here, the number `0x0a` denotes the offset of the runtime code. Let's denote this by `offset`, then the following would be a more optimal constructor code:

PC	**Opcode**	**Stack**
00000	PUSH1 0x41	[0x41]
00002	RETURNDATASIZE	[0, 0x41]
00003	DUP2	[0x41, 0, 0x41]
00004	PUSH1 offset	[offset, 0x41, 0, 0x41]
00006	RETURNDATASIZE	[0, offset, 0x41, 0, 0x41]
00007	CODECOPY	[0, 0x41]
00008	RETURN	[]

Brink: Fixed in PR #47.

Spearbit: Resolved.

4.2.4 Vanity address optimization

Severity: Gas optimization

By finding the appropriate salt for the implementation contract, to make the deployment address have as many zeros as possible, it is possible to save gas for the account factory deployment. This is because one can replace `push20 20-byte-constant` to say, `push16 16-byte-constant`. The latter is 4 bytes shorter than the former, and therefore decreases gas cost for deployment. The following tool can be used to search for `create2` vanity addresses: ERADICATE2

Generally the more time is available for the search, the probability of finding increasing number of leading zeroes goes up. Attaining 2 leading zeroes should be a matter of seconds.

Note: the code of the minimal proxy as well as the code in `proxyOwner()` needs to be updated to support a short address.

Note: a vanity address is most relevant for the `Address.sol`. It might also have a slight advantage for the verifiers. For the minimal proxy, which is deployed per user, it is not important.

Brink: Fixed in PR #47.

Spearbit: Resolved.

4.2.5 Use `bytes.concat` instead of `abi.encodePacked`

Severity: Informational

Context: `AccountFactory.sol#L16-L21`

Since 0.8.4 it is possible to use `bytes.concat`, which expects literals, `bytes`, and `bytesNN` inputs, and is aimed to replace most use cases of `abi.encodePacked`. It is more expressive, avoids the complex rules of `abi.encodePacked`, and the latter is expected to be phased out in the future.

This suggestion applies throughout the code, but especially in `AccountFactory.deployAccount`, where this would improve readability for showing the layout clearly:

```
-    bytes memory initCode = abi.encodePacked(
-      ....
-      owner
-    );
+    bytes memory initCode = bytes.concat(
+      ....
+      bytes20(owner)
+    );
```

4.2.6 Use `<address>.code.length` instead of `extcodesize` in inline assembly

Severity: Informational

Context: `SaltedDeployer.sol#L48-L58`

```
   function _isContract(address account) internal view returns (bool) {
     // This method relies on extcodesize, which returns 0 for contracts in
     ↪  construction, since the code is only stored
     // at the end of the constructor execution.
-
-    uint256 size;
-    assembly {
-      size := extcodesize(account)
-    }
-    return size > 0;
+      return account.code.length > 0;
   }
 }
```

Note: In the past (before 0.8.1), this was inefficient (Solidity used to do an `extcodecopy` to copy the entire code to memory and then calculate the size of this byte array in memory instead of directly using `extcodesize`). But since 0.8.1, Solidity would avoid the memory copy and only use `extcodesize`.

Brink: Fixed in PR #47.

Spearbit: Resolved.

4.2.7 Use file-level constant for `SALT`

Severity: Informational

Context:

1. `AccountFactory.sol#L9`.
2. `SaltedDeployer.sol#L20`.

```
   /// @dev Salt used for salted deployments
   bytes32 constant SALT =
↪  0x841eb53dae7d7c32f92a7e2a07956fb3b9b1532166bc47aa8f091f49bcaa9ff5;
```

This salt is duplicated in both the contracts `Account/AccountFactory.sol` and `Deployers/SaltedDeployer.sol`.

It may make sense placing the `SALT` as a file-level constant in `AccountFactory` and using that in `SaltedDeployer`. This would reduce the risk of mismatched salt during future changes.

Brink: No longer relevant after the salt was set to `0` in AccountFactory.

Spearbit: Resolved.

4.2.8 Use constants for offsets in `proxyOwner`

Severity: Informational

Context: `Account.sol#L162`

```
    extcodecopy(address(), mload(0x40), 0x2d, 0x14)
```

The offset `0x2d` is hardcoded in `Account`, but it relies on the layout provided by `AccountFactory`. It may make sense provide a file-level constant in `AccountFactory.sol` and use that:

```
uint256 constant PROXY_OWNER_OFFSET = 0x2d;
```

This would reduce the risk of mismatched offsets during future changes.

4.2.9 Document that the variable `callData` must have location `memory` in the function `deployAndCall`

Severity: Informational

Context: `DeployAndCall.sol#L16`

In `Batched/DeployAndCall.sol` the function

```
function deployAndCall(address owner, bytes memory callData) external payable;
```

has the variable `callData` marked as `memory`. By purely looking at the function signature, one would suggest to use the `calldata` location specifier here. This would be incorrect, because the function body uses assembly code relying on the memory layout caused by the `memory` specifier.

It is suggested to include a comment in the code to ensure the location remains unchanged.

4.2.10 Document that `deployAndCall` may not call the created account

Severity: Informational

Context: `DeployAndCall.sol#L15`

In `Batched/DeployAndCall.sol` the function `deployAndCall(address owner, bytes memory callData)` deploys a new account and calls into it. While it always deploys a new account, it only calls into it if the `callData` is non-empty.

It is adviseable to document this feature in the NatSpec description.

4.2.11 Using underscores to improve readability of hex value

Severity: Informational

Context: `AccountFactory.sol#L19`

Adding underscore as delimiters improves the readability.

```
@@ -15,8 +15,8 @@ contract AccountFactory {
   /// address added to the deployed bytecode. The owner address can be read
   ↪  within a delegatecall by using `extcodecopy`
   function deployAccount(address owner) external returns (address account) {
     bytes memory initCode = abi.encodePacked(
-      //  [*** constructor **][**** eip-1167 ****]...
-      hex'3d604180600a3d3981f3363d3d373d3d3d363d73...
+      //  [*** constructor **]_[**** eip-1167 ****]_...
+      hex'3d604180600a3d3981f3_363d3d373d3d3d363d73_...
       owner
     );
     assembly {
```

Brink: Fixed in PR #47.

Spearbit: Resolved.

5 Custom proxy code

The contract implements a slightly customized version of the EIP-1167 proxy contract.

5.1 Difference to EIP-1167

The main change is to include an owner address to be used for `NotOwner` checks. This is accomplished with concatenating the address owner to the end of the deployed code. It is done without padding.

Two main concerns arise here:

1. Is it possible for execution to flow into data (the owner address)?
2. Is the address correctly and efficiently read in the `NotOwner` check?

The second question has been addressed in the section "use scratch space".

Regarding the first question we need to note that the EVM has no native support for code/data separation, but several proposals are aimed at providing it (e.g. EIP-2327 and EIP-3540). Save this, we can analyze the control flow. The only control flow instruction is `JUMPI`, which is immediately preceded with a `PUSH1` instruction. We can call this a *static jump*, and conclude that both the destination (the jump) and falling through end up at `RETURN` or `REVERT`, respectively. Also note that several other preceding instructions could abort via various reasons.

One best practice the Solidity compiler does is the insertion of a special marker, the `INVALID` (`0xfe`) instruction, between code and data. This could be done here too, with little benefit.

5.2 Deploy-time

The deploy time code is similar to one in an earlier version of EIP-1167. We have suggested a gas optimisation in the section "saving 1 byte off the constructor code".

See also comments in ethereum-magicians.

5.3 Runtime

The runtime code prior to the audit is identical to the one suggested in EIP-1167. We suggest to consider vanity addresses in the section "vanity address optimization". This suggestion is also mentioned in the EIP.

Additionally one could consider a further optimisation of reordering the stack layout to save on `SWAP`/`DUP` instructions. A method has been described in this coinmonks article.

5.4 Remarks

The EIP-1167 proxy code is not easily readable due to optimisations. The varying costs of instructions motivates unconventional code, e.g. `DUPn` and `SWAPn` costs 3 gas, while `RETURNDATASIZE` only costs 2 and is initially set to 0, but can turn to non-zero after certain instructions.

Using `RETURNDATASIZE` is a common practice, but it doesn't come without risks. Proposals could change the semantics of this instruction in certain cases. While this is unlikely to happen, proposals like EIP-2733 do have a slight chance as they do not break existing contract executions, only if they are used in a certain way.

While it is likely a futile attempt to be entirely future proof, one could consider other instructions, such as `MSIZE` (before memory expansion this returns 0), `PC` (is 0 as the first instruction), and using `DUPn` more (which comes at a slight gas increase).

6 Appendix

6.1 Proof of concept for dirty higher order bits

```
pragma abicoder v1;

contract Test {
  bytes32 constant SALT =
↪  0x841eb53dae7d7c32f92a7e2a07956fb3b9b1532166bc47aa8f091f49bcaa9ff5;
  address constant random_address = 0x94324fcF2cC42F702F7dCBEe5e61E947DC9e2D91;

  address immutable proxy = deployAccount(random_address);
  uint256 constant junk = type(uint256).max;

  /// deployAccount from AccountFactory.
  function deployAccount(address owner) internal returns (address account) {
      ...
  }

  /// Note: returns bytes32 instead of address;
  /// so that the compiler does not perform cleanup.
  function proxyOwner(address _proxy) internal view returns (bytes32
↪  _proxyOwner) {
    assembly {
      extcodecopy(_proxy, mload(0x40), 0x2d, 0x14)
      _proxyOwner := mload(sub(mload(0x40), 0x0c))
    }
  }

  /// returns
  /// 0xffffffffffffffffffffffff94324fcf2cc42f702f7dcbee5e61e947dc9e2d91
  /// instead of
  /// 0x94324fcf2cc42f702f7dcbee5e61e947dc9e2d91
  function test() external returns (bytes32) {
      assembly {
          // Store junk at the current free memory
          mstore(mload(0x40), junk)
          // Increment the free memory pointer
          mstore(0x40, add(mload(0x40), 0x20))
      }
      return proxyOwner(proxy);
  }
}
```