[H-1] Nonce missing while generating message to withdraw tokens from L2->L1, which can cause the vault being drained by repeatedly using same message signed by signer(Root Cause + Impact)

Description:

The `L1BossBridge::withdrawTokensToL1` function utilizes an external signature (`v, r, s`) to authorize the withdrawal of tokens from the L2 vault to an L1 address. However, the message hash used for generating and validating this signature does not include a unique, contract-enforced counter (nonce), or any form of message uniqueness check. Since the signature validation is only based on the recipient address (`to`) and the withdrawal amount (`amount`), an attacker can observe a single, valid withdrawal transaction and then reuse the exact same signature multiple times. This allows the attacker to repeatedly call the `L1BossBridge::sendToL1` function with identical inputs, draining the underlying token vault without needing new authorization from the legitimate signer.

Impact:

The vulnerability is rated as High Severity.

- Loss of Funds (Asset Drain): This is the most direct impact. An attacker can repeatedly replay the same valid withdrawal signature indefinitely, leading to the tokens held by the contract's vault being withdrawn multiple times until the funds are completely exhausted.
- Denial of Service (DoS): Even if the funds are not completely depleted, malicious replay transactions can render the L2 -> L1 bridge service unusable or lead to the hijacking of legitimate user withdrawal requests.
- Loss of Trust: The core responsibility of a smart contract is asset protection. This vulnerability directly violates that responsibility, leading to severe lack of confidence and trust from users.

Proof of Concept:

To prove our concept, we formulate the proof of concept into 3 steps.

- User2 bridges tokens from L1->L2
- User bridges tokens from L1->L2 as well, then immediately withdraws them from L2-L1. Signer/Operator would sign message accordingly.
- User repeatedly use this signed message until vault get eventually drained.

The Proof of Concept are sa follows:

```
function testUsercanRepeatedlyWithdraw() public{
        address user2 = makeAddr("user2");
        address user2InL2 = makeAddr("user2InL2");
        vm.prank(deployer);
        token.transfer(address(user2),1000e18);

        //user2 bridges 1000e18 tokens from L1->L2
        vm.startPrank(user2);
        token.approve(address(tokenBridge),1000e18);

        vm.expectEmit(true, true, false, false);
```

```
        emit Deposit(user2, user2InL2, 1000e18);
        tokenBridge.depositTokensToL2(address(user2), address(user2), 1000e18);
        vm.stopPrank();

        //user bridges 1000e18 tokens from L1->L2
        vm.startPrank(user);
        token.approve(address(tokenBridge),1000e18);
        vm.expectEmit(true, true, false, false);
        emit Deposit(user, userInL2, 1000e18);
        tokenBridge.depositTokensToL2(address(user), address(user), 1000e18);
        vm.stopPrank();

        //user withdraw tokens from L2->L1, he will get a message signed by signer/operator
        (uint8 v, bytes32 r, bytes32 s) =
_signMessage(_getTokenWithdrawalMessage(address(user),1000e18), operator.key);
        //user can reaptedly use this message
        vm.startPrank(user);
        tokenBridge.sendToL1(v,r,s,_getTokenWithdrawalMessage(address(user),1000e18));
        tokenBridge.sendToL1(v,r,s,_getTokenWithdrawalMessage(address(user),1000e18));
        vm.stopPrank();
        assertEq(2000e18,token.balanceOf(address(user)));
        assertEq(0,token.balanceOf(address(tokenBridge)));
    }
```

Recommended Mitigation:

To prevent users from repeatedly using same signed message to transfer tokens from vault, I recommend The `L1BossBridge` contract utilize a `nonce` like mechanism to make sure every message can only be used once.

Specifically, it needs to add a mapping variable `messageHashesUsed` at the beginning of the `L1BossBridge` contract. Then it needs to check if the message is unused before making any further operations in function `L1BossBridge::sendToL1`. Finally, log messageHashesUsed[messageHash] as true before using the message.

The detailed difference are as follows:

```
contract L1BossBridge is Ownable, Pausable, ReentrancyGuard {
    ...
+   mapping(bytes32 => bool) public messageHashesUsed;
    ...
    function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message) public
nonReentrant whenNotPaused {
+       bytes32 messageHash = keccak256(message);
+       require(messageHashesUsed[messageHash] != true)   // make sure the message is
unused.
        address signer =
ECDSA.recover(MessageHashUtils.toEthSignedMessageHash(keccak256(message)), v, r, s);

        if (!signers[signer]) {
            revert L1BossBridge__Unauthorized();
        }
+       messageHashesUsed[messageHash] = true;   // set used messageHash mapping variable to
be true
```

```
        (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();
        }
    }
}
```

[L-1] Fail to check if ERC-20's symbol already exists in `TokenFactory::s_tokenToAddress`, which can overwrite previous ERC-20 token' address(Root Cause + Impact)

Description:

In contract `TokenFactory`, when deploying new token, if the input symbol string from function `deployToken(string memory symbol, bytes memory contractBytecode)` already exists, it will overwrite the old mapping of `TokenFactory::s_tokenToAddress`.

Impact:

The vulnerability is rated as Low Severity.
`TokenFactory` will lose the previous token address of mapping variable `TokenFactory::s_tokenToAddress`.

Proof of Concept:

The codes below demonstrate our review. When running code below, it passes with no failure. TokenAddress does not equal to newTokenAddress, which means old mapping slot has been overwritten when calling `TokenFactory::deployToken` using same token symbol.

```
function testAddToken_overwrite() public {
        vm.prank(owner);
        address tokenAddress = tokenFactory.deployToken("TEST", type(L1Token).creationCode);
        vm.prank(owner);
        address newTokenAddress = tokenFactory.deployToken("TEST",
type(L1Token).creationCode);
        assertNotEq(tokenAddress, tokenFactory.getTokenAddressFromSymbol("TEST"));
    }
```

Recommended Mitigation:

Check if string symbol already exists in `TokenFactory::s_tokenToAddress`.

```
function deployToken(string memory symbol, bytes memory contractBytecode) public onlyOwner
returns (address addr) {
+       require(s_tokenToAddress[symbol]!=address(0));
        assembly {
            addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode))
        }
        s_tokenToAddress[symbol] = addr;
        emit TokenDeployed(symbol, addr);
    }
```xxxxxxxxxx     let accountBalanceSlot := keccak256(ptr, 0x40)    let accountBalance :=
sload(accountBalanceSlot)+   if lt(accountBalance, value){+       mstore(0x00, shl(224,
0xe450d38c))+       mstore(add(0x00, 4), account)+       mstore(add(0x00, 0x24),
accountBalance)+       mstore(add(0x00, 0x44), value)+       revert(0, 0x64)+    }+
sstore(accountBalanceSlot, sub(accountBalance, value))diff
```
```