// Private Smart Contract Security Assessment 03.25.2024 - 04.01.2024

Neon Labs - EVM SPL Contract

Neonlabs

HALBORN

Neon Labs - EVM SPL Contract - Neonlabs

Prepared by: HALBORN

Last Updated 04/12/2024

Date of Engagement by: March 25th, 2024 - April 1st, 2024

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
11	0	0	2	5	4

TABLE OF CONTENTS

1. Introduction

NeonEVM Labs engaged Halborn to conduct a security assessment on their smart contracts beginning on **25/03/2024** and ending on **01/04/2024.** The security assessment was scoped to the smart contracts provided to the Halborn team.

2. Assessment Summary

The team at Halborn was provided one week for the engagement and assigned a full-time security engineer to evaluate the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some security risks that were mostly addressed by the **NeonEVM Labs team**.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance code coverage and quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions. (`solgraph`)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes. Manual testing by custom scripts.
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (`MythX`)
- Static Analysis of security for scoped contract, and imported functions. (`Slither`)
- Testnet deployment. (Brownie, Anvil, Foundry)

Out-Of-Scope

- External libraries and financial-related attacks.
- New features/implementations after/with the **remediation commit IDs**
- Changes that occur outside of the scope of PRs.

4. RISK METHODOLOGY

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the LIKELIHOOD of a security incident and the IMPACT should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

- **10** - CRITICAL
- **9 - 8** - HIGH
- **7 - 6** - MEDIUM
- **5 - 4** - LOW
- **3 - 1** - VERY LOW AND INFORMATIONAL

5. SCOPE

FILES AND REPOSITORY

(a) Repository: neon-contracts

(b) Assessed Commit ID: 86fb7cf

(c) Contracts in scope:

- ERC20ForSPLFactory.sol
- ERC20ForSPLMintableFactory.sol
- ERC20ForSPLBackbone.sol
- ERC20ForSPL.sol
- ERC20ForSPLMintable.sol

Out-of-Scope:

REMEDIATION COMMIT ID:

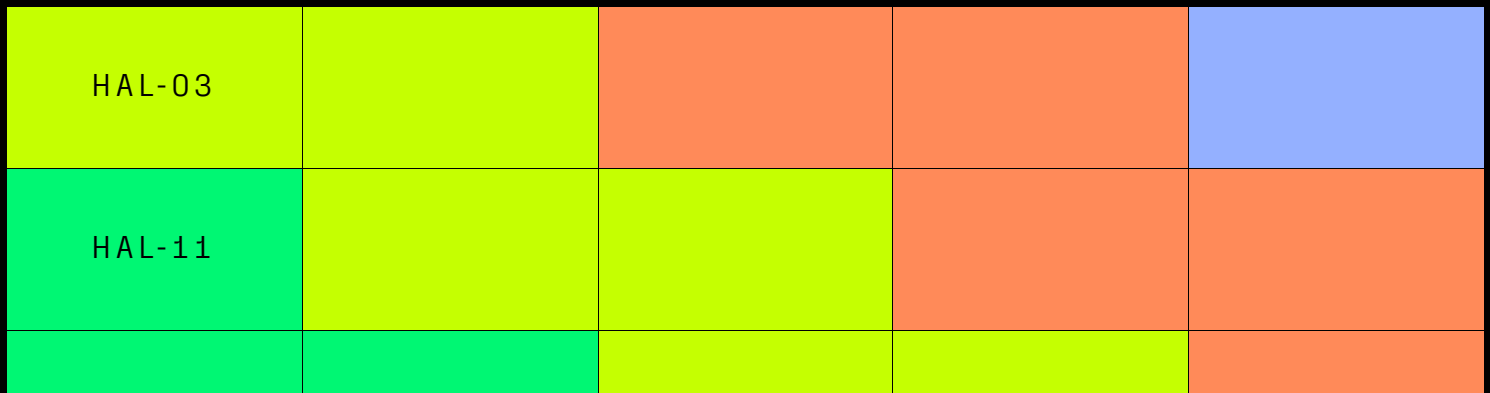
- 9f1c445
- 16de9d6
- 1e91746
- f68dfb4

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	2	5	4

IMPACT X LIKELIHOOD



		HAL-06		
	HAL-02 HAL-04 HAL-01 HAL-07 HAL-09			
HAL-08 HAL-10 HAL-12				

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-03 - INHERITANCE CONFLICT IN STORAGE SLOTS	Medium	PARTIALLY SOLVED - 04/03/2024
HAL-06 - SOLANA ACCOUNT INITIALIZATION OVERSIGHT	Medium	NOT APPLICABLE
HAL-02 - FRONT-RUNNING/GAS-GRIEFING IN TOKEN DEPLOYMENT	Low	SOLVED - 04/03/2024
HAL-04 - ALLOWANCE FRONT-RUNNING RISK	Low	RISK ACCEPTED - 04/02/2024
HAL-01 - STORAGE SLOTS MODIFICATION RISKS	Low	PARTIALLY SOLVED - 04/03/2024
HAL-07 - TWO STEP OWNERSHIP TRANSFER	Low	RISK ACCEPTED
HAL-09 - ALLOWANCES SHOULD BE LIMITED TO TYPE(UINT64).MAX	Low	SOLVED - 04/03/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-08 - OWNER CAN RENOUNCE OWNERSHIP	Informational	ACKNOWLEDGED
HAL-11 - LACK OF INPUT VALIDATION IN ADDALREADYEXISTINGERC20FORSPL()	Informational	SOLVED - 04/03/2024
HAL-10 - MISSING EVENT EMISSION ON ADDALREADYEXISTINGERC20FORSPL()	Informational	SOLVED - 04/03/2024
HAL-12 - OMISSION OF __UUPSUPGRADEABLE_INIT()	Informational	SOLVED - 04/02/2024

7. FINDINGS & TECH DETAILS

7.1 (HAL-03) INHERITANCE CONFLICT IN STORAGE SLOTS

// MEDIUM

Description

The `ERC20ForSPLMintable.sol` contract incorporates the `OwnableUpgradeable.sol`, which is a modified variant of the OpenZeppelin (OZ) upgradeables contract by `NeonEVM`. The purpose of these modifications is gas optimization during deployment. A particular change involves the relocation of the `ownableStorageLocation` variable.

The original `ownableStorageLocation` is designated by a hash computation: `keccak256(abi.encode(uint256(keccak256("openzeppelin.storage.Ownable")) - 1)) & ~bytes32(uint256(0xff))`.

In contrast, the modified version simplifies this location to a direct reference: `0x0000000000000000000000000000000000000000000000000000000000000002`. This alteration means that any contract inheriting from this modified library would have its owner address stored at storage slot 2. However, the `ERC20ForSPLMintable.sol` also extends `ERC20ForSPLBackBone.sol`, which initializes storage slots without reserving slot 2. Notably, the `beacon` variable occupies slot 0, and `tokenMint` occupies slot 1 and slot 2 is used by `_allowances` so `owner` and `_allowances` are sharing the same slot. Due to how mappings store memory in solidity, there is no way to exploit it in this implementation, however during further updates, if an error is made and slot 2 is accessible externally it could lead to a critical scenario.

```
contract ERC20ForSPLBackbone {
    ISPLToken public constant SPL_TOKEN =
ISPLToken(0xFf00000000000000000000000000000000000004);
    IMetaplex public constant METAPLEX =
IMetaplex(0xff00000000000000000000000000000000000005);

    address public beacon; // Slot 0
    bytes32 public tokenMint; // Slot 1
    mapping(address => mapping(address => uint256)) private _allowances;
    // ..... //
}
```

Proof of Concept

The following Foundry test prove that both `owner` and `_allowances` can be fetched from the same slot (`0x0..02`) :

```
function test_Collision() public {
        mintable.initialize("_testCoin","TC","",9,address(this));
        uint256 address2 = uint256(uint160(address(0x02)));
        uint256 addressThis = uint256(uint160(address(this)));
```

```
        bytes32 allowanceSlot = keccak256(abi.encodePacked(address2,
keccak256(abi.encodePacked(addressThis, uint256(2)))));

        // address(2) approve 100_000 from slot 2
        mintable.approve(address(0x02),100000);
        bytes32 allowanceSlotValue = vm.load(address(mintable), allowanceSlot);
        console2.log("Allowance : ",uint256(allowanceSlotValue));

        // Fetch ownerSlot value from slot 2
        bytes32 ownerSlot = vm.load(address(mintable), bytes32(uint256(2)));
        console2.log("Owner : ", address(uint160(uint256(ownerSlot))));

    }
```

```
Ran 1 test for test/Counter.t.sol:CounterTest
[PASS] test_Collision() (gas: 93528)
Logs:
  Allowance :  100000
  Owner :  0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 7.07ms (1.29ms CPU time)

Ran 1 test suite in 261.87ms (7.07ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:M/D:N/Y:N (6.3)

Recommendation

It is strongly recommended that NeonEVM reverts to utilizing the original OpenZeppelin libraries that conform to EIP-7201: Namespaced Storage Layout. This EIP prescribes a storage structure that mitigates such conflicts. Alternatively, NeonEVM could reserve slot 2 specifically for `OwnableStorageLocation` in `ERC20ForSPLBackBone.sol` by setting it manually like it is done in `ERC20ForSPLFactory` and then set `_allowances`. While this approach may diminish the risk, it does not eliminate the possibility of future storage collisions. Adopting EIP-7201's guidelines is the preferred solution to ensure compatibility and upgrade safety across inherited contracts.

Remediation Plan

PARTIALLY SOLVED: The **Neon Labs team** partially fixed the issue by booking the 2nd memory storage slot for **_owner** value.

Remediation Hash

https://github.com/neonlabsorg/neon-contracts/commit/9f1c445ca0bda42a6e3485530b796574250a6f87

7.2 (HAL-06) SOLANA ACCOUNT INITIALIZATION OVERSIGHT

// MEDIUM

Description

The smart contract functions `transfer()`, `claimTo()` within `ERC20ForSPLBackBone`, and `mint()` to `ERCForSPLMintable` incorporate a critical verification step to ascertain whether the receiver's account is an initialized Solana account. This verification involves a conditional check:

```
if (SPL_TOKEN.isSystemAccount(toSolana)) {
      SPL_TOKEN.initializeAccount(_salt(to), tokenMint);
}
```

This condition ensures that if the target account on the Solana network is not initialized, it undergoes initialization with the appropriate token mint.
However, there is an omission of similar checks in two additional functions:

- `transferSolana(bytes32 to, uint64 amount)`
- `approveSolana(bytes32 spender, uint64 amount)`

```
function transferSolana(bytes32 to, uint64 amount) public returns (bool) {
        address from = msg.sender;
        bytes32 fromSolana = solanaAccount(from);

    //E @audit Ensure `to` account is an initialized account
        SPL_TOKEN.transfer(fromSolana, to, uint64(amount));

        /// .. ///
}

function approveSolana(bytes32 spender, uint64 amount) public returns (bool) {
        address from = msg.sender;
        bytes32 fromSolana = solanaAccount(from);

    //E @audit Ensure `spender` account is an initialized account
        if (amount > 0) {
            SPL_TOKEN.approve(fromSolana, spender, amount);
        } else {
            SPL_TOKEN.revoke(fromSolana);
        }

        /// .. ///
}
```

In these functions, the transactions proceed without verifying the initialization status of the `to` or `spender` Solana accounts, potentially leading to interactions with unprepared account states on the Solana network.
The absence of initialization checks in `transferSolana` and `approveSolana` functions can lead to unsuccessful transactions or unintended behaviors when attempting to interact with Solana accounts that are not yet set up to receive tokens or approvals.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:N/S:U (5.0)

Recommendation

To mitigate this vulnerability, it is recommended to integrate the Solana account initialization check into the `transferSolana` and `approveSolana` functions. The following conditional initialization logic should be added:

```
if (SPL_TOKEN.isSystemAccount(solanaAccount(to))) {
            SPL_TOKEN.initializeAccount(_salt(to), tokenMint);
}
```

This adjustment ensures that before proceeding with a transfer or approval to a Solana account, the contract verifies the account's initialization status and initializes it if necessary, aligning the security and operational integrity of these functions with those already implementing this check.

Remediation Plan

NOT APPLICABLE: Due to not having Ethereum-like addresses of variables `bytes32` it is not possible to initialize them, in the end the `transferSolana` and `approveSolana` functions will revert if a non-initialized account is used.

7.3 (HAL-02) FRONT-RUNNING/GAS-GRIEFING IN TOKEN DEPLOYMENT

// LOW

Description

The `addAlreadyExistingERC20ForSPL` function within the `ERC20ForSPLFactory` smart contract exhibits susceptibility to front-running and gas-griefing attacks. This function is designed to associate pre-existing ERC20 tokens on the `neonEVM` with their corresponding SPL tokens on Solana, requiring inputs of token mint addresses (`Solana`) and ERC20 addresses (`neonEVM`). A crucial safeguard is implemented to prevent the association of a Solana token mint address with multiple `neonEVM` token addresses, enforced by:

`if (tokensData[tokenMints[i]].token != address(0)) revert AlreadyExistingERC20ForSPL();`

Once the address of a `tokenMints` is set, it is no longer possible to modify it.

Details

- The process is compromised when an attacker, with knowledge of the impending execution of `addAlreadyExistingERC20ForSPL`, strategically invokes the `deploy` function with a token mint address that is about to be linked. Since `deploy` does not discern between new and existing tokens on `neonEVM`, it proceeds to create a new `ERC20` token contract for the `SPL` token if not already associated.
- This preemptive action by an attacker can result in the legitimate operation by the owner being reverted due to the token already being registered, thereby obstructing the owner's attempt to correctly map pre-existing tokens.

```
    function addAlreadyExistingERC20ForSPL(bytes32[] memory tokenMints,
address[] memory alreadyExistingTokens) external onlyOwner {
        uint tokensLen = alreadyExistingTokens.length;
        if (tokensLen != tokenMints.length) revert InvalidTokenData();

        for (uint i; i < tokensLen; ++i) {
            if (tokensData[tokenMints[i]].token != address(0)) revert
AlreadyExistingERC20ForSPL();

            tokensData[tokenMints[i]] = Token({
                token: alreadyExistingTokens[i],
                state: State.AlreadyExisting
            });
            tokens.push(alreadyExistingTokens[i]);
        }
    }

    function deploy(bytes32 tokenMint) external returns(address) {
        if (tokensData[tokenMint].token != address(0)) revert
AlreadyExistingERC20ForSPL();
```

```
        BeaconProxy token = new BeaconProxy(
            address(this),
            abi.encodeWithSelector(ERC20ForSPL(address(0)).initialize.selector,
tokenMint)
        );

        tokensData[tokenMint] = Token({
            token: address(token),
            state: State.New
        });
        tokens.push(address(token));

        emit TokenDeploy(tokenMint, address(token));
        return address(token);
    }
```

Proof of Concept

Following the scenario described below:

- Bob, an admin, wants to deploy tokens `[0,1,2,3,4,5,6]` that are already deployed and have their addresses on `neonEVM (0x0,0x1,0x2,0x3,0x4,0x5,0x6)`
- Alice, a malicious user spots that Bob wants to attach these tokens to their `neonEVM` addresses but wants to prevent owner from doing it and call `deploy(3)` before Bob call to `addAlreadyExistingERC20ForSPL([0,1,2,3,4,5,6],[0x0,0x1,0x2,0x3,0x4,0x5,0x6])`

By doing this Alice will make Bob call revert and prevent a fair deployment of token 3.
This act not only forces Bob's transaction to fail but also grants Alice the power to repeatedly disrupt Bob's attempts to map the token mint `3` to its `neonEVM` counterpart.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:L/Y:N (3.1)

Recommendation

To address this security flaw, it is advised to introduce a functionality allowing the removal of erroneously or maliciously deployed tokens. Such a mechanism would empower the contract owner to rectify the token registration state, ensuring the integrity of token mappings remains intact. The proposed function, `removeToken(bytes32 tokenMint)`, should adhere to the following logic in `ERC20ForSPLFactory.sol` and `ERC20ForSPLMintableFactory.sol`:

```
function removeToken(bytes32 tokenMint) external onlyOwner {
    if (tokensData[tokenMint].token == address(0)) revert
NotDeployedERC20ForSPL();
```

```
    tokensData[tokenMint].token = address(0);
}
```

Remediation Plan

SOLVED: The **Neon Labs team** implemented a 3rd choice on the kind of tokens added, allowing the owner to replace already added tokens and removing the possibility for this issue to happen.

Remediation Hash

https://github.com/neonlabsorg/neon-contracts/commit/16de9d6c1b1ee0f74d2d6259480122fa91dc0068

7.4 (HAL-04) ALLOWANCE FRONT-RUNNING RISK

// LOW

Description

During the code review, It has been noticed that the ERC20 implementation is not resistant to the well-known ERC20 race condition issue, also known as the allowance front-running problem. This issue occurs when a user attempts to update their token allowance for a spender while the spender is simultaneously trying to use the existing allowance. If the spender's transaction is confirmed before the user's allowance update, the user might inadvertently grant the spender a higher allowance than intended.

```
function approveSolana(bytes32 spender, uint64 amount) public returns (bool) {
        address from = msg.sender;
        bytes32 fromSolana = solanaAccount(from);

        if (amount > 0) {
            SPL_TOKEN.approve(fromSolana, spender, amount);
        } else {
            SPL_TOKEN.revoke(fromSolana);
        }

        emit Approval(from, address(0), amount);
        emit ApprovalSolana(from, spender, amount);
        return true;
  }
  function _approve(address owner, address spender, uint256 amount) internal {
        if (owner == address(0) || spender == address(0)) revert EmptyAddress();

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }
```

Proof of Concept

Let's suppose this scenario :

1. Alice initiates the `approve(Bob, 500)` or `approveSolana(Bob,500)`function, granting Bob the permission to utilize 500 tokens.
2. Subsequently, Alice reconsiders and calls `approve(Bob, 1000)`or `approveSolana(Bob, 1000)` which amends Bob's spending allowance to 1000 tokens.
3. Bob observes the transaction and swiftly invokes `transferFrom(Alice,X, 500)` before its mining completion.
4. If Bob's transaction is mined before Alice's, Bob will successfully transfer 500 tokens. Once Alice's transaction has been mined, Bob can proceed to call `transferFrom(Alice, X, 1000)`.
5. Bob has transferred 1500 tokens even though this was not Alice's intention.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:L/Y:N (3.1)

Recommendation

To mitigate the race condition issue and enhance the security and reliability of the token implementation, it is advised to implement the `increaseAllowance` and `decreaseAllowance` functions, instead of only using the approve function for modifying allowances.

Remediation Plan

RISK ACCEPTED: The **Neon Labs team** accepted the risk of this issue and will not implement a fix.

7.5 (HAL-01) STORAGE SLOTS MODIFICATION RISKS

// LOW

Description

`NeonEVM` has undertaken modifications to the OpenZeppelin upgradeable contract framework, specifically altering the storage locations of essential variables. These modifications encompass the reassignment of `OwnableStorageLocation` in `ownableUpgradeable.sol`, `IMPLEMENTATION_SLOT`, and `BEACON_SLOT` in `ERC1967Utils.sol` to simplified, consecutive storage slots.
The following alterations have been identified within `NeonEVM`'s fork:

- `OwnableStorageLocation` **Modification:**
  - Original: Computed via `keccak256(abi.encode(uint256(keccak256("openzeppelin.storage.Ownable")) - 1)) & ~bytes32(uint256(0xff))`
  - Modified To: `0x0000000000000000000000000000000000000000000000000000000000000002`
- `IMPLEMENTATION_SLOT` **Modification:**
  - Original: Derived from the `keccak-256` hash of `"eip1967.proxy.implementation"` subtracted by 1.
  - Modified To: `0x0000000000000000000000000000000000000000000000000000000000000001`
- `BEACON_SLOT` **Modification:**
  - Original: Based on the `keccak-256` hash of `"eip1967.proxy.beacon"` subtracted by 1.
  - Modified To: `0x0000000000000000000000000000000000000000000000000000000000000000`

To incorporate these changes, `NeonEVM` adjusted the initial storage slots in the `ERC20ForSPLFactory` and `ERC20ForSPLMintableFactory` contracts for corresponding variables.

```
contract ERC20ForSPLFactory is OwnableUpgradeable, UUPSUpgradeable {
    address private _implementation;
    address private _uupsImplementation;
    address private _owner;
    // ... //
 }

 contract ERC20ForSPLMintableFactory is OwnableUpgradeable, UUPSUpgradeable {
    address private _implementation;
    address private _uupsImplementation;
    address private _owner;
    // ... //
 }
```

The practice of modifying storage slots for critical variables to achieve gas efficiency poses significant risks to further developments. Inadvertent overwriting or alteration of these slots during upgrades could lead to severe consequences.

Proof of Concept

Tests have been conducted about the gas consumption, here are the results :

- Tests with custom implementation (forked and modified OZ libraries):

```
  Test init
    ERC20ForSPL tests

Creating instance of just now deployed ERC20ForSPL contract with address  0x014F23A70DC098530B614998D22c5D526cbd3404

      ✔ Deploy using modified OZ libraries + transferOwnership + upgradeToAndCall (315472720 gas)

·-------------------------------------------------------|---------------------------|-------------|-----------------------------·
|                 Solc version: 0.8.24                  ·  Optimizer enabled: false  ·  Runs: 200  ·  Block limit: 6718946 gas  |
·····················|·····························|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  Methods                                                                                                                      |
·····················|·····························|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  Contract            ·  Method                     ·  Min  ·  Max  ·  Avg  ·  # calls  ·  usd (avg)  |
·····················|·····························|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  ERC20ForSPLFactory  ·  addAlreadyExistingERC20ForSPL  ·  -  ·  -  ·  2334640  ·  1  ·  -  |
·····················|·····························|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  ERC20ForSPLFactory  ·  deploy                     ·  -  ·  -  ·  22365520  ·  1  ·  -  |
·····················|·····························|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  ERC20ForSPLFactory  ·  upgradeToAndCall           ·  -  ·  -  ·  10000  ·  1  ·  -  |
·····················|·····························|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  Deployments                                       ·                                  ·  % of limit  ·             |
·····················|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  ERC1967Proxy                                      ·  -  ·  -  ·  19325080  ·  287.6 %  ·  -  |
·····················|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  ERC20ForSPL                                       ·  -  ·  -  ·  90230920  ·  1342.9 %  ·  -  |
·····················|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  ERC20ForSPLFactory                                ·  -  ·  -  ·  90133480  ·  1341.5 %  ·  -  |
·····················|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  ERC20ForSPLFactoryV2                              ·  -  ·  -  ·  91073080  ·  1355.5 %  ·  -  |
·-------------------------------------------------------|-------------|-------------|-------------|-------------|-------------·

  1 passing (1m)
```

- Tests with official OZ libraries:

```
  Test init
    ERC20ForSPL tests

Creating instance of just now deployed ERC20ForSPL contract with address  0x05bb9b40780e74DF4bA21FE27f6222183b5D7781

      ✔ Deploy using not modified OZ libraries + transferOwnership + upgradeToAndCall (321778480 gas)

·-------------------------------------------------------|---------------------------|-------------|-----------------------------·
|                 Solc version: 0.8.24                  ·  Optimizer enabled: false  ·  Runs: 200  ·  Block limit: 6718946 gas  |
·····················|·····························|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  Methods                                                                                                                      |
·····················|·····························|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  Contract            ·  Method                     ·  Min  ·  Max  ·  Avg  ·  # calls  ·  usd (avg)  |
·····················|·····························|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  ERC20ForSPLFactory  ·  addAlreadyExistingERC20ForSPL  ·  -  ·  -  ·  2334640  ·  1  ·  -  |
·····················|·····························|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  ERC20ForSPLFactory  ·  deploy                     ·  -  ·  -  ·  23527840  ·  1  ·  -  |
·····················|·····························|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  ERC20ForSPLFactory  ·  upgradeToAndCall           ·  -  ·  -  ·  10000  ·  1  ·  -  |
·····················|·····························|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  Deployments                                       ·                                  ·  % of limit  ·             |
·····················|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  ERC1967Proxy                                      ·  -  ·  -  ·  21865480  ·  325.4 %  ·  -  |
·····················|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  ERC20ForSPL                                       ·  -  ·  -  ·  90230920  ·  1342.9 %  ·  -  |
·····················|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  ERC20ForSPLFactory                                ·  -  ·  -  ·  91435000  ·  1360.9 %  ·  -  |
·····················|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  ERC20ForSPLFactoryV2                              ·  -  ·  -  ·  92374600  ·  1374.8 %  ·  -  |
·-------------------------------------------------------|-------------|-------------|-------------|-------------|-------------·

  1 passing (1m)
```

In conclusion, the efficiency gains may not justify the risks involved around aforementioned modifications.

BVSS

AO:A/AC:M/AX:M/R:N/S:U/C:L/A:M/I:M/D:N/Y:N (3.1)

Recommendation

The modification of critical storage slots is deemed a hazardous practice, prone to errors in future upgrades. It is recommended that `NeonEVM` reconsider the strategy for utilizing these storage slots. Employing an abstract contract with a storage gap could mitigate risks by limiting future modifications. However, the most effective risk elimination method is to revert to the unmodified versions of `OpenZeppelin` contracts that adhere to EIP-7201: Namespaced Storage Layout, ensuring a standardized and safe approach to storage slot management.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.24;

abstract contract UpgradeableStorage {

            address private _implementation;
            address private _uupsImplementation;
            address private _owner;

            // Large storage gap for future additions
            uint256[47] private __gap; // Adjust the size of the gap based on
expected future storage needs

            // ... //
    }
```

Remediation Plan

PARTIALLY SOLVED: The **Neon Lab team** accepts the issue as a known one and will not implement any modification.

Remediation Hash

https://github.com/neonlabsorg/neon-contracts/commit/9f1c445ca0bda42a6e3485530b796574250a6f87

7.6 (HAL-07) TWO STEP OWNERSHIP TRANSFER

// LOW

Description

The ownership of the contracts can be lost as the `ERC20ForSPLFactory`, `ERC20ForSPLMintableFactory`, `ERC20ForSPLMintable` contract is inherited from the `OwnableUpgradeable` contract and their ownership can be transferred in a single-step process. The address the ownership is changed to should be verified to be active or willing to act as the owner.

```
/**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        if (newOwner == address(0)) {
            revert OwnableInvalidOwner(address(0));
        }
        _transferOwnership(newOwner);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Internal function without access restriction.
     */
    function _transferOwnership(address newOwner) internal virtual {
        OwnableStorage storage $ = _getOwnableStorage();
        address oldOwner = $._owner;
        $._owner = newOwner;
        emit OwnershipTransferred(oldOwner, newOwner);
    }
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:P/S:U (2.5)

Recommendation

It is advised to use the `Ownable2StepUpgradeable` library over the `OwnableUpgradeable` library or implement similar two-step ownership transfer logic into the contract.

Remediation Plan

RISK ACCEPTED: The **Neon Labs team** accepted the risk of this issue and won't take any action since the Neon Labs team states that the owner is already a multi-sig address.

7.7 (HAL-09) ALLOWANCES SHOULD BE LIMITED TO TYPE(UINT64).MAX

// LOW

Description

As `NeonEVM` is a cross virtual machine between Solana and Ethereum and Solana doesn't support numbers bigger than `type(uint64).max` it is advised to limit all modification of balance number to `type(uint64).max` in order to prevent tx revert on the Solana side:

```
function _spendAllowance(address owner, address spender, uint256 amount)
internal {
    uint256 currentAllowance = allowance(owner, spender);
    // @audit change to type(uint64).max
    if (currentAllowance != type(uint256).max) {
        if (currentAllowance < amount) revert InvalidAllowance();
        _approve(owner, spender, currentAllowance - amount);
    }
}

function _approve(address owner, address spender, uint256 amount) internal {
    // @audit add amount limitation to type(uint64).max
    if (owner == address(0) || spender == address(0)) revert EmptyAddress();

    _allowances[owner][spender] = amount;
    emit Approval(owner, spender, amount);
}
```

BVSS

AO:A/AC:L/AX:L/C:L/I:L/A:L/D:N/Y:N/R:P/S:C (2.3)

Recommendation

It is recommended to prevent users from submitting amounts more than `type(uint64).max` within **ERC20ForSPLBackbone.sol** functions `_spendAllowance()` and `_approve()`.

Remediation Plan

SOLVED: The **Neon Labs team** solved the issue by adding `type(uint64).max` input limitations.

Remediation Hash

https://github.com/neonlabsorg/neon-contracts/commit/1e91746d8125ffd0da571b7e423cd1d4cbec3a4e

7.8 (HAL-08) OWNER CAN RENOUNCE OWNERSHIP

// INFORMATIONAL

Description

The `ERC20ForSPLFactory`, `ERC20ForSPLMintableFactory`,`ERC20ForSPLMintable` contracts are inherited from the OwnableUpgradeable contract. The Owner of the contract is usually the account that deploys the contract. As a result, the Owner can perform some privileged functions. In the `OwnableUpgradeable` contracts, the `renounceOwnership()` function is used to renounce the Owner permission. Renouncing ownership before transferring would result in the contract having no Owner, eliminating the ability to call privileged functions.

```
/**
 * @dev Leaves the contract without owner. It will not be possible to call
 * `onlyOwner` functions. Can only be called by the current owner.
 *
 * NOTE: Renouncing ownership will leave the contract without an owner,
 * thereby disabling any functionality that is only available to the owner.
 */
function renounceOwnership() public virtual onlyOwner {
    _transferOwnership(address(0));
}
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:P/S:U (1.3)

Recommendation

It is recommended that the Owner cannot call `renounceOwnership()` without first transferring Ownership to another address. In addition, if a multisignature wallet is used, the call to the `renounceOwnership()` function should be confirmed for two or more users.

Remediation Plan

ACKNOWLEDGED: The **Neon Labs team** acknowledged this issue and won't take any action since Neon Labs states that the owner is already a multi-sig address.

7.9 (HAL-11) LACK OF INPUT VALIDATION IN ADDALREADYEXISTINGERC20FORSPL()

// INFORMATIONAL

Description

The `addAlreadyExistingERC20ForSPL` function in the `ERC20ForSPLFactory` contract lacks comprehensive validation checks for input parameters `tokenMints` and `alreadyExistingTokens`. This omission permits the registration of invalid or zero addresses as ERC20 token addresses corresponding to Solana Token Mints, potentially compromising the integrity of the contract's state and leading to operational disruptions.
No validation checks to ensure that neither elements in the `tokenMints` array are zero bytes (`bytes32(0)`) nor elements in the `alreadyExistingTokens` array are zero addresses (`address(0)`) can lead to the following issues:

- **Erroneous State Modification:** Malicious or accidental addition of zero addresses or zero bytes can pollute the contract's storage, leading to a misleading representation of token mappings.
- **Operational Disruption:** Interaction with invalid addresses may result in failed transactions or unintended behaviors in dependent functionalities, impacting the contract's reliability.

BVSS

AO:S/AC:L/AX:L/C:L/I:L/A:L/D:L/Y:L/R:N/S:U (1.0)

Recommendation

Adding the following 2 checks within `addAlreadyExistingERC20ForSPL()` is recommended:

```
// Validate that the token mint is not zero bytes
if (tokenMints[i] == bytes32(0)) revert InvalidTokenData();

// Validate that the ERC20 token address is not a zero address
if (alreadyExistingTokens[i] == address(0)) revert InvalidTokenData();
```

Remediation Plan

SOLVED: The **Neon Labs team** solved the issue by adding validation checks.

Remediation Hash

https://github.com/neonlabsorg/neon-contracts/commit/16de9d6c1b1ee0f74d2d6259480122fa91dc0068

7.10 (HAL-10) MISSING EVENT EMISSION ON ADDALREADYEXISTINGERC20FORSPL()

// INFORMATIONAL

Description

It was identified that the `addAlreadyExistingERC20ForSPL()` function from the `ERC20ForSPLFactory.sol` contract does not emit any events. These functions can only be used by the Owner to manage the contract. As a result, it might be more difficult for blockchain monitoring systems to detect suspicious behavior related to this feature.

```
    function addAlreadyExistingERC20ForSPL(bytes32[] memory tokenMints, address[]
memory alreadyExistingTokens) external onlyOwner {
        uint tokensLen = alreadyExistingTokens.length;
        if (tokensLen != tokenMints.length) revert InvalidTokenData();

        for (uint i; i < tokensLen; ++i) {
            if (tokensData[tokenMints[i]].token != address(0)) revert
AlreadyExistingERC20ForSPL();

            tokensData[tokenMints[i]] = Token({
                token: alreadyExistingTokens[i],
                state: State.AlreadyExisting
            });
            tokens.push(alreadyExistingTokens[i]);
        }
    }
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:C (0.0)

Recommendation

It is recommended to add events for all important operations to help monitor the contract, as a monitoring system that tracks relevant events would allow the timely detection of compromised system components.

Remediation Plan

SOLVED: The **Neon Labs team** solved the issue by adding an event.

Remediation Hash

https://github.com/neonlabsorg/neon-contracts/commit/16de9d6c1b1ee0f74d2d6259480122fa91dc0068

7.11 (HAL-12) OMISSION OF __UUPSUPGRADEABLE_INIT()

// INFORMATIONAL

Description

The `ERC20ForSPLFactory` and `ERC20ForSPLMintableFactory` contracts, designed for deploying interface contracts for SPL Tokens on Solana, utilize the UUPS (Universal Upgradeable Proxy Standard) upgrade pattern through inheritance from OpenZeppelin's `UUPSUpgradeable`. During the initialization process, these contracts explicitly invoke `__Ownable_init(msg.sender)` to initialize the `OwnableUpgradeable` aspect of their functionality. However, they notably omit the call to `__UUPSUpgradeable_init()`.
While the current implementation of `__UUPSUpgradeable_init()` in OpenZeppelin's contracts does not perform any operations and is essentially a placeholder function, its invocation is considered a best practice for initializing all aspects of inherited upgradeable functionality. This practice enhances code readability, maintains consistency with the initialization pattern of upgradeable contracts, and ensures future compatibility in case the `__UUPSUpgradeable_init()` function is modified to include initialization logic.

Score

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

It is advised to modify the `initialize` functions in both `ERC20ForSPLFactory` and `ERC20ForSPLMintableFactory` contracts to include a call to `__UUPSUpgradeable_init()`. This addition ensures that all inherited functionalities are explicitly initialized, even if they currently do not perform any operations.

Remediation Plan

SOLVED: The **Neon Labs team** solved the issue and added necessary modifier `__UUPSUpgradeable_init()`

Remediation Hash

f68dfb430c137eb989af0611dc93c2df93673143

8. AUTOMATED TESTING

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was `Slither`, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

```
Reentrancy in ERC20ForSPLFactory.deploy(bytes32) (contracts/ERC20ForSPLFactory.sol#127-148):
        External calls:
        - token = new BeaconProxy(address(this),abi.encodeWithSelector(ERC20ForSPL(address(0)).initialize.selector,tokenMint)) (contracts/ERC20ForSPLFactory.sol#133-137)
        State variables written after the call(s):
        - tokensData[tokenMint] = Token({token:address(token),state:State.New}) (contracts/ERC20ForSPLFactory.sol#140-143)
        ERC20ForSPLFactory.tokensData (contracts/ERC20ForSPLFactory.sol#31) can be used in cross function reentrancies:
        - ERC20ForSPLFactory.addAlreadyExistingERC20ForSPL(bytes32[],address[]) (contracts/ERC20ForSPLFactory.sol#96-122)
        - ERC20ForSPLFactory.deploy(bytes32) (contracts/ERC20ForSPLFactory.sol#127-148)
        - ERC20ForSPLFactory.tokensData (contracts/ERC20ForSPLFactory.sol#31)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
ERC1967Utils.upgradeToAndCall(address,bytes) (contracts/openzeppelin-fork/contracts/proxy/ERC1967/ERC1967Utils.sol#83-92) ignores return value by Address.functionDelegateCall(newImplementation,data) (contracts/openzeppelin-fork/contracts/proxy/ERC1967/ERC1967Utils.sol#88)
ERC1967Utils.upgradeBeaconToAndCall(address,bytes) (contracts/openzeppelin-fork/contracts/proxy/ERC1967/ERC1967Utils.sol#173-182) ignores return value by Address.functionDelegateCall(IBeacon(newBeacon).implementation(),data) (contracts/openzeppelin-fork/contracts/proxy/ERC1967/ERC1967Utils.sol#178)
ERC20ForSPLMintable._initialize(string,string,string,uint8) (contracts/ERC20ForSPLMintable.sol#52-62) ignores return value by METAPLEX.createMetadata(mintAddress,_name,_symbol,_uri) (contracts/ERC20ForSPLMintable.sol#60)
ERC20ForSPLMintable.mint(address,uint256) (contracts/ERC20ForSPLMintable.sol#71-87) ignores return value by SPL_TOKEN.initializeAccount(_salt(to),tokenMint) (contracts/ERC20ForSPLMintable.sol#79)
ERC20ForSPLBackbone.claimTo(bytes32,address,uint64) (contracts/ERC20ForSPLBackbone.sol#249-277) ignores return value by SPL_TOKEN.initializeAccount(_salt(to),tokenMint) (contracts/ERC20ForSPLBackbone.sol#261)
ERC20ForSPLBackbone._transfer(address,address,uint256) (contracts/ERC20ForSPLBackbone.sol#316-336) ignores return value by SPL_TOKEN.initializeAccount(_salt(to),tokenMint) (contracts/ERC20ForSPLBackbone.sol#330)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
INFO:Detectors:
BeaconProxy.constructor(address,bytes).beacon (contracts/openzeppelin-fork/contracts/proxy/beacon/BeaconProxy.sol#39) lacks a zero-check on :
                - _beacon = beacon (contracts/openzeppelin-fork/contracts/proxy/beacon/BeaconProxy.sol#41)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
INFO:Detectors:
Reentrancy in ERC20ForSPLFactory.deploy(bytes32) (contracts/ERC20ForSPLFactory.sol#127-148):
        External calls:
        - token = new BeaconProxy(address(this),abi.encodeWithSelector(ERC20ForSPL(address(0)).initialize.selector,tokenMint)) (contracts/ERC20ForSPLFactory.sol#133-137)
        State variables written after the call(s):
        - tokens.push(address(token)) (contracts/ERC20ForSPLFactory.sol#144)
Reentrancy in ERC20ForSPLMintableFactory.deploy(string,string,string,uint8) (contracts/ERC20ForSPLMintableFactory.sol#77-106):
        External calls:
        - token = new BeaconProxy(address(this),abi.encodeWithSelector(ERC20ForSPLMintable(address(0)).initialize.selector,_name,_symbol,_uri,_decimals,msg.sender)) (contracts/ERC20ForSPLMintableFactory.sol#85-96)
        - tokenMint = IERC20ForSPLMintable(address(token)).tokenMint() (contracts/ERC20ForSPLMintableFactory.sol#98)
        State variables written after the call(s):
        - tokens.push(address(token)) (contracts/ERC20ForSPLMintableFactory.sol#102)
        - tokensData[tokenMint] = address(token) (contracts/ERC20ForSPLMintableFactory.sol#101)
Reentrancy in ERC20ForSPLMintable.initialize(string,string,string,uint8,address) (contracts/ERC20ForSPLMintable.sol#32-49):
        External calls:
        - _tokenMint = _initialize(_name,_symbol,_uri,_decimals) (contracts/ERC20ForSPLMintable.sol#42)
                - mintAddress = SPL_TOKEN.initializeMint(bytes32(0),_decimals) (contracts/ERC20ForSPLMintable.sol#59)
                - METAPLEX.createMetadata(mintAddress,_name,_symbol,_uri) (contracts/ERC20ForSPLMintable.sol#60)
        State variables written after the call(s):
        - tokenMint = _tokenMint (contracts/ERC20ForSPLMintable.sol#48)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2
INFO:Detectors:
Reentrancy in ERC20ForSPLBackbone._burn(address,uint256) (contracts/ERC20ForSPLBackbone.sol#300-312):
        External calls:
        - SPL_TOKEN.burn(tokenMint,fromSolana,uint64(amount)) (contracts/ERC20ForSPLBackbone.sol#309)
        Event emitted after the call(s):
        - Transfer(from,address(0),amount) (contracts/ERC20ForSPLBackbone.sol#311)
Reentrancy in ERC20ForSPLBackbone._transfer(address,address,uint256) (contracts/ERC20ForSPLBackbone.sol#316-336):
        External calls:
        - SPL_TOKEN.initializeAccount(_salt(to),tokenMint) (contracts/ERC20ForSPLBackbone.sol#330)
        - SPL_TOKEN.transfer(fromSolana,toSolana,uint64(amount)) (contracts/ERC20ForSPLBackbone.sol#334)
        Event emitted after the call(s):
        - Transfer(from,to,amount) (contracts/ERC20ForSPLBackbone.sol#335)
Reentrancy in ERC20ForSPLBackbone.approveSolana(bytes32,uint64) (contracts/ERC20ForSPLBackbone.sol#185-211):
```

```
Reentrancy in ERC20ForSPLBackbone.approveSolana(bytes32,uint64) (contracts/ERC20ForSPLBackbone.sol#185-211):
        External calls:
        - SPL_TOKEN.approve(fromSolana,spender,amount) (contracts/ERC20ForSPLBackbone.sol#202)
        - SPL_TOKEN.revoke(fromSolana) (contracts/ERC20ForSPLBackbone.sol#205)
        Event emitted after the call(s):
        - Approval(from,address(0),amount) (contracts/ERC20ForSPLBackbone.sol#208)
        - ApprovalSolana(from,spender,amount) (contracts/ERC20ForSPLBackbone.sol#209)
Reentrancy in ERC20ForSPLBackbone.claimTo(bytes32,address,uint64) (contracts/ERC20ForSPLBackbone.sol#249-277):
        External calls:
        - SPL_TOKEN.initializeAccount(_salt(to),tokenMint) (contracts/ERC20ForSPLBackbone.sol#261)
        - SPL_TOKEN.transferWithSeed(_salt(msg.sender),from,toSolana,amount) (contracts/ERC20ForSPLBackbone.sol#274)
        Event emitted after the call(s):
        - Transfer(address(0),to,amount) (contracts/ERC20ForSPLBackbone.sol#275)
Reentrancy in ERC20ForSPLFactory.deploy(bytes32) (contracts/ERC20ForSPLFactory.sol#127-148):
        External calls:
        - token = new BeaconProxy(address(this),abi.encodeWithSelector(ERC20ForSPL(address(0)).initialize.selector,tokenMint)) (contracts/ERC20ForSPLFactory.sol#133-137)
        Event emitted after the call(s):
        - TokenDeploy(tokenMint,address(token)) (contracts/ERC20ForSPLFactory.sol#146)
Reentrancy in ERC20ForSPLMintableFactory.deploy(string,string,string,uint8) (contracts/ERC20ForSPLMintableFactory.sol#77-106):
        External calls:
        - token = new BeaconProxy(address(this),abi.encodeWithSelector(ERC20ForSPLMintable(address(0)).initialize.selector,_name,_symbol,_uri,_decimals,msg.sender)) (contracts/ERC20ForSPLMintableFactory.sol#85-96)
        - tokenMint = IERC20ForSPLMintable(address(token)).tokenMint() (contracts/ERC20ForSPLMintableFactory.sol#98)
        Event emitted after the call(s):
        - TokenDeploy(tokenMint,address(token)) (contracts/ERC20ForSPLMintableFactory.sol#104)
Reentrancy in ERC20ForSPLMintable.mint(address,uint256) (contracts/ERC20ForSPLMintable.sol#71-87):
        External calls:
        - SPL_TOKEN.initializeAccount(_salt(to),tokenMint) (contracts/ERC20ForSPLMintable.sol#79)
        - SPL_TOKEN.mintTo(tokenMint,toSolana,uint64(amount)) (contracts/ERC20ForSPLMintable.sol#85)
        Event emitted after the call(s):
        - Transfer(address(0),to,amount) (contracts/ERC20ForSPLMintable.sol#86)
Reentrancy in ERC20ForSPLBackbone.transferSolana(bytes32,uint64) (contracts/ERC20ForSPLBackbone.sol#217-236):
        External calls:
        - SPL_TOKEN.transfer(fromSolana,to,uint64(amount)) (contracts/ERC20ForSPLBackbone.sol#231)
        Event emitted after the call(s):
        - Transfer(from,address(0),amount) (contracts/ERC20ForSPLBackbone.sol#233)
        - TransferSolana(from,to,amount) (contracts/ERC20ForSPLBackbone.sol#234)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
```

No major issues were found by slither.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.