Alongside - Universal Contracts report (Fees update)

Audit summary

Date: 26/06/2024 - 08/07/2024
Initial commit id: b4dab67dc4d463afd526212eeed085efe74e0364
Fixed commit id: 02e4685eb925cf148666222d976f63520c718a7f
Initial commit id (Fees update): 240d0a49e32b3870a6f35e4ef24a1bbdf931e2a1
Fixed commit id (Fees update): 7503818acaa834733b3baa1711bad7c0edb2bc6e

Severity	Count
Low	3
Informational	4

Issue Severity	Description	Recommendation	Status
Low	Project may fail to be deployed to chains not compatible with Shanghai hard-fork	Update the Solidity compiler version to 0.8.19 or define an EVM version compatible across all intended chains.	SOLVED
Low	Missing event emission when mintedAmt is not equal to amount in the _mint() function	Emit an event in the _mint() function whenever the mint limit is exceeded and a manual executeQueue() call is required.	SOLVED
Low	Possible inconsistency with unused signature nonces	Use a different nonce for each operation to avoid collisions.	RISK ACCEPTED
Informational	Ensure that the WrapFactory contract is deployed with the same account and nonce across all chains	Ensure that the WrapFactory contract is deployed with the same deployer address and nonce across all chains.	SOLVED
Informational	Index parameter can be removed from Epoch struct	Remove the index parameter in the Epoch struct to save gas costs.	SOLVED
Informational	Unused imports	Remove the console2 import from the WrappedAsset contract.	SOLVED
Informational	Redundant gas optimizations in for loops	Remove the unchecked code blocks used to increment the iterator in for loops	SOLVED

About r0bert

r0bert is an independent smart contract security researcher. He serves as a Security Researcher at Spearbit and also leads the Solidity smart contract audit team at a popular Web3 Cybersecurity firm. r0bert has meticulously audited a wide spectrum of notable projects, including Algorithmic stable coins, Lending protocols, Decentralized Exchanges, AMMs, DAOs, Blockchain games...

You can find him on Twitter at 0xr0berth.

LOW - Project may fail to be deployed to chains not compatible with Shanghai hard-fork - SOLVED

Relevant Context

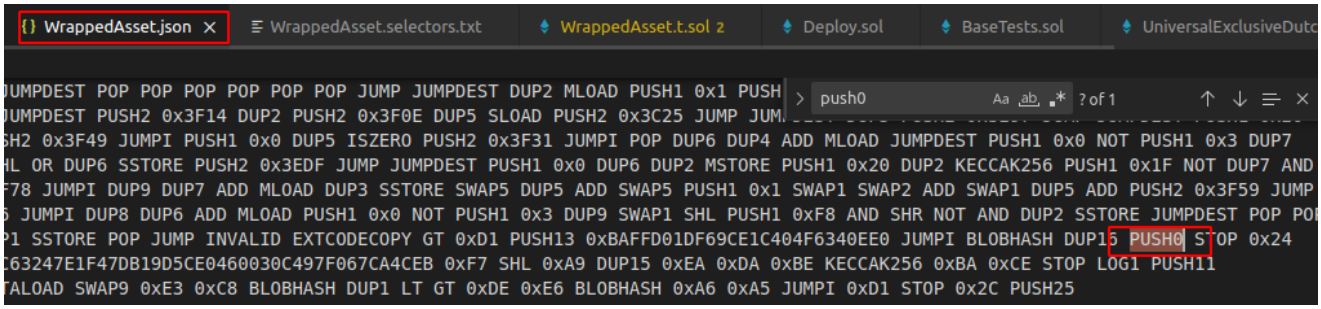
- foundry.toml#L5
- WrapFactory.sol#L2
- WrappedAsset.sol#L2

Description

The current compiler version used across the smart contracts (0.8.25) may produce incompatible bytecode with some of the chains supported by the protocol. The Universal contracts should support and target different chains, such as Ethereum, Polygon, Avalanche, BNB, Optimism, Arbitrum and possibly Linea...

All of the contracts in scope have the pragma version fixed to be compiled using Solidity 0.8.25. This new version of the compiler uses the new PUSH0 opcode introduced in the Shanghai hard-fork, which is now the default EVM version in the compiler and the one being currently used to compile the project.

Here is a part of the bytecode produced for the `WrappedAsset` contract, in which we can see the presence of the PUSH0 opcode (full bytecode can be found in the file out/WrappedAsset.sol/WrappedAsset.json) after compiling with `forge build --force --extra-output evm.bytecode.opcodes`:



This means that the produced bytecode for the different contracts won't be compatible with the chains that don't yet support the Shanghai hard-fork or which is the same, the PUSH0 opcode. You can check the support for this `opcode` in the link below:

- https://www.evmdiff.com/features?feature=opcodes

Currently, Linea is the only blockchain that does not support the PUSH0 opcode.

Recommendation

If considering to deploy in Linea blockchain, update the Solidity compiler version to 0.8.19 or define an evm version, which is compatible across all of the intended chains to be supported by the protocol (see https://book.getfoundry.sh/reference/config/solidity-compiler?highlight=evm_vers#evm_version).

Status: SOLVED

ALONGSIDE: This is already mitigated as there are no plans to deploy in the Linea blockchain. However, a new version of this contract using the 0.8.19 compiler version will be used to deploy in such blockchains.

r0bert: OK.

LOW - Missing event emission when mintedAmt is not equal to amount in the _mint() function - SOLVED

Relevant Context

- WrappedAsset.sol#L412-L421

Description

The `WrappedAsset` contract implements the function `_calculateMintLimit()`:

```
function _calculateMintLimit(address account, uint256 amount) internal
returns (uint256) {
    Epoch memory epoch = lastEpoch[account];
    uint256 limitAmt = mintLimit[account];

    if ((block.timestamp - epoch.startTimestamp) / EPOCH_DURATION != 0) {
        if (amount <= limitAmt) {
            // Create new epoch
            unchecked {
                lastEpoch[account].index++;
            }
            lastEpoch[account].startTimestamp =
block.timestamp.toUint48();
            lastEpoch[account].minted = 0;

            return amount;
        } else {
            // Create new epoch
            unchecked {
                lastEpoch[account].index++;
            }
            lastEpoch[account].startTimestamp =
block.timestamp.toUint48();
            lastEpoch[account].minted = 0;

            return limitAmt;
        }
    } else {
        if (amount + epoch.minted > limitAmt) {
            if (epoch.minted < limitAmt) {
                return limitAmt - epoch.minted;
            } else {
                return 0;
```

```
            }
        } else {
            return amount;
        }
    }
}
```

This function enforces that the mint limit is never surpassed by a minter during an epoch. Consequently, if a minter/merchant tries to mint an amount that will surpass this limit, this function will return the remaining amount that can be minted. For example:

- Mint limit: 200 tokens.
- Current minted tokens during the Epoch: 150 tokens.
- Amount trying to mint by the minter: 200 tokens.
- As 250+100 = 350, which is higher than the 200 tokens set as the mint limit, `_calculateMintLimit()` will return 50 and the remaining 150 will have to be approved and minted manually by the Alongside protocol (`EXECUTOR_ROLE`) through the `executeQueue()` function.

The remaining amount of tokens to mint are saved in the `mapping(bytes32 => OperationState) internal mintQueueState;` .

```
// If the full amount cannot be minted, queue the remaining.
if (mintedAmt != amount) {
    /* Queue the remaining amount, the tail of the queue is used as salt
to avoid collisions
    even if two mint operations with the same arguments come in the same
block.
    */
    bytes32 id = queueId(minter, amount - mintedAmt, block.timestamp,
mintQueueTail);

    mintQueueState[id] = OperationState.Waiting;
    mintQueueTail = id;
}
```

However, no event is emitted when this occurs and therefore the backend would not be able to retrieve this value easily later on during the `executeQueue` call:

```
function executeQueue(address minter, uint256 amount, uint256 timestamp,
bytes32 predecessor)
    external
    onlyRole(EXECUTOR_ROLE)
{
    (bytes32 id, OperationState state) = getQueuedOperation(minter,
amount, timestamp, predecessor);
```

```
    if (state == OperationState.Unset) {
        revert OperationDoesNotExist();
    } else if (state == OperationState.Minted) {
        revert OperationAlreadyMinted();
    } else if (state == OperationState.Expired) {
        revert OperationExpired();
    } else {
        mintQueueState[id] = OperationState.Minted;
        _mint(minter, amount);
    }
}
```

An event should be emitted to:

- Notify the backend that this scenario occurred.
- Notify the backend the different data that forms the id: minter, amount, timestamp & predecessor so the `executeQueue()` call can be generated easily.

On the other hand, events should also be emitted on important state changes, for example:

- When a new mint limit is set through the `setMintLimit()` function.
- When a new chain is whitelisted through the `setChainWhitelist()` function.
- When a user is blacklisted through the `setUserBlacklist()` function.

Recommendation

Consider emitting an event in the `_mint()` function whenever the mint limit is exceeded and a manual `executeQueue()` call is required:

```
function _mint(address minter, address witness, uint256 amount) internal
returns (uint256) {
    uint256 minterAmt = _calculateMintLimit(minter, amount);
    uint256 witnessAmt = _calculateMintLimit(witness, amount);

    // Mint the lowest amount of both.
    uint256 mintedAmt = minterAmt < witnessAmt ? minterAmt : witnessAmt;
    _mint(minter, mintedAmt);

    // Update the epochs with the amounts
    if (mintedAmt != 0) {
        lastEpoch[minter].minted += mintedAmt.toUint192();
        lastEpoch[witness].minted += mintedAmt.toUint192();
    }

    // If the full amount cannot be minted, queue the remaining.
```

```
    if (mintedAmt != amount) {
        /* Queue the remaining amount, the tail of the queue is used as
salt to avoid collisions
        even if two mint operations with the same arguments come in the
same block.
        */
        bytes32 id = queueId(minter, amount - mintedAmt, block.timestamp,
mintQueueTail);

                emit OperationQueued(minter, amount - mintedAmt,
block.timestamp, mintQueueTail); // <--------------------

        mintQueueState[id] = OperationState.Waiting;
        mintQueueTail = id;
    }

    return mintedAmt;
}
```

Moreover, it is also recommended to emit events on important state changes:

- When a new mint limit is set through the `setMintLimit()` function.
- When a new chain is whitelisted through the `setChainWhitelist()` function.
- When a user is blacklisted through the `setUserBlacklist()` function.

Status: SOLVED

ALONGSIDE: Solved in the following commit id: 922a2db95e1c2aed3ba6d1913c0a9b6ab9ec4d27.

r0bert: Fix OK.

LOW - Possible inconsistency with unused signature nonces - RISK ACCEPTED

Relevant Context

- WrappedAsset.sol#L157
- WrappedAsset.sol#L178
- WrappedAsset.sol#L198
- WrappedAsset.sol#L229
- WrappedAsset.sol#L259

Description

In the `WrappedAsset` contract the following functions make use of the `_useNonce()` function as a mitigation against signature replay attacks:

Function Signature	Signatures Provided By	Nonce Consumed By
`mint(uint256 amount, uint256 expiration, bytes memory attestationSig)`	`WITNESS_ROLE`	`MINTER` /caller
`mint(address minter, uint256 amount, uint256 expiration, bytes memory minterSig, bytes memory attestationSig)`	`WITNESS_ROLE` & `MINTER_ROLE`	`MINTER`
`burn(address burner, uint256 amount, uint256 expiration, bytes memory signature)`	`BURNER_ROLE`	Caller
`initiateBridge(address user, uint256 amount, uint256 destinationChain, uint256 expiration, bytes memory signature)`	User	User
`finishBridge(address user, address witness, uint256 amount, uint256 originChain, uint256 expiration, bytes memory signature)`	`WITNESS_ROLE`	Witness

However, let's imagine the following scenario:

1. A merchant with the `BURNER_ROLE` provides a signature for Alice using the nonce 1, so she can call the `burn(address burner, uint256 amount, uint256 expiration, bytes memory signature)` function using the merchant's signature.
2. Alice does not perform the call yet and the Merchant executes a buy in Coinbase and consequently the backend (`WITNESS_ROLE`) provides the Merchant a signature to call

the `mint(uint256 amount, uint256 expiration, bytes memory attestationSig)` function with the nonce 1.

3. Alice performs the call to the `burn()` function, consuming the nonce 1 of the Merchant.
4. The Merchant calls now `mint(uint256 amount, uint256 expiration, bytes memory attestationSig)` with the provided `WITNESS` signature, however it reverts as the nonce 1 was already consumed.

Recommendation

Instead of making use of the `_useNonce()` function inherited from the `lib/openzeppelin-contracts-upgradeable/contracts/utils/NoncesUpgradeable.sol`:

```
function _useNonce(address owner) internal virtual returns (uint256) {
    // For each account, the nonce has an initial value of 0, can only be
incremented by one, and cannot be
    // decremented or reset. This guarantees that the nonce never
overflows.
    unchecked {
        // It is important to do x++ and not ++x here.
        return _nonces[owner]++;
    }
}
```

Use a different nonce for each operation. For example:

```
mapping(uint256 => mapping(address => uint256)) public _nonces;

/*
        Use type == 0 for minting
        Use type == 1 for burning
        Use type == 2 for bridge operations
*/
function _useNonce(uint256 type, address owner) internal virtual returns
(uint256) {
    unchecked {
        return _nonces[type][owner]++;
    }
}
```

This way, there is no room for any type of "collision" for using the same nonce signatures in multiple operations.

Status: RISK ACCEPTED

ALONGSIDE: Alongside does not anticipate that merchants will share their signatures with external third parties. In the event a third party takes control of a merchant's key, that person will only be able to initiate actions that send assets to (mint) or pull assets from (redeem) the merchant's wallet, since only whitelisted merchant wallets are permitted to interact with the smart contract. For that reason, the actual impact of such an attack is low.

INFORMATIONAL - Ensure that the WrapFactory contract is deployed with the same account and nonce accross all chains - SOLVED

Relevant Context

- WrapFactory.sol#L13
- WrapFactory.sol#L24-L26

Description

The `WrapFactory` contract implements the function `deployBeaconProxy()`:

```
function deployBeaconProxy(
    string memory name,
    string memory symbol,
    WrappedAsset.Operator[] calldata operators,
    uint256[] calldata chains
) external onlyOwner returns (address) {
    bytes32 _salt = keccak256(abi.encodePacked(name, symbol));

    bytes memory data =
        abi.encodeWithSelector(WrappedAsset.initialize.selector, name,
symbol, owner(), operators, chains);
    return address(new BeaconProxy{salt: _salt}(address(beacon), data));
}
```

This function makes use of the `CREATE2` opcode to deploy a `BeaconProxy`. The main purpose of the function is that all the different `WrappedAssets` contracts with the same `name` and `symbol` are deployed in the same address accross all chains.

The `BeaconProxy` address deployed is calculated as:

$$\text{address} = \text{keccak256}(\text{0xFF} || \text{sender} || \text{salt} || \text{keccak256}(\text{init\_code}))[12:]$$

Where:

- `0xFF` is a constant prefix.
- `sender` is the address of the contract deploying the `BeaconProxy`, or which is the same, the address of `WrapFactory`.
- `salt` is the value derived from `keccak256(abi.encodePacked(name, symbol))`.
- `keccak256(init_code)` is the hash of the initialization code, which includes the `beacon` address.

However, the `WrapFactory` address is deployed in a standard way through the normal `CREATE` opcode. The formula for calculating the address of a contract deployed with `CREATE` is:

$$\text{address} = keccak256(rlp.encode([sender, nonce]))$$

Therefore, in order to guarantee that all the `WrappedAssets` contracts are deployed consistently in the same addresses across all the different chains the `WrapFactory` contract should be deployed with the same deployer address and nonce across all chains.

Recommendation

Ensure that the `WrapFactory` contract is deployed with the same deployer address and nonce across all chains.

Status: SOLVED

ALONGSIDE: This is already mitigated in the `BaseScript` [deployment script]() as the `WrapFactory` will be also deployed using `CREATE2` in a predeterministic address:

```
function _deployFactory() internal returns (address, address) {
    vm.startBroadcast(multisigKey);

    WrappedAsset implementation = new WrappedAsset{salt: deploySalt}();
    WrapFactory factory = new WrapFactory{salt: deploySalt}
(address(implementation), multisig); // <-----------------------------

    vm.stopBroadcast();

    address beacon = address(factory.beacon());

    console2.log("Wrap Factory: ", address(factory));
    console2.log("Upgradeable Beacon: ", beacon);

    return (address(factory), beacon);
}
```

r0bert: Fix OK.

INFORMATIONAL - index parameter can be removed from Epoch struct - SOLVED

Relevant Context

- WrappedAsset.sol#L71
- WrappedAsset.sol#L368-L370
- WrappedAsset.sol#L377-379

Description

The `WrappedAsset` contract implements the `Epoch` struct:

```
struct Epoch {
    uint16 index;
    uint48 startTimestamp;
    uint192 minted;
}
```

However, the `index` parameter which is written in the `_calculateMintLimit()` function does not really provide any utility and is not really used anywhere in the code. Therefore, it is recommended to remove it in order to avoid a `SSTORE` operation and hence reducing the gas costs:

```
function _calculateMintLimit(address account, uint256 amount) internal
returns (uint256) {
    Epoch memory epoch = lastEpoch[account];
    uint256 limitAmt = mintLimit[account];

    if ((block.timestamp - epoch.startTimestamp) / EPOCH_DURATION != 0) {
        if (amount <= limitAmt) {
            // Create new epoch
            unchecked {
                lastEpoch[account].index++; // <------------------
            }
            lastEpoch[account].startTimestamp =
block.timestamp.toUint48();
            lastEpoch[account].minted = 0;

            return amount;
        } else {
            // Create new epoch
            unchecked {
                lastEpoch[account].index++; // <------------------
```

```
            }
            lastEpoch[account].startTimestamp =
block.timestamp.toUint48();
            lastEpoch[account].minted = 0;

            return limitAmt;
        }
    } else {
        if (amount + epoch.minted > limitAmt) {
            if (epoch.minted < limitAmt) {
                return limitAmt - epoch.minted;
            } else {
                return 0;
            }
        } else {
            return amount;
        }
    }
}
```

Recommendation

It is recommended to update the Epoch struct as shown below:

```
struct Epoch {
    uint64 startTimestamp;
    uint192 minted;
}
```

On the other hand, remove the update of the old `Epoch.index` parameter in the `_calculateMintLimit()` function. This way:

1. Timestamps are treated as `uint64` and consequently support higher *epoch*/*Unix* timestamps.
2. A `SSTORE` operation is saved with each `_mint()` call.

Status: SOLVED

ALONGSIDE: Solved in the following commit id: 5e890c1d0eefe07f287f56ae31a73bb234cc5810.

r0bert: Fix OK.

INFORMATIONAL - Unused imports - SOLVED

Relevant Context

- WrappedAsset.sol#L4

Description

The `WrappedAsset` contract imports the `console2` library:

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

While the `console2` library is a useful library for debugging during the development and testing phases, it should not be used in production code.

Recommendation

It is recommended to remove the `console2` import from the `WrappedAsset` contract.

Status: SOLVED

ALONGSIDE: Solved in the following commit id: 02e4685eb925cf148666222d976f63520c718a7f.

r0bert: Fix OK.

INFORMATIONAL - Redundant gas optimizations in for loops - SOLVED

Relevant Context

- WrappedAsset.sol#L133-L135
- WrappedAsset.sol#L140-L142

Description

Solidity 0.8.22 introduces an overflow check optimization that automatically generates an unchecked arithmetic increment of the counter of for loops. This new optimization removes the need for poor unchecked increment patterns in for loop bodies such as the ones shown below:

```
for (uint256 i; i < length;) {
    if (operators[i].operator == address(0)) revert ZeroAddress();
    if (operators[i].role == MINTER_ROLE || operators[i].role ==
WITNESS_ROLE) {
        _grantRole(operators[i].role, operators[i].operator);
        _setMintLimit(operators[i].operator, operators[i].limit);
    } else if (operators[i].role == BURNER_ROLE || operators[i].role ==
EXECUTOR_ROLE) {
        _grantRole(operators[i].role, operators[i].operator);
    } else {
        revert InvalidRole();
    }
    unchecked {
        ++i;
    }
}
length = chains.length;
for (uint256 i; i < length;) {
    _setChainWhitelist(chains[i], true);
    unchecked {
        ++i;
    }
}
```

In contrast, the new optimization enables users to return to the original, more readable code without sacrificing gas efficiency.

The precise conditions under which the new optimization avoids the overflow check are the following:

- The loop condition is a comparison of the form `i < ...` for a local variable i (called "loop counter" from now on).
- This comparison must be performed on the same type as the loop counter, i.e. the type of the right-hand-side must be implicitly convertible to the type of the loop counter, such that the loop counter is not implicitly widened before comparing.
- The loop counter must be a local variable of a built-in integer type.
- The loop expression must be a prefix or postfix increment of the loop counter, i.e. i++ or ++i.
- The loop counter may not be modified in the loop condition or the loop body.

As all these conditions are met in the `WrappedAsset` contract, it is recommended to remove the unchecked code blocks in the `WrappedAsset.initialize()` function as they are redundant.

Reference

- solidity-0.8.22-release-announcement

Recommendation

It is recommended to remove the unchecked code blocks used to increment the iterator in all the for loops as the contracts are already using the `0.8.25` Solidity compiler version.

Status: SOLVED

ALONGSIDE: Solved in the following commit id: 7503818acaa834733b3baa1711bad7c0edb2bc6e.

r0bert: Fix OK.