[M-1] Missing Deadline Check in Forwarder contract leads to Indefinite execution

Description

The `Forwarder::ForwardRequest` struct lacks a deadline parameter for signed requests. Currently, requests only validate nonces and signatures without any time-based expiration mechanism. This means signed transfer requests remain valid indefinitely

Key missing validations:

- No timestamp-based expiration check
- No mechanism to automatically invalidate stale requests

Impact

1. **Indefinite Validity of Requests:**
   Without a deadline, each `ForwardRequest` is potentially valid indefinitely. Once signed, it can be executed at any future time as long as the nonce remains valid. This means that the execution context —such as contract state, or other external conditions—may have changed drastically from when the request was originally signed. Signers thus lose control over the timing of their transactions.

2. **Delayed or Malicious Execution:**
   Relayers or any intermediaries with access to the signed request can withhold its execution. For instance, if a user signs a time-sensitive transfer (e.g., to participate in a discounted token sale or to meet a flash loan deadline), a relayer could execute the transaction well past the intended window, leading to unintended financial outcomes.

These issues reduce user control and trust in the protocol, especially in a meta-transaction system where users delegate gas management to relayers.

Proof of Concept

1. User signs a gasless transfer request for a time-sensitive payment:

```
ForwardRequest {
    from: userAddress,
    to: EURFAddress,
    value: 100e18,  // 100 EURF
    gas: 50000,
    nonce: 5,
    data: transferFunctionData
}
```

2. Relayer can hold this request and execute it at any future time:

```
// Can be called at any time
forwarder.execute(
    storedRequest,
    domainSeparator,
    requestTypeHash,
    suffixData,
    signature
);
```

Recommended Mitigation

Add a deadline parameter to the `ForwardRequest` struct and validate it during execution:

```
struct ForwardRequest {
    address from;
    address to;
    uint256 value;
    uint256 gas;
    uint256 nonce;
    bytes data;
+   uint256 deadline;  // NEW: timestamp after which request expires
}

function execute(
    ForwardRequest calldata req,
    bytes32 domainSeparator,
    bytes32 requestTypeHash,
    bytes calldata suffixData,
    bytes calldata sig
) external payable returns (bool success, bytes memory ret) {
+    require(block.timestamp <= req.deadline, "NGEUR Forwarder: request expired");
    // ... rest of the existing execution logic
}
```

Also update `GENERIC_PARAMS` to include the new parameter:

```
string public constant GENERIC_PARAMS =
+    "address from,address to,uint256 value,uint256 gas,uint256 nonce,bytes
data,uint256 deadline";
```

Tools Used

Manual Review

[M-2] Uninitialized Implementation Contract Vulnerability

Description

The UUPS implementation contract lacks `_disableInitializers()` in its constructor, exposing it to:

- Direct initialization attacks on implementation contract
- Proxy front-running attacks

Impact

Critical Severity - Attackers could:

1. Deploy malicious proxy to implementation
2. Call `initialize()` with their parameters
3. Gain full admin control of token contract

Proof of Concept

1. Attacker deploys EURFToken implementation contract
2. Without initializer lock, attacker deploys proxy pointing to implementation
3. Attacker calls `initialize()` through proxy:
4. Attacker gains full administrative control

Recommended Mitigation

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
+    _disableInitializers();
}
```

Tools Used

Manual Review

[L-1] Wrong Natspec in `Token::setTxFeeRate` function

Description:

The `@param` natspec in `Token::setTxFeeRate` function, describes `newRate` as `"the address of the minter"` but it is a uint256 value which is meant to be set as a new Transaction fee rate

Impact:

Misleads/Confuses developers and auditors

Proof of Concept:

Check line 115-122 in `Token`

Recommended Mitigation:

Write the correct use case of the `newRate` parameter

```
    /**
     * @dev Function to update tx fee rate
-     * @param newRate The address of the minter
+     * @param newRate The new transaction rate to be set
     */
    function setTxFeeRate(uint256 newRate) public override onlyRole(ADMIN) {
        super.setTxFeeRate(newRate);
    }
```

Tools used:

- Manual Review

[L-2] Redundant allowance update functionality in `addMinter` and `updateMintingAllowance`

Description:

The `addMinter` function updates the `minterAllowed` mapping while granting the `MINTER_ROLE`, while `updateMintingAllowance` also updates the same mapping. This creates functional redundancy where both methods can modify the allowance value, despite having different primary purposes (role granting vs allowance adjustment).

https://github.com/code-423n4/2025-01-next-generation/blob/main/contracts/ERC20ControlerMinterUpgradeable.sol#L106-L110

https://github.com/code-423n4/2025-01-next-generation/blob/main/contracts/ERC20ControlerMinterUpgradeable.sol#L127-L134

Impact:

Increases code complexity and wastes gas when both role assignment and allowance setting are needed

Recommended Mitigation:

Separate role management from allowance updates:

```
-   function addMinter(address minter, uint256 minterAllowedAmount) external {
+   function addMinter(address minter) external {
        grantRole(MINTER_ROLE, minter);
```

```
-   minterAllowed[minter] = minterAllowedAmount;
-   emit MinterAllowanceUpdated(minter, minterAllowedAmount);
}

function updateMintingAllowance(...) ... {
    // Existing logic remains (allowance update only)
}
```

[L-3] Use of Deprecated `ABIEncoderV2` Pragma in Contract

Description:

The `Forwarder` contract includes the deprecated `pragma experimental ABIEncoderV2;` statement. While this pragma was necessary in older Solidity versions (<0.8.0) to enable encoding of complex types (e.g., structs, nested arrays), it is **redundant in Solidity 0.8.0+** where the ABI encoder V2 became the default and stable. Using deprecated experimental features in a "new contract" indicates outdated patterns. Read about it in the official solidity documentation here, talking about breaking changes in v8.0+

Impact:

- **Code Quality**: Signals use of obsolete practices, reducing maintainability.
- **Forward Compatibility Risk**: Unclear behavior if compiled with future compiler versions (though unlikely to break).
- **Misleading Context**: Creates confusion about whether the contract targets legacy or modern EVM environments.

Proof of Concept:

The pragma is declared at the top of the `Forwarder` contract:

https://github.com/code-423n4/2025-01-next-generation/blob/main/contracts/Forwarder.sol#L25

```
// Deprecated in newer versions of Solidity
pragma experimental ABIEncoderV2;
```

Recommended Mitigation:

Remove the pragma

```
-   pragma experimental ABIEncoderV2;
    pragma solidity ^0.8.0;
```

Tools Used:

Manual Review