Solady

Security Review

Cantina Public Goods review by:

Saw-Mon and Natalie, Lead Security Researcher

Philogy, Security Researcher

Plotchy, Security Researcher

September 14, 2023

1 Acknowledgements

This public goods security review was made possible by the following:

1.0.1 Key contributors

- Nascent
- OpenSea led by z0age
- Sound.xyz
- 0xSplits
- Vectorized
- Spearbit

1.0.2 Individual Supporters

- noah.eth
- hickuphh3.eth
- gogotheauditor.eth
- bytes032.eth
- clabby.eth
- indreams.eth
- jimbobbins.eth
- aviggiano.eth
- 0xA9D1e08C7793af67e9d92fe308d5697FB81d3E43
- 0x75319dCF2F347b4F7C6F9040a50b4253F7E46681
- 0x3B36Cb2c6826349eEC1F717417f47C06cB70b7Ea
- 0x6A897b32996D342535791a74B8a40E75DDf2486e

For more information regarding the raise please refer to the official blog post announcement. Thank you all for making this space a safer place for all.

- The Cantina.xyz team.

Contents

2 Introduction

2.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

2.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

2.3 Risk assessment

Severity	Description
Critical	*Directly* exploitable security vulnerabilities that need to be fixed.
High	Security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All high issues should be addressed.
Medium	Objective in nature but are not security vulnerabilities. Should be addressed unless there is a clear reason not to.
Low	Subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

2.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. When determining the severity one first needs to determine whether the finding is subjective or objective. All subjective findings are considered of Low severity.

Next it is determined whether the finding can be regarded as a security vulnerability. Some findings might be objective improvements that need to be fixed, but do not impact the project's security overall (Medium).

Finally, objective findings of security vulnerabilities are classified as either critical or major. Critical findings should be directly vulnerable and have a high likelihood of being exploited. High findings on the other hand may require specific conditions that need to be met before the vulnerability becomes exploitable.

3 Security Review Summary

Solady is a project for gas optimized Solidity snippets.

From August 7th to August 25th the Cantina team conducted a review of solady on commit hash 89101d53 with the main scope targeting the following contracts: ERC1967Factory.sol, ERC20.sol, ERC721.sol, ERC1155.sol, LibClone.sol, MerkleProofLib.sol, SignatureCheckerLib.sol and ECDSA.sol

During the abovementioned period of time the team identified a total of **40** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 1
- Medium Risk: 3
- Low Risk: 4
- Gas Optimizations: 16
- Informational: 16

4 Findings

4.1 High Risk

4.1.1 LibClone: Length overflow allows corruption of created proxy

Severity: High Risk

Context: LibClone.sol#L409-L4012,L466-L469,

Description: When creating an immutable proxy with args with either the `clone(address, bytes memory)` or `cloneDeterministic(address, bytes memory, bytes32)` functions it's assumed that the length of the bytes input +100 (`0x62 + 2`) will fit in 2-bytes.

Excerpt from `clone(address, bytes)` (both the non-create2 & create2 variants of this clone function have the same logic & flaw):

```
let extraLength := add(dataLength, 2)

mstore(
    sub(data, 0x5a),
    or(shl(0x78, add(extraLength, 0x62)), 0x6100003d81600a3d39f336602c57343d527f)
)
```

The assumption that "The inserted data length will fit in 2-bytes" is not explicitly checked anywhere. A final length that is larger than 2-bytes is not truncated either, being first bit shifted to the left by 15-bytes (`0x78` bits) and then bitwise OR-ed with the data for insertion. This means that a final length requiring 3 or more non-zero bytes ($\geq$ 65536) to be represented would actually change the meaning of the final bytecode:

```
- OR(0x6100003d81, 0x0001230000) = 0x6101233d81 => `PUSH2 0x0123 RETURNDATASIZE DUP2 ...`
+ OR(0x6100003d81, 0x0200230000) = 0x6300233d81 => `PUSH4 0x00233d81 ...`
```

This allows you to modify the `PUSH2` byte (`0x61`) into a different opcode while still resulting in runnable code. Specifically the first `PUSH2` can be changed into any even `PUSH<n>` opcode (`PUSH4`, `PUSH6`, `PUSH8`, `PUSH10`, ...) depending on the length in the resulting inserted length. Extending the length of the initial push opcode means that subsequent opcodes will be turned from logical operations into part of the word that'll get pushed onto the stack as part of execution. If sufficiently extended it'll consume the op-byte of other subsequent push opcodes, turning their "value bytes" into logical opcodes, e.g.:

```
- 6101023d8160ff => PUSH2 0x01 0x02 RETURNDATASIZE DUP2 PUSH1 0xff
+ 6401023d8160ff => PUSH5 0x01 0x02 0x3d           0x81 0x60  SELFDESTRUCT
```

In the `clone{Deterministic}` functions this can be used to achieve 1 of 2 things:

1. Cause a proxy deployment to fail that would otherwise be considered valid (`predictDeterministicAddress` would still compute an address for the undeployable contract).
2. Cause a valid, empty contract (no bytecode) to deployed

Action 2. might be especially harmful as a library consumer may assume that if the cloning does not revert it must've been successful. Any funds sent to such a contract would be permanently forever lost & inaccessible.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.21;

import {Test} from "forge-std/Test.sol";
import {LibClone} from "solady/utils/LibClone.sol";

contract LibCloneLengthOverflowPoC is Test {
    function test_length_overflow_PoC() public {
        bytes memory d = new bytes(0xe0000);
        address proxy = LibClone.clone(makeAddr("implementation"), d);
        // Successfully created contract
        assertTrue(proxy != address(0));
        // Proxy is however corrupted, having no runtime code.
        assertEq(proxy.code, new bytes(0));

        // Corrupted proxy also silently accepts ETH which is then permanently stuck.
        hoax(makeAddr("sender"), 1 ether);
        (bool success,) = proxy.call{value: 1 ether, gas: 2100}("");
        assertTrue(success);
        assertEq(proxy.balance, 1 ether);
    }
}
```

While the length of the data (`0xe0000` bytes = 0.91 Mb) required to deploy a corrupted proxy may seem atypical it is feasible to create a `bytes memory` value of that size in a contract under mainnet gas constraints costing min. ~1.7M gas in memory expansion and EIP-3860 initcode costs.

Recommendation: It is recommended the length of `data` is checked in both `clone(address implementation, bytes memory data)` and `cloneDeterministic(address implementation, bytes memory data, bytes32 salt)` not to exceed 65,435 such that the final max runtime size (65,435 + 2 + 0x62 = 65535) never exceeds 2 bytes.

Cantina: Fixed in PR 548 using the recommendation.

4.2 Medium Risk

4.2.1 ERC1967Factory: Unsafe memory pointer allocation

Severity: Medium Risk

Context: ERC1967Factory#L287-425

Description: In the `ERC1967Factory` contract the `_initCode` function is responsible for allocating memory and for writing the deployment bytecode for proxies to memory. In an effort to save gas the function *does not* adjust the memory pointer to indicate that the section of memory storing the bytecode is now in use. Not only is this not memory safe, creating a pointer to memory that may be reallocated/used by other parts of the code can lead to bugs if a developer attempts to inherit from and use the `ERC1967Factory` contract and subsequently implicitly allocates some memory.

Standalone the `ERC1967Factory` contract seems to be correct, however, this issue is still relevant because there's no indication to potential users of the Solady library that the `ERC1967Factory.sol:ERC1967Factory` contract is not intended for 3rd party use. The file sits under the library's `src/utils/` folder along with other libraries intended for use by library consumers & does not have any comments directly indicating that it's not for consumption by library users.

Beyond being unsafe the `_initCode` function also violates Solidity's conventions. Typically `bytes memory` pointers in Solidity are expected to point to a word of memory storing the length of the data directly followed by the data itself. The memory pointer created by the `_initCode` directly points to a static piece of data, offset by 19 bytes and not having any length.

Recommendation: If the `ERC1967Factory` contract is intendend **solely** for standalone use & deployment as an on-chain component. Add comments reflecting this fact and move the file to its own folder with a name such as `standalone` or `component` denoting that it's different from the libraries intended for direct use and integration with the contracts of 3rd parties.

Furthermore, use a more "neutral" datatype such as `uint256`/`bytes32` or even a custom type for the return value from the `_initCode` function to indicate that it's *not* a normal memory pointer.

If intended for 3rd party use ensure the `_initCode` function updates the free memory pointer to reflect its reserved memory area. Furthermore ensure that the return `bytes memory` value follows Solidity's conventions, pointing to a word with the data's length followed by the data itself.

Cantina: Fixed by changing the return type to `bytes32` and also cautionary comments have been added in PR 547.

4.2.2 ERC20: Memory unsafe assembly is not future proof

Severity: Medium Risk

Context: ERC20.sol#L397-L417

Description: The `DOMAIN_SEPARATOR()` method in the `ERC20` mixin has 2 direct inline-assembly blocks both of which are marked "memory safe" despite violating memory safety guarantees. Specifically, it stores a reference to and writes to memory which may not be free.

1. The method locally caches the free memory pointer in `result`:

```
/// @solidity memory-safe-assembly
assembly {
    result := mload(0x40) // Grab the free memory pointer.
}
```

2. Calls `name()`, a memory-using, view method

```
403:         bytes32 nameHash = keccak256(bytes(name()));
```

3. Uses the previously cached free memory pointer to directly write to memory:

```
/// @solidity memory-safe-assembly
assembly {
    let m := result
    // `keccak256("EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)")`.
    // forgefmt: disable-next-item
    mstore(m, 0x8b73c3c69bb8fe3d512ecc4cf759cc79239f7b179b0ffacaa9a75d522b39400f)
    mstore(add(m, 0x20), nameHash)
    // `keccak256("1")`.
    // forgefmt: disable-next-item
    mstore(add(m, 0x40), 0xc89efdaa54c0f20c7adf612882df0950f5a951637e0307cdcb4c672f298b8bc6)
    mstore(add(m, 0x60), chainid())
    mstore(add(m, 0x80), address())
    result := keccak256(m, 0xa0)
}
```

This may not be memory-safe because `name()` may also read and use the free memory pointer which is then overwritten. Standalone this seems to work as the result string of `name()` does not need to persist, being immediately consumed by the `keccak256` function.

Nevertheless, future improvements to Solidity's optimizer may allow for reasonable uses of this library to result in incorrect code.

Potential Scenario:

A developer uses the Solady library, writing a function where `name()` (overridden as a pure function) and `DOMAIN_SEPARATOR()` are both used within a new method:

```
function getMetadata() public view returns (bytes32, string memory) {
    return (DOMAIN_SEPARATOR(), name());
}
```

A more sophisticated Solidity compiler + optimizer compiles the code, considering its structure:

- The `name()` method is pure, meaning it has no side effects or mutable dependencies outside of its (0) paramters
- The `DOMAIN_SEPARATOR()` method relies on the value of `name()`
- All assembly blocks within `DOMAIN_SEPARATOR()` are marked "memory safe"

based on the facts above the optimizer decides to inline `DOMAIN_SEPARATOR()` into the `getMetadata()` function (hypothetical inlined version):

```
function inlined__getMetadata() public view returns (bytes32, string memory) {
    bytes32 result__DOMAIN_SEPARATOR;
    /// @solidity memory-safe-assembly
    assembly {
        result__DOMAIN_SEPARATOR := mload(0x40) // Grab the free memory pointer.
    }
    //  We simply calculate it on-the-fly to allow for cases where the `name` may change.
    string memory name__inlined = name();
    bytes32 nameHash = keccak256(bytes(name__inlined));
    /// @solidity memory-safe-assembly
    assembly {
        let m := result__DOMAIN_SEPARATOR
        // `keccak256("EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)")`.
        // forgefmt: disable-next-item
        mstore(m, 0x8b73c3c69bb8fe3d512ecc4cf759cc79239f7b179b0ffacaa9a75d522b39400f)
        mstore(add(m, 0x20), nameHash)
        // `keccak256("1")`.
        // forgefmt: disable-next-item
        mstore(add(m, 0x40), 0xc89efdaa54c0f20c7adf612882df0950f5a951637e0307cdcb4c672f298b8bc6)
        mstore(add(m, 0x60), chainid())
        mstore(add(m, 0x80), address())
        result__DOMAIN_SEPARATOR := keccak256(m, 0xa0)
    }
    return (result__DOMAIN_SEPARATOR, name__inlined);
}
```

This would then produce incorrect results because the memory unsafe code from `DOMAIN_SEPARATOR()` would cause the data within `name()` to be overwritten.

Recommendation: In `DOMAIN_SEPARATOR()` retrieve the free memory pointer *after* calling `name()`.

Cantina: Recommendation applied in PR 538.

4.2.3 ECDSA: Empty signature can result in valid recovered address

Severity: Medium Risk

Context: ECDSA.sol#L37-58, ECDSA.sol#L192-210

Description: The `ECDSA.recover(bytes32, bytes memory)` and `ECDSA.tryRecover(bytes32, bytes memory)` functions each take a `hash` and `bytes signature` argument carrying the components of an ECDSA signature `r || s || v` and attempt to recover the signer's Ethereum address via the EVM's `ecrecover` precompile. The only difference between the two is that `recover` will revert if no valid signer could be recovered and `tryRecover` fails silently, returning the zero-address.

The `ecrecover` precompile takes 4 words (128 bytes) as input: `hash || bytes31(0) || v || r || s`. To save gas on memory expansion costs, Solady's ECDSA library temporarily overwrites the memory region `0x00-0x80` (including the free memory pointer at `[0x40:0x60)` and the default null pointer at `[0x60:0x80)`) for the precompile's input. To save gas the functions first optimistically copies the 65 bytes of the signature from memory and then validate its length. This means that if the signature is <65 bytes long it may copy out-of-bound data, the length check is meant to still invalidate such signatures.

There is however an edge case that is unaccounted for: uninitialized `bytes memory` objects. These point to `0x60` giving them a length of zero thanks to the default null pointer. If such a signature is validated the recover functions will copy the data at `[0x80:0xc1]` which may have other variables allocated. The optimistic copy will overwrite the null pointer at `0x60` temporarily changing the implicit length to the word at `0xa0`. This can allow a non-zero address to be recovered from an empty signature.

Proof of Concept:

```solidity
// SPDX-License-Identifier: MIT
pragma solidity 0.8.21;

import {Test} from "forge-std/Test.sol";
import {ECDSA} from "solady/utils/ECDSA.sol";

contract ECDSAPoC is Test {
    struct NotASig {
        bytes32 r;
        bytes32 s;
        bytes1 v;
    }

    function testRecoverInvalid() public {
        bytes32 hash = 0x73139abbd176cf7f94893453632c00afefb2776f9805a21d03d176c885cd0d63;
        // Create an unrelated struct that sits in memory and happens to contain the components of a valid
↪   signature.
        NotASig memory noSig = NotASig({
            r: 0x88385877e6c712ef33fbd9c82ed5f3a348bed805e701d1d9b9ff479ff65b7020,
            s: bytes32(uint256(65)),
            v: bytes1(uint8(28))
        });
        // Uninitialized bytes objects (pointer to 0x60).
        bytes memory emptySig;

        // Try to recover from empty signature
        address recovered = ECDSA.recover(hash, emptySig);

        // No revert, non-zero signer recovered.
        assertEq(recovered, 0x5776b19B163da161b8f6fD304dc1a09FA96b16A7);
    }
}
```

Note that both `tryRecover` and `recover` have the same issue, the PoC demonstrates the issue for the `recover` function.

For this edge-case to be exploited in practice certain conditions must be met:

- The signature parameter but be an empty, uninitialized `bytes memory` variable. If created with the syntax `bytes memory signature = new bytes(0);` Solidity may actually allocate memory, if the pointer is not `0x60` the functions will correctly recognize the signatures as invalid.
- The data already in memory at `[0x80:0xc1]` must represent an ECDSA signature that leads to a recovered address.
- The word at `0xa0` must be `65` so that when `0x60` is overwritten the "length" of the signature will be that of a valid signature, this also means that the `s` component of the signature must be `65`.

The recovered address may be a controllable EOA account (`ecrecover` can "recover" addresses for which there is no valid private key for a subset of random inputs `(hash, r, s, v)`), if the `hash` can be a specifically chosen value. This is due to how the s value is computed when generating a signature (k = secure random integer, z = message hash, d_A = private key, n = order of the point G on the curve):

$$s = k^{-1}(z + r \cdot d_A) \mod n$$

Assuming we know the private key d_A We can rearrange the above formula to find a z such that s is `65`:

$$65 = k^{-1}(z + r \cdot d_A) \mod n$$

$$65 \cdot k = z + r \cdot d_A \mod n$$

$$65 \cdot k - r \cdot d_A = z \mod n$$

We now have a valid signature pair (r, s) for our chosen message hash z and the private key d_A (v is an added component not part of the base ECDSA algorithm chosen based on the original point coordinate x_1 and r).

Signature for real private key with `s = 65` proof of concept:

"Find the full runnable PoC in the Appendix section of this report".

```
# Private key in your control (randomly generated and public, do not use to store funds).
private_key = 0xca6e0c197892239353a097f7db686d4a0313f4316bc726bb7d8fe692f4d827ad
public_key = private_key * G
address = as_address(public_key)
print(f'address: 0x{address.hex()}')

# Random *secure integer
# Note: `randint` is not a secure number generator, only for demonstration purposes
k = randint(0, n-1)
print(f'\nk:    {k}')

# Compute r & v
big_k = k * G
v = 27 + big_k.y % 2
print(f'v:    {v}')
r = big_k.x % n
print(f'r:    0x{r:064x}')
s = 65
print(f's:    0x{s:064x}')

# Compute the hash `z` such that the signature `(v, r, s)` is valid for `z`
z = (s * k - r * private_key) % n
print(f'hash: 0x{z:064x}')
```

Recommendation: Validate or at least cache the validity/length of the signature prior to optimistically copying its contents. This will still allow the use of the memory region `[0x00:0x80)` while correctly handling the empty signature edge case.

Vectorized: Fixed in PR 536.

4.3 Low Risk

4.3.1 ERC20: Underflow check can prevent allowance decrease

Severity: Low Risk

Context: ERC20.sol#L199-L208

Description: The `decreaseAllowance` method is intended to allow accounts to partially reduce an allowance they've already made to other accounts will mitigating frontrunning issues arising from the default behaviour of `approve(address, uint256)` as specified by the ERC20 standard.

The original issue arises from the fact that the `approve(address spender, uint allowance)` function **sets** the allowance between `msg.sender` and `spender` rather than decreasing it e.g.:

1. Alice grants Bob an allowance of 100 (allowance: 100)
2. Bob uses the allowance, transferring 20 tokens (allowance: 80)
3. Alice wants to reduce Bob's allowance by 60 to 20, submitting an `approve` transaction
4. Bob sees Alice's transaction in the mem-pool and quickly frontruns them, transferring 80 tokens (allowance: 0)
5. Alice's transaction gets included, **setting** Bob's allowance to 20 again (allowance: 20)

The effect is that Alice has granted an added 20 tokens worth of allowance when their intention was to actually reduce Bob's allowance. The `decreaseAllowance` method is meant to fix this by giving someone like Alice a way to specify that they want to merely decrease an allowance by a certain number, not set it to a new number.

However, due to the method's underflow check in the method allowance changes can be frontrun to partially block them.

Exploit Scenario:

1. Alice grants Bob an allowance of 100 (allowance: 100)
2. Alice wants to decrease allowance, submitting a `decreaseAllowance` transaction to decrease it by 80

3. Bob sees Alice's transaction in the mem-pool, frontrunning it with a transaction that transfers 20.1 tokens from Alice (new allowance: 79.9)
4. Alice's `decreaseAllowance` transaction gets included, due the current allowance (79.9) being lower than the specified decrease (80) the call reverts with the `AllowanceUnderflow` error

Bob's concrete gain from such an attack is minimal but not zero: Bob retains longer optionality around his allowance from Alice. Bob temporarily delayed the allowance decrease, only having to spend `initial_allowance - decrease + 1 token_wei` to prevent the decrease, depending on Alice's liveness Bob may have extended the time for which he holds the remainder of his allowance for a considerable amount of time.

Recommendation: It is recommended to change the logic such that the new allowance is set to `min(0, previous_allowance - delta)` such that if the decrease is higher than the remaining allowance the allowance can still be decreased. Alternatively, add a separate allowance-decreasing method with the suggested semantics. This would mitigate the above issue while still providing the core functionality of giving users a safe way to atomically reduce allowances to a non-zero amount.

Cantina: The endpoint in this context has been removed in PR 538 and thus the issue is not relevant anymore.

4.3.2 Execution ordering of `and()` may lead to unexpected behavior in future compiler versions.

Severity: Low Risk

Context:

- SignatureCheckerLib.sol#L69-L87
- SignatureCheckerLib.sol#L134-L152
- SignatureCheckerLib.sol#L217-L235
- SignatureCheckerLib.sol#L271-L289
- SignatureCheckerLib.sol#L311-L329
- SignatureCheckerLib.sol#L369-L387

Description: The arguments of `and(arg1, arg2)` are expected to always execute in the order of `arg2` and then `arg1`. Solady uses this behavior to ensure the `staticcall` is executed before the `returndatasize` check in the `and()` operation.

```
isValid := and(
    and(
        // .. snip ..
        eq(returndatasize(), 0x20)
    ),
    // Whether the staticcall does not revert.
    // This must be placed at the end of the `and` clause,
    // as the arguments are evaluated from right to left.
    staticcall(
        gas(), // Remaining gas.
        signer, // The `signer` address.
        m, // Offset of calldata in memory.
        add(signature.length, 0x64), // Length of calldata in memory.
        0x00, // Offset of returndata.
        0x20 // Length of returndata to write.
    )
)
```

If `returndatasize` is executed before the `staticcall`, this leads to unexpected behavior.

A test was conducted for all of the following compiler configuration permutations:

```
compiler_versions = [
    "0.8.0", "0.8.1", "0.8.2", "0.8.3", "0.8.4", "0.8.5", "0.8.6",
    "0.8.7", "0.8.8", "0.8.9", "0.8.10", "0.8.11", "0.8.12", "0.8.13",
    "0.8.14", "0.8.15", "0.8.16", "0.8.17", "0.8.18", "0.8.19", "0.8.20", "0.8.21"
]
optimize_runs_values = [1, 200, 10000, 999999]
via_ir_options = [True, False]
```

No anomalies of execution reordering were found. However, this behavior is not guaranteed to be permanent across future compiler versions.

Full runnable PoC.

Recommendation: Assembly operations are in some form subject to reordering by compiler optimizations. It is not recommended to rely on this behavior to be permanent across future compiler version optimizations. Explicitly performing this `staticcall` before the `and()` may prevent possible execution reordering.

```
@@ -65,6 +65,14 @@ library SignatureCheckerLib {
                        if iszero(lt(i, signatureLength)) { break }
                    }
                }
+               let success := staticcall(
+                       gas(), // Remaining gas.
+                       signer, // The `signer` address.
+                       m, // Offset of calldata in memory.
+                       add(signatureLength, 0x64), // Length of calldata in memory.
+                       0x00, // Offset of returndata.
+                       0x20 // Length of returndata to write.
+                   )
                // forgefmt: disable-next-item
                isValid := and(
                    and(
@@ -76,14 +84,7 @@ library SignatureCheckerLib {
                    // Whether the staticcall does not revert.
                    // This must be placed at the end of the `and` clause,
                    // as the arguments are evaluated from right to left.
-                   staticcall(
-                       gas(), // Remaining gas.
-                       signer, // The `signer` address.
-                       m, // Offset of calldata in memory.
-                       add(signatureLength, 0x64), // Length of calldata in memory.
-                       0x00, // Offset of returndata.
-                       0x20 // Length of returndata to write.
-                   )
+                   success
                )
                break
```

In addition, it would be beneficial to notify Solidity Compiler developers of this issue to bring awareness that this is a breaking change for libraries like Solmate and Solady.

Vectorized: Acknowledged.

4.3.3 `predicted` addresses in `predictDeterministicAddress` are not cleaned up

Severity: Low Risk

Context:

- LibClone.sol#L573
- ERC1967Factory.sol#L270

Description: `predicted` is not truncated/masked in this context. Not that the return type of this variable does not in all cases cleanup this parameter (enforced by compiler) if for example the return value is again used in another assembly block. This issue is also present in OpenZeppelin implementations for the similar endpoints.

Recommendation: It would be best to mask and clean up this returned parameter.

Vectorized: Acknowledged.

4.3.4 MerkleProofLib: Multi-proof does not validate that `boolean` flags are clean

Severity: Low Risk

Context: MerkleProofLib.sol#L79-L172,L176-266

Description: Neither the `verifyMultiProof` or `verifyMultiProofCalldata` functions verify that the boolean values in the `flags` arrays are non-dirty (literal `0x0` or `0x1`). Any non-zero value will get interpreted as `true`.

There are 2-ways Solidity will handle dirty boolean values by default:

- From calldata / external source: revert if value is $\notin \{0, 1\}$
- Value accessed by inline-assembly: clean value, interpreting `0` as false (=> `0`) and any non-zero value as `true` (=> `1`)

Solady chooses the second interpretation which is not necessarily wrong. However, as a library, it should take extra care to account for cases where consumers could expect a reasonable alternative and have assumptions based on that behaviour.

The calldata variant (`verifyMultiProofCalldata`) could be especially precarious as Solidity will not validate elements of calldata arrays by default. If the `flags` array is directly passed from calldata and never used elsewhere the `bool` elements contained within could be dirty as they'd never be validated elsewhere. A reasonable library consumer may assume otherwise due to the lack of documentation.

Recommendation: Document the fact that a multi-proof evaluating to `isValid = true` is not an indication that the `bool` elements contained within the `flags[]` array are within the range `[0, 1]`.

Cantina: Fixed in PR 550 by adding comments.

4.4 Gas Optimization

4.4.1 ERC1967Factory: Can Simplify Less-Than-Or-Equal to expression

Severity: Gas Optimization

Context: ERC1967Factory.sol#L364-368,L410,L418

Description: Upgradeable proxies created via the `ERC1967Factory` have a "sub-routine" that allows the factory address to set the implementation & delegate-call the new implementation via a payload that's encoded as: `(storage_key, word_padded_implementation, calldata)`.

When called by the factory, the proxy checks if the calldata is 64 or fewer bytes long. If so it'll skip the delegate-call to the implementation, only setting the storage slot and ending execution.

To check the length it uses the opcode sequence:

```
| Operation    | Stack                |
|--------------|----------------------|
| PUSH1 0x40   | 0x40 ...             |
| DUP1         | 0x40 0x40 ...        |
| CALLDATASIZE | cdz 0x40 0x40 ...    |
| GT           | cdz > 0x40 0x40 ...  |
| ISZERO       | cdz <= 0x40 0x40 ... |
```

Due to the `0x40` (64) being constant the less-than-or-equal check can be optimized to cost 3 gas less by creating an equivalent expression that omits the `ISZERO` negating operation:

```
| Operation    | Stack                |
|--------------|----------------------|
| PUSH1 0x40   | 0x40 ...             |
| PUSH1 0x41   | 0x41 0x40 ...        |
| CALLDATASIZE | cdz 0x41 0x40 ...    |
| LT           | cdz < 0x41 0x40 ...  |
```

Due to `PUSH1` costing the same as `DUP1` the two expressions can switched at no cost, removing the `ISZERO`. The expressions $x \leq 64$ and $x65$ are equivalent for $x \in \mathbb{N}_0$.

Swapping `DUP1` for `PUSH1 0x41` requires 1 additional byte but omitting the `ISZERO` saves one byte meaning two variations also have the same bytecode size.

Recommendation: Optimize the proxy bytecode as suggested, updating the referenced comment and the actual bytecode.

Vectorized: Acknowledged.

4.4.2 Placing clean-up after check can save gas upon failure.

Severity: Gas Optimization

Context: LibClone.sol#L108-L115,L131-L139,L238-L246,L261-L269

Description: To construct the proxy deployment code in memory the `clone` functions temporarily overwrite a part of the free memory pointer slot (`0x40`). The code to restore the free memory pointer slot is placed before the deploy success check (`if iszero(instance) {`). The consequence of this is that failures will consume unnecessary gas by performing clean-up that is not utilized anyway.

Recommendation: Place the free memory pointer clean up *after* the check.

Cantina: Fixed in PR 548.

4.4.3 SignatureCheckerLib: Sub-optimal signer check

Severity: Gas Optimization

Context: SignatureCheckerLib.sol#L49,L121,L204

Description: In the first step of the `isValidSignatureNow` functions `ecrecover` is applied to see if the signature is a simple ECDSA signature *before* checking whether it's a "contract signature" meaning it needs to call the signer to validate the signature.

To save gas Solady checks both if the recovered signer matches together with if the `ecrecover` operation was successful:

```
assembly {
    ...
    if mul(eq(mload(m), signer), returndatasize()) {
    ...
}
```

An expression with the same effect but costing 2 less gas would be:

```
assembly {
    ...
    if iszero(or(sub(mload(m), signer), iszero(returndatasize()))) {
    ...
}
```

We can prove the two expressions are equivalent by transforming them:

```
   mul(eq(mload(m), signer), returndatasize())               [Inline Assembly]
=> (recovered_signer == signer) && (returndatasize > 0)      [Logic]
=> !(!(recovered_signer == signer) || !(returndatasize > 0)) [Logic]
=> !((recovered_signer != signer) || (returndatasize == 0))  [Logic]

Converting back into assembly:
- (returndatasize == 0) => `iszero(returndatasize())` (output: 0 / 1)
- (recovered_signer != signer) => `sub(mload(m), signer)` (output: 0 / n)
- !(x || y) => `iszero(or(x, y))`
```

Note that it's not safe to use the EVM's bitwise-`or` operation as a logical-OR if inputs may be outside of the range $[0; 1]$ **and** the result of the OR is consumed by another computation that expects the values to be in the range $[0; 1]$

The resulting expression is cheaper because:

1. Bitwise-OR (`or`) costs 2 gas less than a multiplication (`mul`) operation
2. The added `iszero` is optimized out of the final bytecode while the original expression would get compiled to bytecode with an additional `ISZERO` operation

Recommendation: Substitute the pattern with the demonstrated improved expression.

Cantina: Recommendation applied in PR 554.

4.4.4 Can save on `MSTORE` by combining some together

Severity: Gas Optimization

Context: ERC20.sol#L367-L369

Description: In order to derive the ERC712 message hash in the `permit` method the code needs to insert the ERC712 message leading bytes `0x1901` into memory. This is done with its own `MSTORE`. This `MSTORE` can be omitted by reusing another `MSTORE` to write both its contents and the leading bytes together.

Current code:

```
owner := shr(96, shl(96, owner))
...
mstore(0x0c, _NONCES_SLOT_SEED)
mstore(0x00, owner)
...
mstore(0, 0x1901)
```

The `_NONCES_SLOT_SEED` writing `MSTORE` can already set the bytes 30 and 31 in memory to `0x19` and `0x01` and memory if the `_NONCE_SLOT_SEED` contains the bytes such that they're put at the right location.

Recommendation: Change the `_NONCES_SLOT_SEED` constant to contain the bytes `0x1901` so that only 1 `MSTORE` is required to write both it (the slot constant) and the ERC712 leading bytes to memory.

Cantina: Recommendation applied in PR 538.

4.4.5 `mstore(m, 0)` can be removed

Severity: Gas Optimization

Context:

- ERC721.sol#L880
- ERC1155.sol#L264
- ERC1155.sol#L409
- ERC1155.sol#L1053
- ERC1155.sol#L1104

Description/Recommendation: Resetting the memory slot at `m` to `0` is not necessary in this context. Since in case of a failed call to `to` with no return data, we would have at `m`, `on...ReceivedSelector` and this value would not be equal to `shl(224, on...ReceivedSelector)`.

```
-   mstore(m, 0)
```

This can probably be applied to other places where this pattern has been used.

Cantina: Fixed in PR 549 and PR 551.

4.4.6 `ERC721.getApproved(...)` can be optimised

Severity: Gas Optimization

Context:

- ERC721.sol#L183

Description/Recommendation: `ERC721.getApproved(...)` can be optimised by removing the right shift when checking the existence of a token owner:

```
-   if iszero(shr(96, shl(96, sload(ownershipSlot)))) {
+   if iszero(shl(96, sload(ownershipSlot))) {
```

Cantina: Fixed PR 549.

4.4.7 `mstore(0x00, id)` can be removed

Severity: Gas Optimization

Context:

- ERC1155.sol#L633

Description/Recommendation: `id` is already stored in the memory slot at `0x00` when one reaches the line in this context. Gas diff needs to be analysed:

```
-   mstore(0x00, id)
```

Cantina: Applied in PR 551.

4.4.8 `n := sub(add(o, n), m)` could be optimised

Severity: Gas Optimization

Context:

- ERC1155.sol#L366

Description/Recommendation: `n := sub(add(o, n), m)` should be 2n + 64, so there is a potential to calculate it in a different fashion to save gas. It is possible that one of the following forms could require less gas:

```
n := add(shl(1, n), 0x40) // or its permutations or
n := shl(1, add(n, 0x20)) // or its permutations or
n := add(add(n, n), 0x40) // or its permutations
```

One would need to try all the different combinations and run a gas diff.

Cantina: Applied in PR 551.

4.4.9 Second assignment of `n` based on `amounts.length` can be removed.

Severity: Gas Optimization

Context:

- ERC1155.sol#L364
- ERC1155.sol#L394

Description/Recommendation: `ids` and `amounts` have the same length. This lines in this context can be removed.

```
-    n := add(0x20, shl(5, amounts.length))
```

```
testMintToRevertingERC155RecipientReverts(uint256) (gas: 1 (0.000%))
testSafeBatchTransfer() (gas: -135 (-0.002%))
testSafeBatchTransferFromToERC1155Recipient(uint256) (gas: -27 (-0.003%))
testSafeBatchTransferFromToNonERC1155RecipientReverts(uint256) (gas: -27 (-0.010%))
testSafeBatchTransferInsufficientBalanceReverts(uint256) (gas: -27 (-0.016%))
testBatchMintToWrongReturnDataERC1155RecipientReverts(uint256) (gas: -54 (-0.017%))
testMintToNonERC155RecipientReverts(uint256) (gas: -22 (-0.021%))
test__codesize() (gas: -10 (-0.023%))
testMintToZeroReverts(uint256) (gas: -8 (-0.024%))
testMintToERC1155Recipient(uint256) (gas: -183 (-0.028%))
testSafeBatchTransferFromToEOA(uint256) (gas: 57 (0.030%))
testSafeTransferFromSelf(uint256) (gas: 44 (0.042%))
testSafeTransferFromToRevertingERC1155RecipientReverts(uint256) (gas: 161 (0.051%))
testBatchMintToRevertingERC1155RecipientReverts(uint256) (gas: 175 (0.055%))
testERC1155Hooks() (gas: -2864 (-0.065%))
testSafeTransferFromToWrongReturnDataERC1155RecipientReverts(uint256) (gas: -180 (-0.066%))
testBatchBurnWithArrayLengthMismatchReverts(uint256) (gas: -29 (-0.068%))
testBurn(uint256) (gas: 57 (0.069%))
testSafeTransferFromInsufficientBalanceReverts(uint256) (gas: -79 (-0.079%))
test__codesize() (gas: -10 (-0.082%))
testMintToEOA(uint256) (gas: 60 (0.084%))
testSafeBatchTransferFromWithArrayLengthMismatchReverts(uint256) (gas: 66 (0.099%))
testBalanceOfBatchWithArrayMismatchReverts(uint256) (gas: -36 (-0.108%))
testSafeTransferFromToERC1155Recipient(uint256) (gas: 977 (0.125%))
testBatchMintToNonERC1155RecipientReverts(uint256) (gas: 236 (0.127%))
testBatchMintToERC1155Recipient(uint256) (gas: -1090 (-0.139%))
testBatchMintToZeroReverts(uint256) (gas: -96 (-0.147%))
testBatchMintToEOA(uint256) (gas: -158 (-0.158%))
testSafeBatchTransferFromToRevertingERC1155RecipientReverts(uint256) (gas: -722 (-0.162%))
testBurnInsufficientBalanceReverts(uint256) (gas: -241 (-0.247%))
testSafeTransferFromSelfInsufficientBalanceReverts(uint256) (gas: 196 (0.272%))
testBatchBalanceOf(uint256) (gas: -268 (-0.286%))
testSafeTransferFromToEOA(uint256) (gas: -735 (-0.654%))
testBatchBurn(uint256) (gas: 2816 (1.766%))
Overall gas change: -2155 (-0.000%)
```

Cantina: Applied in PR 551.

4.4.10 Iterate loops backwards to save gas

Severity: Gas Optimization

Context:

- ERC1155.sol#L324-L325
- ERC1155.sol#L439-L442
- ERC1155.sol#L545-L547
- ERC1155.sol#L686-L688
- ERC1155.sol#L901-L903
- MerkleProofLib.sol#L122-L123

Description: In the context above we have the following loop pattern used which is a forward iteration:

```
let end := shl(5, ids.length)
for { let i := 0 } iszero(eq(i, end)) { i := add(i, 0x20) } {
    ...
}
```

There is a potential to save gas if we iterate backwards:

```
let i := shl(5, ids.length)
for {} i {} {
    i := sub(i, 0x20)
    ...
}
```

can be applied to other similar loops.

```
testBatchMintToWrongReturnDataERC1155RecipientReverts(uint256) (gas: 2 (0.001%))
testSafeBatchTransfer() (gas: 56 (0.001%))
testMintToRevertingERC155RecipientReverts(uint256) (gas: -3 (-0.001%))
testMintToWrongReturnDataERC155RecipientReverts(uint256) (gas: 3 (0.001%))
testSafeBatchTransferFromToNonERC1155RecipientReverts(uint256) (gas: 8 (0.003%))
testBatchMintToRevertingERC1155RecipientReverts(uint256) (gas: -13 (-0.004%))
testMintToERC1155Recipient(uint256) (gas: -30 (-0.005%))
testSafeTransferFromToRevertingERC1155RecipientReverts(uint256) (gas: -18 (-0.006%))
testSafeBatchTransferFromWithArrayLengthMismatchReverts(uint256) (gas: -5 (-0.008%))
testSafeBatchTransferInsufficientBalanceReverts(uint256) (gas: -19 (-0.011%))
testSafeTransferFromToWrongReturnDataERC1155RecipientReverts(uint256) (gas: 30 (0.011%))
testBurn(uint256) (gas: 15 (0.018%))
testBatchBurnWithArrayLengthMismatchReverts(uint256) (gas: -9 (-0.021%))
testMintToZeroReverts(uint256) (gas: -8 (-0.024%))
testMintToNonERC155RecipientReverts(uint256) (gas: -26 (-0.025%))
testSafeBatchTransferFromToRevertingERC1155RecipientReverts(uint256) (gas: -267 (-0.060%))
testBatchBalanceOf(uint256) (gas: -56 (-0.060%))
testBatchMintToERC1155Recipient(uint256) (gas: 583 (0.075%))
testMintToEOA(uint256) (gas: 60 (0.084%))
testBalanceOfBatchWithArrayMismatchReverts(uint256) (gas: -36 (-0.108%))
testSafeTransferFromInsufficientBalanceReverts(uint256) (gas: 109 (0.109%))
testBatchMintToNonERC1155RecipientReverts(uint256) (gas: 214 (0.115%))
testSafeBatchTransferFromToERC1155Recipient(uint256) (gas: -1158 (-0.130%))
testSafeBatchTransferFromToZeroReverts(uint256) (gas: 171 (0.137%))
testSafeBatchTransferFromToEOA(uint256) (gas: 287 (0.152%))
testSafeTransferFromToERC1155Recipient(uint256) (gas: 1199 (0.153%))
testBurnInsufficientBalanceReverts(uint256) (gas: -160 (-0.164%))
testSafeTransferFromSelfInsufficientBalanceReverts(uint256) (gas: 124 (0.172%))
testBatchMintToEOA(uint256) (gas: -235 (-0.234%))
testSafeTransferFromToZeroReverts(uint256) (gas: 182 (0.259%))
testSafeTransferFromToEOA(uint256) (gas: -809 (-0.720%))
testBatchBurn(uint256) (gas: 2565 (1.609%))
Overall gas change: 2756 (0.000%)
```

Recommendation: Apply the above recommendation to this context and compare gas diff to potentially reduce gas cost across different loop patterns.

Cantina: Recommendations applied in PR 551.

4.4.11 ERC1967Factory deploys contracts with extra STOP opcodes

Severity: Gas Optimization

Context:

- ERC1967Factory.sol#L282
- ERC1967Factory.sol#L226

Description: When ERC1967Factory produces proxy contracts the length of initcode is incorrect. The length value is hardcoded as the 0x89 literal, but the actual length of the initcode is dependent on the address of the ERC1967Factory deployment and whether or not it begins with six leading zero bytes.

Factory Address	Proxy Initcode Length
6 or more leading zero bytes	0x82
5 or less leading zero bytes	0x88

In both cases, 0x89 is larger than necessary.

```
function _deploy(
    address implementation,
    address admin,
    bytes32 salt,
    bool useSalt,
    bytes calldata data
) internal returns (address proxy) {
    bytes memory m = _initCode();
    /// @solidity memory-safe-assembly
    assembly {
        // Create the proxy.
        switch useSalt
        case 0 { proxy := create(0, add(m, 0x13), 0x89) }
        default { proxy := create2(0, add(m, 0x13), 0x89, salt) }
        // .. snip ..
```

This inaccuracy is also reflected in the `initCodeHash` function, which is used to calculate the address of the to-be-deployed proxy contract.

```
function initCodeHash() public view returns (bytes32 result) {
    bytes memory m = _initCode();
    /// @solidity memory-safe-assembly
    assembly {
        result := keccak256(add(m, 0x13), 0x89)
    }
}
```

The extra opcodes deployed are not a security issue as the extra memory read will be zero bytes, which are STOP opcodes. However, the extra opcodes will increase the gas cost of deployment by 200gas for each extra STOP.

Recommendation: The length of the proxy initcode in both the `_deploy` and `initCodeHash` functions should be corrected to the exact length of the initcode by checking the address of the Factory.

Alternatively, since the proxy initcode used is dependent on the Factory address, this check can be done once at Factory construction time. By checking the deployment address in the Factory constructor, the unused proxy initcode can be pruned from the Factory runtime code. This saves both Factory deployment costs and proxy deployment costs. Note: Apparently the construction-time trick comes at the cost of Etherscan Verification on the Factory.

Cantina: No changes. Marking as acknowledged.

4.4.12 Storage layout for `ERC1967Factory` can be optimised

Severity: Gas Optimization

Context:

- ERC1967Factory.sol

Description: Currently the only storage parameter for `ERC1967Factory` is the admins of the deployed proxies. The storage slot for this parameter is calculated using:

```
mstore(0x0c, address())
mstore(0x00, proxy)

let adminSlot := keccak256(0x0c, 0x20) // the storage slot for the `admin` of `proxy`
```

We can avoid storing values in memory and hashing that portion by redesign the storage slot such that:

```
let adminSlot := proxy // collision-resistant
```

```
diff --git a/src/utils/ERC1967Factory.sol b/src/utils/ERC1967Factory.sol
index bbe8754..f88a09c 100644
--- a/src/utils/ERC1967Factory.sol
+++ b/src/utils/ERC1967Factory.sol
@@ -82,9 +82,7 @@ contract ERC1967Factory {
     function adminOf(address proxy) public view returns (address admin) {
         /// @solidity memory-safe-assembly
         assembly {
-            mstore(0x0c, address())
```

```
-            mstore(0x00, proxy)
-            admin := sload(keccak256(0x0c, 0x20))
+            admin := sload(proxy)
         }
     }

@@ -94,15 +92,12 @@ contract ERC1967Factory {
         /// @solidity memory-safe-assembly
         assembly {
             // Check if the caller is the admin of the proxy.
-            mstore(0x0c, address())
-            mstore(0x00, proxy)
-            let adminSlot := keccak256(0x0c, 0x20)
-            if iszero(eq(sload(adminSlot), caller())) {
+            if iszero(eq(sload(proxy), caller())) {
                 mstore(0x00, _UNAUTHORIZED_ERROR_SELECTOR)
                 revert(0x1c, 0x04)
             }
             // Store the admin for the proxy.
-            sstore(adminSlot, admin)
+            sstore(proxy, admin)
             // Emit the {AdminChanged} event.
             log3(0, 0, _ADMIN_CHANGED_EVENT_SIGNATURE, proxy, admin)
         }
@@ -128,9 +123,7 @@ contract ERC1967Factory {
         /// @solidity memory-safe-assembly
         assembly {
             // Check if the caller is the admin of the proxy.
-            mstore(0x0c, address())
-            mstore(0x00, proxy)
-            if iszero(eq(sload(keccak256(0x0c, 0x20)), caller())) {
+            if iszero(eq(sload(proxy), caller())) {
                 mstore(0x00, _UNAUTHORIZED_ERROR_SELECTOR)
                 revert(0x1c, 0x04)
             }
@@ -248,9 +241,7 @@ contract ERC1967Factory {
             }

             // Store the admin for the proxy.
-            mstore(0x0c, address())
-            mstore(0x00, proxy)
-            sstore(keccak256(0x0c, 0x20), admin)
+            sstore(proxy, admin)

             // Emit the {Deployed} event.
             log4(0, 0, _DEPLOYED_EVENT_SIGNATURE, proxy, implementation, admin)
```

```
testCdCompressDecompress(bytes) (gas: -151 (-0.024%))
testCdFallbackDecompressor(bytes) (gas: -49 (-0.042%))
testUpgradeAndCallWithRevert() (gas: -139 (-0.052%))
testDeployDeterministicAndCall(uint256) (gas: -185 (-0.053%))
testProxySucceeds() (gas: 165 (0.064%))
testUpgradeAndCall() (gas: -325 (-0.092%))
testChangeAdmin() (gas: -270 (-0.101%))
testUpgrade() (gas: -304 (-0.114%))
testUpgradeWithCorruptedProxy() (gas: -304 (-0.115%))
testFlzCompressDecompress() (gas: -2607 (-0.121%))
testDeployAndCallWithRevert() (gas: -272 (-0.129%))
testDecompressWontRevert(bytes) (gas: -796 (-0.129%))
testDeploy() (gas: -343 (-0.133%))
testDeployAndCall(uint256) (gas: -457 (-0.134%))
testProxyFails() (gas: -357 (-0.138%))
test__codesize() (gas: -53 (-0.156%))
testChangeAdminUnauthorized() (gas: -476 (-0.185%))
testCdFallback() (gas: -10666 (-0.187%))
testDeployBrutalized(uint256) (gas: -84 (-0.190%))
test__codesize() (gas: -53 (-0.247%))
testFlzCompressDecompress2() (gas: -2743 (-0.274%))
testUpgradeUnauthorized() (gas: -894 (-0.330%))
testCdCompressDecompress(uint256) (gas: -2745 (-0.394%))
testFlzCompressDecompress(bytes) (gas: -2673 (-0.398%))
testCdFallback(bytes,uint256) (gas: -6089 (-0.589%))
Overall gas change: -32870 (-0.004%)
```

Recommendation: Note that if the above recommendation is taken into account, we should live a warn-

ing/note for the devs that if this contract gets inherited there are potentials of storage collisions due to the use of customised storage layout for this contract.

Cantina: Fixed in PR 547 which uses the `shl(96, proxy)` as the `adminSlot`.

4.4.13 `calldataload(offset)` can be cached in `verifyCalldata`

Severity: Gas Optimization

Context:

- MerkleProofLib.sol#L62-L66

Description: The effect of caching `calldataload(offset)` is that the other endpoints get optimised according to the current test cases and `forge s --diff .gas-audit --asc`:

```
testVerifyMultiProof(bool,bool,bool,bool,bytes32) (gas: -20 (-0.003%))
testVerifyMultiProofForSingleLeaf(bytes32[],uint256) (gas: -1844 (-0.232%))
Overall gas change: -1864 (-0.000%)
```

Recommendation: Perhaps this caching can be tested in isolation not using `foundry` and analysed whether it would actually affect the gas cost.

Cantina: Marking as acknowledged.

4.4.14 The requirement that all the proof elements are used in `verifyMultiProof` can be simplified

Severity: Gas Optimization

Context:

- MerkleProofLib.sol#L167
- MerkleProofLib.sol#L261

Description: In the context above`proofEnd` cannot be `0` since due to *"Calculation of `proofEnd` in `verifyMultiProof`... can be simplified"*.

```
let proofEnd := add(proof, shl(5, proofLength))
```

and for `proofEnd` to be `0` we would either need to have:

- both `proof` and `proofLength` to be `0` which is impossible since `proof` at the point `proofEnd` was calculated is at least `0x20` unless an overflow happens
- the `add` or `shl` in `proofEnd` expression overflows

Warning : overflow instances need to be checked

and thus line 167 can be simplified to:

```
eq(proofEnd, proof) // make sure all `proof` elements are consumed
```

`forge s --diff .gas-audit --asc`:

```
test__codesize() (gas: -3 (-0.025%))
testVerifyMultiProofIsInvalid() (gas: -266 (-0.042%))
testVerifyMultiProofIsValid() (gas: -275 (-0.044%))
testVerifyProof(bytes32[],uint256) (gas: 360 (0.047%))
testVerifyMultiProof(bool,bool,bool,bool,bytes32) (gas: -311 (-0.049%))
testVerifyMultiProofMalicious() (gas: -9 (-0.112%))
testVerifyMultiProofForHeightTwoTree(bool,bool,bool,bool,bool,bytes32) (gas: -9 (-0.133%))
testVerifyMultiProofForSingleLeaf(bytes32[],uint256) (gas: -7182 (-0.904%))
Overall gas change: -7695 (-0.001%)
```

The same recommendation can be applied to the line 261:

```
eq(proofEnd, proof.offset)
```

The argument that `proofEnd` cannot be zero is slightly different. For example

- `proof.offset` cannot be zero unless the protocol's exposed function signature is not used and also `proof.length` is `0`. and so the call data would need to start with `bytes32(0)` and the `proof.offset` would need to point to the very beginning of call data.
- `add(proof.offset, shl(5, proof.length))` overflows when adding or shifting.

Recommendation: If the above recommendations are implemented it would be great to add comment to these endpoints so that the devs would be aware of the above edge cases and try to avoid them. The default endpoints and abi-decoding by `solc` should avoid those issues, unless the protocol developer define some custom decoding and exchange of call data.

Cantina: Recommendation applied in PR 550.

4.4.15 Calculation of `proofEnd` in `verifyMultiProof` can be simplified

Severity: Gas Optimization

Context:

- MerkleProofLib.sol#L108-L115
- MerkleProofLib.sol#L195-L209

Description: If we end up on line `115` in `verifyMultiProof(...)` we know that `iszero(flagsLength) == 0` since otherwise we would have breaked out of the loop earlier. And so the expression on this line can be simplified to:

```
let proofEnd := add(proof, shl(5, proofLength))
```

Also if we put all the lines after the initial `if` block into an inner block, for some reason one would have quite a few gas saved (without the scoping the `testVerifyMultiProofForSingleLeaf` would have an increase in gas, but the other tests would consume less gas):

```
// For the case where `proof.length + leaves.length == 1`.
if iszero(flagsLength) {
    // `isValid = (proof.length == 1 ? proof[0] : leaves[0]) == root`.
    isValid := eq(mload(xor(leaves, mul(xor(proof, leaves), proofLength))), root)
    break
}

// new scope
{
    // The required final proof offset if `flagsLength` is not zero, otherwise zero.
    let proofEnd := add(proof, shl(5, proofLength))
    ...
    break
}
```

`forge s --diff .gas-audit --asc`:

```
test__codesize() (gas: -4 (-0.033%))
testVerifyProof(bytes32[],uint256) (gas: -270 (-0.035%))
testVerifyMultiProofIsInvalid() (gas: -333 (-0.053%))
testVerifyMultiProofIsValid() (gas: -413 (-0.066%))
testVerifyMultiProof(bool,bool,bool,bool,bytes32) (gas: -481 (-0.076%))
testVerifyMultiProofMalicious() (gas: -14 (-0.174%))
testVerifyMultiProofForHeightTwoTree(bool,bool,bool,bool,bool,bytes32) (gas: -14 (-0.207%))
testVerifyMultiProofForSingleLeaf(bytes32[],uint256) (gas: -7465 (-0.939%))
Overall gas change: -8994 (-0.001%)
```

The same recommendation can be applied to `verifyMultiProofCalldata(...)`, lines 208-209 can be simplified to:

```
let proofEnd := add(proof.offset, shl(5, proof.length))
```

Recommendation: Apply the above recommendations to `verifyMultiProof...` when calculating `proofEnd`

Cantina: Recommendation applied in PR 550.

4.4.16 Memory copying via loop instead of leveraging the identity precompile

Severity: Gas Optimization

Context:

- SignatureCheckerLib.sol#L59-L66,L261-L268
- MerkleProofLib.sol#L123-L125

Description: Both the `SignatureCheckerLib` and `MerkleProofLib` libraries need to copy memory sections from one location to another.

Currently, they both do this via an iterative loop that copies one word at a time from the source to the destination. A cheaper alternative would be to leverage the identity precompile to effectively copy entire memory segments using a constant amount of gas. Note that for smaller memory segments the iterative approach is cheaper due to the fixed cost of the required `STATICCALL` however most use-cases will require larger signatures which is why this trade-off is likely to be beneficial in practice.

Recommendation: Use a `STATICCALL` to the identity precompile (`0x4`) to copy memory. And make sure to leave a warning/note for the devs to check whether the `address(0x4)` is the identity precompile

Vectorized:`SignatureCheckerLib` fixed PR 537.

4.5 Informational

4.5.1 Soft Memory Safety Violations

Severity: Informational

Context: ECDSA.sol#L40,L78,L112,L147,L196,L230,L264,L295, ERC1967Factory.sol#L150-L151,L246-L247

Description: Outside of the other more severe violations of memory safety or general unsafe memory access already mentioned in other issues, there are other less severe violations of memory safety. These are separately grouped together and enumerated here due to these violations having no known consequences in current Solidity versions and being deemed unlikely to have negative consequences in the near future.

Nevertheless, these have to be mentioned as the Solidity compiler is still being actively developed, and these violations while seeming benign now, are mentioned in the official Solidity documentation meaning that future compiler versions may rely on these properties being upheld.

The first, violated repeatedly in `ECDSA.sol` is the expectation that `0x60` will consistently be left at `0`:

> In particular, a memory-safe assembly block may only access the following memory ranges:
>
> - Memory allocated by yourself using a mechanism like the allocate function described above.
> - Memory allocated by Solidity, e.g. memory within the bounds of a memory array you reference.
> - The scratch space between memory offset 0 and 64 mentioned above.
> - Temporary memory that is located after the value of the free memory pointer at the beginning of the assembly block, i.e. memory that is "allocated" at the free memory pointer without updating the free memory pointer. [1]

Note that the default null pointer location (`0x60`) is **not** mentioned in the list of permitted memory accesses and is explicitly noted to be expected to be zero constantly:

> The 32 bytes after the free memory pointer (i.e., starting at 0x60) are meant to be zero permanently [...] [2]

This invariant is violated when functions in the `ECDSA` temporarily overwrite the `0x60` slot for the sake of storing the `s` component of signatures e.g.:

[1] docs.soliditylang.org/en/v0.8.21/assembly.html
[2] docs.soliditylang.org/en/v0.8.21/assembly.html

```
/// @solidity memory-safe-assembly
assembly {
    ...
    mstore(0x60, mload(add(signature, 0x40)))
}
```

Another violation that is explicitly mentioned by the docs is copying to the `0` offset and onwards for the sake of reverting:

> Since this is mainly about the optimizer, these restrictions still need to be followed, even if the assembly block reverts or terminates. As an example, the following assembly snippet is not memory safe, because the value of returndatasize() may exceed the 64 byte scratch space:[3]

```
assembly {
    returndatacopy(0, 0, returndatasize())
    revert(0, returndatasize())
}
```

Which is also done in `ERC1967Factory` on two separate instances.

Recommendation: Due to their current non-impact and the concrete gas savings arising from these abuses of memory safety it is not recommended to currently fix these instances.

However, it is strongly recommended to keep a close watch to changes made to the Solidity compiler and explicitly cap the compatible solidity version via the pragma version statement in these files to the last tested version. This will ensure that the library is not accidentally used with newer unsafe Solidity versions as they come out before they're explicitly evaluated.

Vectorized: Acknowledged. I think we will just add a Solidity badge that shows what versions has Solady been tested on to the README.

4.5.2 `sub(sload(slot), 1)` could potentially underflow in a child contract

Severity: Informational

Context:

- ERC721.sol#L297
- ERC721.sol#L568
- ERC721.sol#L763

Description: There is an assumption in this context that when a token is owned by `from`, the balance of `from` is at least `1`. If not true, the balance will become `type(uint256).max`.

Recommendation: It would be best to add a comment regarding this in case some of the virtual end-points are overridden which can break the assumed invariant.

Cantina: Comments have been added in PR 549.

4.5.3 `ERC721.getApproved(...)` might return a result with dirty bits

Severity: Informational

Context:

- ERC721.sol#L187

Description: The `result` in this context could potentially include dirty bits or extra data.

It is true that currently when the approved address is written in this slot it gets cleaned, but a potential child can override that method (`_approve(...)`).

Recommendation: It would be best to leave a comment regarding this for the users/devs.

Cantina: Contract level comments have been added in PR 549.

[3] docs.soliditylang.org/en/v0.8.21/assembly.html

4.5.4 Document the derivation of the slot and seed storage constants

Severity: Informational

Context:

- ERC20.sol#L58-L88
- ERC721.sol#L118

Description: In the context above the slot or seed storage constants are calculated as follows:

```
// C = keccak256("NAME") >> (8 * (32 - N))
uint256 private constant NAME = C;
```

It would be best to maybe add a prefix (path-like prefix, we can include for example the project and contract name) to the `NAME` to avoid collision in a child contract that inherits from two different contracts using the same scheme.

Example:

The string to hash:

```
solady.tokens.ERC20._TOTAL_SUPPLY_SLOT
```

Recommendation: Document the derivation of the slot and seed storage constants.

Vectorized: Acknowledged.

4.5.5 Leave a note for users/devs to check the precompile requirements for the chains they are planning to deploy

Severity: Informational

Context: `ecrecover` precompile:

- ERC20.sol#L375
- ECDSA.sol#L43-L50
- ECDSA.sol#L81-L88
- ECDSA.sol#L115-L122
- ECDSA.sol#L150-L157
- ECDSA.sol#L198-L205
- ECDSA.sol#L232-L239
- ECDSA.sol#L266-L273
- ECDSA.sol#L297-L304
- SignatureCheckerLib.sol#L39-L46
- SignatureCheckerLib.sol#L111-L118
- SignatureCheckerLib.sol#L194-L201

`identity` precompile:

- ERC721.sol#L872
- ERC1155.sol#L570
- ERC1155.sol#L575
- ERC1155.sol#L710
- ERC1155.sol#L715
- ERC1155.sol#L938
- ERC1155.sol#L943

- ERC1155.sol#L1045
- ERC1155.sol#L1084
- ERC1155.sol#L1090
- ERC1155.sol#L1095

Description: The codes in this context rely on the fact that the static calling the addresses `1` and `4` should perform `ecrecover` and `identity` precompile operations.

Recommendation: The above fact needs to be commented for the devs so that they would check those requirements on the chain they are planning to deploy their contracts.

Cantina: Fixed in:

- PR 573.
- PR 554.
- PR 551.
- PR 549.
- PR 538.

4.5.6 Typo in `ERC1155` comment

Severity: Informational

Context:

- ERC1155.sol#L690

Description/Recommendation: In the comment in this context the word `to` needs to be changed to `from`:

```
-   // Decrease and store the updated balance of `to`.
+   // Decrease and store the updated balance of `from`.
```

Cantina: Fixed in PR 551.

4.5.7 Potential storage collision for child contracts of `ERC1155`

Severity: Informational

Context:

- ERC1155.sol#L83-L104

Description: Since the `ERC1155` abstract contract can be inherited, if the child contract introduces a high-level mapping storage parameter there is potential that there would be storage collision with the `balanceSlot`. The high-level storage parameter would need to have a big position for this collision to happen due to the usage of `_ERC1155_MASTER_SLOT_SEED`.

The `balanceSlot` for an `owner` and `id` is calculated as

```
keccak256(id . (owner << 12 * 8 || _ERC1155_MASTER_SLOT_SEED))
```

So if there was a storage parameter with position `owner << 12 * 8 || _ERC1155_MASTER_SLOT_SEED` and one would wanted to write to the key `id` it would land on the same storage slot.

The above could be rare to happen as one would need a really high storage parameter position.

Recommendation: It would be best to document above to prevent devs to run into this type of issues.

Vectorized: Acknowledged.

4.5.8 Free memory pointer is partially overwritten and then cleared in `predictDeterministicAddress`

Severity: Informational

Context:

- ERC1967Factory.sol#L272
- LibClone.sol#L575

Description: In this context part of the free memory pointer that was written to is cleared:

```
// Restore the part of the free memory pointer that has been overwritten.
mstore(0x35, 0)
```

For some custom memory managements by devs and on a chain with a super cheap gas costs, this can potentially cause an issue as it is assumed that the free memory pointer is always contained in the first `11` bytes.

Recommendation: It would be great to document this and leave comment in the NatSpec for the developers.

Vectorized: Acknowledged. Currently, geth only supports up to `0x1FFFFFFFE0`, ethereum/go-ethereum.

4.5.9 `ERC1967Factory` comment corrections

Severity: Informational

Context:

- ERC1967Factory.sol#L304
- ERC1967Factory.sol#L315
- ERC1967Factory.sol#L330-L331
- ERC1967Factory.sol#L341)
- ERC1967Factory.sol#L350-L353
- ERC1967Factory.sol#L369
- ERC1967Factory.sol#L394
- ERC1967Factory.sol#L383-L384

Description/Recommendation:

In below the other case refers to factories addresses with at least `6` leading `0`s.

- ERC1967Factory.sol#L304, true for the default case, runtime size for default is `0x7f`. For the other case is `0x79`.
- ERC1967Factory.sol#L315, for the other case is `push14 factory`.
- ERC1967Factory.sol#L330-L331, `0` missing on the stack:

```
- * 54  | SLOAD      | i cds 0 0        | [0..calldatasize): calldata     |
- * 5a  | GAS        | g i cds 0 0      | [0..calldatasize): calldata     |
+ * 54  | SLOAD      | i 0 cds 0 0      | [0..calldatasize): calldata     |
+ * 5a  | GAS        | g i 0 cds 0 0    | [0..calldatasize): calldata     |
```

- ERC1967Factory.sol#L341),ERC1967Factory.sol#L369, ERC1967Factory.sol#L394, `0x52` offset is for the default case.
- ERC1967Factory.sol#L350-L353, there might be other unused stack items present. Perhaps might be good to indicate that with `...`.
- ERC1967Factory.sol#L383-L384, `0` missing on the stack:

```
- * 35  | CALLDATALOAD   | i t 0 0        | [0..t): extra calldata     |
- * 5a  | GAS            | g i t 0 0      | [0..t): extra calldata     |
+ * 35  | CALLDATALOAD   | i 0 t 0 0      | [0..t): extra calldata     |
+ * 5a  | GAS            | g i 0 t 0 0    | [0..t): extra calldata     |
```

Cantina: 2/6 of the recommendations have been applied in PR 547. Marking as acknowledged.

4.5.10 The clone implementations in `LibClone` are slightly different than the original `clones-with-immutable-args`

Severity: Informational

Context:

- LibClone.sol#L396-L413
- LibClone.sol#L453-L470
- wighawag/clones-with-immutable-args

Description: The clone implementations in `LibClone` are slightly different than the original `clones-with-immutable-args`.

If `calldata` is empty, it will not `delegate` the call to the implementation contract and instead accepts native tokens if any and logs it:

```
CALLDATASIZE        ; cds, if no calldata it will not delegate
PUSH1 0x2c          ; 0x2c cds
JUMPI               ;

CALLVALUE           ; v
RETURNDATASIZE      ; 0, v
MSTORE              ; | m[0..32) : v
PUSH32 0x9e4ac34f21c619cefc926c8bd93b54bf5a39c7ab2127a895af1cc0691d7e3dff
MSIZE
RETURNDATASIZE      ; 0, 32, h
LOG1
STOP
```

Recommendation: This is a diverging behaviour compared to original clone with the immutable args implementation and can cause in some cases funds to be locked up. There is already a comment section in the disassembly comments, but it should also be added to the NatSpec comments as a note or warning.

Cantina: Comments were added in PR 548.

4.5.11 Document the invariants and requirements for `verifyMultiProof...`

Severity: Informational

Context:

- MerkleProofLib.sol#L167
- MerkleProofLib.sol#L261

Description: If we end up on the lines in this context for example:

```
or(iszero(proofEnd), eq(proofEnd, proof))
```

and if `or(iszero(proofEnd), eq(proofEnd, proof))` (or its simplified version *"The requirement that all the proof elements are used in `verifyMultiProof...` can be simplified"*) holds, one can deduce that we should have at least one leaf. This is due to the below invariant:

$$F = (L - 1) + P$$

and the fact that we would need to consume all proof elements. If we had L = 0 then F = P − 1 and we would only do P − 1 hash computation and at each around we can consume at most one proof element so at most we would consume P − 1 proof element. Thus at least one proof element would remain not used which would break `eq(proofEnd, proof)`.

paramater	description
F	flag count
L	leaf count
P	proof count

so in general it is important that at least one leaf is provided when $F \neq 0$. And thus writing the above invariant in that form by using $L - 1$ makes sense.

Recommendation: Document the invariants and requirements for `verifyMultiProof...`.

Cantina: Recommendation applied in PR 550.

4.5.12 `add(leavesLength, proofLength)` used in `verifyMultiProof` can potentially overflow

Severity: Informational

Context:

- MerkleProofLib.sol#L106
- MerkleProofLib.sol#L193

Description: `add(leavesLength, proofLength)` in this context can potentially overflow. The conditions for the overflow are exact and require the protocol dev to either perform custom memory manipulation or allow external actors to perform custom memory manipulation through their exposed endpoints. This would not happen in normal expected usage of the library.

In a worst case scenario, this overflow can be used to verify a proof for any number of any arbitrary leaves. To show the impact concretely in an example, a protocol developer using this function for a merkle tree token airdrop could have an attacker claim the airdrop for any number of their own addresses, even if they were not included in the airdrop.

The worst case scenario PoC requires providing an empty `flags` array and memory manipulation of the `leaves` and `proof` arrays:

```
function testOverflowMultiVerify() public {
    bytes32[] memory proof = new bytes32[](0);
    bytes32 root = bytes32(0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef);
    bool[] memory flags = new bool[](0);

    // initialize a first leaf of leaf==root, while the rest are arbitrary
    bytes32[] memory leaves = new bytes32[](5);
    leaves[0] = root;
    leaves[1] = bytes32(uint(root) + 1);
    leaves[2] = bytes32(uint(root) + 2);
    leaves[3] = bytes32(uint(root) + 3);
    leaves[4] = bytes32(uint(root) + 4);


    // change len of proof and leaves to overflow the addition of:
    // add(leavesLength, proofLength)
    assembly {
        mstore(proof, 0x8000000000000000000000000000000000000000000000000000000000000000)
        mstore(leaves, 0x8000000000000000000000000000000000000000000000000000000000000001)
    }
    bool isValid = MerkleProofLib.verifyMultiProof(proof, root, leaves, flags);
    // All leaves appear validated
    assertTrue(isValid);
}
```

All of the leaves are validated, even though all but the first leaf is an arbitrary value.

It is worth noting that this issue does not pertain to the similar function `verifyMultiProofCalldata`, as the length of dynamic arrays in calldata are compared against `2^64 - 1` and will revert if larger (verified in solc versions `>=0.8`). This means the addition of two dynamic array lengths in calldata will never overflow.

Recommendation: It would be best to document and leave a NatSpec comment about this issue for devs. Custom memory operations before calling `verifyMultiProof` should be heavily scrutinized or avoided.

Cantina: Comments where added in PR 550.

4.5.13 Stricter requirement for `ERC1271`'s return data size

Severity: Informational

Context:

- SignatureCheckerLib.sol#L74
- SignatureCheckerLib.sol#L139
- SignatureCheckerLib.sol#L222
- SignatureCheckerLib.sol#L276
- SignatureCheckerLib.sol#L316
- SignatureCheckerLib.sol#L374

Description: `eq(returndatasize(), 0x20)` requirement for the `ERC1271` return data size in this context is stricter than OpenZeppelin's:

```
result.length >= 32
```

This change in OpenZeppelin has been discussed in this issue: OpenZeppelin/openzeppelin-contracts/issues/4035

The argument being that the `solc` compiler itself is not so strict on the returned size when being decoded to `bytes32` and it would just make sure that it would meet the minimum size required for the decoded tuple:

```
function <functionName>(headStart, dataEnd) <arrow> <valueReturnParams> {
  if slt(sub(dataEnd, headStart), <minimumSize>) { <revertString>() }
  <decodeElements>
}
```

Recommendation: No action is needed, but it would be great to be aware of this issue and maybe document that the requirements are more strict. Also since the `ERC1271`'s interface is written in 'Solidity' (it is not language/compiler agnostic) the requirements for the return value are not so precisely defined.

One should also be aware that in some edge cases due to this difference between implementation for an `ERC1271` contract Solady and OpenZeppelin might return different values.

Cantina: The requirement check is removed all together which can also allow return data size of less than `32` in PR 554.

This would not be an issue since we have `mstore(d, 0x40)` and for the memory slot at `d` to be equal to `f` one would need to remove the previous value of `0x40` through the `staticcall`.

4.5.14 Assembly block marked as `memory-safe` could in some cases cause `MSIZE` to change

Severity: Informational

Context: ECDSA.sol#L38-L40 ECDSA.sol#L194-L196

Description: In the context above when working with the `signature`, `mload(add(signature, A))` can potentially change `msize` if it causes the memory to expand.

- `libyul/SideEffects.h#L76-L78`

This can potentaillly have an effect on different `ir` optimisations (`libyul/optimiser/*`) and needs to be investigated.

Recommendation: It might be best to check the `signature` length before reading from the memory slots after `signature` unless this is proven to be safe when using the `solc` compiler.

Cantina: Marking as acknowledged.

4.5.15 `SHR(A, SHL(A, X))` can be replaced by masking `X`

Severity: Informational

Context:

- Ownable.sol#L90
- Ownable.sol#L104
- ERC20.sol#L348-L349
- ERC20.sol#L480
- ERC721.sol#L183
- ERC721.sol#L235
- ERC721.sol#L376
- ERC721.sol#L456
- ERC721.sol#L535
- ERC721.sol#L542
- ERC721.sol#L594
- ERC721.sol#L599
- ERC721.sol#L680-L681
- ERC721.sol#L867
- ERC1155.sol#L167
- ERC1155.sol#L1039
- ERC1155.sol#L1079
- ERC2981.sol#L103
- ERC2981.sol#L137
- ECDSA.sol#L112
- ECDSA.sol#L264
- SignatureCheckerLib.sol#L30
- SignatureCheckerLib.sol#L104
- SignatureCheckerLib.sol#L170
- SignatureCheckerLib.sol#L187
- SignatureCheckerLib.sol#L344

Description: Depending on the number of optimisation runs `SHR(A, SHL(A, X))` gets replaced by `AND(X, mask)`:

Ref:

- `libevmasm/RuleList.h#L525`

Recommendation: In the opening call, the client mentioned that they prefer using shifts over masks. It might make sense to inline this optimisation rule to have a more consistent runtime code across different optimisation steps/settings:

```
and(X, mask)
```

Vectorized: Acknowledged.

4.5.16 Inconsistent use of literals vs. constants for revert signatures

Severity: Informational

Context: ECDSA.sol#L54,92,126,161

Description: Some files in Solady library such as the `ECDSA` & `LibClone` libraries, use literals directly for customer error selectors vs. other files such as `ERC1967Factory` which uniquely define constants for selectors, which are then referenced throughout the file:

Example (from ECDSA#L53-L56):

```
if iszero(returndatasize()) {
    mstore(0x00, 0x8baa579f) // `InvalidSignature()`.
    revert(0x1c, 0x04)
}
```

Example (from ERC1967Factory#L24-L25,L100-L103):

```
/// @dev `bytes4(keccak256(bytes("Unauthorized()")))`.
uint256 internal constant _UNAUTHORIZED_ERROR_SELECTOR = 0x82b42900;
```

```
if iszero(eq(sload(adminSlot), caller())) {
    mstore(0x00, _UNAUTHORIZED_ERROR_SELECTOR)
    revert(0x1c, 0x04)
}
```

While having no concrete security impact it does hinder readability. Especially the literal variant may be inconvenient for readers as it may be hard to remember literals requiring each instance to be independently verified. Defining a constant allows verifying the validity of the signature once and having a more readable reference to it. Furthermore, the comment explaining what selector the literal represents would not have to be maintained.

Recommendation: Move to a consistent style across files, ideally the constant-variant.

Cantina: Marked as acknowledged.

5 Appendix

5.0.1 PoC: Empty signature can result in valid recovered address

```python
from typing import NamedTuple, Self
from Crypto.Hash import keccak
from random import randint


class Int(NamedTuple('Int', [('inner', int)])):
    def __add__(self, other: Self) -> Self:
        return Int((self.inner + other.inner) % F_p)

    def __mul__(self, other: Self) -> Self:
        return Int((self.inner * other.inner) % F_p)

    def __truediv__(self, other: Self) -> Self:
        return self * (other ** -1)

    def __pow__(self, e: int) -> Self:
        # Python's `pow` built-in supports modular arithmetic
        return Int(pow(self.inner, e, F_p))

    def __neg__(self) -> Self:
        return Int((-self.inner) % F_p)

    def __sub__(self, other: Self) -> Self:
        return self + -other


class Point(NamedTuple('Point', [('x', int), ('y', int)])):
    def coords(self) -> tuple[Int, Int]:
        return (*map(Int, self),)

    def __neg__(self) -> Self:
        return Point(self.x, (-self.y) % F_p)._validate()

    def __sub__(self, other: Self) -> Self:
        return self + -other

    def __add__(self, other: Self) -> Self:
        if self.identity():
            return other
        if other.identity():
            return self

        x1, y1 = self.coords()
        x2, y2 = other.coords()
        if x1 == x2:
            # Negation, return identity early
            if y1 != y2:
                return Point(0, 0)
            l = (Int(3) * (x1*x1) + a) / (Int(2) * y1)
        else:
            l = (y2 - y1) / (x2 - x1)

        x3 = l**2 - x1 - x2
        y3 = l * (x1 - x3) - y1

        return Point(x3.inner, y3.inner)._validate()

    def __mul__(self, x: int) -> Self:
        x = x % n
        y: Self = Point(0, 0)  # 0
        acc: Self = self
        for _ in range(x.bit_length()):
            if x & 1:
                y += acc
            x >>= 1
            acc += acc
        return y

    def __rmul__(self, x: int) -> Self:
        return self * x

    def _validate(self) -> Self:
        x, y = self.coords()
        assert self.identity() or y * y == x**3 + a * x + b
        return self

    def identity(self):
        return self.x == 0 and self.y == 0


def as_address(p: Point) -> str:
    hash = keccak.new(digest_bits=256)
    hash.update(p.x.to_bytes(32, 'big'))
    hash.update(p.y.to_bytes(32, 'big'))
    return f'0x{hash.digest()[12:32].hex()}'
```

```
# secp256k1 parameters
F_p = 0xFFFFFFFF_FFFFFFFF_FFFFFFFF_FFFFFFFF_FFFFFFFF_FFFFFFFF_FFFFFFFE_FFFFFC2F
n = 0xFFFFFFFF_FFFFFFFF_FFFFFFFF_FFFFFFFE_BAAEDCE6_AF48A03B_BFD25E8C_D0364141
G = Point(
    0x79BE667E_F9DCBBAC_55A06295_CE870B07_029BFCDB_2DCE28D9_59F2815B_16F81798,
    0x483ADA77_26A3C465_5DA4FBFC_0E1108A8_FD17B448_A6855419_9C47D08F_FB10D4B8
)
a = Int(0)
b = Int(7)


def main():
    # Private key in your control
    private_key = 0xca6e0c197892239353a097f7db686d4a0313f4316bc726bb7d8fe692f4d827ad
    public_key = private_key * G
    print(f'address: {as_address(public_key)}')

    # Random "secure" integer.
    # Note: `randint` is not a secure number generator, only for demonstration purposes
    k = randint(0, n-1)
    print(f'\nk:    {k}')

    # Compute r & v
    big_k = k * G
    v = 27 + big_k.y % 2
    r = big_k.x % n

    # Compute the hash `z` such that the signature `(v, r, s)` is valid for `z`
    s = 65
    z = (s * k - r * private_key) % n

    print(f'\nhash: 0x{z:064x}')
    print(f'v:    {v}')
    print(f'r:    0x{r:064x}')
    print(f's:    0x{s:064x}')


if __name__ == '__main__':
    main()
```

5.0.2 PoC Execution ordering of and() may lead to unexpected behavior in future compiler versions

```
import subprocess
import csv

"""
Compiler Optimization Flags:

Selecting Optimizations
By default the optimizer applies its predefined sequence of optimization steps to the generated assembly. You can override
↪  this sequence and supply your own using the --yul-optimizations option:

solc --optimize --ir-optimized --yul-optimizations 'dhfoD[xarrscLMcCTU]uljmul:fDnTOc'
The order of steps is significant and affects the quality of the output. Moreover, applying a step may uncover new
↪  optimization opportunities for others that were already applied, so repeating steps is often beneficial.


f"~/.sum/{compiler_version}/solc-{compiler_version} --bin-runtime {file_path} --no-cbor-metadata {optimization_flags}"

--optimize:
    ALWAYS ON
--optimize-runs:
   ALWAYS ON, CHANGE NUM. Use 1, 200, 10000, 999999
--via-ir:
    toggle its inclusion or exclusion as a flag
"""

def check_opcodes(bytecode):
    # print(len(bytecode))
    # print(bytecode)
    call_found = False
    rds_after_call = False
    i = 0
    while i < len(bytecode) - 1:
        # print(i)
        opcode = bytecode[i:i+2]
        # convert opcode to hex uint8
        opcode = int(opcode, 16)
        # check for push. Anything in range 5F-7F will be a push
        if opcode >= 0x5F and opcode <= 0x7F:
            # get the length of the push
            push_length = opcode - 0x5F
            # advance past the push bytes
            i += push_length*2
        elif opcode == 0xFA:
```

```
            call_found = True
        elif opcode == 0x3D and not call_found:
            rds_after_call = False
            return rds_after_call
        elif opcode == 0x3D and call_found:
            rds_after_call = True
            pass
        i += 2
    return rds_after_call

def compile_solidity(file_path, compiler_version, optimize_runs=None, via_ir=None):
    optimize_flags = f"--optimize --optimize-runs {optimize_runs} {('--experimental-via-ir' if via_ir else '')}" if
    ↪  optimize_runs is not None else ""
    cmd = f"~/.svm/{compiler_version}/solc-{compiler_version} --bin-runtime {file_path} --metadata-hash none {optimize_flags}"
    print(cmd)
    process = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    stdout, _ = process.communicate()
    binary_part = stdout.split(b"Binary of the runtime part:")[-1].strip()
    return binary_part.decode()

compiler_versions = [
    "0.8.0", "0.8.1", "0.8.2", "0.8.3", "0.8.4", "0.8.5", "0.8.6",
    "0.8.7", "0.8.8", "0.8.9", "0.8.10", "0.8.11", "0.8.12", "0.8.13",
    "0.8.14", "0.8.15", "0.8.16", "0.8.17", "0.8.18", "0.8.19", "0.8.20", "0.8.21"
] # List of compiler versions
optimize_runs_values = [1, 200, 10000, 999999]
via_ir_options = [True, False]
file_path = "./src/OrderingTest.sol"

# Open the CSV file for writing
with open('results.csv', 'w', newline='') as csvfile:
    # Create a CSV writer object
    writer = csv.writer(csvfile)

    # Write the header row
    writer.writerow(['Compiler Version', 'Optimize Runs', 'Via IR', 'Bytecode Len', 'RDS After Call'])

    # Iterate through the compiler versions
    for compiler_version in compiler_versions:
        # print(f"Running {compiler_version}")
        # Run without optimization
        binary_part = compile_solidity(file_path, compiler_version)
        binary_part = ''.join(filter(lambda x: x in '0123456789abcdefABCDEF', binary_part))
        if len(binary_part) == 0:
            pass
        else:
            result = check_opcodes(binary_part)
            writer.writerow([compiler_version, "NA", "False", str(len(binary_part)), result])

        # Run with various optimization options
        for optimize_runs in optimize_runs_values:
            for via_ir in via_ir_options:
                binary_part = compile_solidity(file_path, compiler_version, optimize_runs, via_ir)
                # print(binary_part)
                binary_part = ''.join(filter(lambda x: x in '0123456789abcdefABCDEF', binary_part))
                if len(binary_part) == 0:
                    pass
                else:
                    result = check_opcodes(binary_part)
                    writer.writerow([compiler_version, optimize_runs, via_ir, str(len(binary_part)), result])
```