// HALBORN

DeBridge - EVM to Solana Serializer

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **July 5th, 2023 - July 12th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	07/10/2023	Przemyslaw Swiatowiec
0.2	Document Updates	07/07/2023	Przemyslaw Swiatowiec
0.3	Final Draft	07/11/2023	Przemyslaw Swiatowiec
0.4	Draft Review	07/11/2023	Grzegorz Trawinski
0.5	Draft Review	07/11/2023	Piotr Cielas
0.6	Draft Review	07/12/2023	Gabi Urrutia
1.0	Remediation Plan	08/30/2023	Przemyslaw Swiatowiec
1.1	Remediation Plan Review	08/30/2023	Piotr Cielas
1.2	Remediation Plan Review	08/31/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Piotr Cielas	Halborn	Piotr.Cielas@halborn.com
Grzegorz Trawinski	Halborn	Grzegorz.Trawinski@halborn.com
Przemyslaw Swiatowiec	Halborn	Przemyslaw.Swiatowiec@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

The EVM to Solana Serializer serves the purpose of declaring and serializing the DeBridge ExternalInstruction data structure. By utilizing this library, developers can create legitimate and secure calls to Solana smart contracts on the supported EVM chains. These calls can then be bridged to Solana using DeBridge's generic messaging and cross-chain interoperability protocol, enabling seamless communication and interaction between different blockchain ecosystems.

DeBridge engaged Halborn to conduct a security assessment on their smart contracts beginning on July 5th, 2023 and ending on July 12th, 2023. The security assessment was scoped to the Solidity library provided in the evm-sol-serializer GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

1.2 ASSESSMENT SUMMARY

Halborn was provided one week for the engagement and assigned a full-time security engineer to assessment the security of the Solidity library in scope. The security team consists of a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessments is to:

- Identify potential security issues within the library.
- Ensure that the library functionality operates as intended.

In summary, Halborn identified some code quality improvements, which were acknowledged by the DeBridge team. The main ones are the following:
- Pinning the DeBridgeSolana library to a fixed Solidity version.
- Introducing for loops optimization.

- Removing commented code.
- Adding NatSpec to the DeBridgeSolana.sol library functions.

1.3 TEST APPROACH & METHODOLOGY

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

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

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A)	1
	Specific (AO:S)	0.2
Attack Cost (AC)	Low (AC:L)	1
	Medium (AC:M)	0.67
	High (AC:H)	0.33
Attack Complexity (AX)	Low (AX:L)	1
	Medium (AX:M)	0.67
	High (AX:H)	0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 SEVERITY COEFFICIENT

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2.4 SCOPE

Code repositories:

1. EVM to Solana Serializer

- Repository: evm-sol-serializer
- Commit ID: 44ef6fd9909f813440f4a474e6ffe05db80a0ca4
- Library in scope:

  1. DeBridgeSolana.sol (contracts/library/DeBridgeSolana.sol)

Out-of-scope

- Third-party libraries and dependencies.
- DeBridge cross chain communication.
- Bridging and execution of ExternalInstruction.

Please note that the project documentation states that developers are responsible for preparing and validating the values included in the data struct to ensure their correctness. The documentation emphasizes the importance of developer diligence and accuracy when providing data to the library, as it plays a significant role in maintaining the overall integrity and reliability of the system. **Due to this statement, validation for data used in the EVM to Solana Serializer library was not included in the scope of this assessment.**

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	0	4

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) FLOATING PRAGMA	Informational (0.0)	ACKNOWLEDGED
(HAL-02) LOOP GAS USAGE OPTIMIZATION	Informational (0.0)	ACKNOWLEDGED
(HAL-03) COMMENTED CODE	Informational (0.0)	ACKNOWLEDGED
(HAL-04) INCOMPLETE NATSPEC DOCUMENTATION	Informational (0.0)	ACKNOWLEDGED

FINDINGS & TECH DETAILS

4.1 (HAL-01) FLOATING PRAGMA - INFORMATIONAL (0.0)

Description:

The pragma annotation in the DeBridgeSolana.sol library is not pinned to exact version (floating pragma).

Contracts and libraries should be deployed with the same compiler version and flags used during development and testing. Locking the pragma helps to ensure that contracts do not accidentally get deployed using another pragma. For example, an outdated pragma version might introduce bugs that affect the contract system negatively.

Code Location:

Listing 1: DeBridgeSolana.sol (Line 1)

```
// SPDX-License-Identifier: LGPL-3.0-only
pragma solidity ^0.8.0;

library DeBridgeSolana {
```

BVSS:

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

Recommendation:

Consider locking the pragma version in the library.

For example: `pragma solidity 0.8.19;`

Remediation Plan:

ACKNOWLEDGED: The DeBridge team acknowledged this issue.

4.2 (HAL-02) LOOP GAS USAGE OPTIMIZATION - INFORMATIONAL (0.0)

Description:

It was identified that the for loop employed in the contracts can be gas optimized by incrementing iterator using `++i` instead of `i++`.

In Solidity, using the `++i` notation is generally considered better than `i++` or `i += 1` in `for` loops, primarily due to the difference in gas cost. The `++i` notation incurs lower gas costs compared to the post-increment `i++` or `i += 1` for unsigned integers, even with the optimizer enabled. The pre-increment operation in Solidity is more gas-efficient, typically costing around 5 gas per iteration.

Code Location:

Listing 2: DeBridgeSolana.sol (Line 116)

```
  data = abi.encodePacked(uint64(seeds.length).
    ↳ uint64ToLittleEndian());
  for (uint i = 0; i < seeds.length; i++) {
    data = abi.encodePacked(data, seeds[i]);
  }
```

Listing 3: DeBridgeSolana.sol (Line 166)

```
  // beware! This loop starts at (1) rather than (0) because the
    ↳ first (zeroed) element
  // is being inlined onto previous abi.encodePacked() call. This
    ↳ saves 500 gas
  for (uint i = 1; i < pss.length; i++) {
    data = abi.encodePacked(data, pss[i].u64.uint64ToLittleEndian
    ↳ (), pss[i].data);
  }
```

Listing 4: DeBridgeSolana.sol (Line 182)

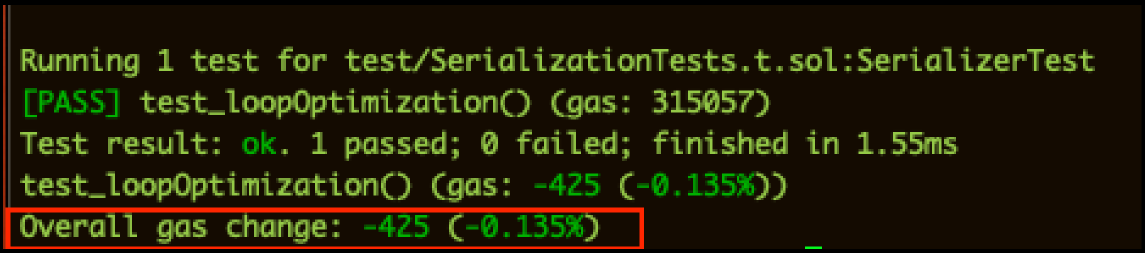
```
for (uint i = 0; i < dss.length; i++) {
  data = abi.encodePacked(data, dss[i].data);
}
```

Listing 5: DeBridgeSolana.sol (Line 218)

```
for (uint i = 0; i < ams.length; i++) {
  data = abi.encodePacked(
    data,
    // inline call of `serialize(DeBridgeSolana.AccountMeta memory
    ↳ )` for optimization
    ams[i].pubkey,
    ams[i].is_signer,
    ams[i].is_writable
  );
...
}
```

Gas Consumption Benchmark Tests:

The forge gas snapshot feature was used to measure the difference between non-optimized and optimized-code:



```
Running 1 test for test/SerializationTests.t.sol:SerializerTest
[PASS] test_loopOptimization() (gas: 315057)
Test result: ok. 1 passed; 0 failed; finished in 1.55ms
test_loopOptimization() (gas: -425 (-0.135%))
Overall gas change: -425 (-0.135%)
```

Figure 1: Gas difference after loop optimization

BVSS:

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

Recommendation:

It is recommended to use ++i (preincrementation) instead of i++ (postincrementation) to increment the loop iterator.

Remediation Plan:

ACKNOWLEDGED: The DeBridge team acknowledged this issue.

4.3 (HAL-03) COMMENTED CODE - INFORMATIONAL (0.0)

Description:

The codebase contains several instances of commented-out code. These commented sections can impact code readability negatively, making it more challenging for developers to understand the code.

Code Location:

Listing 6: DeBridgeSolana.sol (Line 88)

```
87 function getArbitrarySeed(bytes memory vec) internal pure returns (bytes memory) {
88   // require(vec.length <= (2**64 - 1), "U64");
89
90   return abi.encodePacked(hex'00000000', uint64(vec.length).uint64ToLittleEndian(), vec);
91 }
```

Listing 7: DeBridgeSolana.sol (Line 113)

```
112 function serialize(bytes[] memory seeds) internal pure returns (bytes memory data) {
113   // require(seeds.length <= (2**64 - 1), "U64");
114
115   data = abi.encodePacked(uint64(seeds.length).uint64ToLittleEndian());
116   ...
117 }
```

Listing 8: DeBridgeSolana.sol (Line 142)

```
141 function serialize(bytes memory vec) internal pure returns (bytes memory data) {
142   // require(vec.length <= (2**64 - 1), "U64");
143
```

```
  data = abi.encodePacked(uint64(vec.length).uint64ToLittleEndian
↳ (), vec);
}
```

Listing 9: DeBridgeSolana.sol (Line 153)

```
if (pss.length == 0) {
  data = hex'00';
} else {
  // require(pss.length <= (2**64 - 1), "U64");
...
}
```

Listing 10: DeBridgeSolana.sol (Line 178)

```
if (dss.length == 0) {
  data = hex'00';
} else {
  // require(dss.length <= (2**64 - 1), "U64");

  data = abi.encodePacked(hex'01', uint64(dss.length).
↳ uint64ToLittleEndian());

  for (uint i = 0; i < dss.length; i++) {
    data = abi.encodePacked(data, dss[i].data);
  }
}
```

Listing 11: DeBridgeSolana.sol (Line 209)

```
function serialize(
    DeBridgeSolana.AccountMeta[] memory ams
 ) internal pure returns (bytes memory data) {
    // require(ams.length <= (2**64 - 1), "U64");

    // inlining vector length on an higher level saves 500 gas {{{
    // data = abi.encodePacked(uint64(ams.length).
↳ uint64ToLittleEndian());
    // }}}

    for (uint i = 0; i < ams.length; i++) {
    ...
}}
```

BVSS:

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

Recommendation:

It is recommended to remove commented lines to enhance the overall clarity and maintainability of the codebase.

Remediation Plan:

ACKNOWLEDGED: The DeBridge team acknowledged this issue.

4.4 (HAL-04) INCOMPLETE NATSPEC DOCUMENTATION - INFORMATIONAL (0.0)

Description:

The Natspec documentation is a form of inline documentation that provides essential information about a project. It serves as a valuable resource for internal developers, external developers, auditors, and end users. Internal developers can refer to the documentation to understand the project’s functionality and make modifications or additions to the codebase. External developers who intend to integrate their own projects with the verified project can also benefit from the Natspec documentation, as it provides guidance on how to interact with the project’s smart contracts. Auditors can review the documentation to gain a comprehensive understanding of the project’s design and implementation. Additionally, end users can access the Natspec documentation through various blockchain explorers, thus enhancing their understanding of the project’s features and functionalities.

Code Location:

All functions in the DeBridgeSolana.sol library.

BVSS:

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

Recommendation:

It is recommended to provide comprehensive and accurate Natspec documentation for the project. This documentation should cover all relevant smart contracts, functions, and their respective inputs and outputs. The documentation should be easily accessible and ideally integrated with popular blockchain explorers. By providing clear and thorough documentation, the project can facilitate its development, integration, auditing, and usage processes for all stakeholders involved.

Remediation Plan:

ACKNOWLEDGED: The DeBridge team acknowledged this issue.

MANUAL TESTING

In the manual testing phase, the following scenarios were simulated. The scenarios listed below were selected based on the severity of the vulnerabilities Halborn was testing the program for.

5.1 DIFFERENCE IN SERIALIZATION OUTPUT IN RUST AND IN THE VERIFIED LIBRARY

Description:

The testing methodology employed for this security assessment primarily focused on the serialization of the ExternalInstruction data struct. The approach involves setting up test environments for both Solidity and Rust, where the ExternalInstruction is serialized in Rust using the serde library. The same parameters are then serialized using the EVM to Solana library in Solidity, and the resulting outputs are compared.

The testing process involves meticulously comparing the serialization outputs of various ExternalInstructions. By conducting these comparisons, the security assessment aims to ensure that the verified Solidity library produces consistent and accurate results when serializing the ExternalInstruction data struct, aligning with the serialization performed in Rust. This comparative analysis allows for thorough validation of the serialization process and ensures that the library functions reliably and as intended.

Examples of tested scenarios:
- serialization of edge cases for uint64 values - value 0 and U64::MAX (18446744073709551615),
- serialization of different values for Solana public keys,
- ExternalInstructions serialization with and without optional fields,
- enum and constant values serialization,
- serialization of different vectors with different length containing AccountMeta, DataSubstitution and PubkeySubstitution structs,
- serialization of different vectors of bytes representing Instruction

data field.

Script for serialization in Rust

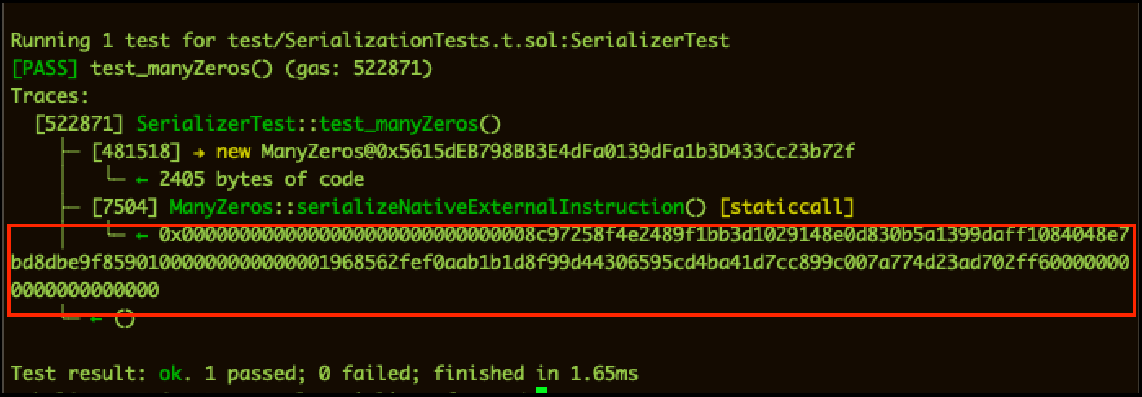
Listing 12: ExternalInstruction.rs (Line 5)

```rust
#[derive(Debug, Clone, PartialEq, Eq, Deserialize, Serialize)]
pub struct ExternalInstruction {
    pub reward: Amount,
    pub expense: Option<Amount>,
    pub execute_policy: ExecutePolicy,
    pub pubkey_substitutions: Option<Vec<(u64, PubkeySubstitution)
↳ >>,
    pub data_substitutions: Option<Vec<DataSubstitution>>,
    pub instruction: Instruction,
}

pub type Amount = u64;

#[derive(Debug, Clone, PartialEq, Eq, Serialize, Deserialize)]
pub enum ExecutePolicy {
    Empty,
    MandatoryBlock,
}

#[derive(Debug, Clone, PartialEq, Eq, Serialize, Deserialize)]
pub enum SeedVariants {
    Arbitrary(Vec<u8>),
    SubmissionAuth,
}

#[derive(Debug, Clone, PartialEq, Eq, Serialize, Deserialize)]
pub enum PubkeySubstitution {
    SubmissionAuthWallet {
        token_mint: Pubkey,
    },
    BySeeds {
        program_id: Pubkey,
        seeds: Vec<SeedVariants>,
        bump: Option<u8>,
    },
}

#[derive(Debug, Clone, Copy, Serialize, Deserialize)]
pub struct AccountMeta {
    pub pubkey: Pubkey,
    pub is_signer: bool,
    pub is_writable: bool,
}

impl PartialEq for AccountMeta {
    fn eq(&self, other: &Self) -> bool {
        self.pubkey == other.pubkey
    }
}

impl Eq for AccountMeta {}

#[derive(Debug, Eq, PartialEq, Clone, Serialize, Deserialize)]
pub struct Instruction {
    pub program_id: Pubkey,
    pub accounts: Vec<AccountMeta>,
    pub data: Vec<u8>,
}

#[derive(Debug, Clone, PartialEq, Eq, Serialize, Deserialize)]
pub enum DataSubstitution {
    SubmissionAuthWalletAmount {
        account_index: usize,
        offset: usize,
        is_big_endian: bool,
        subtraction: u64,
    },
}

mod tests {
    #[test]
    fn test_compareWithRust() {
        let ix = ExternalInstruction {...};
        let test_bytes = bincode::serialize(&ix).unwrap();
        let test_hex = hex::encode(&test_bytes);
        let from_evm = String::from("...");
        assert_eq!(test_hex, from_evm);
    }
}
```

Results:

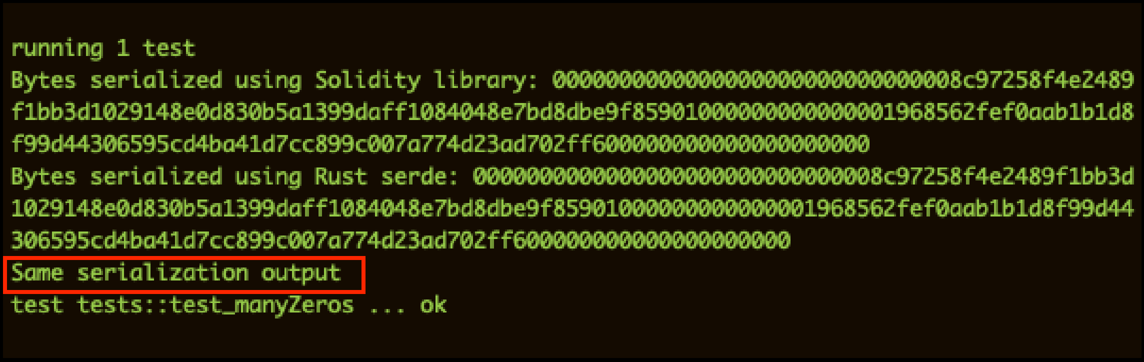
No vulnerabilities were identified.

Serialization of ExternalInstruction struct without optional fields



```
Running 1 test for test/SerializationTests.t.sol:SerializerTest
[PASS] test_manyZeros() (gas: 522871)
Traces:
  [522871] SerializerTest::test_manyZeros()
    ├─ [481518] → new ManyZeros@0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
    │   └─ ← 2405 bytes of code
    ├─ [7504] ManyZeros::serializeNativeExternalInstruction() [staticcall]
    │   └─ ← 0x0000000000000000000000000000008c97258f4e2489f1bb3d1029148e0d830b5a1399daff1084048e7
bd8dbe9f8590100000000000000001968562fef0aab1b1d8f99d44306595cd4ba41d7cc899c007a774d23ad702ff60000000
0000000000000
    └─ ← ()

Test result: ok. 1 passed; 0 failed; finished in 1.65ms
```

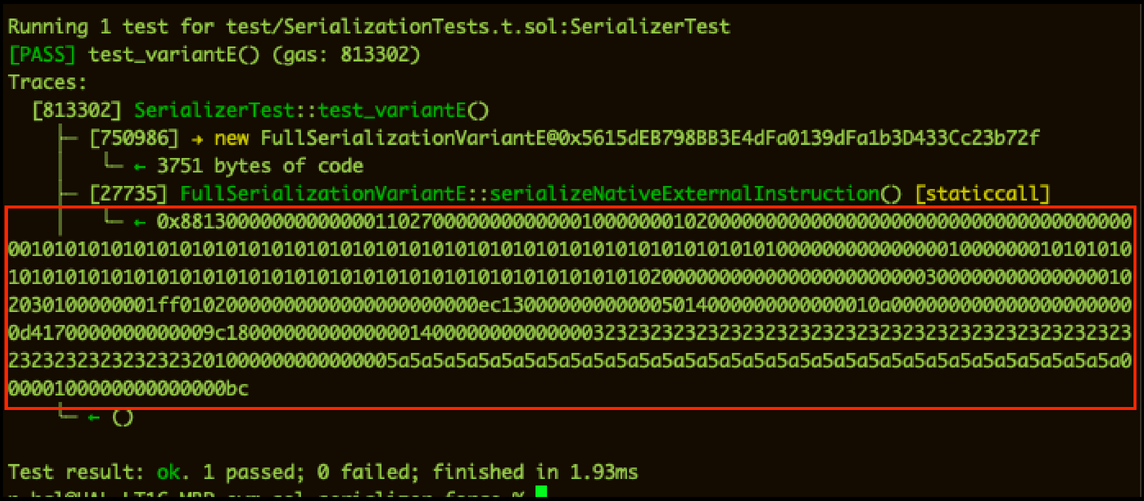
Figure 2: Serialized struct bytes in using Solidity library.



```
running 1 test
Bytes serialized using Solidity library: 0000000000000000000000000000008c97258f4e2489
f1bb3d1029148e0d830b5a1399daff1084048e7bd8dbe9f8590100000000000000001968562fef0aab1b1d8
f99d44306595cd4ba41d7cc899c007a774d23ad702ff600000000000000000000
Bytes serialized using Rust serde: 0000000000000000000000000000008c97258f4e2489f1bb3d
1029148e0d830b5a1399daff1084048e7bd8dbe9f8590100000000000000001968562fef0aab1b1d8f99d44
306595cd4ba41d7cc899c007a774d23ad702ff600000000000000000000
Same serialization output
test tests::test_manyZeros ... ok
```

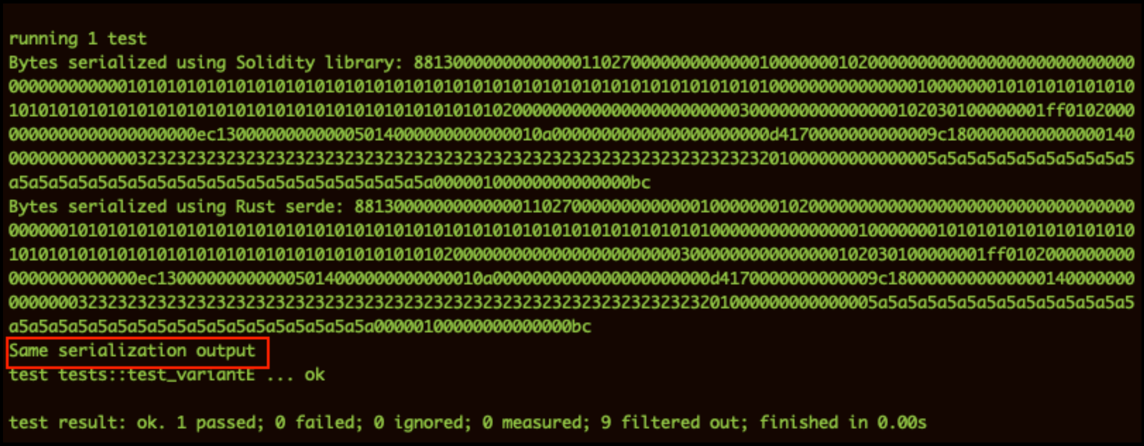
Figure 3: Comparison of Solidity and Rust serialization.

Serialization of ExternalInstruction struct with optional fields



```
Running 1 test for test/SerializationTests.t.sol:SerializerTest
[PASS] test_variantE() (gas: 813302)
Traces:
  [813302] SerializerTest::test_variantE()
    ├─ [750986] → new FullSerializationVariantE@0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
    │   └─ ← 3751 bytes of code
    ├─ [27735] FullSerializationVariantE::serializeNativeExternalInstruction() [staticcall]
    │   └─ ← 0x8813000000000000000110270000000000000001000000010200000000000000000000000000000000000000
0010101010101010101010101010101010101010101010101010101010101010101000000000000000001000000010101010
1010101010101010101010101010101010101010101010101010102000000000000000000000000030000000000000000010
2030100000001ff0102000000000000000000000000ec13000000000000050140000000000000010a000000000000000000000
0d4170000000000009c18000000000000000140000000000000003232323232323232323232323232323232323232323
2323232323232323232010000000000000005a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a0
0000100000000000000000bc
    └─ ← ()

Test result: ok. 1 passed; 0 failed; finished in 1.93ms
```

Figure 4: Serialized struct bytes in using Solidity library.



```
running 1 test
Bytes serialized using Solidity library: 88130000000000000001102700000000000000010000000102000000000000000000000000000
0000000000001010101010101010101010101010101010101010101010101010101010101000000000000000000100000001010101010101010
1010101010101010101010101010101010101010101010101020000000000000000000000000300000000000000000102030100000001ff0102000
000000000000000000ec1300000000000005014000000000000000010a000000000000000000000000d4170000000000009c1800000000000000140
0000000000003232323232323232323232323232323232323232323232323232323232323232323232010000000000000005a5a5a5a5a5a5a5a5
a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a000001000000000000000bc
Bytes serialized using Rust serde: 88130000000000000001102700000000000000010000000102000000000000000000000000000000000
00000010101010101010101010101010101010101010101010101010101010101010100000000000000000001000000010101010101010101010
101010101010101010101010101010101010101010101020000000000000000000000000300000000000000000102030100000001ff0102000000000
0000000000000ec1300000000000005014000000000000000010a000000000000000000000000d4170000000000009c1800000000000000140000000
000000032323232323232323232323232323232323232323232323232323232323232323201000000000000000005a5a5a5a5a5a5a5a5a5a5a5
a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a000001000000000000000bc
Same serialization output
test tests::test_variantE ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 9 filtered out; finished in 0.00s
```

Figure 5: Comparison of Solidity and Rust serialization.

AUTOMATED TESTING

6.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.
The security team assessed all findings identified by the Slither software, however, findings with severity Information and Optimization are not included in the below results for the sake of report readability.

Results:

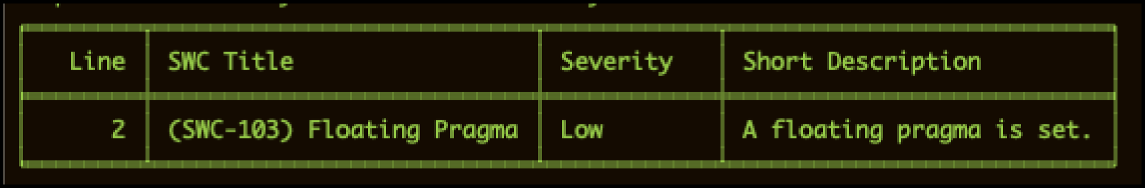
Slither did not identify any vulnerabilities in the verified library.

6.2 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and to identify low-hanging fruits on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the smart contracts and sent the compiled results to the analyzers in order to locate any vulnerabilities.

Results:



Line	SWC Title	Severity	Short Description
2	(SWC-103) Floating Pragma	Low	A floating pragma is set.

Figure 6: MythX output for DeBridgeSolana.sol library

The finding obtained as a result of the MythX scan was examined and reported as HAL-01 in the previous section of this report.

THANK YOU FOR CHOOSING

HALBORN