deBridge - Multisignature

Solana Program Security Audit

Prepared by: **Halborn**

Date of Engagement: **March 29th, 2022 - April 19th, 2022**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	04/18/2022	Mateusz Garncarek
0.2	Document Edit	04/19/2022	Mateusz Garncarek
0.3	Draft Review	04/19/2022	Gabi Urrutia
1.0	Remediation Plan	05/12/2022	Mateusz Garncarek
1.1	Remediation Plan Review	05/16/2022	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
Mateusz Garncarek	Halborn	Mateusz.Garncarek@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

deBridge engaged Halborn to conduct a security assessment on their Solana programs beginning on March 29th, 2022 and ending April 19th, 2022. The security assessment was scoped to the Solana programs provided in the GitHub repository debridge-finance/de-multi-signature. deBridge is a cross-chain interoperability and liquidity transfer protocol that allows truly decentralized transfer of assets between various blockchains. The cross-chain intercommunication of deBridge programs is powered by the network of independent oracles/validators which are elected by deBridge governance.

1.2 AUDIT SUMMARY

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

The purpose of this audit to achieve the following:

- Ensure that Solana programs functions are intended.
- Identify potential security issues with the Solana programs.

In summary, Halborn identified some security risks that were addressed and acknowledged by the deBridge team.

1.3 TEST APPROACH & METHODOLOGY

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

- Research into architecture, purpose, and use of the platform.
- Manual code review and walkthrough.
- Manual assessment of use and safety for the critical Rust variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual assessment to determine access control issues such as missing ownership checks, missing signer checks, and Solana account confusions.
- Fuzz testing. (Halborn custom fuzzing tool)
- Scanning of Rust files for vulnerabilities. (cargo audit)
- Detecting usage of unsafe Rust code. (cargo-geiger)

RISK METHODOLOGY:

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

RISK SCALE - LIKELIHOOD

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

RISK SCALE - IMPACT

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

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

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

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

1.4 SCOPE

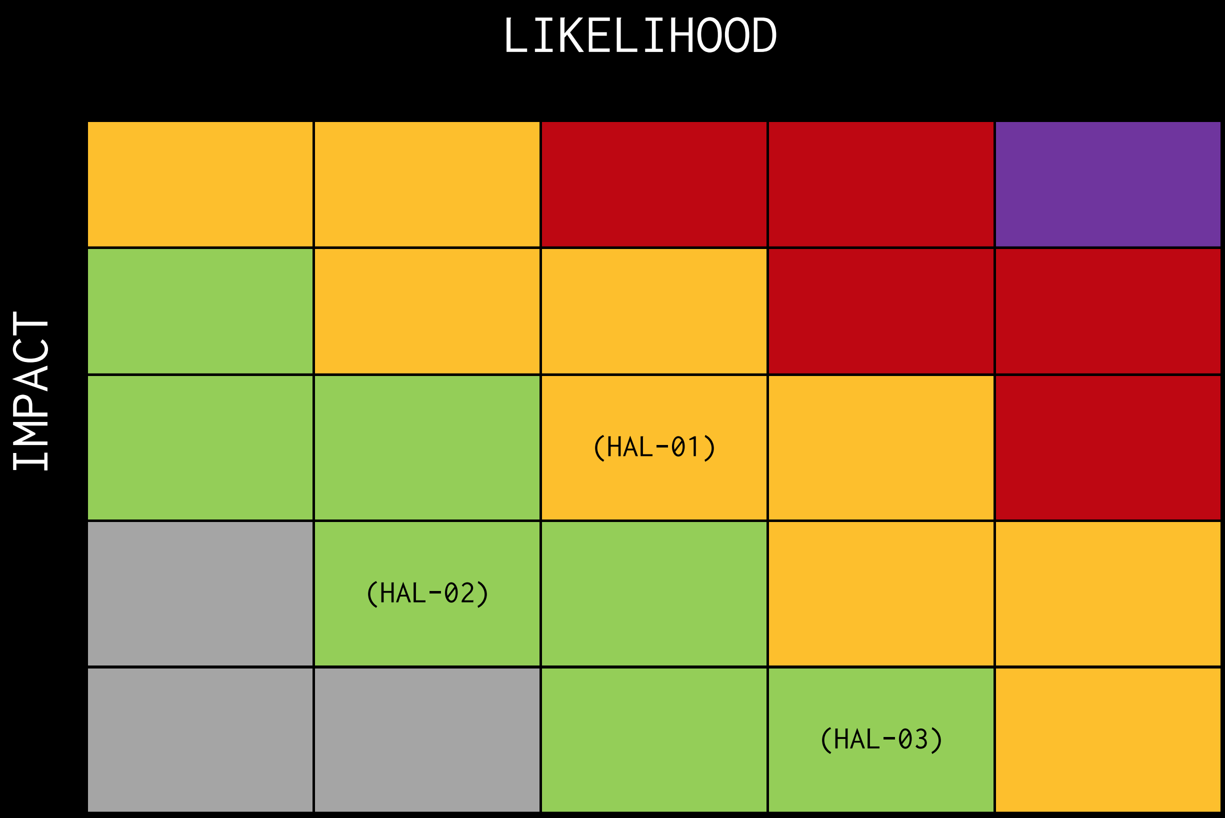
1. Repository: debridge-finance/de-multi-signature
2. Commit ID: 21bef49fba944a791b771a8f53ab702ae1264263
3. Programs in-scope:
   (a) de-multi-signature
   (b) de-program-updater

OUT-OF-SCOPE:

- Other Solana programs in the repository
- Economics attacks
- Third-party dependencies

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	2	0



SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) ARITHMETIC ERRORS	Medium	SOLVED - 10/05/2022
(HAL-02) TAUTOLOGY EXPRESSIONS	Low	ACKNOWLEDGED
(HAL-03) USAGE OF VULNERABLE CRATES	Low	ACKNOWLEDGED

FINDINGS & TECH DETAILS

3.1 (HAL-01) ARITHMETIC ERRORS - MEDIUM

Description:

Serious arithmetic errors include integer overflow/underflow. In computer programming, integer overflow/underflow occurs when an arithmetic operation attempts to create a numeric value that is outside the range that can be represented by a given number of bits, either greater than the maximum or less than the minimum representable value. Although integer overflows and underflows do not cause Rust to panic in the release mode, the consequences could be dire if the result of those operations is used in financial calculations.

Code Location:

Integer Overflow/Underflow

Listing 1: programs/de-multi-signature/lib.rs (Line 1313)

```
1305 impl<'info> RegisterChange for Account<'info, IntegrityChecker> {
1306     fn register_changes<S: TryBorrowProposalStorage>(
1307         &self,
1308         proposal_storage: &S,
1309         offset: usize,
1310         change_len: NonZeroUsize,
1311     ) -> Result<Hash> {
1312         let begin_chunk = offset / LEAF_CHUNK_SIZE;
1313         let end_chunk = (offset + change_len.get()) / LEAF_CHUNK_SIZE;
1314         let storage = proposal_storage.try_borrow_proposal_storage()?;
1315         IntegrityChecker::update_hash_sum(
1316             self,
1317             begin_chunk,
1318             IntegrityChecker::calculate_slot(&storage.as_ref(), begin_chunk)?,
1319             if begin_chunk == end_chunk {
1320                 None
```

```
            } else {
                Some(IntegrityChecker::calculate_slot(
                    &storage.as_ref(),
                    end_chunk,
                )?)
            },
        )?;
        self.try_get_proposal_hash()
    }
}

```

Listing 2: programs/de-multi-signature/lib.rs (Line 952)

```
938 impl Instruction {
939     fn pop(storage: impl AsRef<[u8]>, offset: &mut u64) -> Result<Option<Box<Instruction>>> {
940         if storage.as_ref().len() <= *offset as usize {
941             return Ok(None);
942         }
943
944         let mut ptr = storage.as_ref().split_at(*offset as usize).1;
945
946         let remaning_len = ptr.len();
947         let ix = BorshDeserialize::deserialize(&mut ptr).map_err(|err| {
948             msg!("Error while deserialize: {:?}", err);
949             ErrorCode::WrongInstructionDeserializing
950         })?;
951
952         *offset += (remaning_len - ptr.len()) as u64;
953
954         Ok(Some(ix))
955     }
956 }
```

Listing 3: programs/de-program-updater/lib.rs (Line 415)

```
408 impl<'info> CalculateSlot for AccountInfo<'info> {
409     fn calculate_slot(&self, index: usize) -> Result<Hash> {
410         let storage = self.try_borrow_data()?;
411         let offset =
```

```
            UpgradeableLoaderState::buffer_data_offset().expect("
↳ Unreachable. Constant offset");
        // TODO #9 Add check for bpf loader account
        Ok(merkle_tree::calculate_leaf(
            &storage[offset + index * LEAF_CHUNK_SIZE
                ..storage.len().min(offset + (index + 1) *
↳ LEAF_CHUNK_SIZE)],
        ))
    }
}
```

Division

Listing 4: programs/de-multi-signature/lib.rs (Lines 1312,1313)

```
1305 impl<'info> RegisterChange for Account<'info, IntegrityChecker> {
1306     fn register_changes<S: TryBorrowProposalStorage>(
1307         &self,
1308         proposal_storage: &S,
1309         offset: usize,
1310         change_len: NonZeroUsize,
1311     ) -> Result<Hash> {
1312         let begin_chunk = offset / LEAF_CHUNK_SIZE;
1313         let end_chunk = (offset + change_len.get()) /
   ↳ LEAF_CHUNK_SIZE;
1314         let storage = proposal_storage.try_borrow_proposal_storage
   ↳ ()?;
1315         IntegrityChecker::update_hash_sum(
1316             self,
1317             begin_chunk,
1318             IntegrityChecker::calculate_slot(&storage.as_ref(),
   ↳ begin_chunk)?,
1319             if begin_chunk == end_chunk {
1320                 None
1321             } else {
1322                 Some(IntegrityChecker::calculate_slot(
1323                     &storage.as_ref(),
1324                     end_chunk,
1325                 )?)
1326             },
1327         )?;
1328         self.try_get_proposal_hash()
1329     }
```

```
1330 }
```

Risk Level:

Likelihood - 3
Impact - 3

Recommendation:

It is recommended to use safe and verified math libraries (such as checked_add, checked_div) for consistent arithmetic operations throughout the Solana program system. Consider using Rust safe arithmetic functions for primitives instead of standard arithmetic operators.

References

Safe arithmetic operations for primitives: u8, u32, u64.

Remediation Plan:

SOLVED: The DeBridge team fixed this issue in commit ccdb347082cfaace9ccb02aba0e1ab5dd1f84412.

3.2 (HAL-02) TAUTOLOGY EXPRESSIONS - LOW

Description:

Some accounts initialized by several instruction handlers derive their addresses from user-supplied seeds and bumps. Anchor simplifies account initialization by providing a collection of #[account] attribute macro parameters, including seed and bump. Those two are used when the address of the account to be initialized is a PDA.

Anchor allows program developers to have users provide bumps that are used in PDA generation. In such case, the attribute might look like so:

Listing 5

```
#[ account (init , seeds = [seed1 , seed2 ], bump = bump )]
```

Alternatively, programs may calculate bumps automatically:

Listing 6

```
#[ account (init , seeds = [seed1 , seed2 ], bump )]
```

In either case, the bump should be equal to the one calculated with the Pubkey::find_program_address function.

According to official Solana documentation there's a 50/50 percent chance of generating a correct address given a collection of seeds and a u8 bump, which means there could be over 120 valid bumps for a given collection of seeds.
Considering all the above, having users to provide bumps is not only unnecessary but also misleading as perfectly valid bumps are rejected by the framework.

Code Example Location:

Listing 7: programs/de-multi-signature/lib.rs (Line 685)

```
    fn create_multi_signature_storage_address(
        name: &str,
        bump: u8,
    ) -> result::Result<Pubkey, PubkeyError> {
        let bump = [bump];
        let mut seeds = get_multisignature_storage_seeds(name);
        seeds.push(&bump);
        Pubkey::create_program_address(seeds.as_slice(), &ID)
    }
```

Listing 8: programs/de-multi-signature/lib.rs (Line 481)

```
    pub struct ExecuteProposal<'info>
        #[account(
            mut,
            seeds = [Proposal::SEED, multisignature_storage.key().
 ↳ as_ref(), &proposal_index.to_be_bytes()],
            bump = proposal.proposal_bump,
            has_one = proposal_storage,
            has_one = proposed_by,
        )]
```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

Use Anchor to find the bump and generate the derived address and change the cypher_user attribute parameters like so:

Listing 9

```
#[account(
    init,
```

```
    seeds = [B_CYPHER_USER, cypher_group.key().as_ref(), owner.key.as_ref()],
    bump,
    payer = owner,
    space = size_of::<CypherUser>() + 8 // 8 bytes for acc discriminator
)]
```

This ensures the bump value is calculated with the `Pubkey::find_program_address` function.

Remediation Plan:

ACKNOWLEDGED: The DeBridge team acknowledged this finding.

3.3 (HAL-03) USAGE OF VULNERABLE CRATES - LOW

Description:

It was observed that the project uses crates with known vulnerabilities.

Code Location:

ID	package	short description
RUSTSEC-2020-0159	chrono	potential segfault in localtimer invocations
RUSTSEC-2022-0013	regex	denial of service
RUSTSEC-2020-0071	time	potential-segfault in the time crate

Risk Level:

Likelihood - 4
Impact - 1

Recommendation:

Even if those vulnerable crates cannot impact the underlying application, it is advised to be aware of them. Also, it is necessary to set up dependency monitoring to always be alerted when a new vulnerability is disclosed in one of the project crates.

Remediation Plan:

ACKNOWLEDGED: The DeBridge team acknowledged this finding.

AUTOMATED TESTING

4.1 VULNERABILITIES AUTOMATIC DETECTION

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and vulnerabilities. Among the tools used was cargo audit, a security scanner for vulnerabilities reported to the RustSec Advisory Database. All vulnerabilities published in https://crates.io are stored in a repository named The RustSec Advisory Database. cargo audit is a human-readable version of the advisory database which performs a scanning on Cargo.lock. Security Detections are only in scope. All vulnerabilities shown here were already disclosed in the above report. However, to better assist the developers maintaining this code, the auditors are including the output with the dependencies tree, and this is included in the cargo audit output to better know the dependencies affected by unmaintained and vulnerable crates.

Results:

ID	package	short description
RUSTSEC-2020-0159	chrono	potential segfault in localtimer invocations
RUSTSEC-2022-0013	regex	denial of service
RUSTSEC-2020-0071	time	potential-segfault in the time crate

4.2 UNSAFE RUST CODE DETECTION

Description:

Halborn used automated security scanners to assist with the detection of well-known security issues and vulnerabilities. Among the tools used was cargo-geiger, a security tool that lists statistics related to the usage of unsafe Rust code in a core Rust codebase and all its dependencies.

Results:

There are too many nested crates, therefore only the crates 4 levels deep and above are shown.

de-multi-signature

```
Metric output format: x/y
    x = unsafe code used by the build
    y = total unsafe code found in the crate

Symbols:
    🔒  = No `unsafe` usage found, declares #![forbid(unsafe_code)]
    ❓  = No `unsafe` usage found, missing #![forbid(unsafe_code)]
    ☢  = `unsafe` usage found

Functions  Expressions  Impls  Traits  Methods  Dependency

0/0        0/0          0/0    0/0     0/0      ❓ de-multi-signature 0.0.0
0/0        0/0          0/0    0/0     0/0      ❓ ├── anchor-lang 0.20.1
0/0        0/0          0/0    0/0     0/0      ❓ │   ├── anchor-attribute-access-control 0.20.1
0/0        8/8          0/0    0/0     0/0      ☢ │   │   ├── anchor-syn 0.20.1
```

de-program-updater

```
Metric output format: x/y
    x = unsafe code used by the build
    y = total unsafe code found in the crate

Symbols:
    🔒  = No `unsafe` usage found, declares #![forbid(unsafe_code)]
    ❓  = No `unsafe` usage found, missing #![forbid(unsafe_code)]
    ☢  = `unsafe` usage found

Functions  Expressions  Impls  Traits  Methods  Dependency

0/0        0/0          0/0    0/0     0/0      ❓ de-program-updater 0.0.0
0/0        0/0          0/0    0/0     0/0      ❓ ├── anchor-lang 0.20.1
0/0        0/0          0/0    0/0     0/0      ❓ │   ├── anchor-attribute-access-control 0.20.1
0/0        8/8          0/0    0/0     0/0      ☢ │   │   ├── anchor-syn 0.20.1
```

THANK YOU FOR CHOOSING

HALBORN