HALBORN

DeBridge - extcall

Solana Program Security Assessment

Prepared by: **Halborn**

Date of Engagement: **September 18th, 2023 - October 13th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	10/02/2023
0.2	Draft Version	10/13/2023
0.3	Draft Review	10/13/2023
1.0	Remediation Plan	11/03/2023
1.1	Remediation Plan Review	11/05/2023

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

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

The extcall program by DeBridge allows persisting Solana instruction data in on chain accounts for later execution. Selected users called creators are tasked with creating instruction bundles and executing those bundles, for which they can be rewarded by instruction authors.

DeBridge engaged Halborn to conduct a security assessment on their Solana programs and supporting code, beginning on September 18th, 2023 and ending on October 13th, 2023 . The security assessment was scoped to the programs provided in the GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

1.2 ASSESSMENT SUMMARY

The team at Halborn was provided 4 weeks for the engagement and assigned 1 full-time security engineer to review the security of the programs in scope. The security engineer is a blockchain and Solana program security expert with advanced penetration testing and Solana program hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to identify potential security issues within the programs.

In summary, Halborn identified some security risks that were mostly addressed by the DeBridge team.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of a manual review of the source code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the program assessment. While manual testing is recommended to uncover flaws in business logic, processes, and implementation; automated testing techniques help enhance

coverage of 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 assessment:

- Research into the architecture, purpose, and use of the platform.
- Manual program source code review to identify business logic issues.
- Mapping out possible attack vectors
- Thorough assessment of safety and usage of critical Rust variables and functions in scope that could lead to arithmetic vulnerabilities.
- Finding unsafe Rust code usage (cargo-geiger)
- Scanning dependencies for known vulnerabilities (cargo audit).
- Local runtime testing (solana-test-framework)

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) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 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) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 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. Project Name

- Repository: extcall
- Release Tag: v1.1.0
- Assessment scope:
    1. crates/client
    2. craters/extcall-core
    3. derive/discriminator
    4. programs/extcall-program

Out-of-scope:
- third-party libraries and dependencies
- financial-related attacks

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	3	2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) IMMUTABLE EXTCALLMETA IF SPL TOKEN MINT IS SELECTED FOR PAYMENTS	Low (2.5)	SOLVED - 11/02/2023
(HAL-02) INCORRECT PROVIDED AMOUNT REPORTED	Low (2.5)	SOLVED - 11/02/2023
(HAL-03) IMMUTABLE PROTOCOL AUTHORITY	Low (2.5)	SOLVED - 11/02/2023
(HAL-04) DUPLICATE ENTRIES IN WHITELISTS	Informational (0.5)	SOLVED - 11/02/2023
(HAL-05) EXTCALL META ACCOUNTS CANNOT BE CLOSED	Informational (0.0)	ACKNOWLEDGED

FINDINGS & TECH DETAILS

4.1 (HAL-01) IMMUTABLE EXTCALLMETA IF SPL TOKEN MINT IS SELECTED FOR PAYMENTS - LOW (2.5)

Description:

The purpose of a ExtcallMeta account is to persist metadata of the execution of instructions stored in a corresponding ExtcallStorage account. Because instruction executors can be rewarded by the instruction creators, the ExtcallMeta account must store information on what type of token the executors will be rewarded with: native SOL or some custom SPL Token.

If an ExtcallMeta account is initialized with invalid parameters or if a creator wishes to fund the execute wallet more, the InitializeExtcallMeta can be sent to the program again.

If an SPL Token is selected, the InitializeExtcallMeta instruction handler conveniently creates the reward wallet account on behalf of the user. However, this prevents the instruction handler from completing its execution successfully when a creator wants to deposit more funds in the ExtcallMeta execution wallet because the runtime reverts the account create operation if it already exists. This renders every ExtcallMeta account with SPL Token selected effectively immutable.

Code Location:

Listing 1: extcall/programs/extcall-program/src/lib.rs (Line 557)

```
548 pub struct InitializeExtcallMeta<'info> {
549 #[account(
550     seeds = [
551         State::SEED,
552     ],
553     bump = state.bump,
554 )]
555 pub state: Account<'info, State>,
```

```
556 #[account(
557     init_if_needed,
558     seeds = [
559         EXTCALL_META_SEED,
560         extcall_id.as_slice()
561     ],
562     bump,
563     space = ExtcallMeta::SPACE,
564     payer = liquidity_source_auth,
565 )]
566 pub extcall_meta: Account<'info, ExtcallMeta>,
```

Listing 2: extcall/programs/extcall-program/src/lib.rs (Line 638)

```
637 pub struct InitializeExtcallMeta<'info> {
638     #[account(
639         seeds = [
640             State::SEED,
641         ],
642         bump = state.bump,
643     )]
644     pub state: Account<'info, State>,
645     #[account(
646         init_if_needed,
647         seeds = [
648             EXTCALL_META_SEED,
649             extcall_id.as_slice()
650         ],
651         bump,
652         space = ExtcallMeta::SPACE,
653         payer = liquidity_source_auth,
654     )]
655     pub extcall_meta: Account<'info, ExtcallMeta>,
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:C/A:N/D:C/Y:N/R:N/S:U (2.5)

Recommendation:

Consider verifying if the execute wallet is already initialized before attempting to create it.

Remediation Plan:

SOLVED: The DeBridge team solved this issue in commit ff1b331.

4.2 (HAL-02) INCORRECT PROVIDED AMOUNT REPORTED - LOW (2.5)

Description:

The purpose of a ExtcallMeta account is to persist metadata of the execution of instructions stored in a corresponding ExtcallStorage account. Because instruction executors can be rewarded by the instruction creators, the ExtcallMeta account must store information on what type of token the executors will be rewarded with: native SOL or some custom SPL Token.

If a creator wishes to fund the execute wallet more, the InitializeExtcallMeta instruction can be sent to the program again. If the native mint was selected, the program is going to debit the execute wallet the user-supplied provided_amount.

However, because the ExtcallMeta.provided_amount is reset every time the InitializeExtcallMeta instruction handler executes, the provided_amount is always equal to the amount the most recent deposit was for, erasing any trace of previous deposits.

Code Location:

Listing 3: extcall/programs/extcall-program/src/lib.rs (Lines 86,103)

```
82 pub fn initialize_extcall_meta(
83     ctx: Context<InitializeExtcallMeta>,
84     extcall_id: [u8; 32],
85     external_instructions_hash: [u8; 32],
86     provided_amount: u64,
87     provided_token: ProvidedToken,
88     fallback_authority: Pubkey,
89 ) -> Result<()> {
90     let ExtcallMetaBumps { wallet, auth } = ctx.accounts.
↳ claim_liquidity(
91         &U256::from(extcall_id),
92         &provided_token,
```

```
93          NonZeroU64::new(provided_amount).ok_or_else(|| {
94              msg!("Provided amount is zero");
95              ProgramError::InvalidArgument
96          })?,
97      )?;
98
99      *ctx.accounts.extcall_meta = ExtcallMeta {
100         extcall_id,
101         state: ExtcallMetaState::Initializing {},
102         hash: external_instructions_hash,
103         provided_amount,
104         provided_token,
105         provided_authority: fallback_authority,
106         bump: *ctx.bumps.get("extcall_meta").unwrap(),
107         execute_wallet_bump: wallet,
108         execute_auth_bump: auth,
109     };
110
111     Ok(())
112 }
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:C/A:N/D:C/Y:N/R:N/S:U (2.5)

Recommendation:

Consider accumulating the `provided_amount` instead of resetting it on every `InitializeExtcallMeta` call.

Remediation Plan:

SOLVED: The DeBridge team solved this issue in commit `ff1b331` by disabling the reinitialization of the `ExtcallMeta` accounts.

4.3 (HAL-03) IMMUTABLE PROTOCOL AUTHORITY - LOW (2.5)

Description:

The program State account is managed by a designated protocol_authority . This user is allowed to define lists of creators who can upload external instructions on chain and can execute them. The program exports an UpdateState instruction; however, it does not allow updating the protocol_authority address, which may lead to a security incident if this account is compromised.

Code Location:

Listing 4: extcall/programs/extcall-program/src/lib.rs (Line 38)

```
35 pub struct State {
36     protocol_authority: Pubkey,
37     whitelist: Vec<Pubkey>,
38     bump: u8,
39 }
```

Listing 5: extcall/programs/extcall-program/src/lib.rs

```
68 pub fn initialize_state(ctx: Context<InitializeState>, whitelist:
↳ Vec<Pubkey>) -> Result<()> {
69     *ctx.accounts.state = State {
70         protocol_authority: ctx.accounts.protocol_authority.key(),
71         whitelist,
72         bump: *ctx.bumps.get("state").unwrap(),
73     };
74     Ok(())
75 }
76
77 pub fn update_state(ctx: Context<UpdateState>, whitelist: Vec<
↳ Pubkey>) -> Result<()> {
78     ctx.accounts.state.whitelist = whitelist;
79     Ok(())
80 }
```

```
81
82 pub fn initialize_extcall_meta(
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:C/A:N/D:C/Y:N/R:N/S:U (2.5)

Recommendation:

Consider adding the option to set the protocol_authority account with the UpdateState instruction.

Remediation Plan:

SOLVED: The DeBridge team solved this finding in commit ade0371 by introducing the SetProtocolAuthority instruction.

4.4 (HAL-04) DUPLICATE ENTRIES IN WHITELISTS - INFORMATIONAL (0.5)

Description:

The extcall-program is designed to execute Solana instructions stored in on-chain accounts. The program allows selected users, called creators, to upload instruction data to ExtcallStorage accounts. Creators are whitelisted with the InitializeState and UpdateState instructions based on a vector of Pubkeys provided by the program's protocol_authority.

Neither of the instruction handlers performs deduplication of this externally supplied vector, which contributes to the program's centralization, may corrupt the State account storage.

Code Location:

Listing 6: extcall/programs/extcall-program/src/lib.rs (Lines 70,78)

```
68 pub fn initialize_state(ctx: Context<InitializeState>, whitelist:
↳ Vec<Pubkey>) -> Result<()> {
69     *ctx.accounts.state = State {
70         protocol_authority: ctx.accounts.protocol_authority.key(),
71         whitelist,
72         bump: *ctx.bumps.get("state").unwrap(),
73     };
74     Ok(())
75 }
76
77 pub fn update_state(ctx: Context<UpdateState>, whitelist: Vec<
↳ Pubkey>) -> Result<()> {
78     ctx.accounts.state.whitelist = whitelist;
79     Ok(())
80 }
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (0.5)

Recommendation:

Consider deduplicating the `whitelist` vector before updating the `State` account.

Remediation Plan:

SOLVED: The DeBridge team solved this issue in commit `c02fd32`.

4.5 (HAL-05) EXTCALL META ACCOUNTS CANNOT BE CLOSED - INFORMATIONAL (0.0)

Description:

The purpose of a ExtcallMeta account is to persist metadata of the execution of instructions stored in a corresponding ExtcallStorage account, however while it is possible to close the ExtcallStorage accounts when execution is finished, the ExtcallMeta accounts cannot be removed from storage and their creators cannot recover the rent locked.

BVSS:

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

Recommendation:

Consider adding an instruction that allows the ExtcallMeta account creators to close them.

Remediation Plan:

ACKNOWLEDGED: The DeBridge team acknowledged this finding. They commented that:

> "It is possible to do a realloc to an empty account to recover the funds, but not to release the key"

and

> “(this finding) is an additional security measure to reuse the same storage. That is why we consider realloc(0) and not close”

However, Halborn added that: “not all funds can be recovered this way. The rent-exempt threshold for an empty account will not be withdrawn.”

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.

SCENARIO	EXPECTED RESULT	EVALUATION
only one State account per program allowed	True	pass
only the protocol_authority account can update the State account	True	PASS
only one ExtcallMeta account allowed per a unique extcall ID	True	PASS
only whitelisted creators can create ExtcallMeta accounts	True	PASS
on ExtcallMeta account (re)initialization, the provided amount is transferred to the execution wallet	True	PASS
creators can always update ExtcallMeta accounts	True	FAIL - please see HAL-01
on ExtcallMeta account (re)initialization, the total provided amount is reported in the ExtcallMeta account	True	FAIL - please see HAL-02
only one ExtcallStorage account allowed per a unique extcall ID and unique owner	True	PASS
only the owner of the ExtcallStorage account can update it	True	PASS
the ExtcallStorage account can be updated only if the execution has not started yet	True	PASS
the ExtcallStorage account accurately reports the number of instruction it stores	True	PASS

only the ExtcallStorage owner can execute the stored instructions	True	PASS
only the ExtcallStorage owner can execute the stored instructions	True	PASS
the instructions in the ExtcallStorage account are executed only if the execute_wallet account has enough balance to pay all rewards	True	PASS
if any instruction in the ExtcallStorage account requires to be executed in the same block, the ExecuteExtcall instruction handler reverts unless all saved instructions were executed	True	PASS
only the owner of the ExtcallStorage account can close it	True	PASS
the ExtcallStorage accounts can be closed only if their metastate is not Execution	True	PASS
only the provided_authority can withdraw funds from ExtcallMeta accounts	True	PASS
the provided_authority accounts can withdraw funds from their ExtcallMeta accounts only if those accounts are initialing or instructions are being executed	True	PASS

5.1 AUTOMATED ANALYSIS

Description:

Halborn used automated security scanners to assist with the 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 reviewers 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-2022-0093	ed25519-dalek	Double Public Key Signing Function Oracle Attack on ‘ed25519-dalek‘
RUSTSEC-2023-0065	tungstenite	Tungstenite allows remote attackers to cause a denial of service

5.2 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:

The only crate with unsafe Rust code was debridge-externall-call.

THANK YOU FOR CHOOSING

// HALBORN