Squads v4

Security Assessment

October 19, 2023

Prepared for:
Vladimir Guguiev
Squads Protocol

Prepared by: **Samuel Moelius and Vara Prasad Bandaru**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

About Trail of Bits 1

Notices and Remarks 2

Table of Contents 3

Project Summary 4

Executive Summary 5

Project Goals 8

Project Targets 9

Project Coverage 10

Codebase Maturity Evaluation 11

Summary of Findings 13

Detailed Findings 14

1. Reliance on vulnerable dependencies 14
2. Insufficient linter use 16
3. Lack of build instructions 18
4. Functions invariant and invalidate_prior_transactions called in wrong order 20
5. Insufficient test coverage 23
6. Insufficient logging 25
7. Attacker can front-run multisig creation transaction 27
8. Program uses same set of ephemeral keys for all transactions in a batch 29
9. Inefficient lookup table account verification during transaction execution 31

A. Vulnerability Categories 34

B. Code Maturity Categories 37

C. Non-Security-Related Findings 39

D. Fix Review Results 43

Detailed Fix Review Results 44

E. Fix Review Status Categories 48

Project Summary

Contact Information

The following project managers were associated with this project:

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineering directors were associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Samuel Moelius, Consultant
samuel.moelius@trailofbits.com

Vara Prasad Bandaru, Consultant
vara.bandaru@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
September 7, 2023	Pre-project kickoff call
September 15, 2023	Status update meeting #1
September 22, 2023	Delivery of report draft
September 22, 2023	Report readout meeting
October 13, 2023	Delivery of comprehensive report
October 19 2023	Report updated to refer to revised documentation

Executive Summary

Engagement Overview

Squads Protocol engaged Trail of Bits to review the security of its Squads v4 multisig wallet for the Solana blockchain. It includes features common to multisig wallets, such as allowing members to propose and vote on transactions. It also includes less common features such as assigning permissions to individual members, and creating “spending limits” that do not need to be individually approved.

A team of two consultants conducted the review from September 11 to September 22, 2023, for a total of four engineer-weeks of effort. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

We consider the following to be some of the project’s strengths:

- The code is generally well commented. The purpose of nearly every struct and account field is documented. Remarks explaining why the code behaves as it does feature prominently throughout.

- Account size calculations are a notable example of the previous bullet. Each account size is a sum with the purpose of each summand made explicit. In separate audits, we have pointed to this practice as one to emulate.

- The code uses invariants to verify the correctness of program state changes. More specifically, each such invariant is a function containing various assertions about the program’s state. These functions are called near the end of instructions that modify the program’s state. We have not seen this practice used by Solana programs elsewhere, but we expect to recommend that others emulate this practice as well.

We consider the following as areas where the project could be improved:

- The tools `cargo-audit` and Clippy are not run regularly. The former checks a project’s dependencies against the RustSec Advisory Database. The latter catches common mistakes in Rust code. Both are considered industry standard tools and should be run on nearly every Rust project.

- The program does very little logging. Logging helps blockchain monitoring tools detect when an attack is underway. It also helps users review the program’s transaction history.

- The program is somewhat unclear on the guarantees that it provides, and the threats it protects against. There are many ways that one could use a Squads multisig to "shoot themselves in the foot." However, it is not clear which such scenarios are meant to be allowed, and which may be allowed unintentionally.

  As an example, the Squads multisig program has a built-in timelock feature. When enabled, the feature requires that a specified amount of time pass before a change to a wallet's configuration takes effect. However, the timelock is itself part of a wallet's configuration. Hence, setting the timelock to an unreasonably large value can effectively render a wallet unusable. If not explicitly prohibited, such scenarios should be documented to warn developers of their possibility.

  Addendum (2023-10-19): Squads Protocol informed us that the just-described scenario involving timelocks has been mitigated by changes to the user interface (a component not reviewed by Trail of Bits). Moreover, Squads Protocol has begun documenting ways that the Squads multisig program could be misused to help developers avoid them. Such documentation can be found at the following urls:

  - https://github.com/Squads-Protocol/v4#responsibility
  - https://docs.squads.so/main/frequently-asked-questions/key-aspects-of-using-a-multisig

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Squads Protocol take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Run cargo-audit and Clippy regularly.** These tools can find vulnerabilities in code and running them is very low cost. They should be run on nearly every Rust project.

- **Expand the program's use of logging.** Ensure that each instruction includes sufficient information to allow an off-chain monitoring tool to detect an attack. Further details are given in TOB-SQUADS-6.

- **Expand and improve the program's documentation.** Try to state precisely the guarantees that a Squads multisig wallet provides. Consider giving specific examples of scenarios, and describing how Squads multisig wallet handles them. Taking these steps will help users to understand which threats they are protected against, and which threats they are still vulnerable to.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	1
Medium	1
Low	0
Informational	6
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Auditing and Logging	1
Data Validation	1
Patching	2
Testing	2
Undefined Behavior	2

Project Goals

The engagement was scoped to provide a security assessment of the Squads multisig wallet. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can an instruction be executed without proper approval (e.g., without reaching the necessary threshold)?
- Can funds be withdrawn from a vault other than through the transaction approval process or through a spending limit?
- Can a multisig wallet be "bricked" (i.e., rendered inoperable)?
- Can one instance of a multisig wallet affect other instances?
- Can spending limits be bypassed/exceeded?
- Are permissions implemented and checked correctly?
- Is the protocol vulnerable to front-running?

Project Targets

The engagement involved a review and testing of the following target.

v4

Repository	https://github.com/Squads-Protocol/v4
Version	`e5dfcee142c16d9255c92157d092d79568179b19` `7d79e69c03892be468cce8acb51b6748b6262fbb`
Type	Rust/Anchor
Platform	Solana

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **Documentation review:** We carefully reviewed the Hyperdrive Hackathon documentation.
- **Static analysis:** We ran cargo-audit and Clippy over the codebase, and reviewed their results.
- **Test coverage review:** We verified that both the conventional Rust and the Anchor tests pass. We compared the program code to the Anchor tests to determine which parts of the instructions were tested, and which were not.
- **Manual review:** We manually reviewed of all program code, paying special attention to the following areas:
    - Proposal acceptance, rejection, and cancellation
    - Member addition and removal
    - Permissions checking
    - Spending limit use
    - Config authorities
    - Transaction execution

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- As mentioned earlier, the project would benefit from better documentation of its threat model. In particular, it is unclear which ways of “shooting oneself in the foot” are meant to be forbidden. It is possible that we dismissed some such scenarios when we should not have.
- No fuzzing was performed during this assessment. The project’s use of invariants makes it a perfect candidate for fuzzing. However, the protocol is also highly stateful. Developing fuzzing harnesses for stateful protocols tends to require significant effort. Given the short duration of the engagement, we felt it was better to focus on manual review.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	Checked arithmetic is used appropriately. Calculations of moderate complexity or higher are documented.	**Satisfactory**
Auditing	Very little logging is done. The logging that is done does not appear sufficient to allow an off-chain monitoring system to detect an ongoing attack.	**Weak**
Authentication / Access Controls	Each instruction is guarded by a validate function, which checks that necessary conditions are met before the instruction is executed. This pattern is applied consistently to each instruction, which facilitates code review. The conditions checked in each validate function are appropriate.	**Satisfactory**
Complexity Management	The code is well commented and well structured, and seems to have been written with the reader in mind. The code employs invariant functions to help ensure that instructions leave the program in only valid states. However, the project does not employ tools that are considered industry standards.	**Moderate**
Cryptography and Key Management	Cryptography and key management are handled by the underlying blockchain (Solana), and are not within the purview of this project.	**Not Applicable**
Decentralization	A Squads multisig wallet is accessible to only a select group of individuals by design. Hence, decentralization is not applicable.	**Not Applicable**
Documentation	The code is well documented internally via comments. The documentation provided for the assessment is	**Moderate**

	comprehensive, but is targeted at a specific event (a hackathon). The project's `README` lacks certain essential information (e.g., build instructions). Finally, the documentation should better explain which threats the program does and does not protect against, and clearly define which actors are involved, and their permissions.	
Front-Running Resistance	The project features some inherent and unavoidable front-running risks (e.g., proposals leak information about pending transactions). Additionally, wallet creation is vulnerable to front-running.	**Moderate**
Low-Level Manipulation	Low-level manipulation features most prominently in the small vector implementation. This part of the code is well commented and well tested.	**Satisfactory**
Testing and Verification	The project has a non-trivial set of Anchor tests, which pass. However, several important parts of the code are not covered by any test.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Reliance on vulnerable dependencies	Patching	**Undetermined**
2	Insufficient linter use	Patching	**Informational**
3	Lack of build instructions	Testing	**Informational**
4	Functions invariant and invalidate_prior_transactions called in wrong order	Data Validation	**Informational**
5	Insufficient test coverage	Testing	**Informational**
6	Insufficient logging	Auditing and Logging	**Informational**
7	Attacker can front-run multisig creation transaction	Access Controls	**High**
8	Program uses same set of ephemeral keys for all transactions in a batch	Undefined Behavior	**Medium**
9	Inefficient lookup table account verification during transaction execution	Undefined Behavior	**Informational**

Detailed Findings

1. Reliance on vulnerable dependencies	
Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Patching	Finding ID: TOB-SQUADS-1
Target: `Cargo.lock`	

Description

Although dependency scans did not uncover a direct threat to the codebase, `cargo-audit` identified dependencies with known vulnerabilities. It is important to ensure that dependencies are not malicious. Problems with Rust dependencies could have a significant effect on the system as a whole. The table below shows the output detailing the identified issues.

Package	Advisory	Cargo.lock version	Upgrade to
`ed25519-dalek`	Double public key signing function oracle attack on `ed25519-dalek`	1.0.1	≥ 2.0.0
`h2`	Resource exhaustion vulnerability in h2 may lead to denial of service (DoS)	0.3.15	≥ 0.3.17
`time`	Potential segfault in the `time` crate	0.1.45	≥ 0.2.23
`webpki`	webpki: CPU denial of service in certificate path building	0.22.0	≥ 0.22.1
`ansi_term`	`ansi_term` is unmaintained	0.12.1	N/A
`dlopen_derive`	`dlopen_derive` is unmaintained	0.2.4	N/A

`atty`	Potential unaligned read	0.2.14	N/A
`borsh`	Parsing borsh messages with ZST which are not-copy/clone is unsound	0.9.3	N/A
`tokio`	`tokio::io::ReadHalf<T>::unsplit` is Unsound	1.24.1	N/A
`crossbeam-channel`	Yanked	0.5.6	0.5.8
`ed25519`	Yanked	1.5.2	1.5.3

Table 1.1: Vulnerabilities reported by `cargo-audit`

Note that several of the vulnerabilities can be eliminated by simply updating the associated dependency.

Exploit Scenario
Mallory notices that the Squads multisig program exercises code from one of the vulnerable dependencies in table 1.1. Mallory uses the associated bug to steal funds from Alice's multisig wallet.

Recommendations
Short term, ensure that dependencies are up to date and verify their integrity after installation. As mentioned above, several of the vulnerabilities currently affecting Squads v4's dependencies can be eliminated by simply upgrading the associated dependencies. Thus, keeping dependencies up to date can help prevent many vulnerabilities.

Long term, consider integrating automated dependency auditing into the development workflow. If dependencies cannot be updated when a vulnerability is disclosed, ensure that the codebase does not use it and is not affected by the vulnerable functionality of the dependency.

2. Insufficient linter use

Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-SQUADS-2
Target: Various source files	

Description

The Squads multisig program does not appear to be linted by Clippy regularly. The Clippy linter contains hundreds of lints to help catch common mistakes and improve Rust code. Clippy should be run on most projects regularly.

Running Clippy with no flags (not even `-W clippy::pedantic`) over the Squads multisig program produces several warnings. Examples appear in figure 2.1.

```
warning: useless conversion to the same type: `std::ops::Range<u8>`
   -->
programs/squads_multisig_program/src/instructions/batch_add_transaction.rs:117:47
    |
117 |           let ephemeral_signer_bumps: Vec<u8> = (0..args.ephemeral_signers)
    |  _______________________________________________^
118 | |             .into_iter()
    | |________________________^ help: consider removing `.into_iter()`:
`(0..args.ephemeral_signers)`
    |
    = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#useless_conversion
    = note: `#[warn(clippy::useless_conversion)]` on by default

warning: this expression creates a reference which is immediately dereferenced by
the compiler
   -->
programs/squads_multisig_program/src/instructions/config_transaction_execute.rs:175:
29
    |
175 | ...                     &SEED_PREFIX,
    |                         ^^^^^^^^^^^^ help: change this to: `SEED_PREFIX`
    |
    = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#needless_borrow
    = note: `#[warn(clippy::needless_borrow)]` on by default
...
```

Figure 2.1: Sample warnings produced by running Clippy over the codebase

Exploit Scenario
Mallory uncovers a bug in the Squads multisig program. The bug might have been caught if Squads developers had enabled additional lints.

Recommendations
Short term, review all of the warnings currently generated by Clippy's default lints. Address those for which it makes sense to do so. Disable others using `allow` attributes. Taking these steps will produce cleaner code, which in turn will reduce the likelihood that the code contains bugs.

Long term, take the following steps:

- Regularly run Clippy with `-W clippy::pedantic` enabled. The pedantic lints provide additional suggestions to help improve the quality of code.

- Regularly review Clippy lints that have been allowed to see whether they should still be given such an exemption. Allowing a Clippy lint unnecessarily could cause bugs to be missed.

3. Lack of build instructions

Severity: **Informational**	Difficulty: **High**
Type: Testing	Finding ID: TOB-SQUADS-3
Target: `README.md`	

Description

The Squads v4 repository contains information for verifying the deployed program, but lacks other essential information. The repository's `README` should include at least the following:

- Instructions for building the project
- Instructions for running the built artifacts
- Instructions for running the project's tests

For example, running the project's Anchor tests is nontrivial. Figure 3.1 shows how the tests are run in CI:

```
  - name: Build Program
    run: yarn build

  - name: Replace Program keypair in target/deploy
    run: |
        echo -n "${{ secrets.MULTISIG_PROGRAM_KEYPAIR }}" >
./target/deploy/squads_multisig_program-keypair.json

  - name: Run Tests
    run: yarn test
```

Figure 3.1: How the project's Anchor tests are run in CI
(`.github/workflows/reusable-tests.yaml#75–83`)

Steps like those in figure 3.1 should be documented to help ensure that developers perform them correctly.

Exploit Scenario

Alice, a Solana developer, tries to build the Squads multisig program, but a mistake in her procedure causes it to behave incorrectly.

Recommendations

Short term, add the minimum information listed above to the repository's `README`. This will help developers to build, run, and test the project.

Long term, as the project evolves, ensure that the `README` is updated. This will help ensure that they do not communicate incorrect information to users.

4. Functions invariant and invalidate_prior_transactions called in wrong order

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SQUADS-4
Target: `programs/squads_multisig_program/src/state/{multisig.rs, multisig_config.rs}`	

Description

In several places within the Squads multisig program, the `invariant` method is called prior to the `invalidate_prior_transactions` method. However, `invalidate_prior_transactions` modifies the `stale_transaction_index` field, which `invariant` checks. Hence, it would make more sense to call `invariant` after calling `invalidate_prior_transactions`.

The definition of `invalidate_prior_transactions` and the relevant portion of `invariant` appear in figures 4.1 and 4.2, respectively. An example where this issue occurs appears in figure 4.3.

```
/// Makes the transactions created up until this moment stale.
/// Should be called whenever any multisig parameter related to the voting
consensus is changed.
pub fn invalidate_prior_transactions(&mut self) {
    self.stale_transaction_index = self.transaction_index;
}
```

Figure 4.1: Definition of `invalidate_prior_transactions` (`programs/squads_multisig_program/src/state/multisig.rs#175–179`)

```
// `state.stale_transaction_index` must be less than or equal to
`state.transaction_index`.
require!(
    stale_transaction_index <= transaction_index,
    MultisigError::InvalidStaleTransactionIndex
);
```

Figure 4.2: Relevant portion of `invariant` (`programs/squads_multisig_program/src/state/multisig.rs#166–170`)

```
    pub fn multisig_add_member(ctx: Context<Self>, args: MultisigAddMemberArgs)
-> Result<()> {
        ...
        multisig.invariant()?;

        multisig.invalidate_prior_transactions();

        Ok(())
    }
```

Figure 4.3: Example where `invariant` is called prior to `invalidate_prior_transactions` (`programs/squads_multisig_program/src/instructions/multisig_config.rs#79–113`)

This issue affects the following locations:

- `programs/squads_multisig_program/src/instructions/multisig_config.rs#L108-L110`
- `programs/squads_multisig_program/src/instructions/multisig_config.rs#L139-L141`
- `programs/squads_multisig_program/src/instructions/multisig_config.rs#L159-L161`
- `programs/squads_multisig_program/src/instructions/multisig_config.rs#L176-L178`
- `programs/squads_multisig_program/src/instructions/multisig_config.rs#L196-L198`

Note that this finding is informational since the existence of the `invariant` method exceeds industry norms.

Exploit Scenario

The `invalidate_prior_transactions` method is modified in a way that causes it to no longer satisfy the `invariant` method. The Squads multisig program is deployed with the modification. The bug might have been caught prior to deployment if the methods had been called in the reverse order.

Recommendations

Short term, reverse the order of the calls to `invariant` and `invalidate_prior_transactions` in all of the locations listed above. Doing so will ensure that `invalidate_prior_transactions`'s modifications of the `stale_transaction_index` field preserve the invariant.

Long term, as new instructions are added to the multisig program, ensure that `invariant` is always the last thing called. Adopting such a policy will help to prevent future code from introducing bugs.

5. Insufficient test coverage	
Severity: **Informational**	Difficulty: **High**
Type: Testing	Finding ID: TOB-SQUADS-5
Target: `test` subdirectory	

Description

Significant portions of the multisig program are untested. Squads Protocol should strive for near 100% test coverage to help ensure confidence in the code.

The following are some of the limitations of the program's current Anchor tests:

- The three instructions `multisig_remove_member`, `multisig_set_time_lock`, and `multisig_set_config_authority` are untested (programs/squads_multisig_program/src/lib.rs#L42-L64).
- The use of ephemeral signer seeds in batch transitions is untested (programs/squads_multisig_program/src/instructions/batch_add_transaction.rs#L119-L131).
- Of the six possible `ConfigActions`, only SetTimeLock is tested (programs/squads_multisig_program/src/instructions/config_transaction_execute.rs#L139-L275).
- The following blocks within `config_transaction_execute` are untested:
  - programs/squads_multisig_program/src/instructions/config_transaction_execute.rs#L114-L135
  - programs/squads_multisig_program/src/instructions/config_transaction_execute.rs#L279-L285
  - programs/squads_multisig_program/src/instructions/config_transaction_execute.rs#L312-L316
- The use of periods in spending limits is untested (programs/squads_multisig_program/src/instructions/spending_limit_use.rs#L156-L170).

Additional testing might have revealed TOB-SQUADS-8, for example.

Exploit Scenario

A bug is found in one of the above pieces of case. The bug could have been exposed by more thorough Anchor tests.

Recommendations
Short term, add or expand the project's Anchor tests to address each of the above noted deficiencies. Doing so will help increase confidence in the multisig program's code.

Long term, regularly review the multisig program's tests. Doing so will help ensure that the tests are relevant and that all important conditions are tested.

6. Insufficient logging

Severity: **Informational**	Difficulty: **High**
Type: Auditing and Logging	Finding ID: TOB-SQUADS-6
Target: various source files	

Description

Log messages generated during program execution aid in monitoring, baselining behavior, and detecting suspicious activity. Without log messages, users and blockchain monitoring systems cannot easily detect behavior that falls outside the baseline conditions. This may prevent malfunctioning programs or malicious attacks from being discovered.

At a minimum, each instruction should log the following information:

- The instruction's arguments
- Addresses of significant accounts involved, and the roles they played
- The instruction's effects
- Non-obvious code paths taken by the instruction

Note that program logs are displayed in tools Solana Explorer and Solana Beach. Thus, such information benefits users (i.e., humans) in addition to blockchain monitoring tools.

The Squads multisig does perform some limited logging now using the `msg!` macro. The following is a complete list of the locations where the macro is used:

- `config_transaction_create.rs:99`
- `vault_transaction_create.rs:129`
- `spending_limit_use.rs:231`
- `batch_create.rs:100`
- `batch_add_transaction.rs:142`
- `batch_add_transaction.rs:143`

Exploit Scenario

An attacker discovers a vulnerability in the Squads multisig program and exploits it. Because the actions generate no log messages, the behavior goes unnoticed until there is follow-on damage, such as financial loss.

Recommendations

Short term, add the minimal logging information listed above to each instruction. Doing so will make it easier for blockchain monitoring systems to detect problems with Squads multisig wallets. It will also make it easier for users to review past transactions.

Long term, consider using a blockchain monitoring system to track any suspicious behavior in the programs. A monitoring mechanism for critical events would quickly detect any compromised system components. Additionally, develop an incident response plan if Squads Protocol does not already have one. Doing so will help ensure that issues are dealt with promptly and without confusion.

7. Attacker can front-run multisig creation transaction

Severity: **High**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-SQUADS-7
Target: `programs/squads_multisig_program/src/instructions/multisig_create.rs`	

Description

A multisig account is derived from an unauthenticated `create_key`. An attacker can front-run a user's multisig creation transaction and create the multisig with their own parameters, allowing them to perform transactions from that multisig. The attacker can steal tokens from the multisig vaults if the user is unaware of the front-running and continues to use the multisig.

A multisig account is a PDA derived from the key of the `create_key` account (figure 7.1).

```
#[derive(Accounts)]
#[instruction(args: MultisigCreateArgs)]
pub struct MultisigCreate<'info> {
   #[account(
       init,
       payer = creator,
       space = Multisig::size(args.members.len()),
       seeds = [SEED_PREFIX, SEED_MULTISIG, create_key.key().as_ref()],
       bump
   )]
   pub multisig: Account<'info, Multisig>,

   /// A random public key that is used as a seed for the Multisig PDA.
   /// CHECK: This can be any random public key.
   pub create_key: AccountInfo<'info>,
```

Figure 7.1: Accounts struct for the `multisig_create` instruction in (`programs/squads_multisig_program/src/instructions/multisig_create.rs#20–34`)

The `create_key` account is not authenticated; any user can call the `multisig_create` instruction with any `create_key` account and initialize the corresponding multisig account.

As a result, an attacker monitoring new transactions can check for multisig creation transactions and front-run these transactions by copying the `create_key` account and then creating the multisig account themselves.

Because the attacker is creating the multisig, they can set their own values for members. The attacker could make minimal changes to the members list that go unnoticed and then perform operations on the multisig (e.g., transfer tokens from the vaults), after some activity by the original users.

Exploit Scenario

1. Alice, a user of Squads protocol, tries to create a multisig. Eve creates the multisig using Alice's `create_key` with a modified `members` list. Alice notices the modification and tries to create another multisig account. However, Eve constantly front-runs Alice's transactions and hinders her experience of using Squads protocol.

2. Bob, another user of Squads protocol, tries to create a multisig with threshold set to three and members to 10 keys. Eve creates the multisig using Bob's `create_key` with a modified `members` list containing three of her own keys. Bob does not notice the modification and his team continues to use the multisig. After some time, the tokens in the multisig vaults accumulate to one million USD. Eve uses her keys in the members list and steals the tokens.

Recommendations

Short term, check that the `create_key` account has signed the `multisig_create` instruction. This allows only the owner of the `create_key` to create the corresponding multisig.

Long term, consider the front-running risks while determining the authentication requirements for the instructions in the program. Use negative tests to ensure the program meets those requirements.

8. Program uses same set of ephemeral keys for all transactions in a batch

Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-SQUADS-8
Target: `programs/squads_multisig_program/src/instructions/{batch_add_instruction.rs, batch_execute_transaction.rs}`	

Description

The same set of ephemeral keys is used for multiple transactions of a batch. As a result, the transactions may fail to execute if they require the ephemeral keys to be unique for each of the transactions.

The ephemeral keys are temporary PDA accounts that sign the vault transactions. They are intended to be used as one-time keys or accounts by the transaction. For example, an ephemeral key could be used to create a mint account or a token account, as the key is not required after setting the proper authority. Because multiple accounts cannot be created using the same ephemeral account, this use case requires that ephemeral keys are unique for each of the transactions.

However, for a batch transaction, which may contain multiple vault transactions, the ephemeral keys are derived from the batch transaction's account key. As a result, the derived ephemeral keys will be the same for all transactions in that batch (figure 8.1).

```
/// Add a transaction to the batch.
#[access_control(ctx.accounts.validate())]
pub fn batch_add_transaction(ctx: Context<Self>, args: BatchAddTransactionArgs) -> Result<()> {
   [...]
   let ephemeral_signer_bumps: Vec<u8> = (0..args.ephemeral_signers)
       .into_iter()
       .map(|ephemeral_signer_index| {
           let ephemeral_signer_seeds = &[
               SEED_PREFIX,
               batch_key.as_ref(),
               SEED_EPHEMERAL_SIGNER,
               &ephemeral_signer_index.to_le_bytes(),
           ];

           let (_, bump) =
               Pubkey::find_program_address(ephemeral_signer_seeds, ctx.program_id);
```

```

                bump
            })
            .collect();
```

Figure 8.1: A snippet of `batch_add_transaction` instruction in (`programs/squads_multisig_program/src/instructions/batch_add_transaction.rs#106–132`)

If two of the transactions in a batch try to create a new account using the ephemeral key, then the second transaction will fail to execute, as an account would have already been initialized under the ephemeral key by the first transaction. Also, because the transactions in the batch are executed sequentially, the transactions after the failed transaction cannot be executed either. The users would have to create new transactions and go through the transaction approval process again to perform the operations.

Exploit Scenario

Bob and his team use the Squads protocol for treasury management. The team intends to deploy a new protocol. The deployment process involves successful execution of a batch transaction from the multisig, which creates new mint accounts for the program and transfers tokens from the multisig vaults to the program-owned token accounts.

Bob, unaware of the issue, creates a batch such that the first and second transactions create mint accounts using the ephemeral keys. The next five transactions transfer tokens and handle other operations required for deployment. The batch transaction is approved by the team and Bob tries to execute each transaction after the timelock of one week.

The execution of the second transaction fails and the transactions after that cannot be executed. Bob, unsure of the issue, creates regular vault transactions for the last six unexecuted transactions. The team has to approve each transaction and Bob has to execute them after the timelock. This creates a noticeable delay in the deployment of Bob's protocol.

Recommendations

Short term, derive the ephemeral keys using the key of the account used to store the individual transactions in the batch. Doing so will result in unique ephemeral keys for each transaction.

Long term, as recommended in TOB-SQUADS-5, expand the project's tests to ensure that the program is behaving as expected for all intended use cases.

9. Inefficient lookup table account verification during transaction execution

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-SQUADS-9
Target: `programs/squads_multisig_program/src/utils/executable_transaction_message.rs`	

Description

The program does not strictly verify that lookup table accounts given at the time of execution exactly match with the accounts given at the time of transaction initialization. The current implementation performs unnecessary on-chain computation and wastes gas.

The Squads protocol allows the usage of the address lookup tables for vault transactions. Each transaction account stores the list of address lookup table accounts and indices of the accounts that will be used by the transaction. The addresses stored in the lookup table account are expected to not change after the initialization of the transaction. This guarantee is provided by checking that the accounts are owned by the `solana_address_lookup_table` program.

The `vault_transaction_execute` instruction executes the approved transactions. The instruction takes a list of address lookup table accounts that are verified to be the same as the accounts that are given at the time of initialization in `ExecutableTransactionMessage::new_validated` (figure 9.1).

```
    /// `address_lookup_table_account_infos` - AccountInfo's that are expected to correspond to the lookup tables mentioned in `message.address_table_lookups`.
    /// `vault_pubkey` - The vault PDA that is expected to sign the message.
    pub fn new_validated(
        message: &'a VaultTransactionMessage,
        address_lookup_table_account_infos: &'a [AccountInfo<'info>],
        [...]
    ) -> Result<Self> {
        // CHECK: `address_lookup_table_account_infos` must be valid `AddressLookupTable`s
        //         and be the ones mentioned in `message.address_table_lookups`.
        require_eq!(
            address_lookup_table_account_infos.len(),
            message.address_table_lookups.len(),
            MultisigError::InvalidNumberOfAccounts
        );
        let lookup_tables: HashMap<&Pubkey, &AccountInfo> =
address_lookup_table_account_infos
        .iter()
        .map(|maybe_lookup_table| {
            // The lookup table account must be owned by SolanaAddressLookupTableProgram.
            require!(
                maybe_lookup_table.owner == &solana_address_lookup_table_program::id(),
                MultisigError::InvalidAccount
            );
            // The lookup table must be mentioned in `message.address_table_lookups`.
            require!(
                message
                    .address_table_lookups
                    .iter()
                    .any(|lookup| &lookup.account_key == maybe_lookup_table.key),
                MultisigError::InvalidAccount
            );
            Ok((maybe_lookup_table.key, maybe_lookup_table))
        })
        .collect::<Result<HashMap<&Pubkey, &AccountInfo>>>()?;
```

Figure 9.1: A snippet of `ExecutableTransactionMessage::new_validated` function showing the validations performed on lookup table accounts (`programs/squads_multisig_program/src/utils/executable_transaction_message.rs#28–62`)

The `new_validated` function iterates over the `address_lookup_table_account_infos` and checks that each of the accounts is owned by the `solana_address_lookup_table` program and that the account is present in `message.address_table_lookups`. The function does not strictly validate that both lists match. Requiring that `message.address_table_lookups` and `address_lookup_table_account_infos` list the address lookup tables in the same order would result in a more efficient implementation.

Exploit Scenario

Alice, a Solana developer, writes code that uses a Squads multisig wallet's accounts lookup table feature. Alice is unnecessarily charged for gas. The unnecessary charges would be eliminated by a more efficient implementation of the accounts lookup table feature.

Recommendations

Short term, rewrite the code to zip `message.address_table_lookups` together with `address_lookup_table_account_infos`, and go through each of the pairs to verify that they match. Doing so will help protect against incomplete checks and will also make the implementation more efficient.

Long term, as recommended in TOB-SQUADS-5, expand the project's tests to ensure that the program is behaving as expected for all intended use cases.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels

Severity	Description
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels

Difficulty	Description
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	Description
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Non-Security-Related Findings

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Fix the links in README.md.** The LICENSE and Squads-Protocol urls are invalid.

```
 16    The primary license for Squads Multisig Program V4 is the Business Source
License 1.1 (`BUSL-1.1`), see [LICENSE](./LICENSE). The following exceptions are not
licensed under the BULS-1.1, but are licensed separately as follows:
 17
 18    - The file
<https://github.com/Squads-Protocol/v4/blob/main/programs/multisig/src/utils/system.
rs> is derived from code released under the [Apache 2.0
license](https://github.com/coral-xyz/anchor/blob/master/LICENSE) at
<https://github.com/coral-xyz/anchor/blob/714d5248636493a3d1db1481f16052836ee59e94/l
ang/syn/src/codegen/accounts/constraints.rs#L1126-L1179>.
 19    - The file
<https://github.com/Squads-Protocol/v4/blob/main/programs/multisig/src/utils/small_v
ec.rs> is derived from code released under both the [Apache 2.0
license](https://github.com/near/borsh-rs/blob/master/LICENSE-APACHE) and the [MIT
license](https://github.com/near/borsh-rs/blob/master/LICENSE-MIT) at
<https://github.com/near/borsh-rs/blob/master/borsh/src/de/hint.rs> and
<https://github.com/near/borsh-rs/blob/master/borsh/src/ser/mod.rs>.
```

Figure C.1: Broken links in README.md (README.md#16–19)

- **Rewrite the code in small_vec.rs to not use nested question marks.** The code in figure C.2 would be both more clear and more concise if written as in figure C.3. The code in figure C.4 could similarly be rewritten.

```
 33    writer.write_all(
 34       &(u8::try_from(self.len()).map_err(|_| std::io::ErrorKind::InvalidInput)?)
 35           .to_le_bytes(),
 36    )?;
```

Figure C.2: Code from small_vec.rs that could be rewritten to not use nested question marks (programs/squads_multisig_program/src/utils/small_vec.rs#33–36)

```
 33    let len = u8::try_from(self.len()).map_err(|_|
std::io::ErrorKind::InvalidInput)?;
 34    writer.write_all(&len.to_le_bytes())?;
```

Figure C.3: The code from figure C.2 rewritten to not use nested question marks

```
 46    writer.write_all(
 47       &(u16::try_from(self.len()).map_err(|_|
std::io::ErrorKind::InvalidInput)?)
            .to_le_bytes(),
    )?;
```

Figure C.4: Additional code from `small_vec.rs` that could be rewritten to not use nested question marks
(`programs/squads_multisig_program/src/utils/small_vec.rs#46–49`)

- **Add a check to verify the length of preBalances in `batch-sol-transfer.ts`.** In particular, add a line like the highlighted one in figure C.5. Doing so will help improve the test's robustness.

```
// Fetch the member balances before the batch execution.
const preBalances = [] as number[];
for (const member of Object.values(members)) {
 const balance = await connection.getBalance(member.publicKey);
 preBalances.push(balance);
}
assert.strictEqual(Object.values(members).length, preBalances.length);
```

Figure C.5: Loop from `batch-sol-transfer.ts`. The highlighted code does not appear in the original, but adding it would make the test more robust.
(`tests/suites/examples/batch-sol-transfer.ts#226–231`)

- **Calculate month periods accurately.** The code currently defines a month to be 30 days (figure C.6). When a spending limit is reset, its `last_reset` field is set to a multiple of this period (figure C.7). For the special case of a month, the code could instead calculate the offset into the month when the spending limit was created, and set the `last_reset` field to that offset within the current month. Doing so would eliminate the drift that currently results from using month periods, and in turn reduce the likelihood of accounting errors.

```
pub fn to_seconds(&self) -> Option<i64> {
   match self {
       Period::OneTime => None,
       Period::Day => Some(24 * 60 * 60),
       Period::Week => Some(7 * 24 * 60 * 60),
       Period::Month => Some(30 * 24 * 60 * 60),
   }
}
```

Figure C.6: Definition of a month period in seconds
(`programs/squads_multisig_program/src/state/spending_limit.rs#93–100`)

```
// last_reset = last_reset + periods_passed * reset_period,
spending_limit.last_reset = spending_limit
```

```
        .last_reset
        .checked_add(periods_passed.checked_mul(reset_period).unwrap())
        .unwrap();
```

Figure C.7: Setting of the `last_reset` field by a `spending_limit_use` instruction (`programs/squads_multisig_program/src/instructions/spending_limit_use.rs#164–168`)

- **Either document that the `Permissions::from_vec` function (figure C.8) is unused, or remove it.** Unused functions risk becoming out of sync with other parts of the code. `Permissions::from_vec` should either be documented as unused to help mitigate this risk, or removed to eliminate the risk.

```
    pub fn from_vec(permissions: &[Permission]) -> Self {
        let mut mask = 0;
        for permission in permissions {
            mask |= *permission as u8;
        }
        Self { mask }
    }
```

Figure C.8: `Permissions::from_vec` function (`programs/squads_multisig_program/src/state/multisig.rs#251–257`)

- **Ensure that a member can participate in proposal voting and transaction voting by validating the member's key before adding it to the list.** The program works with the assumption that each of the members is a valid key and can sign transactions. However, there are no validations to ensure this fact. The invariant method could be updated to check for default public keys, and the instructions which add new members could be updated to perform signer checks on the member accounts wherever possible.

- **Rewrite the code in figure C.11 to access the `key()` of the account without the call to `.to_account_info()`.** Removing the call to `to_account_info` will be more efficient as the `to_account_info` clones the account info which is not necessary to access the `key`.

```
    multisig.create_key = ctx.accounts.create_key.to_account_info().key();
```

Figure C.9: Code from `multisig_create.rs` that could be updated to be more efficient (`programs/squads_multisig_program/src/instructions/multisig_create.rs#62`)

- **Add documentation listing all intended privileges of the `config_authority`.** Proper user documentation about the privileges of the `config_authority` will allow the users to make informed decisions and will protect them from working under inaccurate assumptions about the operations that could be performed by `config_authority`.

- **Consider imposing an upper bound on timelocks.** If multisig members accidentally approve an unreasonably large timelock, they will have to wait for it to pass before the timelock can be set to a more reasonable value.

D. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On September 27, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Squads Protocol team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the nine issues described in this report, Squads Protocol has resolved four issues, has partially resolved two issues, and has not resolved the remaining three issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Reliance on vulnerable dependencies	Unresolved
2	Insufficient linter use	Resolved
3	Lack of build instructions	Partially Resolved
4	Functions invariant and invalidate_prior_transactions called in wrong order	Resolved
5	Insufficient test coverage	Partially Resolved
6	Insufficient logging	Unresolved
7	Attacker can front-run multisig creation transaction	Resolved
8	Program uses same set of ephemeral keys for all transactions in a batch	Unresolved
9	Inefficient lookup table account verification during transaction execution	Resolved

Detailed Fix Review Results

TOB-SQUADS-1: Reliance on vulnerable dependencies

Unesolved. The client provided the following context for this finding's fix status (emphasis added in bold):

> *The program and SDK reference a number of crates in the program's repository. These dependencies are tightly coupled with the underlying protocols crates, see: https://docs.rs/solana-sdk/1.16.14/solana_sdk/ (for ed25519-dalek^1.0.1 dependency). After investigating, the culprits are the following:*
>
> > *solana-sdk*
> > *solana-client*
> > *solana-program*
> > *spl-associated-token-account*
>
> *These crates are viewed by the team as systemic dependencies across the Solana blockchain ecosystem.* ***Compiling the program to SBF/BPF with dependencies that fall outside of the core Solana crates may create more unintended consequences for the program that will be difficult to handle due to the divergence and misalignment.*** *The team will reach out to the Solana Foundation to encourage updating the crates dependencies and make an effort to pin any crates that will allow bytecode compilation without issue. Additionally the team will implement the practice of running cargo-audit regularly.*

We disagree with the emphasized conclusion. Table 1.1 of TOB-SQUADS-1 listed four errors (red) and seven warnings (yellow). Simply running `cargo update` would eliminate three of the errors (h2, `time`, and `webpki`) and three of the warnings (`tokio`, `crossbeam-channel`, and `ed25519`).

Moreover, the vulnerable dependencies affect the project's `Cargo.lock` file, not the Solana ecosystem as a whole. Put another way, if the Squads multisig project were newly created as of today, it would have a `Cargo.lock` file that does not reference the vulnerable h2, `time`, `webpki`, `tokio`, `crossbeam-channel`, and `ed25519` dependencies.

Hence, we continue to recommend that Squads Protocol eliminate reliance on the dependencies in table 1.1 where feasible (e.g., by running `cargo update`).

TOB-SQUADS-2: Insufficient linter use

Resolved in commit `e7f48dc`. Clippy now produces no warnings when run on the Squads multisig program. However, the client writes (emphasis added in bold):

> *As recommended, cargo clippy has been run to clean up the codebase, and also has been added to the CI pipeline via a github workflow action to ensure* ***cargo clippy runs on all builds that are submitted via Pull Request****.*

While we can see Clippy being installed CI (figure D.1), we can find no evidence of it actually being run in the current head of the main branch (eabda3b). That is, the bolded statement appears inaccurate. We recommend that Squads Protocol investigate this issue to ensure that the stated policy is enacted.

```
name: Install Rust toolchain
uses: actions-rs/toolchain@v1
with:
 profile: minimal
 toolchain: stable
 components: rustfmt, clippy
```

Figure D.1: Installation of Clippy in CI
(`.github/actions/setup-rust/action.yaml#14–19`)

TOB-SQUADS-3: Lack of build instructions
Partially resolved in the current head of the main branch (eabda3b). The project's `README` now contains the minimal information listed in TOB-SQUADS-3. However, the provided instructions for running the tests do not work outright. In particular, the instructions do not contain steps similar to those of figure 3.1, though such steps still appear to be necessary.

TOB-SQUADS-4: Functions invariant and invalidate_prior_transactions called in wrong order
Resolved in commit b133644. The calls were reordered as recommended.

TOB-SQUADS-5: Insufficient test coverage
Partially resolved in the current head of the branch `add-missing-sdk-tests` (f7b38d0). That commit contains several tests not present in the commit reviewed during the assessment (7d79e69). However, not all of the newly added tests pass.

TOB-SQUADS-6: Insufficient logging
Unresolved. The client provided the following context for this finding's fix status:

> *While important, the team will communicate the risks to users and encourage them to use tools like Sec3's Watchtower, in addition to adding monitoring to the Squads team's own infrastructure. The team currently maintains a robust server architecture for indexing and monitoring analytics for all mutisigs and transactions used by the program, and will take the necessary steps to incorporate alerts for any unusual or erratic behavior. Companies like Dune and Top Ledger are currently running detailed analytics across Squads multisig programs.*

TOB-SQUADS-7: Attacker can front-run multisig creation transaction
Resolved in commit b133644. A `create_key` account passed to a `multisig_create` instruction is now required to be a signer. Moreover, Squads Protocol added a test (figure D.2) to verify that the change works as intended.

```
  it("error: missing signature from `createKey`", async () => {
   const creator = await generateFundedKeypair(connection);
   ...
   // Missing signature from `createKey`.
   tx.sign([creator]);

   await assert.rejects(
     () => connection.sendTransaction(tx, { skipPreflight: true }),
     /Transaction signature verification failure/
   );
  });
```

Figure D.2: Verification of the `create_key` signature requirement (`tests/suites/multisig-sdk.ts#69–104`)

TOB-SQUADS-8: Program uses same set of ephemeral keys for all transactions in a batch
Unresolved. The client provided the following context for this finding's fix status:

> *While it may create an irritating experience for a user if an ephemeral key is used in a batch transaction, the Squads team will provide clear instructions and clarity around the use of the batch transactions, and how developers may want to use a standard transaction to accomplish the same goals. The client application provides numerous features to easily perform advanced transaction instructions with various program templates, and we believe this can be mitigated through education, while encouraging users to only use the batch transaction for more straightforward instruction handling. Basic vault transactions can include multiple instructions, and the Batch Transaction functionality is considered a "luxury" as it allows multiple transactions to be approved under one multisig transaction (For example, send 100 assets with one transaction to approve, rather than approving 100 individual transactions).*

TOB-SQUADS-9: Inefficient lookup table account verification during transaction execution
Resolved in commit b133644. (Note that, as of this writing, the commit has not been merged into the main branch.) The client rewrote the code so as to eliminate the use of `iter` on line 56 of Figure 9.1.

However, we encourage Squads Protocol to continue to look for ways to improve the `new_validated` function's efficiency. In particular, we suspect that its use of a `HashMap` could be eliminated. More specifically, we think it should be possible to obtain an address lookup table's `AccountInfo` by indexing into the `address_lookup_table_account_infos` variable, without using its public key to index into a `HashMap`. We encourage Squad's Protocol to consider this idea.

E. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.