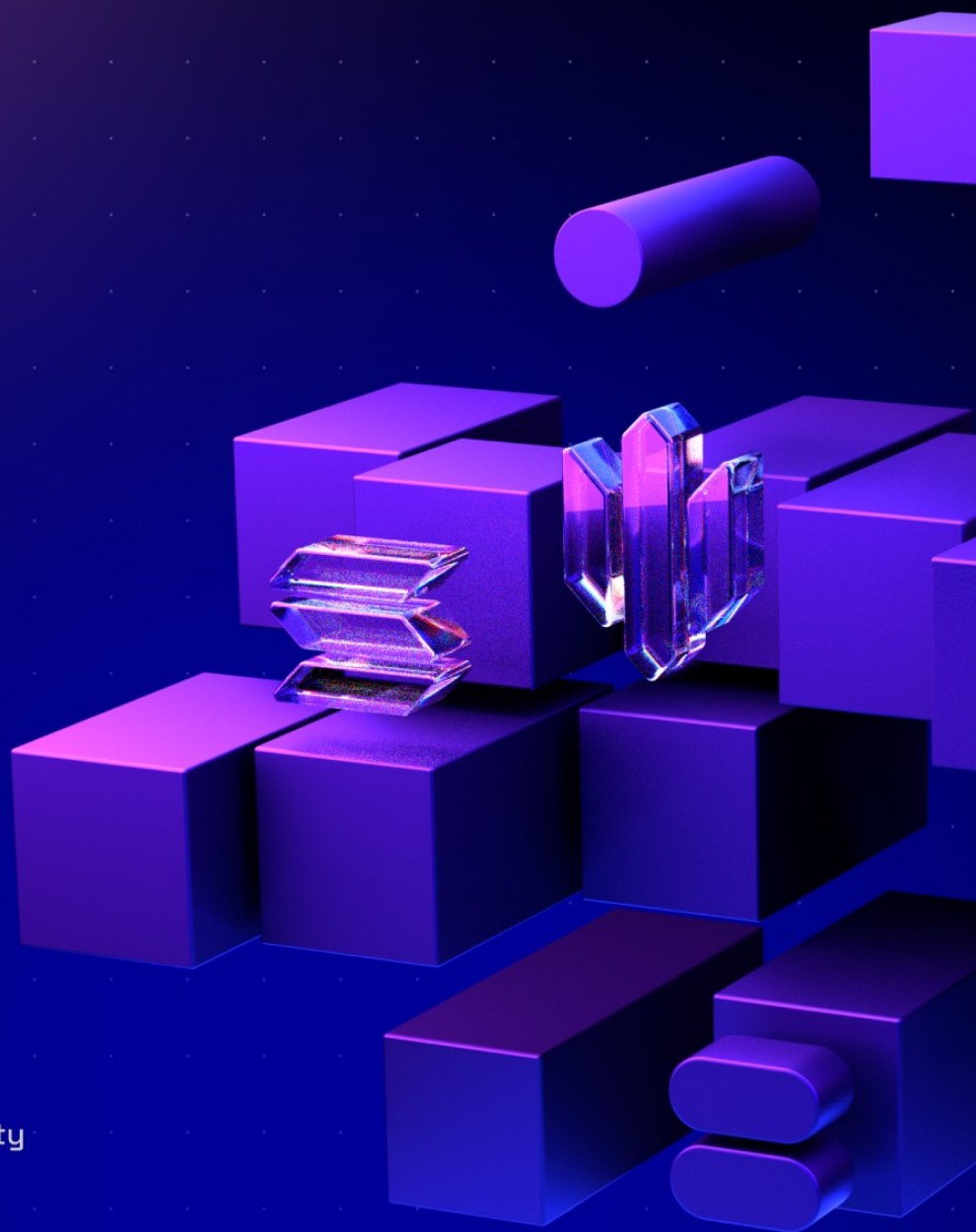
ackee
blockchain security

Wormhole

Worldcoin World ID State Root Bridge

2.10.2024



Ackee Blockchain Security

Contents

1. Document Revisions

1.0-draft	Draft Report	27.09.2024
1.0	Final Report	01.10.2024
1.1	Fix Review	02.10.2024

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain Security

Ackee Blockchain Security is an in-house team of security researchers performing security audits focusing on manual code reviews with extensive fuzz testing for Ethereum and Solana. Ackee is trusted by top-tier organizations in web3, securing protocols including Lido, Safe, and Axelar.

We develop open-source security and developer tooling Wake for Ethereum and Trident for Solana, supported by grants from Coinbase and the Solana Foundation. Wake and Trident help auditors in the manual review process to discover hardly recognizable edge-case vulnerabilities.

Our team teaches about blockchain security at the Czech Technical University in Prague, led by our co-founder and CEO, Josef Gattermayer, Ph.D. As the official educational partners of the Solana Foundation, we run the School of Solana and the Solana Auditors Bootcamp.

Ackee's mission is to build a stronger blockchain community by sharing our knowledge.

Ackee Blockchain a.s.
Rohanske nabrezi 717/4
186 00 Prague, Czech Republic
https://ackee.xyz
hello@ackee.xyz

2.2. Audit Methodology

The Ackee Blockchain Security auditing process follows a routine series of steps:

1. **Code review**
    a. High-level review of the specifications, sources, and instructions provided to us to make sure we understand the project's size, scope, and functionality.
    b. Detailed manual code review, which is the process of reading the source code line-by-line to identify potential vulnerabilities. We focus mainly on common classes of Solana program vulnerabilities, such as:

       missing ownership checks, missing signer authorization, signed CPI of unverified programs, cosplay of Solana accounts, missing rent exemption assertion, bump seed canonicalization, incorrect accounts closing, casting truncation, numerical precision errors, arithmetic overflows or underflows.

    c. Comparison of the code and given specifications, ensuring that the program logic correctly implements everything intended.
    d. Review of best practices to improve efficiency, clarity, and maintainability.
2. **Testing and automated analysis**
    a. Run client's tests to ensure that the system works as expected, potentially write missing unit or fuzzy tests using our testing framework Trident.
3. **Local deployment + hacking**
    a. The programs are deployed locally, and we try to attack the system and break it. There is no specific strategy here, and each project's attack attempts are unique to its implementation.

2.3. Finding Classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in *configuration* (system settings or parameters, such as deployment scripts, compiler configurations, using multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**N/A**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or *configuration*, but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or *configuration* was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review Team

The following table lists all contributors to this report. For authors of the specific revision, see the “Revision team” section in the respective “Report revision” chapter.

Member’s Name	Position
Andrej Lukačovič	Lead Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Worldcoin World ID State Root Bridge is a protocol that enables the bridging of the Worldcoin World ID state root from Ethereum to Solana. The Worldcoin World ID utilizes Semaphore, with a single set containing public keys (or identity commitments) for each verified user. A commitment to this set is then replicated to other blockchains, in this case to Solana, where the new Merkle root is stored, allowing verified users to prove their personhood.

Revision 1.0

Wormhole engaged Ackee Blockchain Security to perform a security review of the Worldcoin World ID State Root Bridge protocol with a total time donation of 11 engineering days in a period between September 6 and September 27, 2024, with Andrej Lukačovič as the lead auditor.

The audit was performed on the commit `70f034`[1] and `a6f479`[2] respectively the scope was the following:

- Solana World ID Program, excluding external dependencies
- Solana World ID On-Chain Template, excluding external dependencies.

Bridging Worldcoin's World ID from Ethereum to Solana relies heavily on off-chain components known as Guardians. Guardians sign a query from the Ethereum blockchain that contains the state root of the Worldcoin World ID. This signed query is then submitted to the Solana blockchain to store the new state root. These off-chain components were not in the scope of the audit.

The audit began by understanding the high-level goals of the project, followed by a deep dive into the program's logic. In the initial phase, we implemented fuzz tests, which were particularly helpful for two reasons: to enhance our understanding of the project's core concepts and to begin

fuzzing as early as possible, increasing the likelihood of identifying bugs. For fuzzing, we used Trident. See the Pull Request with a complete fuzz test code.

In the later stages of the audit, we shifted focus to a manual review of the project, paying special attention to the following:

- ensuring there is no frontrunning possible during the initialization process (e.g. Config Initialization);
- ensuring all Config-related instructions can only be executed by the associated authority;
- ensuring correct deserialization and serialization of instruction inputs;
- ensuring that no reinitialization or denial of service is possible during the posting of Guardian Signatures to the Solana blockchain;
- ensuring there is no possibility of spoofing the Guardian Signatures and Guardian Set accounts when posting a new state root from the Ethereum blockchain;
- ensuring all posted Guardian Signatures are verified in a sequential order and that no signature can be posted multiple times to achieve quorum artificially;
- ensuring the Proof of Personhood verification process is correctly implemented and that no sensitive data leaks are present.

Our review resulted in 5 findings, ranging from Info to High severity.

The most severe finding, H1, presents the potential for a denial of service during the verification process of proof of personhood. If there are relatively large gaps (compared to the `root_expiry`) between newly submitted root hashes from Ethereum, two undesirable scenarios can arise.

Ackee Blockchain Security recommends Wormhole:

- ensure that there is always at least one active root available for verification, and prevent possibility of all roots being removed;
- ensure that the off-chain components are functioning correctly and adhere to best security practices;
- ensure that the Guardian Sets accounts stored on-chain are well protected, and there is no possibility for an attacker to tamper with these accounts;
- avoid using unchecked arithmetic. Although the likelihood of exploiting unchecked arithmetic in unintended ways may be low, potential issues still exist and could lead to catastrophic outcomes.

See Report Revision 1.0 for the system overview and trust model.

Revision 1.1

The fix review was completed on commit `152df3`[3]. The fixes were made in multiple smaller commits. Thus, this commit hash refers to the latest commit after the changes.

[1] full commit hash: `70f0346c3bea804a6131903d208f3df47ca7d8ee`

[2] full commit hash: `a6f4799f493ccfa67f6a9b6b54618beb7a0975cd`

[3] full commit hash: `152df36ad58047fd910a77d44cbd07f25d0b8a12`

4. Findings Summary

The following section summarizes findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- *Description*
- *Exploit scenario* (if severity is low or higher)
- *Recommendation*
- *Fix* (if applicable).

Summary of findings:

Critical	High	Medium	Low	Warning	Info	Total
0	1	2	0	1	1	5

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
H1: The latest available root may be inactive and potentially undesirably removed	High	1.0	Fixed
M1: Possible arithemtic overflow during root `is_active` check	Medium	1.0	Fixed
M2: Missing `mut` constraint	Medium	1.0	Fixed
W1: Possible Reinitialization	Warning	1.0	Fixed

Finding title	Severity	Reported	Status
I1: Signature Malleability due to accepting S values with high and also low order	Info	1.0	Acknowledged

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member's Name	Position
Andrej Lukačovič	Lead Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

Worldcoin World ID State Root Bridge is a Solana program written using the Anchor Framework, dedicated to bridging and storing the Worldcoin World ID state root from Ethereum to Solana.

The off-chain components included in the bridging process are the Guardians and the State Bridge Service as shown in Figure. The main goal of these off-chain components is to obtain the state root from the Ethereum World ID Identity Manager, have it signed by the Guardians, and post the signatures along with the query message to the Solana blockchain. The Solana World ID Protocol then verifies that the Guardians signed the query and ensures that the number of signatures satisfies the quorum. At the time of writing, there are 19 independent Wormhole Guardians available (see Dashboard), and the quorum required by the Solana World ID Protocol is at least 13.

Subsequently, users who have already been verified with their corresponding Proof of Personhood can use the derived credentials to prove that they are human by referencing the stored state root on the Solana blockchain. For this purpose, the Solana World ID on-chain template program was created as an example of how such a verification workflow can be implemented using Cross Program Invocation to the Solana World ID Program.

For a better understanding of the Worldcoin World ID protocol, from the

verification process of personhood with the Orb to the application of Zero Knowledge proofs, we highly recommend reading the full Worldcoin World ID Whitepaper A New Identity and Financial Network.

Trust Model

The availability of data (i.e. the latest state root) on Solana is in the hands of the off-chain components. As mentioned earlier, a quorum must be achieved for the new state root to be stored on Solana. There is no possible attack scenario where an attacker could spoof signatures and post a falsified state root to Solana without first reaching the required quorum of Guardian signatures.

However, hypothetically, if an attacker were to simultaneously take control of all the Guardians, they could generate spoofed messages mimicking the Ethereum World ID Identity Manager and submit those messages to the Solana blockchain, potentially leading to catastrophic scenarios. Nonetheless, the Guardians network appears to be sufficiently decentralized, reducing the likelihood of such an attack.

Lastly, if the quorum is not achieved, no new state root can be submitted to the Solana blockchain. In a hypothetical scenario where multiple Guardians experience an outage, there wouldn't be enough Guardians functioning to achieve the quorum, resulting in a Denial of Service. However, the Guardians network seems to be adequately decentralized, mitigating the risk of this issue.

Fuzzing

During the audit, manually-guided fuzz tests were developed to assess the protocol's correctness, security, and robustness. Fuzz test templates are generated from the IDL created by Anchor and then implemented based on user needs. Notably, Trident allows the specification of flows or invariant

checks.

Flows are important for helping the fuzzer better cover valid instruction sequences. On the other hand, invariant checks allow for the detection of undesired changes made during instruction execution. When an instruction is successfully invoked, the user can specify multiple invariant checks to ensure that the contents of the accounts were updated as expected during the execution.

Over the period of fuzzing, two common types of failures can occur, a panic during instruction execution or a failure of the specified invariant check. The former can happen, for instance, when an unchecked arithmetic overflow is detected, while the latter is triggered by behavior that is defined as undesired (e.g. undesired balance change).

Multiple flows were implemented in the fuzz tests, as detailed in Appendix B. All required inputs, which would have been difficult to generate randomly, were obtained from the typescript tests (specifically signatures, query messages, and credentials submitted by the user to prove their personhood).

The Solana World ID on-chain template was also included in the fuzzing process, as Trident allows for the specification of flows across multiple programs. This allowed the simulation to closely mimic real-world scenarios.

Findings

The following section presents the list of findings discovered in this revision.

H1: The latest available root may be inactive and potentially undesirably removed

High severity issue

Impact:	High	Likelihood:	Medium
Target:	verify_groth16_proof.rs, verify_and_execute.rs, clean_up_root.rs	Type:	Denial of service

Description

`CleanUpRoot` allows anyone to close a root account that is no longer active. Although the instruction verifies that the root is inactive, it heavily depends on the periodic update of root accounts. In a scenario where the `root_expiry` is set too low and the new state root has not yet been submitted to the Solana blockchain:

- users would be unable to verify their proofs since the latest available root is inactive;
- it becomes possible to remove the most up-to-date state root.

This issue can be partially mitigated by updating the `root_expiry` in the Config account. However, if this update is not performed quickly enough, an attacker could invoke `CleanUpRoot` on the `refund_recipient's` behalf and close the latest available state root.

This behavior might be supported by the batching of commitments, as outlined in the Worldcoin Whitepaper:

> *A batcher monitors the work queue. When*
>
> *1) a sufficiently large number of commitments are queued or*

> *2) the oldest commitment has been queued for too long, the batcher will take a batch of keys from the queue to process.*
>
> — Worldcoin Whitepaper

This means that new commitments are not necessarily propagated on-chain immediately, which could result in less frequent root updates, increasing the likelihood of this scenario.

Exploit scenario

If the `root_expiry` in the Config is set too low relative to the frequency at which new roots are submitted, there is a risk that `CleanUpRoot` could be called on the latest available root. Once a root is no longer active, users will also be unable to verify their proofs.

When there are no new commitments on the Ethereum blockchain, no new root hashes will be submitted to the Solana blockchain, meaning in order for users to verify their proofs the `root_expiry` needs to be manually increased. However if the new `root_expiry` is not updated fast enough anyone can call `CleanUpRoot` instruction and close the latest available root. As a result, the closed root account will need to be reinitialized, or the chain will have to wait for a new root hash to be submitted.

Recommendation

Our recommendation is to:

- allow users to verify the proof with the latest available root (i.e. the one stored in LatestRoot) without checking for active status;
- ensure that the root being removed in the `CleanUpRoot` instruction is not the one currently stored in the LatestRoot account.

Fix 1.1

This issue has been fixed in accordance with the recommendation.

Go back to Findings Summary

M1: Possible arithemtic overflow during root `is_active` check

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	clean_up_root.rs, verify_groth16_proof.rs, verify_and_execute.rs	Type:	Arithmetics

Description

Unchecked arithmetic within the `is_active` function implemented for root can lead to a denial of service. As specified in the root `Cargo.toml` file, the `overflow-checks` option is set to `true`, meaning a panic will occur if an overflow is detected.

Listing 1. Excerpt from Cargo file

```
[profile.release]
overflow-checks = true
```

As shown in the snippet below, the addition operation is prone to overflow. The `read_block_time` is fetched from Ethereum, while the `config_root_expiry` is set by the Config authority to any arbitrary value. This creates the potential for undesired behavior, where a panic in this function would result in a denial of service for the `CleanUpRoot` and `VerifyGroth16Proof` instructions.

Listing 2. Excerpt from Root Implementation

```
pub fn is_active(&self, timestamp: &u64, config_root_expiry: &u64) -> bool {
    let read_block_time_in_secs = self.read_block_time / 1_000_000;
    let expiry_time = read_block_time_in_secs + config_root_expiry;
    expiry_time >= *timestamp
}
```

Exploit scenario

The likelihood of this exploit is considered low. This is because an attacker cannot directly update the root with malicious values to trigger a denial of service, as the required signatures from the Guardians serve as a safeguard, and the attacker has no access to modify the `config_root_expiry`. However, if the `config_root_expiry` is mistakenly set too high, it could still result in the previously mentioned denial of service.

Recommendation

Fix the unchecked arithmetic and ensure that the operation cannot overflow in any scenario. One possible implementation of the fix could be as follows:

Listing 3. Excerpt from Root Implementation

```
pub fn is_active(&self, timestamp: &u64, config_root_expiry: &u64) -> bool {
    let read_block_time_in_secs = self.read_block_time / 1_000_000;
    let result = read_block_time_in_secs.checked_add(*config_root_expiry);
    let expiry_time = match result {
        Some(result) => result,
        None => u64::MAX,
    };
    expiry_time >= *timestamp
}
```

Fix 1.1

This issue has been fixed using an approach similar to the recommendations, which is also valid. The arithmetic overflow is resolved by using the `saturating_add` function, which will result in `u64::MAX` in case an overflow occurs.

Go back to Findings Summary

M2: Missing `mut` constraint

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	clean_up_root.rs, close_signatures.rs, update_root_with_query.rs	Type:	Access control

Description

In several instances within the source code, the `mut` constraint is missing for accounts that will receive funds after another account is closed. This constraint is crucial because it specifies that the account must be writable (i.e. capable of having lamports added to its balance). The account's mutability is reflected in the program's IDL, where the corresponding account is marked as writable. If the mutability of an account is not explicitly set, it can lead to transaction failures due to attempting to modify the balance of a read-only account. Particularly these Instructions are missing the constraint.

The `UpdateRootWithQuery` instruction is missing the `mut` constraint for the `refund_recipient`. If the `payer` and `refund_recipient` refer to the same account, the issue may go unnoticed since the `payer` is marked as mutable. However, if these two fields correspond to different accounts, the instruction cannot be processed successfully, even with valid signatures, because the balance of the `refund_recipient` cannot be updated without it being writable.

Listing 4. Excerpt from UpdateRootWithQuery

```
// ...
/// CHECK: This account is the refund recipient for the above signature_set
#[account(address = guardian_signatures.refund_recipient)]
refund_recipient: AccountInfo<'info>,
// ...
```

The `CleanUpRoot` instruction also contains this issue. If the transaction is signed by the `refund_recipient`, who is marked as the fee payer responsible for paying transaction fees (and thus mutable), the instruction can be successfully processed. However, if the instruction is invoked by someone else (i.e. the `refund_recipient` is not the fee payer and therefore not marked as mutable), the instruction cannot be processed.

Listing 5. Excerpt from CleanUpRoot

```
// ...
/// CHECK: This account is the refund recipient for the above root.
#[account(address = root.refund_recipient)]
refund_recipient: AccountInfo<'info>,
// ...
```

The `CloseSignatures` instruction presents a third scenario. In this case, the likelihood of the issue is low because the `refund_recipient` is marked as a signer, making it most likely that the `refund_recipient` will invoke this instruction. As a result, this account will also be the fee payer (and therefore mutable by default).

Listing 6. Excerpt from CloseSignatures

```
// ...
#[account(address = guardian_signatures.refund_recipient)]
refund_recipient: Signer<'info>,
// ...
```

Exploit scenario

The likelihood of this issue is marked as medium. From an attacker's perspective, there are limited possibilities to exploit this vulnerability. However, the issue is still present and can cause problems during real-time operation. One such scenario could occur when the authority submitting signatures to the Solana blockchain is different from the payer responsible

for covering rent fees for the new Root Account. In this case, the execution would not complete successfully, and you would be forced to either use the same authority for both instructions, modify the program or manually update the IDL.

The remaining instruction, namely `CleanUpRoot`, would still be executable under the current setup. However, the intended logic—where anyone can invoke this instruction to send the rent assets back to the `refund_recipient`—would not function as expected.

Recommendation

We recommend using the `mut` constraint in all of the mentioned account contexts.

Fix 1.1

This issue has been fixed in accordance with the recommendation.

Go back to Findings Summary

W1: Possible Reinitialization

Impact:	Warning	Likelihood:	N/A
Target:	post_signatures.rs	Type:	Reinitialization

Description

The `PostSignatures` instruction defines the `guardian_signatures` account with the `init_if_needed` constraint, allowing the `payer`—or, more appropriately, the authority—to append to the signatures list across multiple transactions.

Listing 7. Excerpt from PostSignatures

```
pub struct PostSignatures<'info> {
    #[account(mut)]
    payer: Signer<'info>,
    // ...

    #[account(
        init_if_needed,
        payer = payer,
        space = 8 +
            GuardianSignatures::compute_size(usize::from(total_signatures))
    )]
    guardian_signatures: Account<'info, GuardianSignatures>,
}
```

Using this constraint can often lead to a reinitialization attack. A reinitialization attack occurs when an attacker (or even an unaware team member) can reset an already initialized (and typically modified) account back to its initial state. Due to the behavior of the `init_if_needed` constraint, which does not call `create_account` (allocate, assign, transfer) if the account is already initialized, it is crucial for developers to correctly verify that the account was not previously initialized. Failing to do so could result in the stored data being reset.

Exploit scenario

In this particular case, the likelihood of a reinitialization attack is rather low. However, there are a few important considerations to be aware of:

- reinitialization is possible if the `PostSignatures` instruction is called with a specified `total_signatures` but an empty signatures vector. In this scenario, if a subsequent `PostSignatures` instruction is called (for example, by a different `payer`), the `guardian_signatures` account could be updated with this new authority (i.e. new `refund_recipient`);
- if the `guardian_signatures` account address was derived as a PDA, the attack scenario would be even more likely. Currently, the `guardian_signatures` is expected to be a regular public key, which decreases the likelihood of an attack, as the attacker would need to know the private key of the `guardian_signatures` account to call `PostSignatures` and update the `refund_recipient`. However, if the `guardian_signatures` were to have its address derived as a PDA, the attacker would not need possession of the private key, allowing them to reinitialize yet only with the empty initial signatures array;
- the impact of an attack could lead to a denial of service, especially when considering M2. If an attacker is able to reinitialize the account to a state where their account is now the `refund_recipient`, you would not be able to perform `UpdateRootWithQuery`, as the `refund_recipient` is not marked as writable, thus preventing you from submitting the next state root.

Recommendation

We recommend implementing the IsInitialized trait for the `GuardianSignatures` along with a new struct field `is_initialized`. One possible solution could look like the following code snippet.

Listing 8. Excerpt from guardian_signatures.rs

```
// ...
pub struct GuardianSignatures {
    pub is_initialized: bool,
    // other fields
}

impl IsInitialized for GuardianSignatures {
    fn is_initialized(&self) -> bool {
        self.is_initialized
    }
}
```

Fix 1.1

The issue was fixed by disallowing empty initial `guardian_signatures`. This resolves the problem, as re-initialization is no longer possible, the `guardian_signatures` field within the `GuardianSignatures` will always contain at least one signature after initialization.

Go back to Findings Summary

I1: Signature Malleability due to accepting S values with high and also low order

Impact:	Info	Likelihood:	N/A
Target:	update_root_with_query.rs	Type:	Data validation

Description

During the audit, the client was informed that the `solana_program::secp256k1_recover` function does not prevent signature malleability.

Signature malleability in ECDSA occurs when a valid signature can be altered in such a way that the modified signature remains valid without changing the underlying message. This happens because ECDSA signatures consist of two components, r and s, where s can be replaced with its additive inverse modulo the curve order, resulting in a different signature that still verifies correctly.

In this case, due to the implementation of `solana_program::secp256k1_recover`, signature malleability is possible since the order of s is included in the signature. However, to create the additive inverse modulo the curve order of the s value, an attacker would need access to the original signature created by any Guardian. Additionally, this type of attack is typically associated with double-spending scenarios, where signature uniqueness is critical. Therefore, we do not think there are any security risks posed by signature malleability in this context. Nevertheless, we think it is important to highlight this fact, particularly for potential future expansions of the project.

Quoutes

The client was already familiar with the issue before being informed.

> *Using secp256k1_recover without additional checks can result in Signature Malleability. Are you aware of this?*
>
> — Ackee Team

> *I'm aware of signature malleability, though I'm not sure how it's relevant in this context (as you mention). The signature simply needs to be valid against the recomputed hash and is not used in any other way (such as for replay protection). Producing a second, valid signature based on the first should have no impact (please let us know if you see something different!).*
>
> — Wormhole Team

Recommendation

It is good practice to accept `s` values of only a particular order. However, since the signatures are generated by a decentralized guardian network, it is better not to alter them, as it is understandably impossible to enforce each guardian in such a network to generate signatures of a specific order.

Go back to Findings Summary

Report Revision 1.1

Revision Team

Revision team is the same as in Report Revision 1.0.

Fuzzing

The issues described in Appendix B that were found during fuzzing have been successfully resolved.

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain Security, Wormhole: Worldcoin World ID State Root Bridge, 2.10.2024.

Appendix B: Trident Findings

This section lists the outputs from the Trident tool used for fuzz testing during the audit. Complete source code of the fuzz tests was provided to the client.

B.1. Fuzzing

For the best fuzzing experience and to increase coverage for fuzz tests, some instruction inputs were generated with the help of Typescript tests. Specifically, these included signatures from the Guardians, root hashes, and inputs required for the proof of personhood.

The following table lists all implemented execution flows in the Trident fuzzing framework.

ID	Flow	Added	Status
F0	Test the main protocol functionality, from posting signatures to the verification instruction	1.0	**Fail** (M1)
F1	Test the admin-related instructions	1.0	Success
F2	Test updating root with different accounts for `payer` and `refund_recipient`	1.0	**Fail** (M2)
F3	Test closing root with `refund_recipient` as an immutable account	1.0	**Fail** (M2)
F4	Test updating root but guardian signatures are slightly randomly modified	1.0	Success
F5	Test verifying proof of personhood but proof is slightly randomly modified	1.0	Success
F6	Test the Solana World ID on-chain template with CPI to the Solana World ID program	1.0	**Fail** (M1)

ID	Flow	Added	Status
F7	Submitted different recipient than expected will not result in a successful `VerifyAndExecute` invocation	1.0	Success
F8	Test verifying proof but nullifier is slightly randomly modified	1.0	Success

Table 4. Trident fuzzing flows

The following table lists all implemented invariant checks in the Trident fuzzing framework.

ID	Invariant	Added	Status
IV1	Only the config authority can update allowed staleness	1.0	Success
IV2	Only the config authority can update root expiry	1.0	Success
IV3	Only the config authority can initiate ownership transfer	1.0	Success
IV4	Only the current authority or the pending authority can claim Config ownership	1.0	Success
IV5	If the `refund_recipient` is writable, `UpdateRootWithQuery` correctly returns rent fees to their account	1.0	Success
IV6	Randomly modifying one byte of the signature will result in `UpdateRootWithQuery` success only if the changed byte matches the reference value	1.0	Success

ID	Invariant	Added	Status
IV7	Randomly modifying three bytes of the proof will result in `VerifyGroth16Proof` success only if the changed bytes match the reference values	1.0	Success
IV8	Randomly modifying one byte of the nullifier will result in `VerifyGroth16Proof` success only if the changed byte match the reference value	1.0	Success

Table 5. Trident fuzzing invariants

ackee
blockchain security

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz