Adevar Labs

Prepared for:
DoubleZero

November 18, 2025

Passport & Sentinel Audit Report

Version: after-fix report

Table of Contents

1. Executive Summary

Executive Summary

About DoubleZero

DoubleZero is a decentralized network protocol designed for efficient and reliable data transmission. This audit specifically focused on the passport and sentinel modules, which manage access requests to the DoubleZero networks, ensuring verified actors access the network.

Audit Summary

The following table summarizes the distribution of identified vulnerabilities by risk level:

Risk Level	Count	Fixed	Acknowledged
Critical	0	0	0
High	1	1	0
Medium	2	2	0
Low	0	0	0

Enhancement Opportunities: 1

Overall Security Posture

The DoubleZero team demonstrated a high level of expertise in Solana program development. Significantly, no issues were found in the on-chain component (Passport) of the audit. The team's decision to forgo the Anchor framework in favor of native Solana programming reflects a sophisticated approach to optimize execution costs and maintain precise flow control.

After Fix Review Summary

All findings have been addressed by the DoubleZero team. The team implemented an architectural change of the Passport program where access mode is now encoded directly in the access request PDA.

This eliminates signature scanning entirely, resolving both incorrect signature ordering issue and the multiple access request extraction bug.

It was also confirmed that the Sentinel will be automatically restarted after unexpected shutdowns.

Before Fix Review Summary

The audit identified 3 security findings in the Sentinel component:

- An issue where the Sentinel incorrectly processes multiple access requests in a single transaction, extracting the wrong service key.
- An incorrect signature ordering logic that could make the Sentinel select the wrong transaction.
- A lack of documented restart mechanisms for cross-network state synchronization.

Launch Recommendations

To ensure a smooth and secure launch, we offer the following recommendations:

I. Mandatory Before Launch

- Fix all High and Medium severity issues.

II. Strongly Recommended

- Develop comprehensive tests for the Sentinel component, covering multi-instruction transactions, network failure scenarios, and edge cases identified in the findings.
- Document a procedure to ensure that the off-chain entity is restarted properly

III. Post-Launch Monitoring

- Test backfill operations under high transaction volumes and network congestion.
- Monitor suspicious activity that might try to spoof legitimate events and bypass filters.

Audit Scope

- **Repository:** doublezero-solana (private)
- **Commit Hash:** **e71c7101a98f71d94dc99831a069dabdee8ee144**
- **Files/Modules in Scope:** programs/passport/src/*.rs

- **Repository:** doublezero-offchain (private)
- **Commit Hash:** **c036b7ecad0e15db8a05d124f7f9c728bb8a0730**
- **Files/Modules in Scope:** crates/sentinel/src/*.rs

Fix commits:

- doublezero-solana: caf1cea
- doublezero-offchain: 3111cff

2. Assumptions and Considerations

Assumptions & Considerations

- A single Sentinel instance processes all access requests.
- RPC endpoints return correct information about the network, such as transaction content or validator participation information.
- Deposit/fee structure creates proper economic incentives without enabling griefing attacks.

Scope Limitations

Time-Limited Assessment: The security assessment was conducted over a fixed-time period. As such, not all potential security vulnerabilities may have been identified. Security assessment is a point-in-time exercise, and new vulnerabilities may emerge following the completion of this assessment.

Defined Scope: The assessment was limited to the systems, applications, and networks explicitly defined in the scope. Systems outside the defined scope, even if they interact with in-scope systems, were not assessed. A list of in-scope assets is provided in the Introduction section of this report.

3. Severity Definitions

Each issue identified in this report is assigned a severity level based on two dimensions: **Impact** and **Likelihood**. These dimensions help project our team's understanding of both the potential consequences of a vulnerability and how likely a vulnerability is to be discovered and exploited in the real world.

Impact

Impact reflects the potential consequences of the issue–particularly on **project funds**, **user funds**, and the **availability or integrity** of the protocol.

- **High Impact**: Successful exploitation could result in a complete loss of user or protocol funds, disruption of core protocol functionality, or permanent loss of control over critical components.
- **Medium Impact**: Exploitation could cause significant disruption or partial loss of funds, but not a total compromise. May impact some users or non-core functionality.
- **Low Impact**: The issue has minor or negligible consequences. It may affect edge cases, expose metadata, or degrade performance slightly without putting funds or core logic at serious risk.

Likelihood

Likelihood reflects how easy a vulnerability is to discover and exploit by an attacker, as well as how economically attractive the exploit is to an attacker.

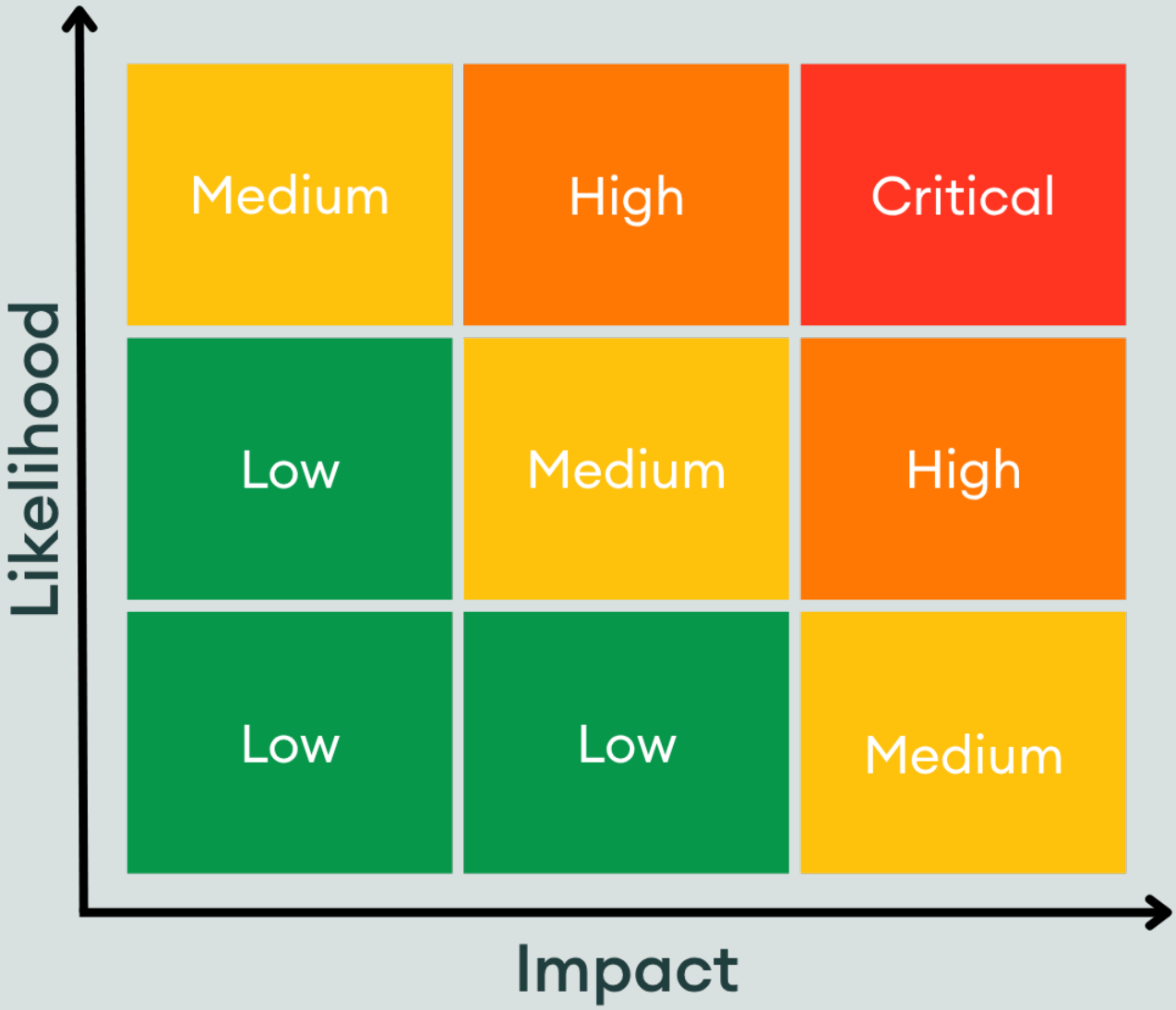
- **High Likelihood**: The vulnerability is trivially exploitable. This means it can be exploited by a wide range of actors without privileged access rights, with minimal capital requirements and low financial risks.
- **Medium Likelihood**: This type of vulnerability can be found and exploited with moderate effort. It might require a significant capital investment, but with manageable financial risk.
- **Low Likelihood**: Exploitation of these vulnerabilities is often technically unfeasible or requires highly specialized conditions. They may require extraordinary effort or a significant financial risk for an attacker, with a high chance of failure and minimal potential return.

Severity Classification Matrix

By combining **Impact** and **Likelihood**, we assign a severity level using the matrix below:

- **Critical**: High impact + high likelihood (e.g. a bug that could allow anyone to drain a substansial amount of protocol funds with minimal effort)
- **High**: High impact with medium likelihood, or medium impact with high likelihood
- **Medium**: Moderate impact and/or discoverability
- **Low**: Minimal impact or unlikely to be exploited

This structured approach helps teams prioritize fixes and mitigate the most dangerous threats first.



Severity Matrix

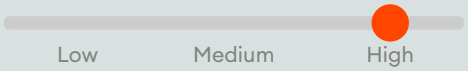
4. Findings

H01: Incorrect Use of .first() Instead of .last() in get_signatures_for_address

Status: Resolved

Impact:

Likelihood:



Severity: High

Location: https://github.com/AdevarLabs/doublezero-sentinel-offchain/blob/842c00e9abe5c7eb258408656d584d794739100a/crates/sentinel/src/client/solana.rs#L136-L141

>_ solana.rs RUST

```rust
134:         let access_ids = futures::stream::iter(accounts)
135:             .then(|(pubkey, _acct)| async move {
136:                 let signatures = self.client.get_signatures_for_address(&pubkey).await
        ?;
137:
138:                 let creation_signature: Signature = signatures
139:                     .first()
140:                     .ok_or(Error::MissingTxnSignature)
141:                     .and_then(|sig| sig.signature.parse().map_err(Error::from))?;
142:
143:                 self.get_access_request_from_signature(creation_signature)
```

Description:

The function **get_signatures_for_address** currently retrieves all signatures associated with an account and then takes **.first()**, assuming this corresponds to the creation signature.

However, the Solana RPC method **getSignaturesForAddress** returns signatures in reverse chronological order (newest → oldest). This means:

- **.first()** = most recent signature
- **.last()** = oldest (creation) signature

This mismatch introduces the following risk:

- An attacker can send an arbitrary lamport transfer (or any transaction).
- This artificially adds more recent signatures, pushing the true creation signature further back.
- As a result, the backfill logic may select the wrong transaction.
- In particular, the **backfill_timer** handler could encounter a DoS condition when **deserialize_access_request_ids** fails to find the expected **compiled_ix** targeting the **passportId**.

Recommendation:

Update the logic to correctly fetch the first-ever transaction.
Use **.last()** instead of **.first()** when identifying the creation signature.

Developer Response:

After a design review with the DoubleZero team, the architecture was changed: access mode is now encoded directly in the access request PDA.
The Sentinel no longer scans signatures at all. State is read directly from the PDA. This removes the **.first()**/**.last()** dependency entirely and eliminates the whole class of traversal-based failures.
PR #63.
PR #107

M01: Lack of Restart Mechanism Documentation for Cross-Network State Synchronization

Status: Resolved

Impact:

Likelihood:

Severity: Medium

Location: https://github.com/AdevarLabs/doublezero-sentinel-offchain/blob/842c00e9abe5c7eb258408656d584d794739100a/crates/sentinel/src/sentinel/handler.rs#L76-L98

>_ handler.rs RUST

```rust
    }

    async fn handle_access_request(&self, access_ids: AccessIds) -> Result<()> {
        let AccessMode::SolanaValidator { service_key, .. } = access_ids.mode;
        if let Some(validator_ip) = self.verify_qualifiers(&access_ids.mode).await? {
            self.dz_rpc_client
                .issue_access_pass(&service_key, &validator_ip)
                .await?;
            let signature = self
                .sol_rpc_client
                .grant_access(&access_ids.request_pda, &access_ids.rent_beneficiary_key)
                .await?;
            info!(%signature, user = %service_key, "access request granted");
            metrics::counter!("doublezero_sentinel_access_granted").increment(1);
        } else {
            let signature = self
                .sol_rpc_client
                .deny_access(&access_ids.request_pda)
                .await?;
            info!(%signature, user = %service_key, "access request denied");
            metrics::counter!("doublezero_sentinel_access_denied").increment(1);
        }

        Ok(())
    }

    async fn verify_qualifiers(&self, access_mode: &AccessMode) -> Result<Option<Ipv4Addr>> {
```

Description:

The **handle_access_request** function performs sequential operations across two different blockchain networks (DZ Ledger and Solana) without atomicity guarantees.
While the implementation includes retry mechanisms for network failures, the system relies on external process management (systemd) to ensure recovery from failures, which is not documented in the codebase.

Normal Operation:

```
1. issue_access_pass() → DZ Ledger (idempotent - no-op if already set)
2. grant_access() → Solana (closes AccessRequest PDA, releases funds)
```

Failure Scenario - Sentinel Crash After Step 1:

```
1. issue_access_pass() → SUCCESS on DZ Ledger
2. Crash - Sentinel dies before grant_access()
   - DZ Ledger state: Access pass set
   - Solana state: AccessRequest PDA still open with user funds
   - Networks are temporarily desynchronized
```

Recovery on Restart (via systemd):

```
1. Sentinel restarts and polls for open AccessRequest accounts
2. issue_access_pass() → No-op (already exists on DZ Ledger)
3. grant_access() → SUCCESS (closes PDA, releases funds)
   - Cross-network state synchronized
   - No fund lock occurs
```

If the sentinel is deployed without automatic restart capabilities:
- Manual intervention is required after crashes
- Temporary state desynchronization persists

Recommendation:

Add explicit documentation stating that the sentinel must run under process supervision to prevent state inconsistency and fund lock scenarios.
Providing a reference systemd service file in the repository would also be beneficial.

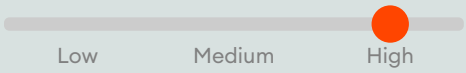
Developer Response:

Although the information was not available at the time of the audit, the Sentinel is planned to be automatically restarted after an unexpected shutdown.

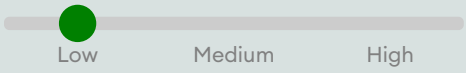
M02: Wrong Service Key Extraction for Multiple Access Requests in Single Transaction

Status: Resolved

Impact:



Likelihood:



Severity: Medium

Location: https://github.com/AdevarLabs/doublezero-sentinel-offchain/blob/842c00e9abe5c7eb258408656d584d794739100a/crates/sentinel/src/client/solana.rs#L221-L248

>_ solana.rs RUST

```rust
219: }
220:
221: fn deserialize_access_request_ids(txn: Transaction) -> Result<AccessIds> {
222:     let signature = txn.signatures.first().ok_or(Error::MissingTxnSignature)?;
223:     let compiled_ix = txn
224:         .message
225:         .instructions
226:         .iter()
227:         .find(|ix| ix.program_id(&txn.message.account_keys) == &passport_id())
228:         .ok_or(Error::InstructionNotFound(*signature))?;
229:     let accounts = compiled_ix
230:         .accounts
231:         .iter()
232:         .map(|&idx| txn.message.account_keys.get(idx as usize).copied())
233:         .collect::<Option<Vec<_>>>()
234:         .ok_or(Error::MissingAccountKeys(*signature))?;
235:     let Ok(PassportInstructionData::RequestAccess(mode)) =
236:         PassportInstructionData::try_from_slice(&compiled_ix.data)
237:     else {
238:         return Err(Error::InstructionInvalid(*signature));
239:     };
240:     match (accounts.get(2), accounts.get(1)) {
241:         (Some(request_pda), Some(payer)) => Ok(AccessIds {
242:             request_pda: *request_pda,
243:             rent_beneficiary_key: *payer,
244:             mode,
245:         }),
246:         _ => Err(Error::InstructionInvalid(*signature)),
247:     }
248: }
249:
250: pub struct PreviousEpochSlots {
```

Description:

When a validator submits multiple **RequestAccess** instructions for different service keys within a single transaction as multiple instructions, the sentinel only processes the first access request repeatedly while completely ignoring the remaining requests. This results in attempting to process the first request multiple times while subsequent requests are never

processed, leaving validators without issued access passes for those service keys and without refunds for their deposits.

Root Cause

The bug exists in **/crates/sentinel/src/client/solana.rs** in the **deserialize_access_request_ids()** function:

```rust
fn deserialize_access_request_ids(txn: Transaction) -> Result<AccessIds> {
    // PROBLEM: Always finds the FIRST passport instruction
    let compiled_ix = txn
        .message
        .instructions
        .iter()
        .find(|ix| ix.program_id(&txn.message.account_keys) == &passport_id())
        //     ^^^^^ Always returns instruction 0, never instruction 1!
        .ok_or(Error::InstructionNotFound(*signature))?;
```

Detailed Attack Scenario

Consider a validator calling request access for two service keys in one transaction and two instructions:

Transaction Structure

```rust
Transaction {
    instructions: [
        // Instruction 0: RequestAccess for service_key_A → creates account_A
        RequestAccess {
            service_key: "service_A",
            validator_id: "validator_123"
        },

        // Instruction 1: RequestAccess for service_key_B → creates account_B
        RequestAccess {
            service_key: "service_B",
            validator_id: "validator_123"
        }
    ]
}
```

Step 1: get_access_requests() Processing
The system calls get_access_requests() which finds both accounts:

```rust
pub async fn get_access_requests(&self) -> Result<Vec<AccessIds>> {
    // Get all AccessRequest accounts
    let accounts = self
        .client
        .get_program_accounts_with_config(&passport_id(), config)
        .await?;
    // accounts = [(account_A, data), (account_B, data)]

    let access_ids = futures::stream::iter(accounts)
```

... continued RUST

```
        .then(|(pubkey, _acct)| async move {
            // This processes EACH account separately
            // Process account_A first, then account_B
```

Step 2: Processing Account A (Correct)

RUST

```
// For account_A:
let signatures = self.client.get_signatures_for_address(&account_A).await?;
// signatures = [shared_transaction_signature]

let creation_signature = signatures.first()?;
// creation_signature = shared_transaction_signature

self.get_access_request_from_signature(creation_signature).await
// ↓ Calls deserialize_access_request_ids()
```

Step 3: Processing Account B (Bug Occurs)

RUST

```
// For account_B:
let signatures = self.client.get_signatures_for_address(&account_B).await?;
// signatures = [shared_transaction_signature] ← SAME signature as account_A!

let creation_signature = signatures.first()?;
// creation_signature = shared_transaction_signature ← SAME as account_A!

self.get_access_request_from_signature(creation_signature).await
// ↓ Calls deserialize_access_request_ids() with SAME transaction
```

Bug: deserialize_access_request_ids()

RUST

```
fn deserialize_access_request_ids(txn: Transaction) -> Result<AccessIds> {
    // PROBLEM: Always finds the FIRST passport instruction
    let compiled_ix = txn
        .message
        .instructions
        .iter()
        .find(|ix| ix.program_id(&txn.message.account_keys) == &passport_id())
        //     ^^^^^ Always returns instruction 0, never instruction 1!
        .ok_or(Error::InstructionNotFound(*signature))?;

    // Extract accounts from the FIRST instruction only
    let accounts = compiled_ix
        .accounts
        .iter()
        .map(|&idx| txn.message.account_keys.get(idx as usize).copied())
        .collect::<Option<Vec<_>>>()?;

    // Extract mode from the FIRST instruction only
    let Ok(PassportInstructionData::RequestAccess(mode)) =
        PassportInstructionData::try_from_slice(&compiled_ix.data);
```

... continued RUST

```
    // Return data from FIRST instruction, regardless of which account we're processing
    match (accounts.get(2), accounts.get(1)) {
        (Some(request_pda), Some(payer)) => Ok(AccessIds {
            request_pda: *request_pda,      // ← Account for A instead of B
            rent_beneficiary_key: *payer,  // ← But these are always from instruction 0
            mode,                          // ← service_key_A always!
        }),
        _ => Err(Error::InstructionInvalid(*signature)),
    }
}
```

Impact - Result Vector from get_access_requests()

RUST

```
// What we get (BUGGY):
vec![
    AccessIds {
        request_pda: account_A,
        mode: AccessMode::SolanaValidator { service_key: "service_A", .. }
    }, // Correct

    AccessIds {
        request_pda: account_A,  // ← Incorrect
        mode: AccessMode::SolanaValidator { service_key: "service_A", .. } // --> wrong
    service
    }
]

// What we should get (CORRECT):
vec![
    AccessIds {
        request_pda: account_A,
        mode: AccessMode::SolanaValidator { service_key: "service_A", .. }
    },
    AccessIds {
        request_pda: account_B,
        mode: AccessMode::SolanaValidator { service_key: "service_B", .. }
    }
]
```

Effects in handle_access_request()

When processing the corrupted entry for account_B:

RUST

```
async fn handle_access_request(&self, access_ids: AccessIds) -> Result<()> {
    let AccessMode::SolanaValidator { service_key, .. } = access_ids.mode;
    // service_key = "service_A" ← WRONG! Should be "service_B"

    if let Some(validator_ip) = self.verify_qualifiers(&access_ids.mode).await? {
        // ■ Issues access pass for WRONG service - which will fail since the access pass for
     the service key and validator IP already exists
        self.dz_rpc_client
            .issue_access_pass(&service_key, &validator_ip) // ← service_A instead of service
    _B!
            .await?;
```

... continued RUST

```
        // ■ This will fail too since the access IDs will give the request PDA for the first
    request which has already been processed
        let signature = self
            .sol_rpc_client
            .grant_access(&access_ids.request_pda, &access_ids.rent_beneficiary_key) // ←
    account_B
            .await?;

        info!(%signature, user = %service_key, "access request granted"); // ← Logs "service_
    A"
    }
}
```

Recommendation:

Modify the code to check if there are multiple access requests using different service keys within one transaction.
The **deserialize_access_request_ids()** function should loop through all passport instructions in the transaction instead of always using **.find()** which returns the first match. Add a **target_request_pda** parameter to match the correct instruction for each account being processed.

Developer Response:

Similarly to the H01 finding, the access mode is now directly encoded into the Access Request PDA, which removes the necessity to fetch the data from the instructions.
PR #63.
PR #107

5. Enhancement Opportunities

E01: Implement Full Account Closure to Prevent Revival Attacks

Location: https://github.com/AdevarLabs/doublezero-passport-solana/blob/main/programs/passport/src/processor.rs#L289-L298

>_ processor.rs RUST

```
287:
288:     // Zero out the access request lamports to close the account
289:     **access_request_lamports = 0;
290:
291:     msg!("Grant {} access", access_request.service_key);
292:     msg!(
293:         "Return {} lamports to {}",
294:         request_refund,
295:         rent_beneficiary_info.key,
296:     );
297:
298:     Ok(())
299: }
300:
```

Description:

Currently, the **try_grant_access** function closes the **AccessRequest** account by zeroing out lamports but leaves the account data intact and doesn't transfer ownership to the system program.

While this function is currently role-protected (only callable by Sentinel), making this method of closing the account safe, if it becomes permissionless in the future, this could create potential for a revival attack.

Potential Benefit:

- Future-proofs the implementation against revival attacks by ensuring proper account closure.
- Provides additional security layers with minimal performance cost.

Recommendation:

Add proper account closure by transferring ownership to the system program and zeroing out the account data.
See example from Anchor.

About Us

About Adevar Labs

Adevar Labs is a boutique blockchain security firm specializing in web3 audits.

Built by a mix of experienced professionals in traditional enterprise and crypto natives who have contributed to some of the most critical projects in blockchain infrastructure.

Our team's background spans companies like Bitdefender, Asymmetric Research, Quantstamp, Chainproof, and Juicebox, and includes experience securing smart contracts, bridges, and L1 and L2 protocols across ecosystems like Solana, Ethereum, Polkadot, Cosmos, and MultiversX.

With over 100 audits completed and a portfolio that includes custom fuzzers, exploit modeling, and runtime testing frameworks, Adevar Labs brings both depth and precision to every engagement.

Our auditors have discovered critical vulnerabilities, built high-impact tooling, and placed in top positions in premier audit competitions including Code4rena and Sherlock.

Team members hold distinctions such as PhDs in software protection, and key roles at Fortune 500 companies, and leadership of flagship conferences like ETH Bucharest.

With team members having publications with over 1,100 academic citations and trusted by projects securing over $500M in on-chain value, Adevar Labs blends elite technical rigor with real-world security impact.

We also collaborate with some of the best independent security researchers in the web3 space.

Projects may optionally request specific contributors to be part of their audit.

Audit Methodology

Our audit methodology is specialized to provide thorough security assessments of Solana programs. We focus explicitly on rigorous manual analysis, detailed threat modeling, and careful validation of implemented fixes to ensure your programs operate securely and as intended.

1. Program Context and Architecture Analysis

Our auditors begin by deeply examining your Solana program's documentation, intended functionality, and account design. We meticulously map out program interactions, instruction processing flows, and state management logic. Special attention is given to understanding how your program interfaces with critical Solana system programs, such as the SPL Token Program, Stake Program, and System Program. Last but not least we check external integrations with other Solana projects and verify if the inputs and return values are handled properly.

2. Threat Modeling

We conduct targeted threat modeling tailored specifically for Solana's execution environment. We carefully define attacker capabilities and identify potential vulnerabilities that may arise from Solana-specific issues, including but not limited to:

- Unauthorized account data manipulation
- Improper ownership or signer verification
- Misuse of Program-Derived Addresses (PDAs)

- Incorrect use of Cross-Program Invocations (CPI)
- Failure to adequately handle account privileges or account states
- Risks stemming from rent-exemption and account initialization logic

3. In-depth Manual Security Review

Our experienced auditors perform an extensive manual security review of your Rust-based Solana programs. This involves a comprehensive line-by-line inspection of source code, focusing on common Solana vulnerabilities including, but not limited to:

- Missing or insufficient ownership checks
- Inadequate signer checks
- Incorrect handling of CPI calls and invocation privileges
- Arithmetic and integer overflow or underflow errors
- Unsafe deserialization and serialization of account data structures
- Improper token transfers and SPL-token logic issues
- Logic flaws in financial operations or state transitions
- Edge-case handling in instruction input validation
- Potential denial-of-service vectors related to transaction execution and account handling

During this stage, we clearly document any discovered vulnerabilities, including detailed descriptions, precise severity ratings, and recommendations for secure implementation. In situations where it is not clear how the vulnerability might be exploited we may also include a detailed proof-of-concept exploit including code snippets and instructions on how the exploit could be performed.

4. Detailed Fix Review and Validation

After the initial audit and your team's subsequent remediation efforts, we perform a comprehensive fix review to ensure the vulnerabilities identified have been effectively resolved.

We verify each fix individually, confirming that:

- Corrections effectively eliminate the security risks
- Changes do not inadvertently introduce new vulnerabilities or regressions
- The fixes align closely with Solana best practices and secure coding guidelines

Our detailed validation ensures that security improvements are robust, complete, and aligned with best practices specific to the Solana development ecosystem.

Confidentiality Notice

This report, including its content, data, and underlying methodologies, is subject to the confidentiality and feedback provisions in your agreement with Adevar Labs. These materials are not to be disclosed, extracted, copied, or distributed except to the extent expressly authorized by Adevar Labs.

Legal Disclaimer

The review and this report are provided by Adevar Labs on an as-is, where-is, and as-available basis. To the fullest extent permitted by law, Adevar Labs disclaims all warranties, expressed or implied, in

connection with this report, its content, and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

You agree that access to and/or use of the report and other results of the review, including any associated services, products, protocols, platforms, content, and materials, will be at your sole risk.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE. This report is based on the scope of materials and documentation provided for a limited review at the time provided.

You acknowledge that blockchain technology remains under development and is subject to unknown risks and flaws. Adevar Labs does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party, or any open-source or third-party software, code, libraries, materials, or information accessible through the report. As with the purchase or use of a product or service in any environment, you should use your best judgment and exercise caution where appropriate.