External Signature Program

Security Assessment

July 20th, 2025 — Prepared by OtterSec

Tamta Topuria tamta@osec.io

Robert Chen notdeghost@osec.io

Table of Contents

© 2025 Otter Audits LLC. All Rights Reserved.

01 — Executive Summary

Overview

Squads engaged OtterSec to assess the `external-signature-program` program. This assessment was conducted between June 26th and July 12th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 9 findings throughout this audit engagement.

In particular, we identified a high-risk vulnerability where the session key authorization and execution instruction fails to verify that the externally signed account is owned by the program, allowing attackers to forge fake accounts with a victim's public key and list themselves as session signers (OS-ESP-ADV-00). Additionally, the lack of a discriminator in payload hashing allows a signature meant for instruction execution to be re-utilized for session key refresh, enabling unintended privilege escalation (OS-ESP-ADV-01).

Furthermore, the payload hash omits the number of execution accounts, allowing a malicious signer to modify the account list and inject unintended instructions without invalidating the user's signature (OS-ESP-ADV-02).

We also provided recommendations to ensure adherence to coding best practices (OS-ESP-SUG-02) and suggested removing the unutilized code instances for improved clarity and maintainability (OS-ESP-SUG-03). We further advised against manually reconstructing `clientDataJSON`, as it may break signature checks and lock users out if authenticators such as Google add fields (OS-ESP-SUG-00).

© 2025 Otter Audits LLC. All Rights Reserved.

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/0xRigel-squads/external-signature-program. This audit was performed against commit f28557e.

A brief description of the program is as follows:

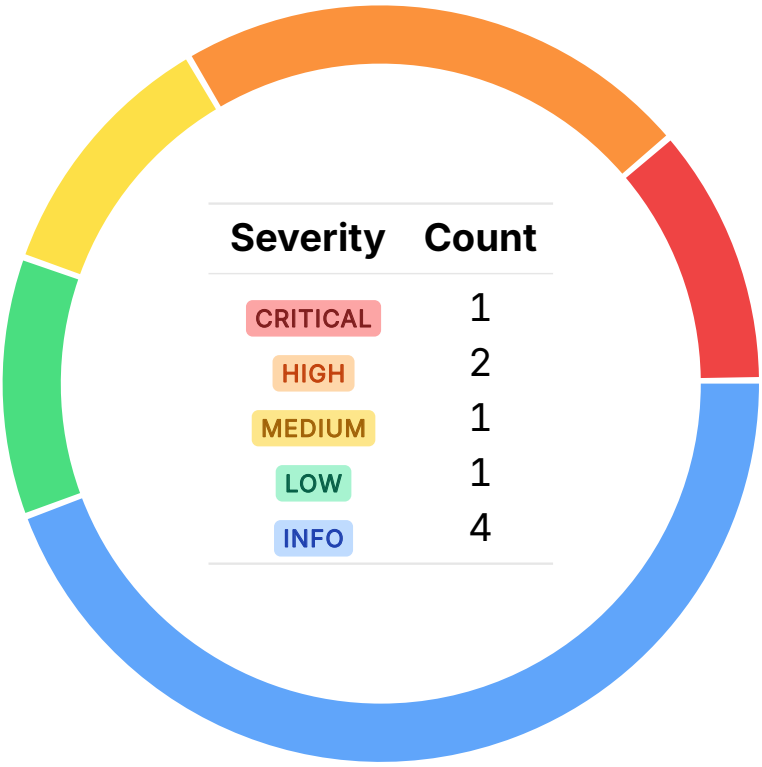
Name	Description
external-signature-program	It enables Solana accounts to be securely controlled using external signature schemes like WebAuthn/Passkeys, allowing passwordless authentication via biometrics or device credentials.

© 2025 Otter Audits LLC. All Rights Reserved.

03 — Findings

Overall, we reported 9 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	1
HIGH	2
MEDIUM	1
LOW	1
INFO	4

© 2025 Otter Audits LLC. All Rights Reserved.

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-ESP-ADV-00	CRITICAL	RESOLVED	`execute_instructions_sessioned` instruction fails to verify that the `ExternallySignedAccount` is owned by the program, allowing attackers to forge fake accounts with a victim's public key and list themselves as session signers.
OS-ESP-ADV-01	HIGH	RESOLVED	The lack of a discriminator in payload hashing allows a signature meant for instruction execution to be re-utilized for session key refresh, enabling unintended privilege escalation.
OS-ESP-ADV-02	HIGH	RESOLVED	The payload hash omits the number of execution accounts, allowing a malicious signer to modify the account list and inject unintended instructions without invalidating the user's signature.
OS-ESP-ADV-03	MEDIUM	RESOLVED	`get_index_difference` fails near slot wraparound boundaries due to incorrect handling of truncated slot values, resulting in rejection of valid instructions.
OS-ESP-ADV-04	LOW	RESOLVED	Solana permits self-reentrancy via CPI, which could lead to unexpected behavior if not explicitly handled.

© 2025 Otter Audits LLC. All Rights Reserved.

Missing Ownership Verification Check CRITICAL OS-ESP-ADV-00

Description

`execute_instructions_sessioned` instruction contains a critical issue due to missing ownership verification of the `ExternallySignedAccount`, which is implicitly verified by write in the other three instructions. Specifically, `load` deserializes the account's data without confirming that it is actually owned by the program (`account.owner != &crate::ID` is not checked). The PDA utilized for signing is computed based on the public key field in the deserialized structure, and not based on the actual account's owner, since the structure is never written in this instruction. Also, the method of invocation of the instruction enables the attacker to include the victim's `ExternallySignedAccount` as a remaining account and still execute a signed instruction with it.

>_ *src/instructions/execute_instructions_sessioned.rs* RUST

```rust
// Sanitizes, checks and loads the context from the account infos and args
pub fn load(
    account_infos: &'a [AccountInfo],
    execution_args: &'a ExecutableInstructionSessionedArgs,
) -> Result<Box<Self>, ProgramError> {
    [...]
    // Load and check the relevant accounts
    let externally_signed_account =
        ExternallySignedAccount::<T>::load(externally_signed_account)?;
    externally_signed_account.is_valid_session_key(session_signer)?;
    [...]
}
```

This occurs as `get_execution_account` re-derives the address from the `public_key` stored in the `ExternallySignedAccount`, resulting in the `executing_account` to be the victim's account, even if the `externally_signed_account` was supplied by the attacker. This implies an attacker may craft a fake account with arbitrary data structured similar to `ExternallySignedAccount`, embed the victim's public key inside it, and list themselves as a valid `session_key` to call transactions with the actual victim's external account signer.

Remediation

Explicitly check that the `ExternallySignedAccount` is owned by the correct program ID.

Patch

Resolved in 95b146f.

© 2025 Otter Audits LLC. All Rights Reserved.

Signature Replay Due to Missing Payload Discriminator HIGH OS-ESP-ADV-01

Description

The issue arises due to the absence of a domain discriminator in the payload hashing logic of `execute_instructions` and `refresh_session_key`. Both the functions, `execute_instructions::get_instruction_payload_hash` and `refresh_session_key::get_refresh_session_key_payload_hash`, compute their signature payloads over similarly structured byte arrays, composed of `slothash`, `nonce_signer`, and additional fields, without including any explicit prefix to distinguish their intent.

As a result, a valid signature for an `execute_instructions` payload may be misinterpreted as authorization for a `refresh_session_key` instruction. This enables a malicious actor to trick a user into signing what appears to be a harmless execution payload (a crank call), and later re-utilize the signature to set that program as the account's `session_key`.

Proof of Concept

1. An attacker prepares a normal-looking `execute_instructions` payload, such as a crank, and presents it to the user:

```rust
>_ payload.rs RUST

//Example payload
[slothash + nonce_signer + <evil program> + u8 + u8 + ins.len(1) + <an instruction with 5
    ↪  bytes>]
```

2. The user perceives this as a harmless crank call and agrees to execute the crank without ext account signer.
3. The attacker then re-utilizes this signed payload in `refresh_session_key`, which interprets the evil program key as the session key and the next byte(s) as a large expiration: `[slothash + nonce_signer + <evil program> + <huge expiration_time>]`.
4. The `ExternallySignedAccount` is then updated with the attacker's session key, granting them access.

Remediation

Add an instruction-type discriminator in each payload hash to prevent cross re-utilization of signatures across instruction types.

© 2025 Otter Audits LLC. All Rights Reserved.

Patch

Resolved in 012d8c5 and 8766f0b.

© 2025 Otter Audits LLC. All Rights Reserved.

Lack of Inclusion of Account List Length HIGH OS-ESP-ADV-02

Description

`execute_instructions::get_instruction_payload_hash` constructs a hash over the nonce, signer, accounts, and instructions, but it fails to include the number of `instruction_execution_accounts` in the signed payload. This omission allows a malicious actor to alter the set of accounts utilized in execution to modify the amount, resulting in the potential execution of different instructions than the user intended. Although this may appear as an unlikely scenario, a carefully chosen second account address may mislead the user into signing a transaction they did not intend to authorize.

>_ *src/instructions/execute_instructions.rs* RUST

```rust
// Gets the instruction payload hash
pub fn get_instruction_payload_hash(&self) -> [u8; 32] {
    [...]
    // Build the instruction execution accounts and their metas as they were
    // passed in
    self.accounts
        .instruction_execution_accounts
        .iter()
        .for_each(|account| {
            instruction_payload.extend_from_slice(account.key().as_ref());
            instruction_payload.push(account.is_signer() as u8);
            instruction_payload.push(account.is_writable() as u8);
        });
    [...]
}
```

Proof of Concept

1. Assuming the user signs a payload that hashes:
   `[slothash + nonce_signer + ACC1 + ACC2 + ins.len(1) + transfer(100 to bob)]`
2. Since the number of `instruction_execution_accounts` (`ACC1`, `ACC2`, ...) is not included in the signed payload, the `nonce_signer` may modify the transaction to include only one remaining account, resulting in the following payload:

   >_ *payload.rs* RUST

   ```
   [slothash + nonce\_signer + ACC1 + ins.len(3) + transfer(1234 to bob) + transfer(1234 to
       ↪  bob) + transfer(100 to bob)].
   ```

3. This payload drops `ACC2` from the actual execution context and adds two extra instructions transferring more funds.

© 2025 Otter Audits LLC. All Rights Reserved.

Remediation

Ensure to include the account list length in the signed payload to bind the user's approval to a fixed execution context.

Patch

Resolved in `012d8c5`.

© 2025 Otter Audits LLC. All Rights Reserved.

Failure Due to Truncated Slot Wraparound MEDIUM OS-ESP-ADV-03

Description

`get_index_difference` incorrectly assumes truncated slots (modulo 1000) are linearly ordered, resulting in failures when slot numbers wrap around. Specifically, slot 863001 truncates to 1, and 862999 to 999. On calculating their difference, the function function raises an error due to underflow since 1 < 999, even though the first slot is ahead of the second. As a result, legitimate transactions may be rejected if issued near these slot wraparound boundaries.

>_ *src/utils/nonce.rs* RUST

```rust
/// Returns the difference between two truncated slots
pub fn get_index_difference(&self, other: &Self) -> Result<u16, ProgramError> {
    // Truncated slot should never be greater than a current slot
    match self.0.checked_sub(other.0) {
        Some(diff) => Ok(diff),
        None => Err(ExternalSignatureProgramError::InvalidTruncatedSlot.into())
    }
}
```

Remediation

Ensure the modular arithmetic handles wraparound correctly.

Patch

Resolved in c7d0e17.

© 2025 Otter Audits LLC. All Rights Reserved.

Unintended Self-Reentrancy Risk LOW OS-ESP-ADV-04

Description

Solana allows programs to invoke themselves via CPI (self-reentrancy), which may be risky if not explicitly accounted for. While the current utilization of a counter appears safe and unaffected, reentrancy may introduce unexpected behavior in future changes. Thus, it will be appropriate to proactively disable self-reentrancy unless it is an intentional design feature.

Remediation

Disable re-entrancy from the CPIs.

Patch

Resolved in 332723f and 012d8c5.

© 2025 Otter Audits LLC. All Rights Reserved.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-ESP-SUG-00	Manually reconstructing `clientDataJSON` is brittle, if authenticators like Google add fields, it may break signature checks and lock users out.
OS-ESP-SUG-01	The `From<Vec<T>>` implementation for `SmallVec<L, T>` lacks a length check, allowing vectors larger than `L`'s max to be stored.
OS-ESP-SUG-02	Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices.
OS-ESP-SUG-03	The codebase contains multiple cases of unutilized code that should be removed for better maintainability and clarity.

© 2025 Otter Audits LLC. All Rights Reserved.

Fragile ClientDataJSON Handling OS-ESP-SUG-00

Description

`reconstruct_client_data_json` manually rebuilds the WebAuthn `clientDataJSON`, rendering it fragile against changes by authenticators such as Google. If new fields are added, signature verification may fail, potentially locking users out of their accounts temporarily.

>_ *src/utils/signatures/p256_webauthn/client_data_json.rs* RUST

```rust
/// Reconstructs the clientDataJson from the params, rp_id and challenge
pub fn reconstruct_client_data_json(
    params: &ClientDataJsonReconstructionParams,
    rp_id: &[u8],
    challenge: &[u8],
) -> Vec<u8> {
    [...]
    // Add Google's extra field if needed
    if params.has_google_extra() {
        json_bytes.extend_from_slice(b",\"other_keys_can_be_added_here\":\"do not compare
            ↪  clientDataJSON against a template. See https://goo.gl/yabPex\"");
    }
    [...]
}
```

Remediation

Utilize the actual raw `clientDataJSON` as input.

Patch

Resolved in 4601c85.

© 2025 Otter Audits LLC. All Rights Reserved.

Missing Capacity Check OS-ESP-SUG-01

Description

The current `From<Vec<T>>` implementation for `SmallVec<L, T>` does not check whether the vector's length fits within the bounds of the length prefix type `L`. This may silently produce an oversized `SmallVec`, resulting in invalid or incompatible serialization and potential deserialization failures.

>_ *src/utils/small_vec.rs* RUST

```rust
impl<L, T> From<Vec<T>> for SmallVec<L, T> {
    fn from(val: Vec<T>) -> Self {
        Self(val, PhantomData)
    }
}
```

Remediation

Add a check in the `From<Vec<T>>` implementation for `SmallVec<L, T>` to ensure that the input vector's length does not exceed the maximum value representable by `L`.

Patch

Resolved in 2efdbaf.

© 2025 Otter Audits LLC. All Rights Reserved.

Code Maturity OS-ESP-SUG-02

Description

1. The current `_` match arm in `precompiles::get_signature_payload` implicitly accepts any unknown precompile ID, risking incorrect offset parsing and signature verification. Instead, the code should explicitly reject unrecognized precompile IDs. This ensures correctness and avoids silent failures.
2. The origin string in `client_data_json::reconstruct_client_data_json` is directly inserted into the JSON without escaping, which may result in malformed JSON if it contains quotes (`"`). To prevent this, the origin should be sanitized by escaping quotes.

>_ src/utils/signatures/p256_webauthn/client_data_json.rs RUST

```rust
/// Reconstructs the clientDataJson from the params, rp_id and challenge
pub fn reconstruct_client_data_json([...]) -> Vec<u8> {
    [...]
    let mut json_bytes: Vec<u8> = Vec::with_capacity(256);
    json_bytes.extend_from_slice(b"{\"type\":\"");
    json_bytes.extend_from_slice(type_str.as_bytes());
    json_bytes.extend_from_slice(b"\",\"challenge\":\"");
    json_bytes.extend_from_slice(challenge_b64url.as_bytes());
    json_bytes.extend_from_slice(b"\",\"origin\":\"");
    json_bytes.extend_from_slice(origin.as_bytes());
    json_bytes.extend_from_slice(b"\",\"crossOrigin\":");
    json_bytes.extend_from_slice(cross_origin.as_bytes());
    [...]
}
```

3. Within `initialize_external_account::process_initialize_external_account`, there is a typographical error in the call to `verfiy_initialization_payload`, where *verify* is spelled as *verfiy*. The call should be updated to `verify_initialization_payload` to ensure the correct method is invoked.

>_ src/instructions/initialize_external_account.rs RUST

```rust
// Processes the initialize external account instruction
pub fn process_initialize_external_account(accounts: &[AccountInfo], data: &[u8]) ->
    ↪  ProgramResult {
    [...]
    // Verify the initialization payload (since we depend on the contents of the
    // account to exist, we do this step last)
    externally_owned_account.verfiy_initialization_payload([...])?;
    Ok(())
}
```

© 2025 Otter Audits LLC. All Rights Reserved.

4. `sha256::hash_into` should be marked as unsafe because it is possible to invoke it with parameters that render its behavior unsafe, such as an invalid `out` parameter.

>_ *src/utils/sha256.rs* RUST

```rust
#[inline(always)]
#[allow(unused)]
// Simple wrapper around the hashv syscall
pub fn hash_into(data: &[&[u8]], out: *mut [u8; 32]) {
    #[cfg(target_os = "solana")]
    unsafe {
        pinocchio::syscalls::sol_sha256(
            data as *const _ as *const u8,
            data.len() as u64,
            out as *mut u8,
        );
    }
}
```

Remediation

Implement the above-mentioned suggestions.

Patch

1. Resolved in 36c3917.
2. Resolved in 4601c85.
3. Resolved in 4601c85.
4. Resolved in 8351898.

© 2025 Otter Audits LLC. All Rights Reserved.

Unutilized Code

OS-ESP-SUG-03

Description

1. The `system_program` account in `InitializeAccounts` is not explicitly validated against Solana's official system program ID. While it is only implicitly utilized through instruction such as Transfer and Assign, those may operate on a similarly named account from the remaining accounts. To ensure correctness and security, it will be safer to explicitly verify that `system_program.key` matches `system_program::ID`.

   >_ *src/instructions/initialize_external_account.rs* RUST

   ```rust
   // Sanitized and checked context for initialization
   pub struct InitializeExternalAccountContext<'a, T: ExternallySignedAccountData> {
       pub slothash: [u8; 32],
       pub accounts: InitializeAccounts<'a, T>,
       pub externally_signed_account_seeds: T::AccountSeeds,
       pub signature_scheme_specific_initialization_data: T::ParsedInitializationData,
       pub session_key: Option<SessionKey>,
   }
   ```

2. `authenticator_data::verify_and_parse_public_key` is never invoked and may thus be removed to avoid dead code.

Remediation

Remove the unutilized code instances highlighted above.

Patch

1. Resolved in ffe8c57.
2. Resolved in 71bef18.

© 2025 Otter Audits LLC. All Rights Reserved.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

© 2025 Otter Audits LLC. All Rights Reserved.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

© 2025 Otter Audits LLC. All Rights Reserved.