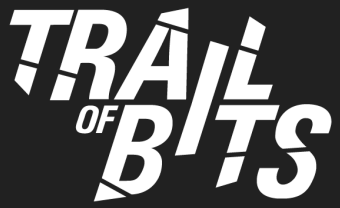
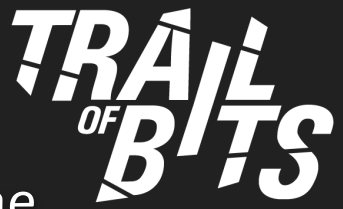
Dylint Can Help You Write More Secure Solana Contracts

Samuel Moelius
sam.moelius@trailofbits.com



Trail of Bits



- We specialize in high-end security technologies, and one of our areas of focus is blockchain.
- We apply real-world research to speed security reviews.
- Tools of ours you might know include:
  - Slither - a static analyzer for Solidity
  - Echidna - a fuzzer for Ethereum
  - Amarna - a static analyzer for Cairo (StarkNet)
  - Tealer - a static analyzer for Teal (Algorand)

Overview

- The Sealevel Attacks
- Dylint
- Lints inspired by the Sealevel Attacks
- Try them!

The Sealevel Attacks

Examples of common exploits unique to the Solana programming model and recommended idioms for avoiding these attacks using the Anchor framework

Anchor

- README description:

  Anchor is a framework for Solana's Sealevel runtime providing several convenient developer tools for writing smart contracts.

- Anchor has both on-chain and off-chain components:
  - On-chain (e.g., types, traits, macros) to assist in writing Solana programs
  - Off-chain to assist in testing Solana programs

The Sealevel Attacks

- Today, the repository contains 11 examples.
- Each example has three versions:
  - `insecure`: a program exhibiting a vulnerability
  - `secure`: a program that mitigates the vulnerability
  - `recommended`: "*the idiomatic version of secure as encouraged by the anchor framework*" – @armaniferrante*

* Armani Ferrante is the creator of the Anchor framework.

The Sealevel Attacks

- 0-signer-authorization
- 1-account-data-matching
- 2-owner-checks
- 3-type-cosplay
- 4-initialization
- 5-arbitrary-cpi
- 6-duplicate-mutable-accounts
- 7-bump-seed-canonicalization
- 8-pda-sharing
- 9-closing-accounts
- 10-sysvar-address-checking

The Sealevel Attacks

- 0-signer-authorization
- 1-account-data-matching
- 2-owner-checks
- 3-type-cosplay
- 4-initialization
- 5-arbitrary-cpi
- 6-duplicate-mutable-accounts
- 7-bump-seed-canonicalization
- 8-pda-sharing
- 9-closing-accounts
- 10-sysvar-address-checking

2-owner-checks

insecure

```
pub fn log_message(
    ctx: Context<LogMessage>
) -> ProgramResult {
    let token = SplTokenAccount::unpack(
        &ctx.accounts.token.data.borrow()
    )?;
    if ctx.accounts.authority.key != &token.owner {
        return Err(ProgramError::InvalidAccountData);
    }
    msg!(
        "Your account balance is: {}",
        token.amount
    );
    Ok(())
}
```

secure

```
pub fn log_message(
    ctx: Context<LogMessage>
) -> ProgramResult {
    let token = SplTokenAccount::unpack(
        &ctx.accounts.token.data.borrow()
    )?;
    if ctx.accounts.token.owner != &spl_token::ID {
        return Err(ProgramError::InvalidAccountData);
    }
    if ctx.accounts.authority.key != &token.owner {
        return Err(ProgramError::InvalidAccountData);
    }
    msg!(
        "Your account balance is: {}",
        token.amount
    );
    Ok(())
}
```

2-owner-checks

insecure

```rust
pub fn log_message(
    ctx: Context<LogMessage>
) -> ProgramResult {
    let token = SplTokenAccount::unpack(
        &ctx.accounts.token.data.borrow()
    )?;
    if ctx.accounts.authority.key != &token.owner {
        return Err(ProgramError::InvalidAccountData);
    }
    msg!(
        "Your account balance is: {}",
        token.amount
    );
    Ok
}
```

secure

```rust
pub fn log_message(
    ctx: Context<LogMessage>
) -> ProgramResult {
    let token = SplTokenAccount::unpack(
        &ctx.accounts.token.data.borrow()
    )?;
    if ctx.accounts.token.owner != &spl_token::ID {
        return Err(ProgramError::InvalidAccountData);
    }
    if ctx.accounts.authority.key != &token.owner {
        return Err(ProgramError::InvalidAccountData);
    }
    msg!(
        "Your account balance is: {}",
```

The “insecure” version doesn’t check `ctx.accounts.token.owner`.
The “secure” version does.

2-owner-checks

insecure

```
pub fn log_message(
    ctx: Context<LogMessage>
) -> ProgramResult {
    let token = SplTokenAccount::unpack(
        &ctx.accounts.token.data.borrow()
    )?;
    if ctx.accounts.authority.key != &token.owner {
        return Err(ProgramError::InvalidAccountData);
    }
    msg!(
        "Your account balance is: {}",
        token.amount
    );
    Ok(())
}
```

secure

```
pub fn log_message(
    ctx: Context<LogMessage>
) -> ProgramResult {
    let token = SplTokenAccount::unpack(
        &ctx.accounts.token.data.borrow()
    )?;
    if ctx.accounts.token.owner != &spl_token::ID {
        return Err(ProgramError::InvalidAccountData);
    }
    if ctx.accounts.authority.key != &token.owner {
        return Err(ProgramError::InvalidAccountData);
    }
    msg!(
        "Your account balance is: {}",
        token.amount
    );
    Ok(())
}
```

2-owner-checks

insecure

```
pub fn log_message(
    ctx: Context<LogMessage>
) -> ProgramResult {
    let token = SplTokenAccount::unpack(
        &ctx.accounts.token.data.borrow()
    )?;
    if ctx.accounts.authority.key != &token.owner {
        return Err(ProgramError::InvalidAccountData);
    }
    msg!(
        "Your account
        token.amount
    );
    Ok(())
}
```

This is not the “owner” of interest.

secure

```
pub fn log_message(
    ctx: Context<LogMessage>
) -> ProgramResult {
    let token = SplTokenAccount::unpack(
        &ctx.accounts.token.data.borrow()
    )?;
    if ctx.accounts.token.owner != &spl_token::ID {
        return Err(ProgramError::InvalidAccountData);
    }
    if ctx.accounts.authority.key != &token.owner {
        return Err(ProgramError::InvalidAccountData);
    }
        "Your account balance is: {}",
        token.amount
    );
    Ok(())
}
```

2-owner-checks

insecure

```
pub fn log_message(
    ctx: Context<LogMessage>
) -> ProgramResult {
    let token = SplTokenAccount::unpack(
        &ctx.accounts.token.data.borrow()
    )?;
    if ctx.accounts.authority.key != &token.owner {
        return Err(ProgramError::InvalidAccountData);
    }
    msg!(
        "Your account balance is: {}",
        token.amount
    );
    Ok(())
}
```

secure

```
pub fn log_message(
    ctx: Context<LogMessage>
) -> ProgramResult {
    let token = SplTokenAccount::unpack(
        &ctx.accounts.token.data.borrow()
    )?;
    if ctx.accounts.token.owner != &spl_token::ID {
        return Err(ProgramError::InvalidAccountData);
    }
    if ctx.accounts.authority.key != &token.owner {
        return Err(ProgramError::InvalidAccountData);
    }
    msg!(
        "Your account balance is: {}",
        token.amount
    );
    Ok(())
}
```

2-owner-checks

Both version versions check the `spl_token` “owner.”

But only the “secure” version checks the `solana_program` “owner.”

```rust
// solana_program::account_info::AccountInfo
pub struct AccountInfo<'a> {
    pub key: &'a Pubkey,
    pub is_signer: bool,
    pub is_writable: bool,
    pub lamports: Rc<RefCell<&'a mut u64>>,
    pub data: Rc<RefCell<&'a mut [u8]>>,
    pub owner: &'a Pubkey,
    pub executable: bool,
    pub rent_epoch: Epoch,
}
```

```rust
// spl_token::state::Account
pub struct Account {
    pub mint: Pubkey,
    pub owner: Pubkey,
    pub amount: u64,
    pub delegate: COption<Pubkey>,
    pub state: AccountState,
    pub is_native: COption<u64>,
    pub delegated_amount: u64,
    pub close_authority: COption<Pubkey>,
}
```

@pencilflip's documentation

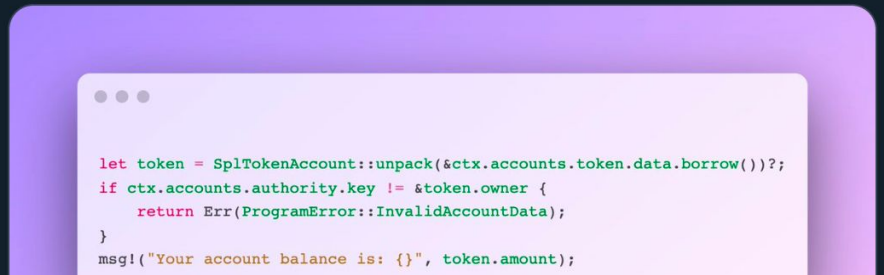
- The Sealevel Attacks' documentation is somewhat sparse...
- But @pencilflip wrote a fantastic Twitter thread describing them:

3) Checking account ownership

Make sure the passed-in accounts are owned by the correct program.

For example, if your instruction expects a token account, it should be owned by the token program.

Don't do this—this code doesn't check to make sure the token account is owned by the SPL token program, so it could be invalid.



Dylint

A tool for running Rust lints from dynamic libraries

Lint

- From Wikipedia's Lint (software):

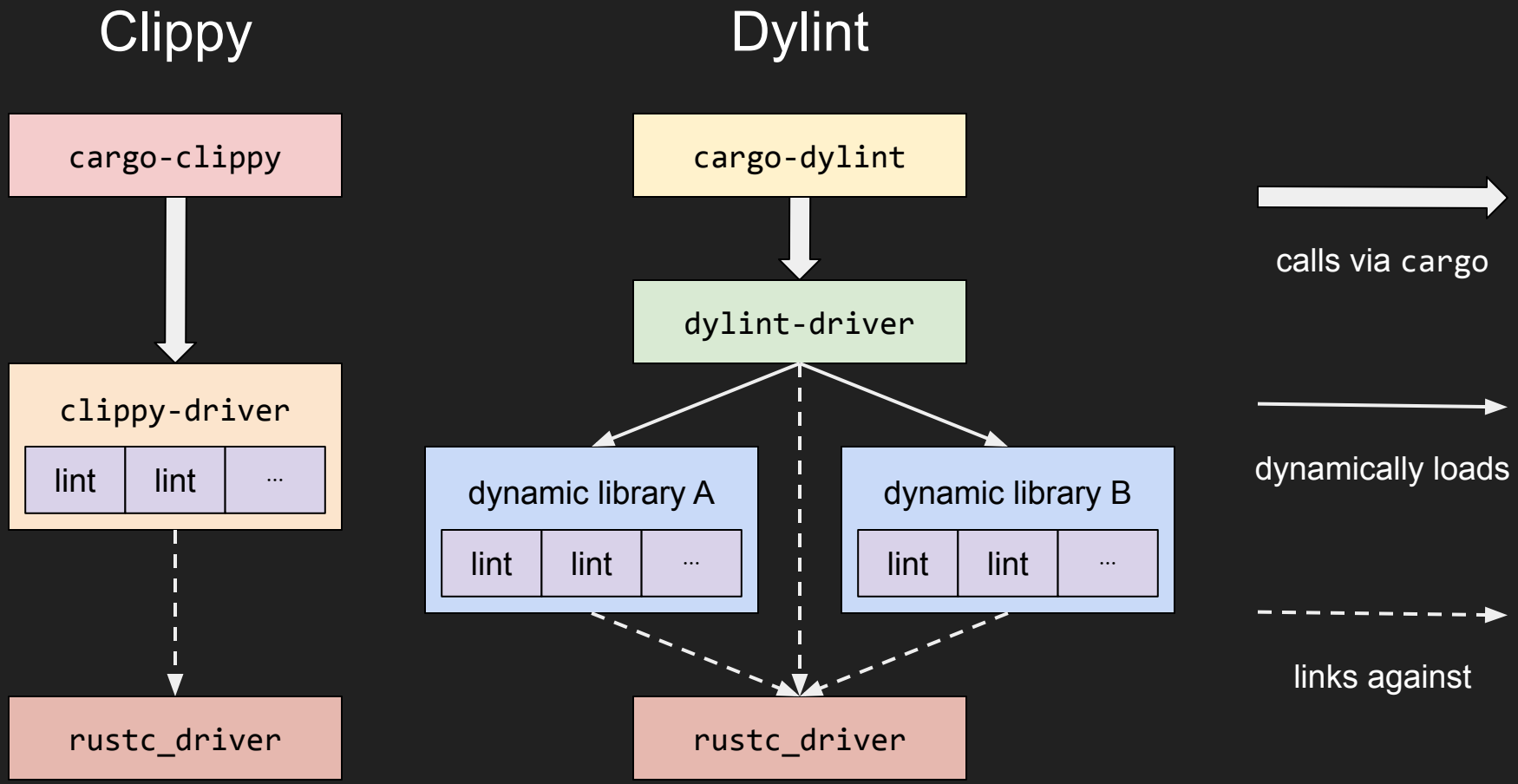
  > *Lint, or a linter, is a static code analysis tool used to flag programming errors, bugs, stylistic errors and suspicious constructs.*

- Example Rust compiler lints:
  - `unreachable_code`, `unused_imports`, `while_true`
- Example Clippy* lints:
  - `too_many_arguments`, `from_over_into`, `redundant_closure`

* Rust's de facto linting tool.

Dylint

- Dylint is similar to Clippy, but...
  - Clippy runs a predetermined, static set of lints.
  - Dylint runs lints from dynamic libraries named by the user.
- Dylint allows developers to maintain their own personal lint collections.


Clippy
Dylint
cargo-clippy
clippy-driver
lint
lint
...
rustc_driver
cargo-dylint
dylint-driver
dynamic library A
lint
lint
...
dynamic library B
lint
lint
...
rustc_driver
calls via cargo
dynamically loads
links against

- Dylint provides a `declare_late_lint!` macro to facilitate writing "one lint libraries."
- A typical Dylint library `lib.rs` has this structure:

```
dylint_linting::declare_late_lint! {
    /// Bad practice documentation
    pub BAD_PRACTICE,
    Warn,
    "bad practice description"
}

impl LateLintPass<'_> for BadPractice {
    fn check_thing(
        &mut self,
        cx: &LateContext<'_>,
        ...
    ) {
        ...
    }
}
```

Writing a Dylint lint

- Writing a Dylint lint...
  - Is essentially no different than writing a Clippy lint...
  - Is essentially no different than writing a Rust compiler lint.
- In each case, the APIs used (e.g., `LateLintPass`) are the same.

Lints inspired by the Sealevel Attacks

Status

- `0-signer-authorization`
  ✅ `missing_signer_check`
- `1-account-data-matching`
  ❌ No clear lintable condition
- `2-owner-checks`
  ✅ `missing_owner_check`
- `3-type-cosplay`
  ✅ `type_cosplay`
- `4-initialization`
  ❌ No clear lintable condition
- `5-arbitrary-cpi`
  ✅ `arbitrary_cpi`
- `6-duplicate-mutable-accounts`
  🚧 In development
- `7-bump-seed-canonicalization`
  ✅ `bump_seed_canonicalization`
- `8-pda-sharing`
  ❌ No clear lintable condition
- `9-closing-accounts`
  ✅ `insecure_account_close`
- `10-sysvar-address-checking`
  🚧 In development

Status

- `0-signer-authorization`
  ✅ `missing_signer_check`
- `1-account-data-matching`
  ❌ No clear lintable condition
- `2-owner-checks`
  ✅ `missing_owner_check`
- `3-type-cosplay`
  ✅ `type_cosplay`
- `4-initialization`
  ❌ No clear lintable condition
- `5-arbitrary-cpi`
  ✅ `arbitrary_cpi`
- `6-duplicate-mutable-accounts`
  🚧 In development
- `7-bump-seed-canonicalization`
  ✅ `bump_seed_canonicalization`
- `8-pda-sharing`
  ❌ No clear lintable condition
- `9-closing-accounts`
  ✅ `insecure_account_close`
- `10-sysvar-address-checking`
  🚧 In development

Example:
Missing owner check

Missing owner check: `declare_late_lint!`

```
dylint_linting::declare_late_lint! {
    /// **What it does:**
    ///
    /// This lint checks that for each account referenced in a program, that there is a
    /// corresponding owner check on that account. Specifically, this means that the owner
    /// field is referenced on that account.
    ///
    /// **Why is this bad?**
    ///
    /// The missing-owner-check vulnerability occurs when a program uses an account, but does
    /// not check that it is owned by the expected program. This could lead to vulnerabilities
    /// where a malicious actor passes in an account owned by program `X` when what was expected
    /// was an account owned by program `Y`. ...
    pub MISSING_OWNER_CHECK,
    Warn,
    "using an account without checking if its owner is as expected"
}
```

Missing owner check: check_fn

```
fn check_fn(
    &mut self,
    cx: &LateContext<'tcx>,
    _: FnKind<'tcx>,
    _: &'tcx FnDecl<'tcx>,
    body: &'tcx Body<'tcx>,
    span: Span,
    _: HirId,
) {
    if !span.from_expansion() {
        let accounts = get_referenced_accounts(cx, body);
        for account_expr in accounts {
            if !contains_owner_use(cx, body, account_expr) {
                span_lint(
                    cx,
                    MISSING_OWNER_CHECK,
                    account_expr.span,
                    "this Account struct is used but there is no check on its owner field",
                );
            }
        }
    }
}
```

Missing owner check: check_fn

```
fn check_fn(
    &mut self,
    cx: &LateContext<'tcx>,
    _: FnKind<'tcx>,
    _: &'tcx FnDecl<'tcx>,
    body: &'tcx Body<'tcx>,
    span:
    _: Hir
) {
    if !sp
        le
        fo
                                    cx,
                                    MISSING_OWNER_CHECK,
                                    account_expr.span,
                                    "this Account struct is used but there is no check on its owner field",
                                );
                            }
                        }
                    }
                }
```

`2-owner-checks/insecure` was insecure because it didn't check the `solana_program::account_info::AccountInfo` `owner` field.

Missing owner check: check_fn

```
fn check_fn(
    &mut self,
    cx: &LateContext<'tcx>,
    _: FnKind<'tcx>,
    _: &'tcx FnDecl<'tcx>,
    body: &'tcx Body<'tcx>,
    span: Span,
    _: HirId,
) {
    if !span.from_expansion() {
        let accounts = get_referenced_accounts(cx, body);
        for account_expr in accounts {
            if !contains_owner_use(cx, body, account_expr) {
                span_lint(
                    cx,
                    MISSING_OWNER_CHECK,
                    account_expr.span,
                    "this Account struct is used but there is no check on its owner field",
                );
            }
        }
    }
}
```

Missing owner check: check_fn

```rust
fn check_fn(
    &mut self,
    cx: &LateContext<'tcx>,
    _: FnKind<'tcx>,
    _: &'tcx FnDecl<'tcx>,
    body: &'tcx Body<'tcx>,
    span: Span,
    _: HirId,
) {
    if !span.from_expansion() {
        let accounts = get_referenced_accounts(cx, body);
        for account_expr in accounts {
            if !contains_owner_use(cx, body, account_expr) {
                span_lint(
                    cx,
                    MISSING_OWNER_CHECK,
                    account_expr.span,
                    "this Account struct is used but there is no check on its owner field",
                );
            }
        }
    }
}
```

Called on each function, method, or closure.

Missing owner check: check_fn

```
fn check_fn(
    &mut self,
    cx: &LateConte
    _: FnKind<'tcx
    _: &'tcx FnDec
    body: &'tcx Bo
    span: Span,
    _: HirId,
) {
    if !span.from_expansion() {
        let accounts = get_referenced_accounts(cx, body);
        for account_expr in accounts {
            if !contains_owner_use(cx, body, account_expr) {
                span_lint(
                    cx,
                    MISSING_OWNER_CHECK,
                    account_expr.span,
                    "this Account struct is used but there is no check on its owner field",
                );
            }
        }
    }
}
```

Collects expressions whose type is:

`solana_program::account_info::AccountInfo`

Missing owner check: check_fn

```rust
fn check_fn(
    &mut self,
    cx: &LateContext<'tcx>,
    _: FnKind<'tcx>,
    _: &'tcx FnDecl<'tcx>,
    body: &'tcx Body<'tcx>,
    span: Span,
    _: HirId,
) {
    if !span.from_expansion() {
        let accounts = get_referenced_accounts(cx, body);
        for account_expr in accounts {
            if !contains_owner_use(cx, body, account_expr) {
                span_lint(
                    cx,
                    MISSING_OWNER_CHECK,
                    account_expr.span,
                    "this Account struct is used but there is no check on its owner field",
                );
            }
        }
    }
}
```

Asks: does the function body contain an expression of the following form?

⟨*account_expr*⟩ . owner

Missing owner check: check_fn

```
fn check_fn(
    &mut self,
    cx: &LateContext<'tcx>,
    _: FnKind<'tcx>,
    _: &'tcx FnDecl<'tcx>,
    body: &'tcx Body<'tcx>,
    span: Span,
    _: HirId,
) {
    if !span.from_expansion() {
        let accounts = get_referenced_accounts(cx, body);
        for account_expr in accounts {
            if !contains_owner_use(cx, body, account_expr) {
                span_lint(
                    cx,
                    MISSING_OWNER_CHECK,
                    account_expr.span,
                    "this Account struct is used but there is no check on its owner field",
                );
            }
        }
    }
}
```

Emits a warning when the answer is “no.”

Output on 2-owner-checks/insecure:

```
warning: this Account struct is used but there is no check on its owner field
  --> src/lib.rs:14:46
   |
14 |         let token = SplTokenAccount::unpack(&ctx.accounts.token.data.borrow())?;
   |                                              ^^^^^^^^^^^^^^^^^^^
   |
   = note: `#[warn(missing_owner_check)]` on by default

warning: `owner-checks-insecure` (lib) generated 1 warning
```

Output on `2-owner-checks/secure`:

(Nothing)

Status

- `0-signer-authorization`
  ✅ `missing_signer_check`
- `1-account-data-matching`
  ❌ No clear lintable condition
- `2-owner-checks`
  ✅ `missing_owner_check`
- `3-type-cosplay`
  ✅ `type_cosplay`
- `4-initialization`
  ❌ No clear lintable condition
- `5-arbitrary-cpi`
  ✅ `arbitrary_cpi`
- `6-duplicate-mutable-accounts`
  🚧 In development
- `7-bump-seed-canonicalization`
  ✅ `bump_seed_canonicalization`
- `8-pda-sharing`
  ❌ No clear lintable condition
- `9-closing-accounts`
  ✅ `insecure_account_close`
- `10-sysvar-address-checking`
  🚧 In development

Status

- `0-signer-authorization`
  ✅ `missing_signer_check`
- `1-account-data-matching`
  ❌ No clear lintable condition
- `2-owner-checks`
  ✅ `missing_owner_check`
- `3-type-cosplay`
  ✅ `type_cosplay`
- `4-initialization`
  ❌ No clear lintable condition
- `5-arbitrary-cpi`
  ✅ `arbitrary_cpi`
- `6-duplicate-mutable-accounts`
  🚧 In development
- `7-bump-seed-canonicalization`
  ✅ `bump_seed_canonicalization`
- `8-pda-sharing`
  ❌ No clear lintable condition
- `9-closing-accounts`
  ✅ `insecure_account_close`
- `10-sysvar-address-checking`
  🚧 In development

A note on Type cosplay

3-type-cosplay

insecure

```
pub fn update_user(ctx: Context<...>) -> ProgramResult {
    let user = User::try_from_slice(
        &ctx.accounts.user.data.borrow()
    ).unwrap();
    if ctx.accounts.user.owner != ctx.program_id {
        return Err(ProgramError::IllegalOwner);
    }
    if user.authority != ctx.accounts.authority.key() {
        return Err(ProgramError::InvalidAccountData);
    }
    msg!("GM {}", user.authority);
    Ok(())
}
#[derive(BorshSerialize, BorshDeserialize)]
pub struct User {
   authority: Pubkey,
}
#[derive(BorshSerialize, BorshDeserialize)]
pub struct Metadata {
   account: Pubkey,
}
```

secure

```
pub fn update_user(ctx: Context<...>) -> ProgramResult {
    let user = User::try_from_slice(
        &ctx.accounts.user.data.borrow()
    ).unwrap();
    if ctx.accounts.user.owner != ctx.program_id {
        return Err(ProgramError::IllegalOwner);
    }
    if user.authority != ctx.accounts.authority.key() {
        return Err(ProgramError::InvalidAccountData);
    }
    msg!("GM {}", user.authority);
    Ok(())
}
#[derive(BorshSerialize, BorshDeserialize)]
pub struct User {
   discriminant: AccountDiscriminant,
   authority: Pubkey,
}
#[derive(BorshSerialize, BorshDeserialize)]
pub struct Metadata {
   discriminant: AccountDiscriminant,
...
```

3-type-cosplay

insecure

```
pub fn update_user(ctx: Context<...>) -> ProgramResult {
    let user = User::try_from_slice(
        &ctx.accounts.user.data.borrow()
    ).unwrap();
    if ctx.accounts.user.owner != ctx.program_id {
        return Err(ProgramError::IllegalOwner);
    }
    if user.authority != ctx.accounts.authority.key() {
        return Err(ProgramError::InvalidAccountData);
    }
    msg!("GM {}", user.authority);
    Ok(())
}
#[derive(BorshSerialize, BorshDeserialize)]
pub struct User {
   authority: Pubkey,
}
#[derive(BorshSerialize, BorshDeserialize)]
pub struct Metadata {
   account: Pubkey,
}
```

3-type-cosplay

insecure

```rust
pub fn update_user(ctx: Context<...>) -> ProgramResult {
    let user = User::try_from_slice(
        &ctx.accounts.user.data.borrow()
    ).unwrap();
    if ctx.accounts.user.owner != ctx.program_id {
        return Err(ProgramError::IllegalOwner);
    }
    if user.authority != ctx.accounts.authority.key() {
        return Err(ProgramError::InvalidAccountData);
    }
    msg!("GM {}", user.authority);
    Ok(())
}
#[derive(BorshSerialize, BorshDeserialize)]
pub struct User {
   authority: Pubkey,
}
#[derive(BorshSerialize, BorshDeserialize)]
pub struct Metadata {
   account: Pubkey,
}
```

insecure-anchor

```rust
pub fn update_user(ctx: Context<...>) -> ProgramResult {
    let user = User::try_from_slice(
        &ctx.accounts.user.data.borrow()
    ).unwrap();
    if ctx.accounts.user.owner != ctx.program_id {
        return Err(ProgramError::IllegalOwner);
    }
    if user.authority != ctx.accounts.authority.key() {
        return Err(ProgramError::InvalidAccountData);
    }
    msg!("GM {}", user.authority);
    Ok(())
}
#[account]
pub struct User {
   authority: Pubkey,
}
#[derive(BorshSerialize, BorshDeserialize)]
pub struct Metadata {
   account: Pubkey,
}
```

3-type-cosplay

insecure

insecure-anchor

[–] An attribute for a data structure representing a Solana account.

`#[account]` generates trait implementations for the following traits:

- `AccountSerialize`
- `AccountDeserialize`
- `AnchorSerialize`
- `AnchorDeserialize`
- `Clone`
- `Discriminator`
- `Owner`

When implementing account serialization traits the first 8 bytes are reserved for a unique account discriminator, self described by the first 8 bytes of the SHA256 of the account's Rust ident.

As a result, any calls to `AccountDeserialize`'s `try_deserialize` will check this discriminator. If it doesn't match, an invalid account was given, and the account deserialization will exit with an error.

Output on `3-type-cosplay/insecure`:

```
warning: type does not have a proper discriminant. It may be indistinguishable when deserialized.
  --> src/lib.rs:12:20
   |
12 |         let user = User::try_from_slice(&ctx.accounts.user.data.borrow()).unwrap();
   |                    ^^^^
   |
   = note: `#[warn(type_cosplay)]` on by default
   = help: add an enum with at least as many variants as there are struct definitions

warning: `type-cosplay-insecure` (lib) generated 1 warning
```

Output on `3-type-cosplay/insecure-anchor`:

```
warning: `User` type implements the `Discriminator` trait. If you are attempting to deserialize
         here, you probably want try_deserialize() instead.
  --> src/lib.rs:12:20
   |
12 |         let user = User::try_from_slice(&ctx.accounts.user.data.borrow()).unwrap();
   |                    ^^^^^^^^^^^^^^^^^^^^
   |
   = note: `#[warn(type_cosplay)]` on by default
   = help: otherwise, make sure you are accounting for this type's discriminator in your
deserialization function

warning: `type-cosplay-insecure-anchor` (lib) generated 1 warning
```

Try them!

Try them!

1. Install `cargo-dylint` and `dylint-link`:

   ```
   cargo install cargo-dylint dylint-link
   ```

2. Add the following to your workspace's `Cargo.toml` file:

   ```
   [workspace.metadata.dylint]
   libraries = [{
       git = "https://github.com/crytic/solana-lints",
       pattern = "lints/*",
   }]
   ```

3. Run cargo-dylint:

   ```
   cargo dylint --all
   ```

The guys that did the actual work

Victor Wei

Andrew Haberlandt

Solana Lints
https://github.com/crytic/solana-lints

Dylint
https://github.com/trailofbits/dylint

The Sealevel Attacks
https://github.com/coral-xyz/sealevel-attacks

Thank you. Questions?

Samuel Moelius
sam.moelius@trailofbits.com