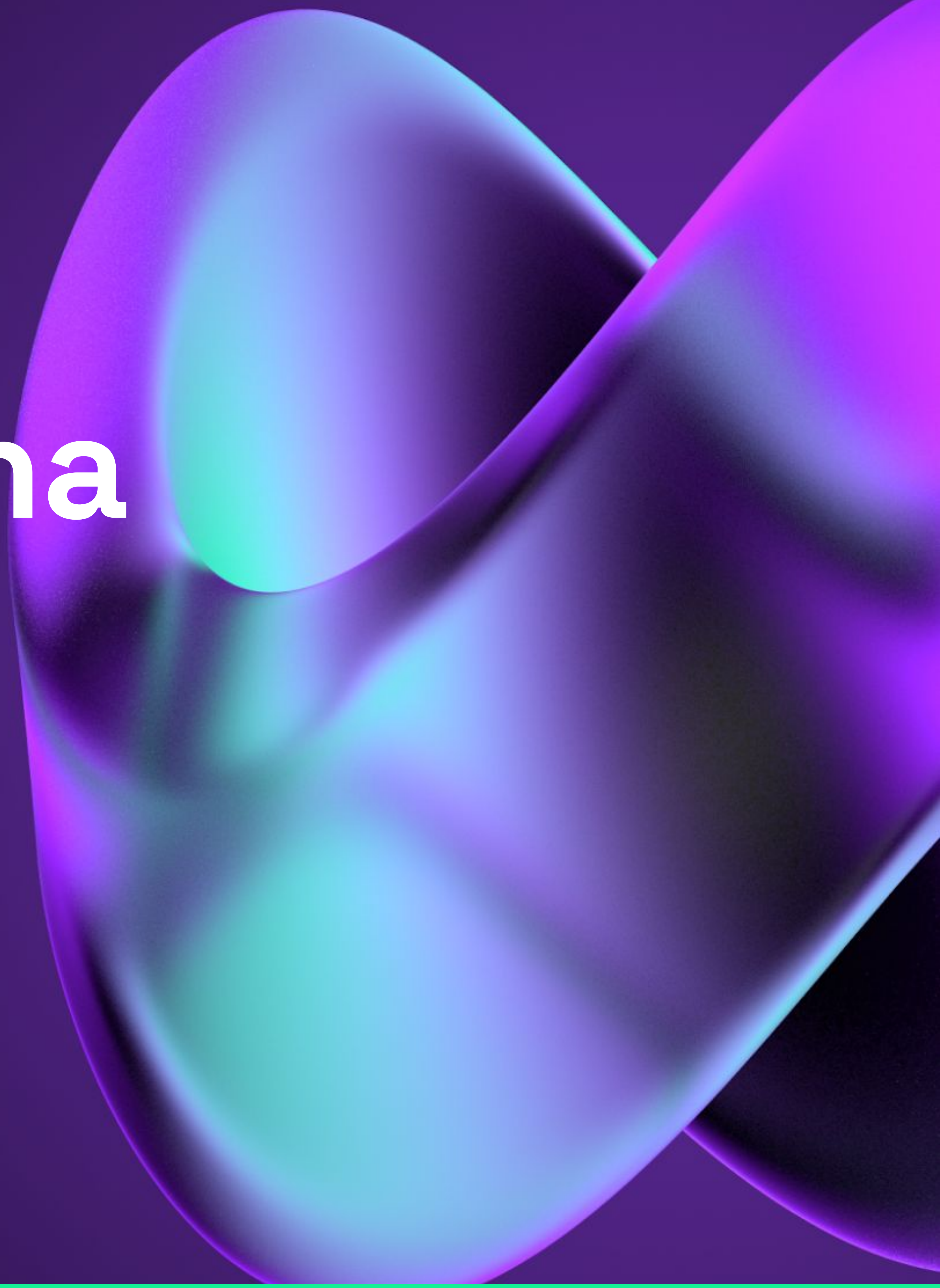
ackee blockchain security

School of Solana

LECTURE 7

Security



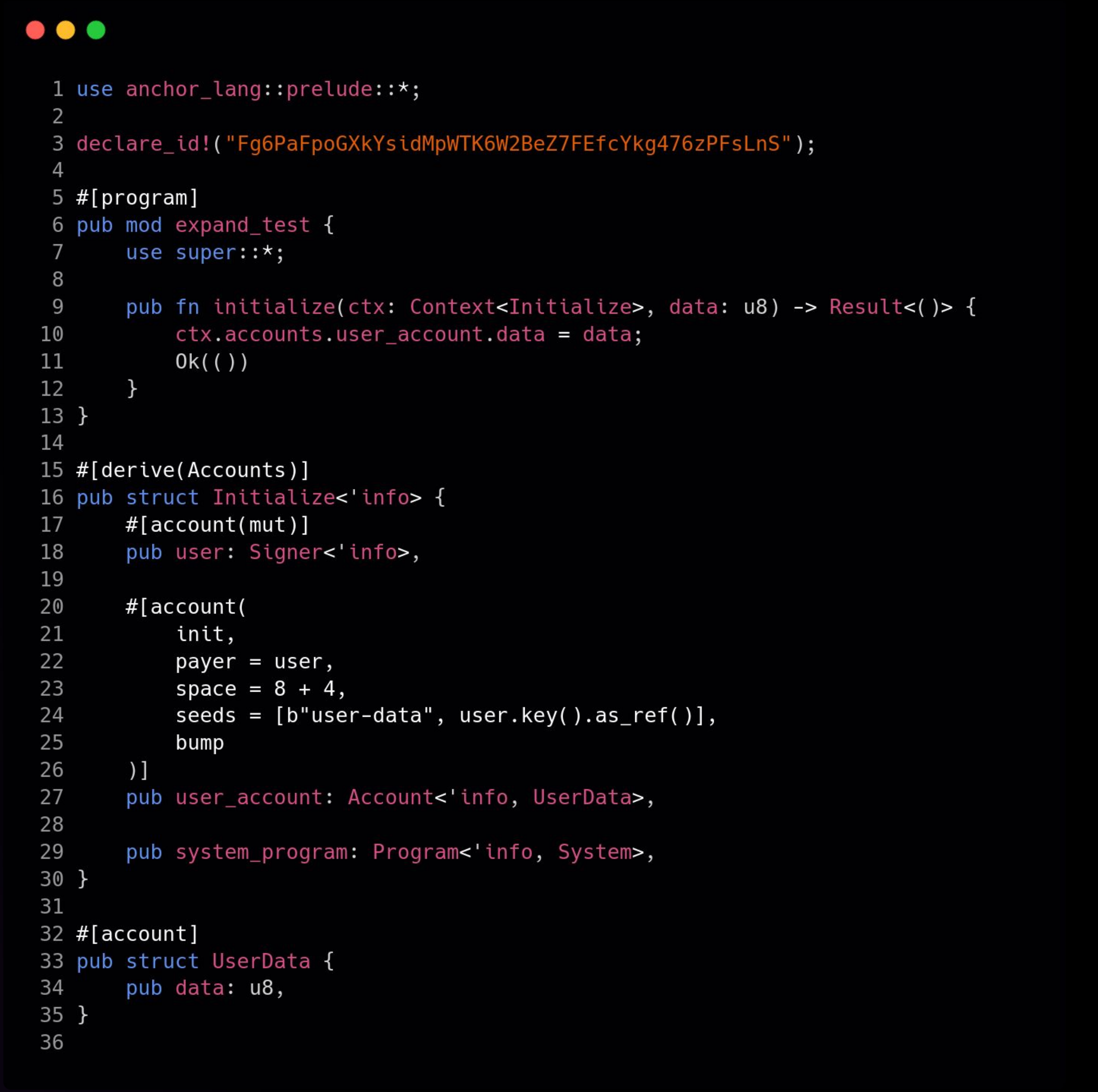
Adam Hrazdira

Motivation

- In 2022, over $4B were lost in the crypto space.
- Over $500M were stolen on Solana ecosystem.
  - Wormhole ~ $338M
  - Cashio ~ $52M
  - Crema Finance ~ $8M
  - Mango Markets ~ $116M

Basic Security Tips

- Use Anchor in most cases.
  - Saves a lot of boilerplate code.
  - Most Solana specific checks within Context<> structs.
  - Easier for others to review your code.



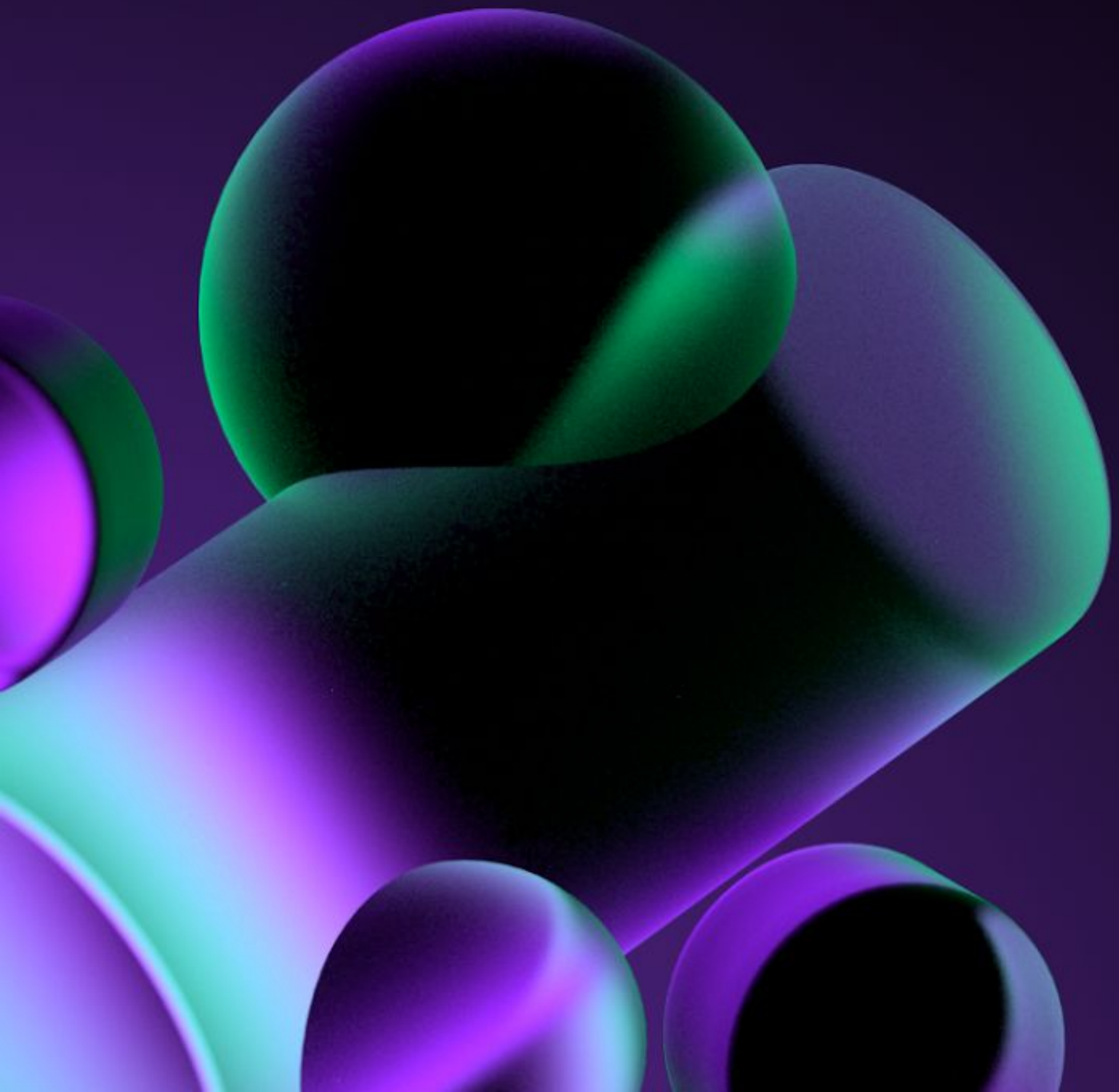
```rust
use anchor_lang::prelude::*;

declare_id!("Fg6PaFpoGXkYsidMpWTK6W2BeZ7FEfcYkg476zPFsLnS");

#[program]
pub mod expand_test {
    use super::*;

    pub fn initialize(ctx: Context<Initialize>, data: u8) -> Result<()> {
        ctx.accounts.user_account.data = data;
        Ok(())
    }
}

#[derive(Accounts)]
pub struct Initialize<'info> {
    #[account(mut)]
    pub user: Signer<'info>,

    #[account(
        init,
        payer = user,
        space = 8 + 4,
        seeds = [b"user-data", user.key().as_ref()],
        bump
    )]
    pub user_account: Account<'info, UserData>,

    pub system_program: Program<'info, System>,
}

#[account]
pub struct UserData {
    pub data: u8,
}

```

- Run `cargo expand` to expand macros.

Basic Security Tips

- Test extensively.
  - Focus on unhappy path scenarios. Think like a hacker.
- Have your project audited.
  - We all make mistakes.

Common Security Exploits



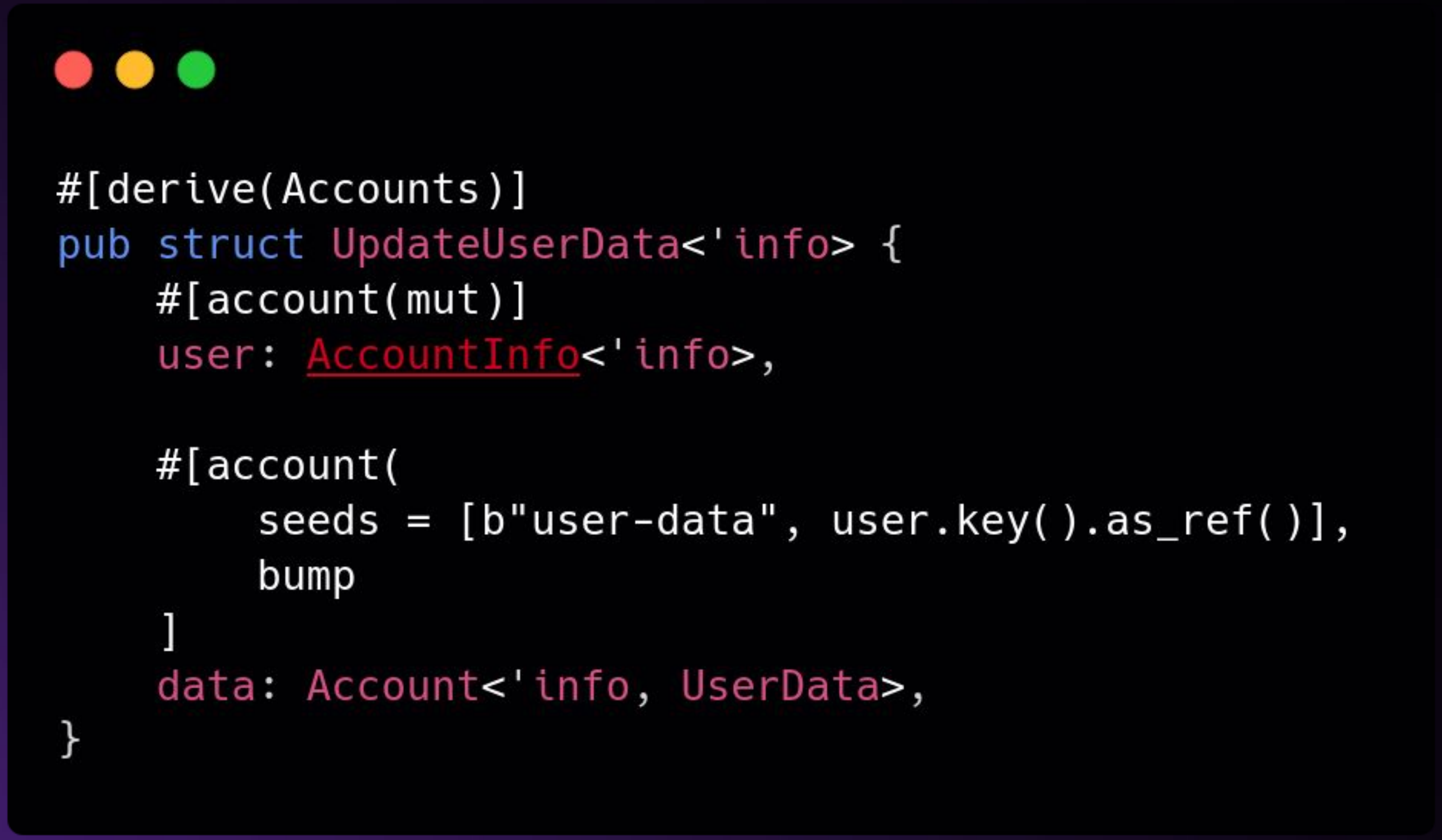
ackee | blockchain security

Common Security Exploits

1. Signer check
2. Address check
3. Ownership check
4. Arbitrary CPI
5. Math & logic issues
6. Reinitialization and revival attacks
7. Other

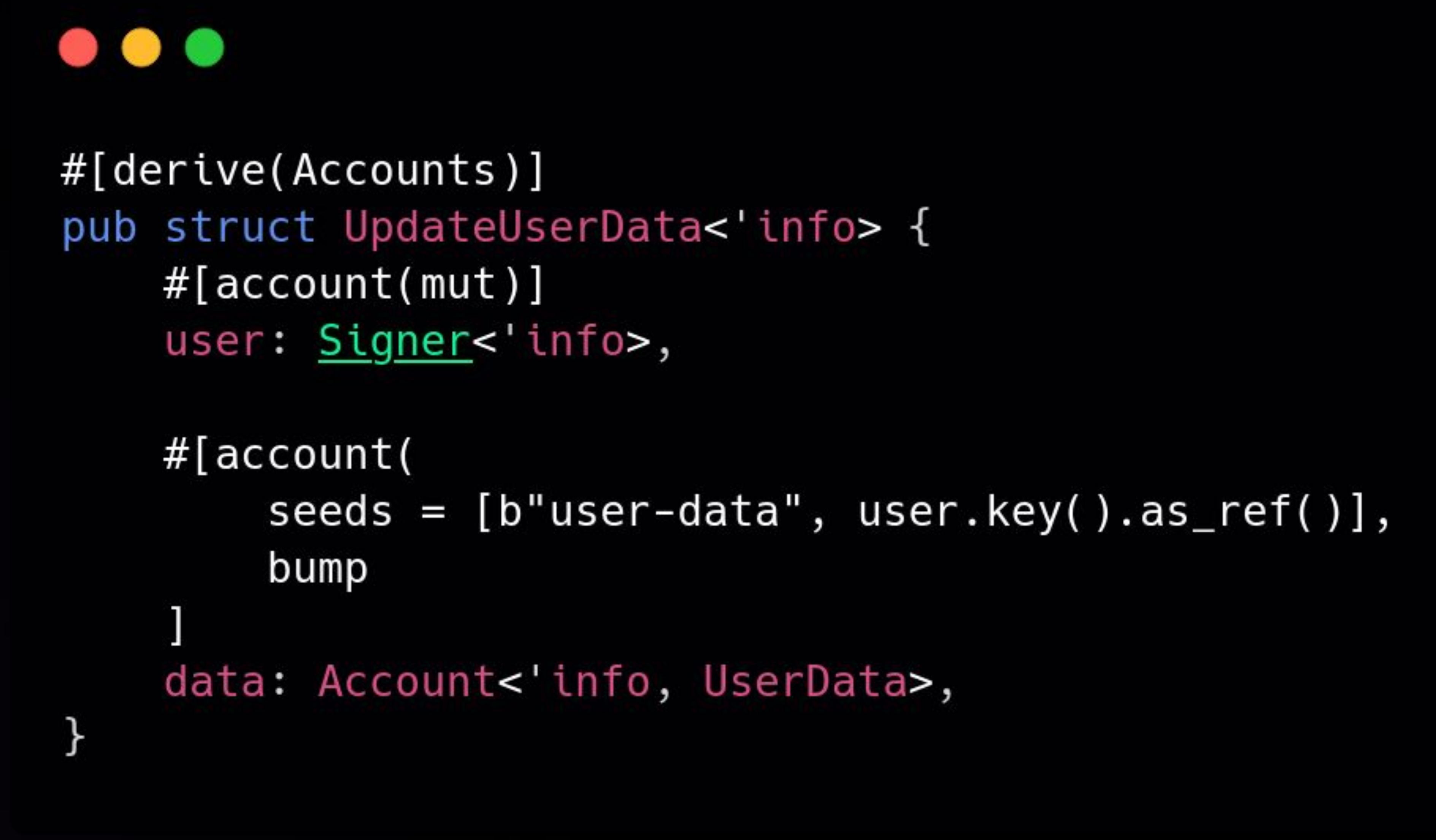
#1 Signer Check

- Verify that the right parties have signed a transaction.



```
#[derive(Accounts)]
pub struct UpdateUserData<'info> {
    #[account(mut)]
    user: AccountInfo<'info>,

    #[account(
        seeds = [b"user-data", user.key().as_ref()],
        bump
    ]
    data: Account<'info, UserData>,
}
```

Not signed and insecure.
Anyone who knows the user Pubkey can send this transaction.

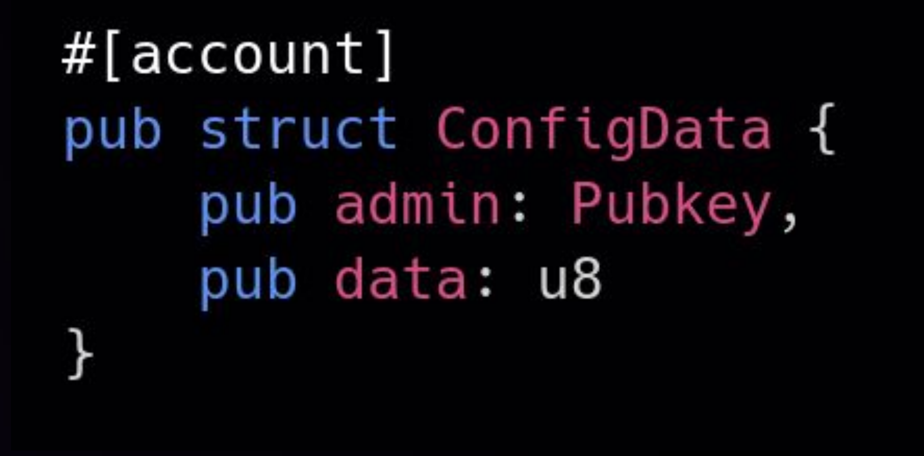


```
#[derive(Accounts)]
pub struct UpdateUserData<'info> {
    #[account(mut)]
    user: Signer<'info>,

    #[account(
        seeds = [b"user-data", user.key().as_ref()],
        bump
    ]
    data: Account<'info, UserData>,
}
```

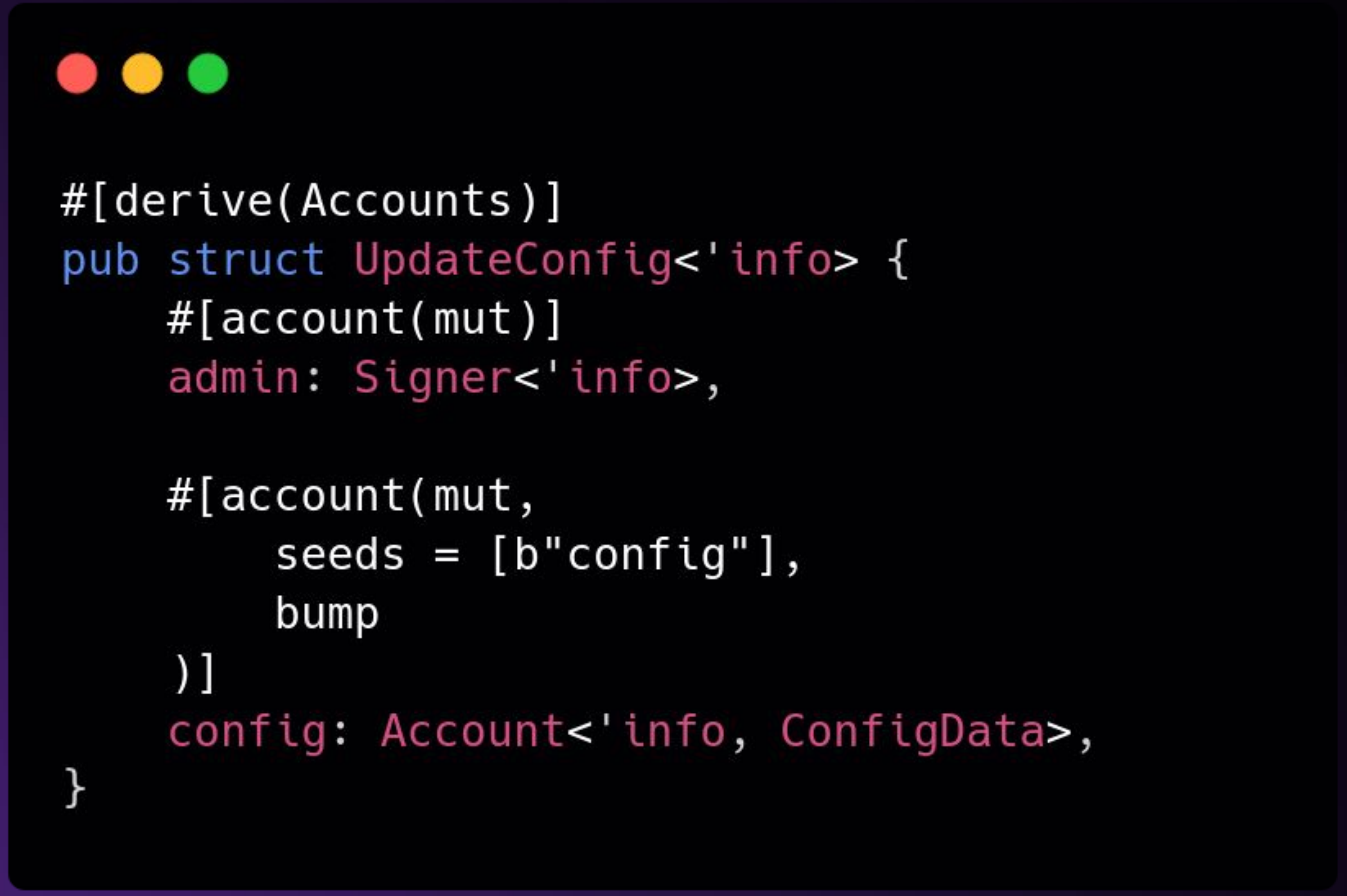
Signed and secure. Use Anchor’s Signer<’info> type.

#2 Address Check

- Verify that an account has the expected address (public key).

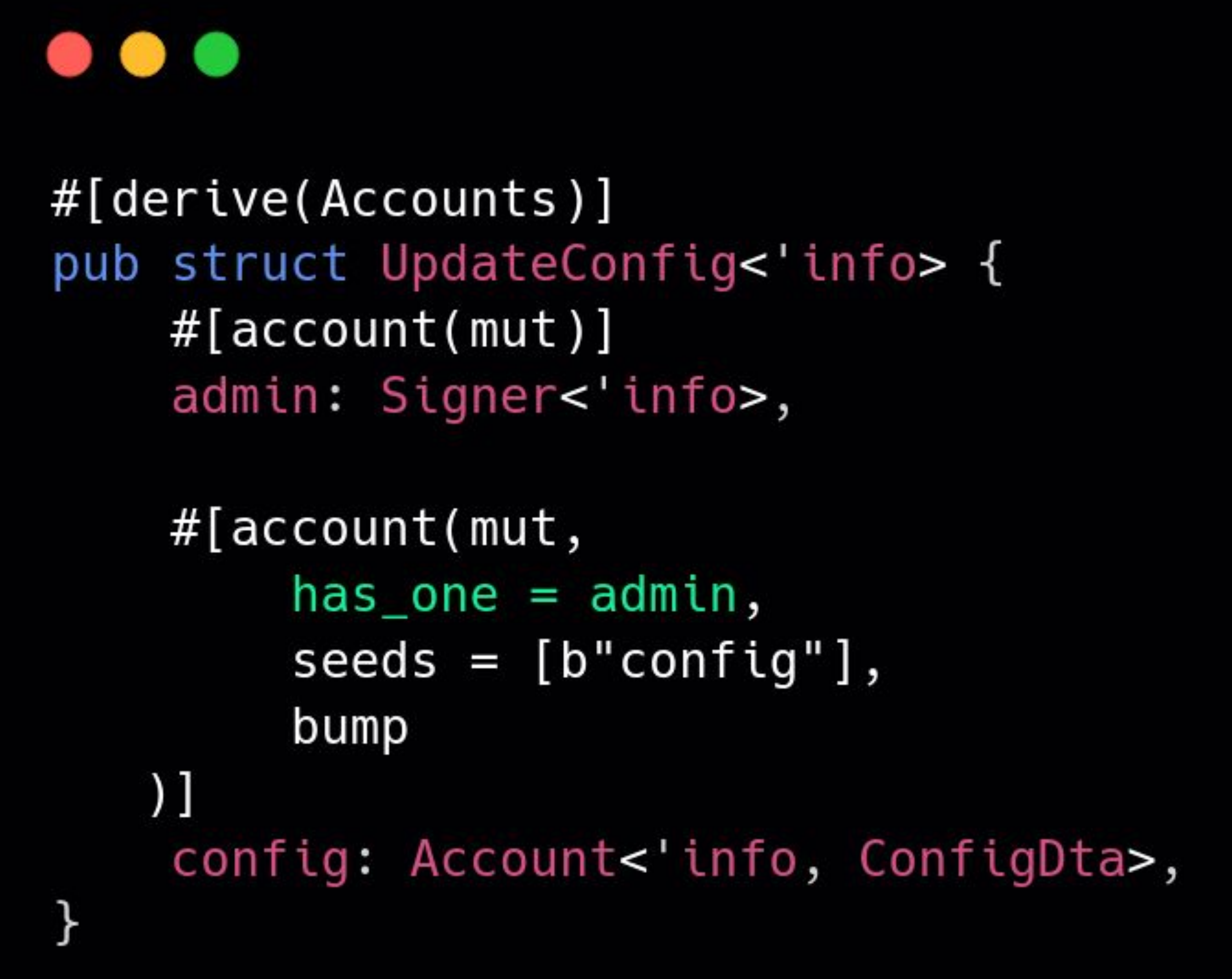


```
#[account]
pub struct ConfigData {
    pub admin: Pubkey,
    pub data: u8
}
```



```
#[derive(Accounts)]
pub struct UpdateConfig<'info> {
    #[account(mut)]
    admin: Signer<'info>,

    #[account(mut,
        seeds = [b"config"],
        bump
    )]
    config: Account<'info, ConfigData>,
}
```

Insecure.
Signer is not associated with the config account's admin Pubkey. Anyone can sign the transaction.

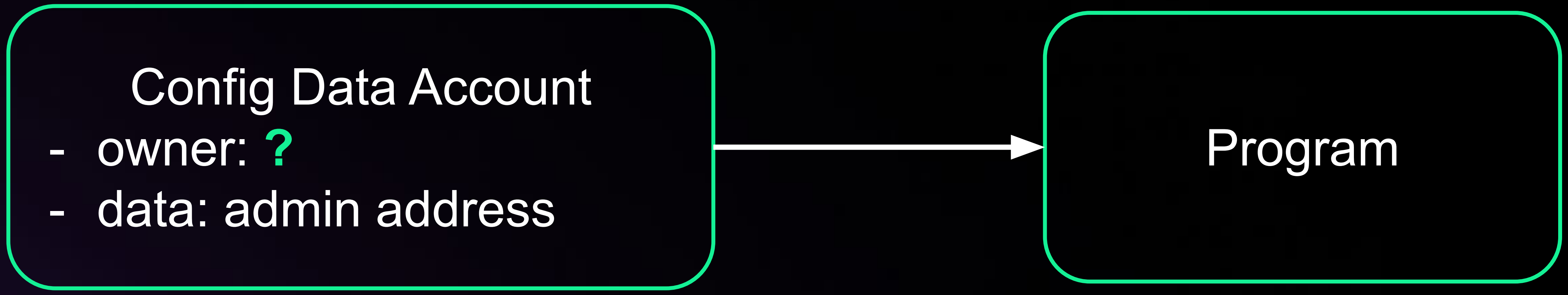


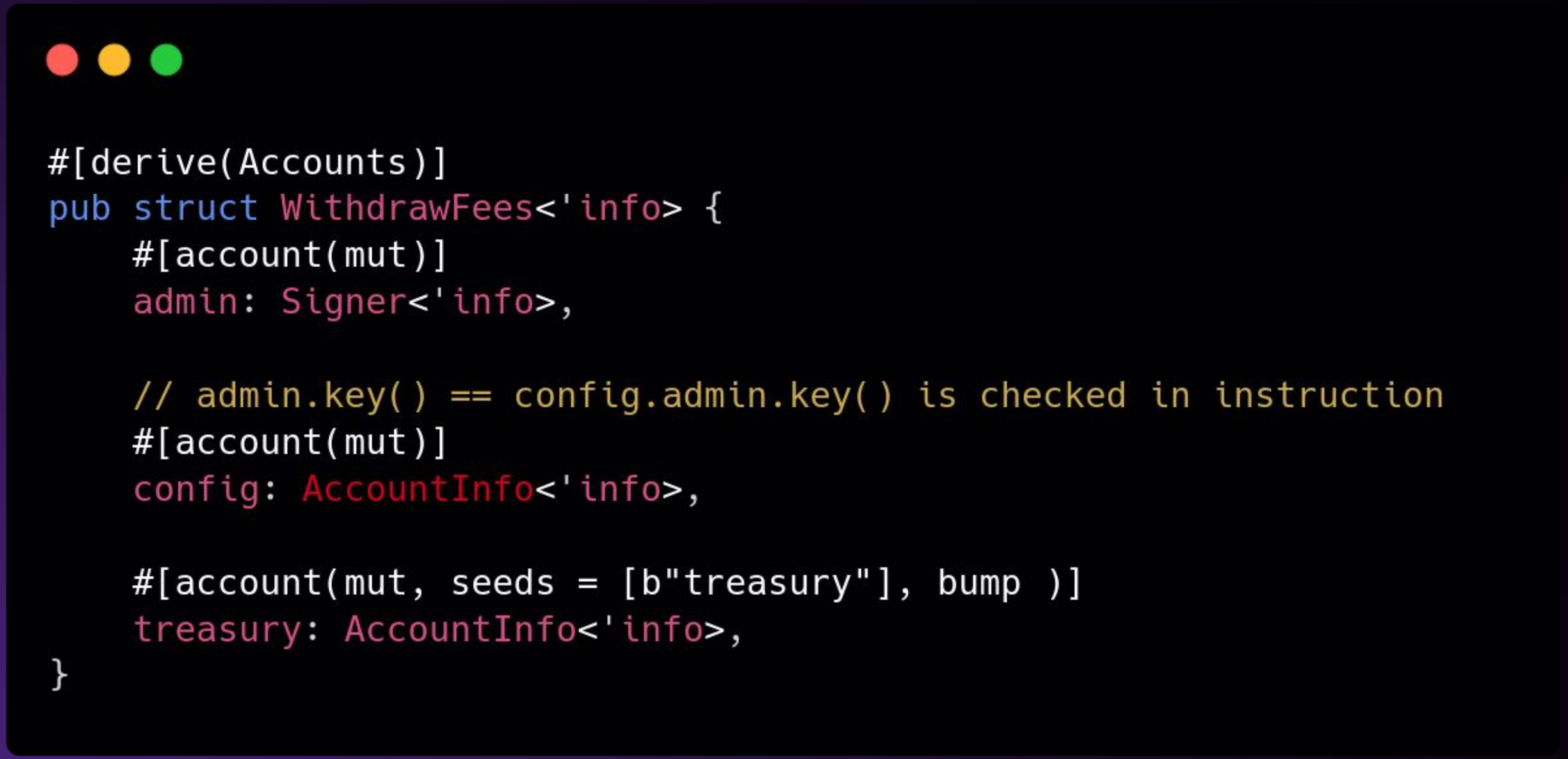
```
#[derive(Accounts)]
pub struct UpdateConfig<'info> {
    #[account(mut)]
    admin: Signer<'info>,

    #[account(mut,
        has_one = admin,
        seeds = [b"config"],
        bump
    )]
    config: Account<'info, ConfigDta>,
}
```

Secure. Use for example has_one constraint.

#3 Ownership Check

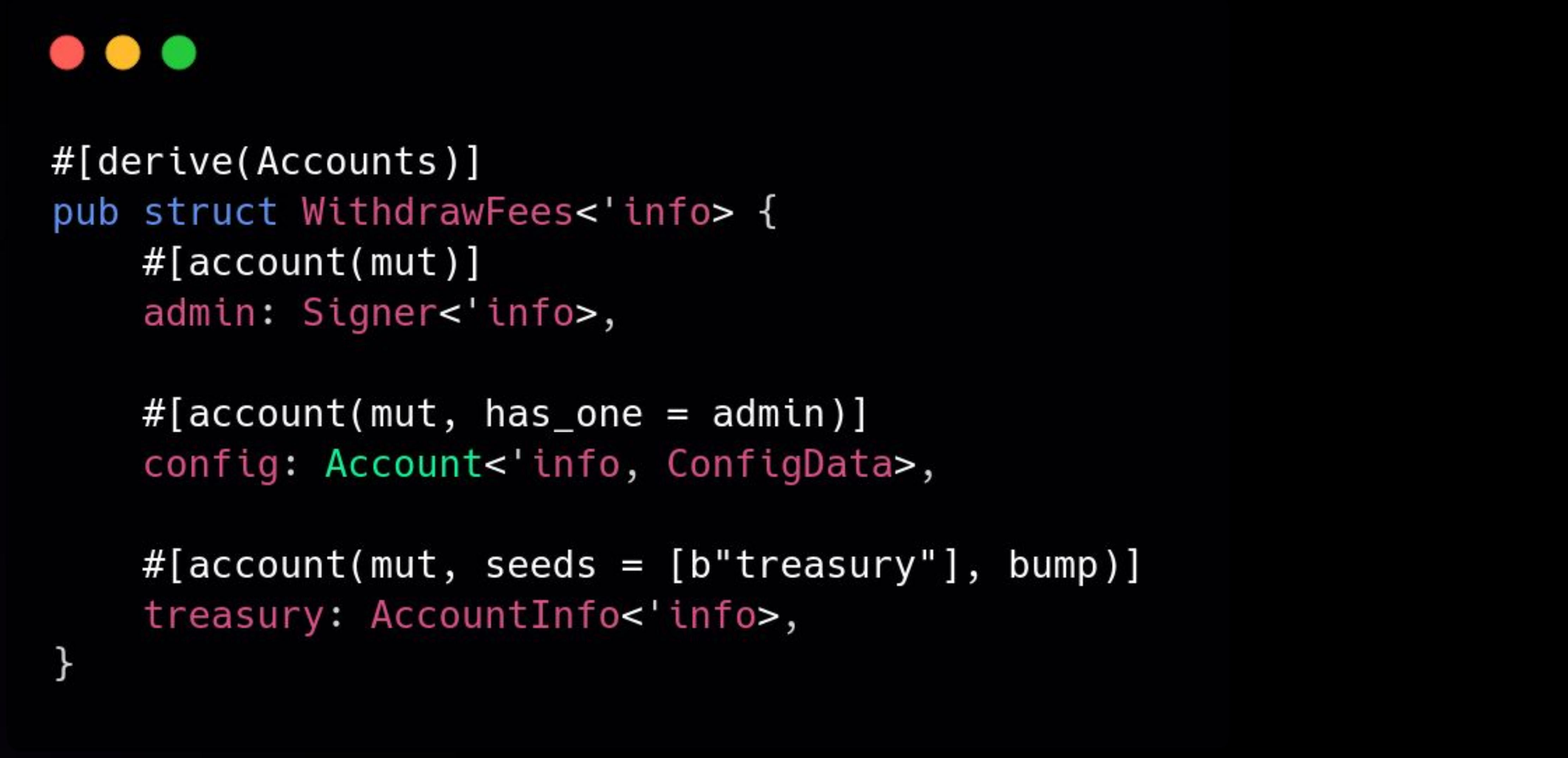
- Verify that an account is owned by the expected program.





```
#[derive(Accounts)]
pub struct WithdrawFees<'info> {
    #[account(mut)]
    admin: Signer<'info>,

    // admin.key() == config.admin.key() is checked in instruction
    #[account(mut)]
    config: AccountInfo<'info>,

    #[account(mut, seeds = [b"treasury"], bump )]
    treasury: AccountInfo<'info>,
}
```

Insecure. Config account might be owned by another program and any account with the required data structure might be supplied.



```
#[derive(Accounts)]
pub struct WithdrawFees<'info> {
    #[account(mut)]
    admin: Signer<'info>,

    #[account(mut, has_one = admin)]
    config: Account<'info, ConfigData>,

    #[account(mut, seeds = [b"treasury"], bump)]
    treasury: AccountInfo<'info>,
}
```

Secure. Use Anchor’s Account<’info, T> type that checks the owner.

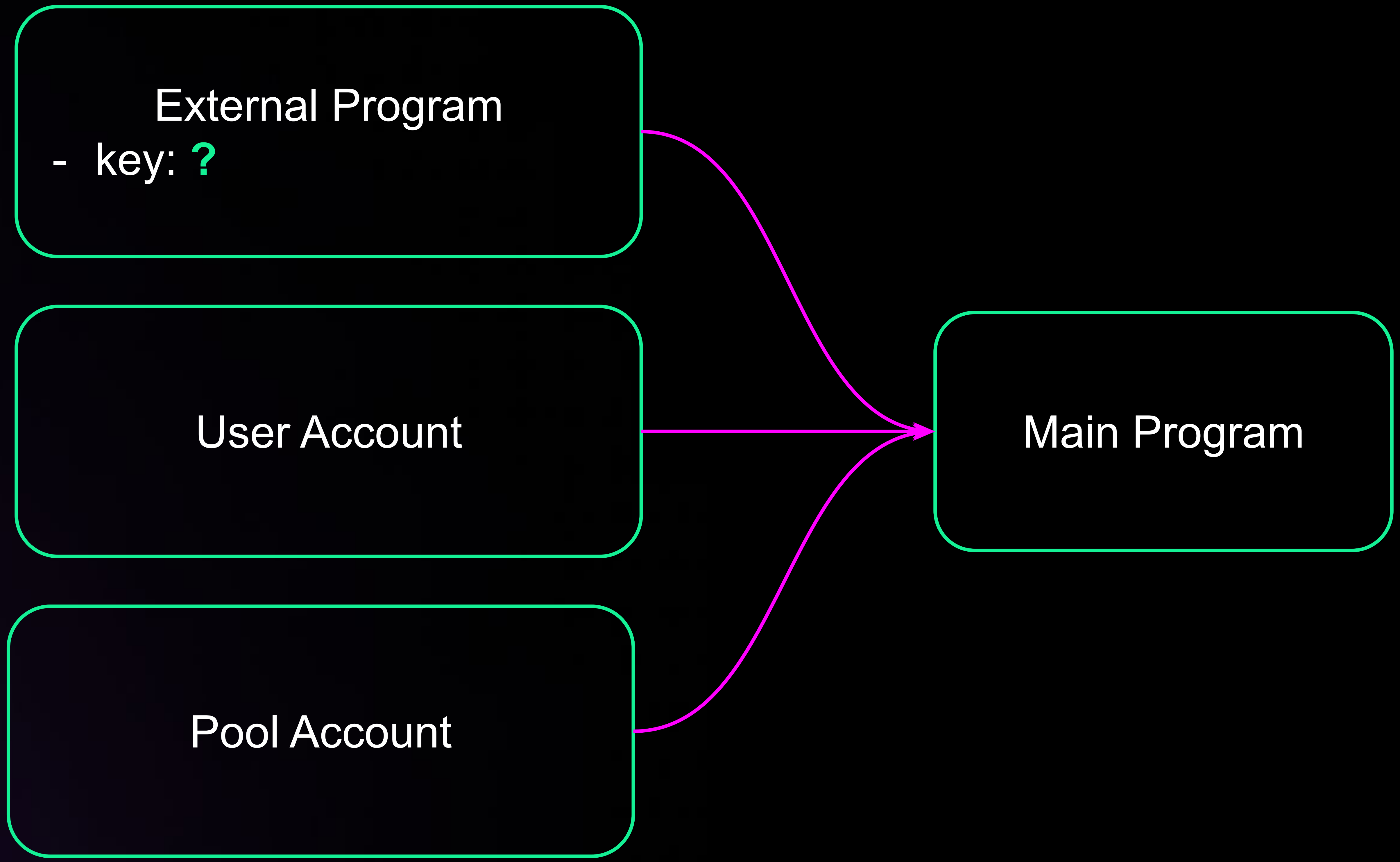
#4 Arbitrary CPI (Cross Program Invocation)

- Verify that the target program you want to invoke has correct address.

1. Main program invokes an external program to transfer funds from user account to pool account and logs the event.

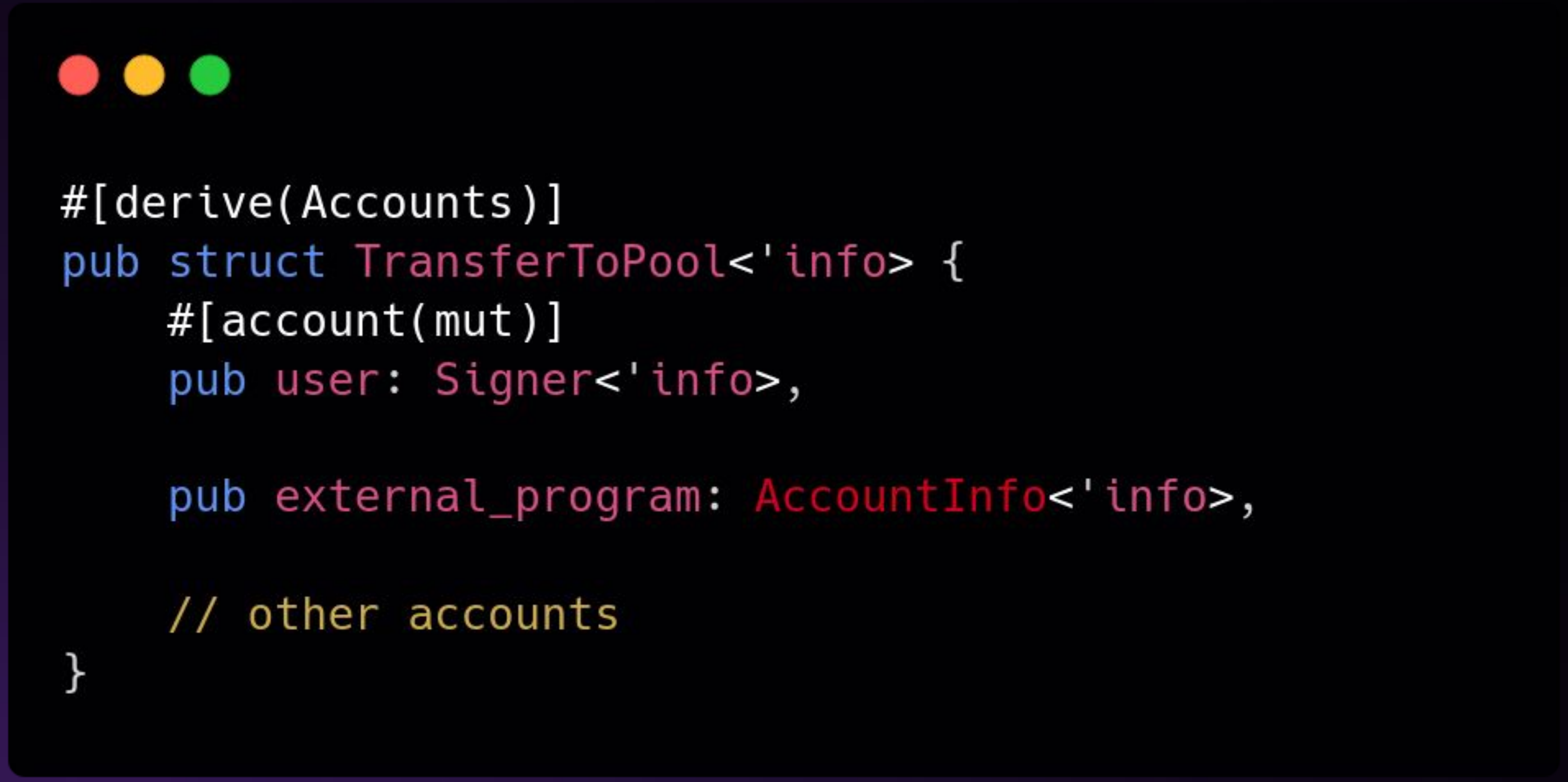
1. External program verifies the correct address of the pool and transfers the funds from user to the pool.

1. If the main program does not verify the address of the external program, an arbitrary malicious program can be supplied.



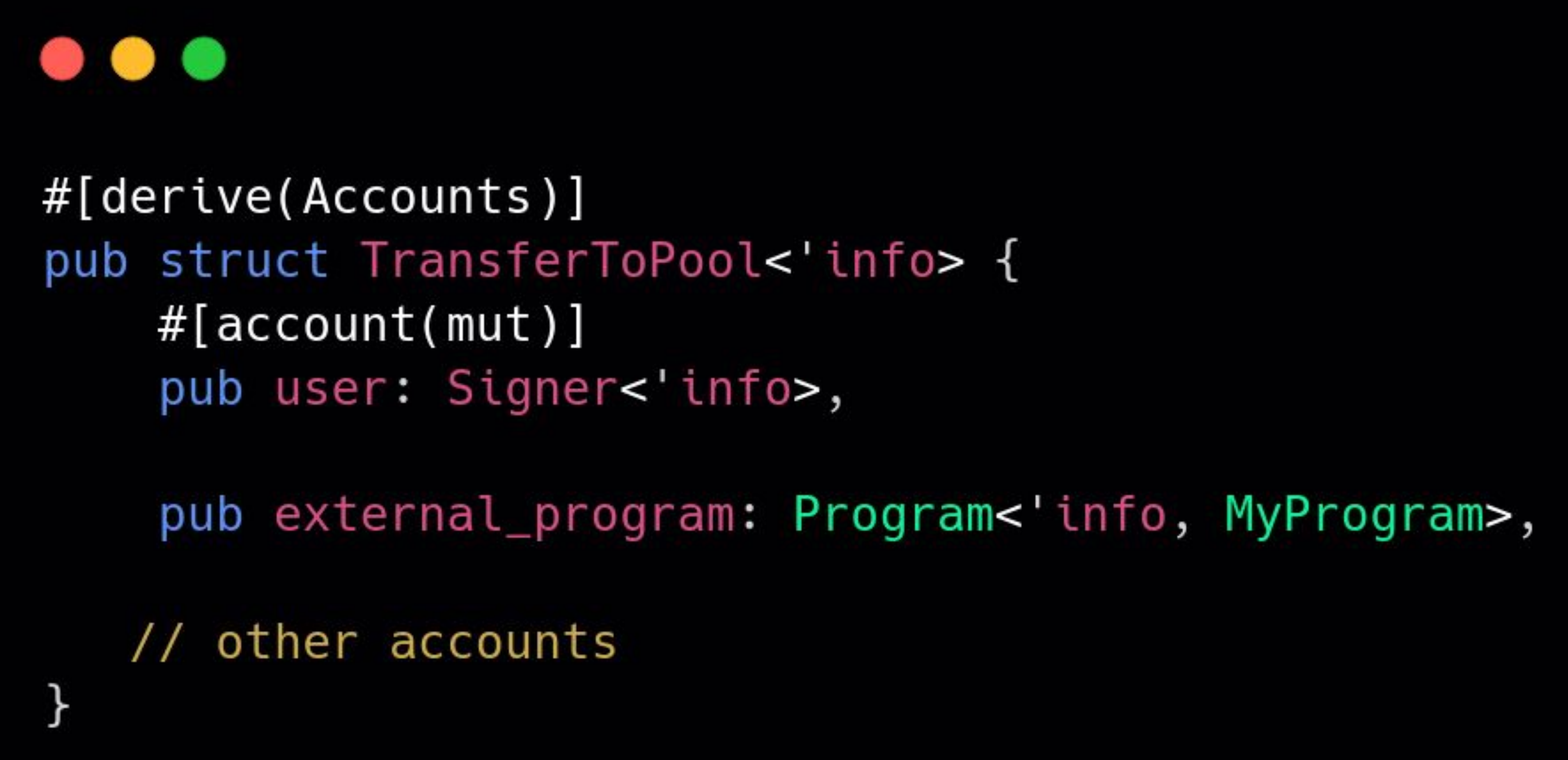
#4 Arbitrary CPI (Cross Program Invocation)

- Verify that the target program you want to invoke has correct address.



```
#[derive(Accounts)]
pub struct TransferToPool<'info> {
    #[account(mut)]
    pub user: Signer<'info>,

    pub external_program: AccountInfo<'info>,

    // other accounts
}
```

Insecure. Arbitrary program might be supplied.



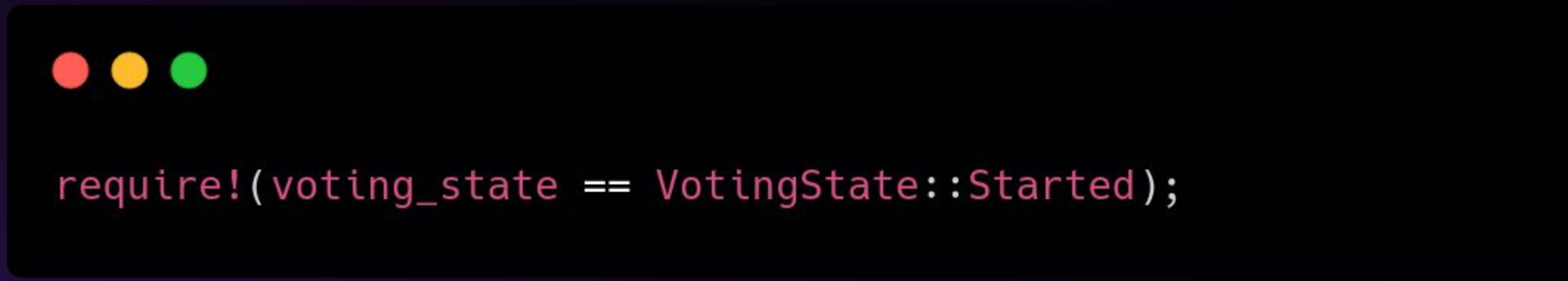
```
#[derive(Accounts)]
pub struct TransferToPool<'info> {
    #[account(mut)]
    pub user: Signer<'info>,

    pub external_program: Program<'info, MyProgram>,

   // other accounts
}
```

Secure. Use Anchor's Program<'info, T> type that checks the program's address.

- Programs that work out of the box are System, Token and AssociatedToken.
- Other programs must have the CPI modules generated.
  - https://www.anchor-lang.com/docs/cross-program-invocations

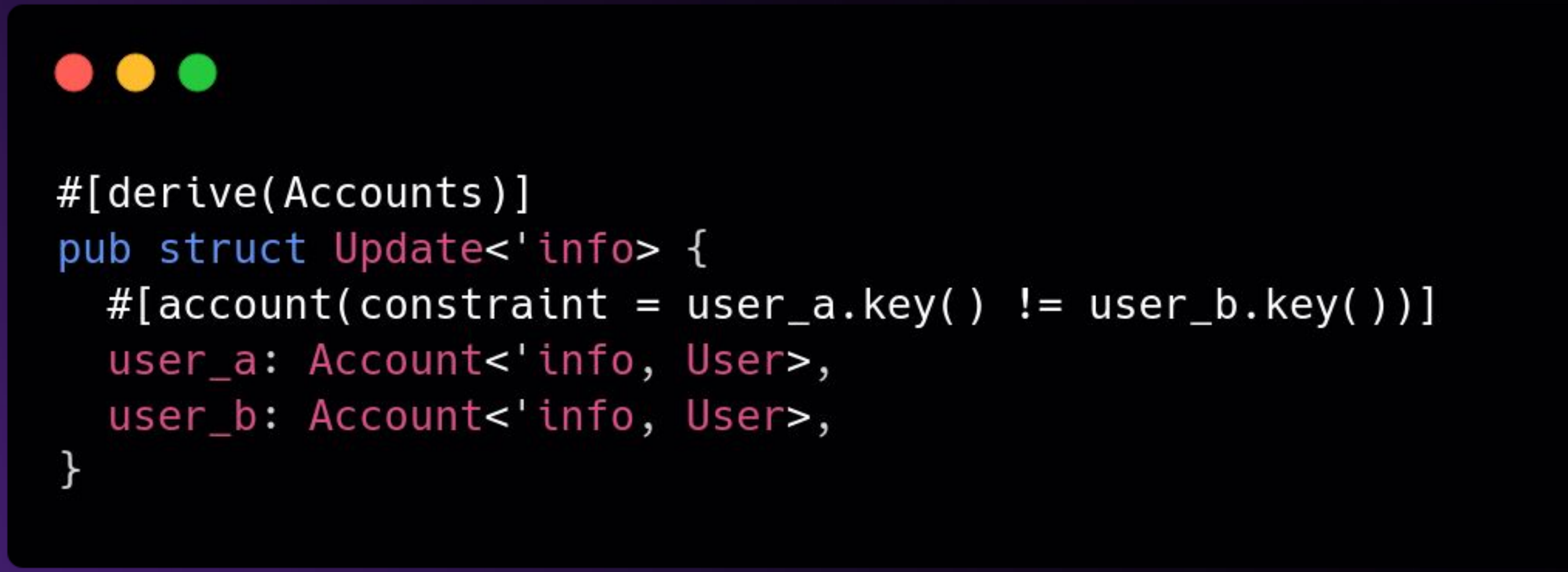
#5 Math & Logic Issues

- Beware of arithmetics and precision issues.
- Validate account data and instruction parameters.
- Make sure instructions are executed in correct order.



```
require!(voting_state == VotingState::Started);
```

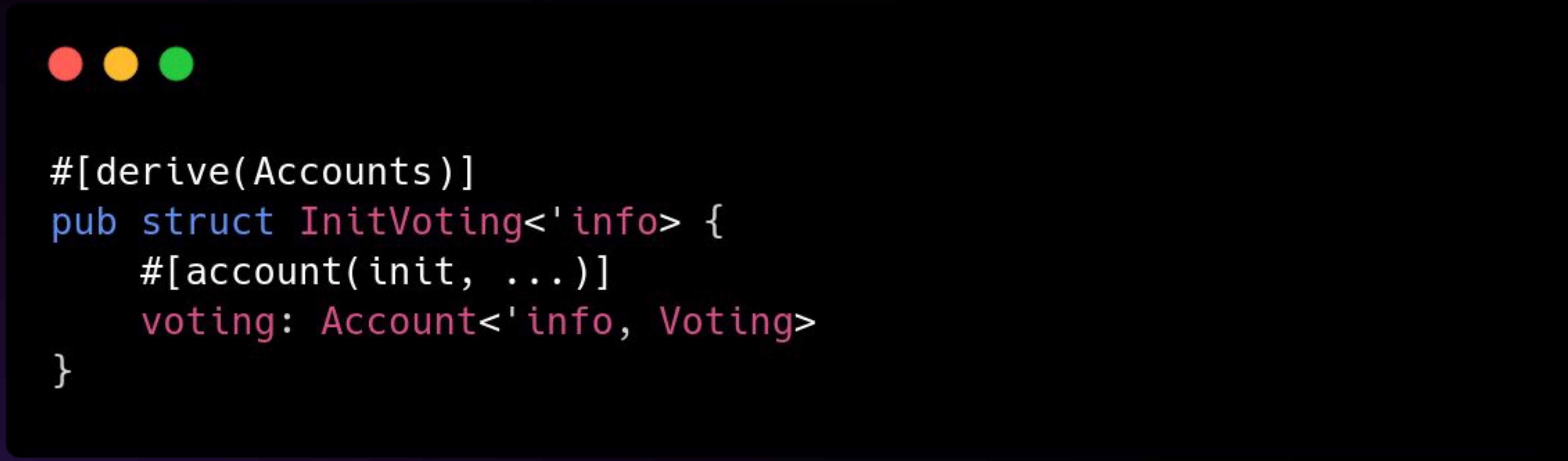
- Prevent unintended behavior when passing duplicate accounts.



```
#[derive(Accounts)]
pub struct Update<'info> {
  #[account(constraint = user_a.key() != user_b.key())]
  user_a: Account<'info, User>,
  user_b: Account<'info, User>,
}
```

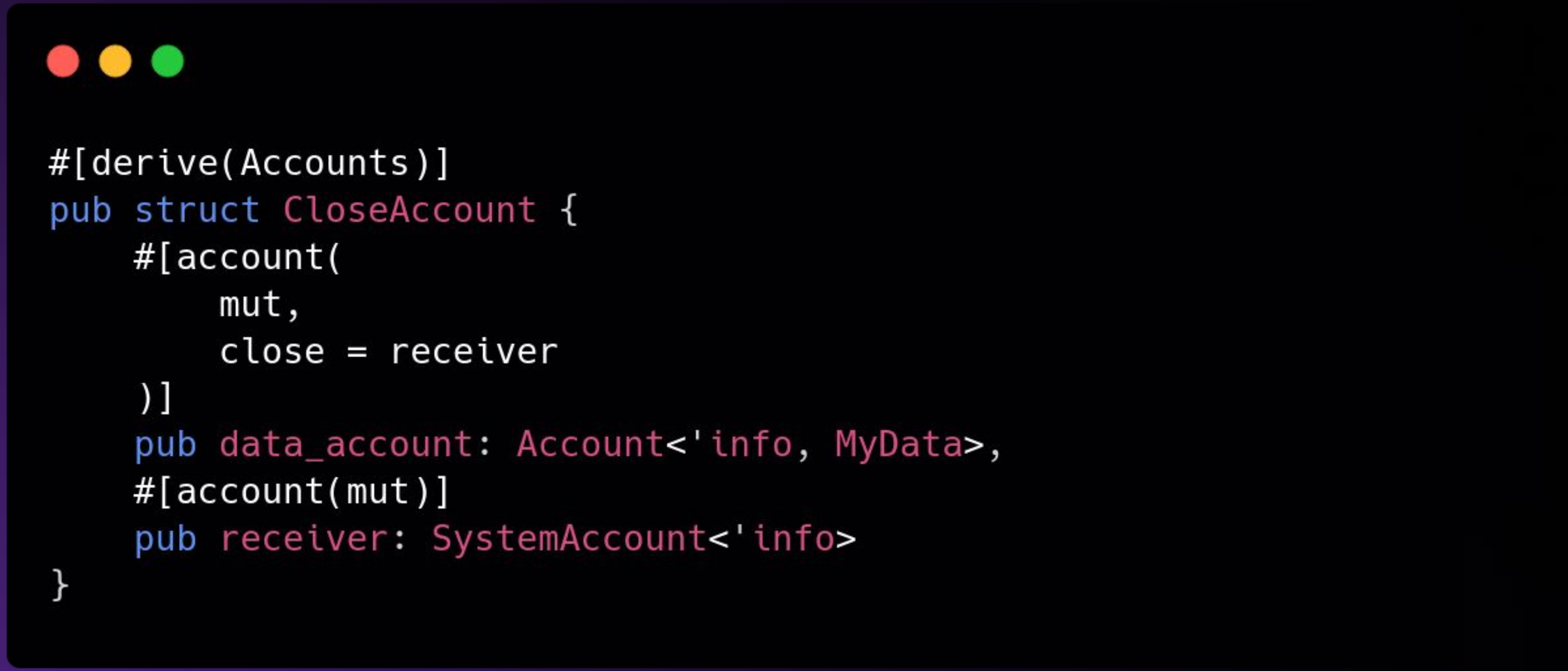
#6 Reinitialization and Revival Attacks

- You don’t want to re-initialize an already initialized account.



```
#[derive(Accounts)]
pub struct InitVoting<'info> {
    #[account(init, ...)]
    voting: Account<'info, Voting>
}
```

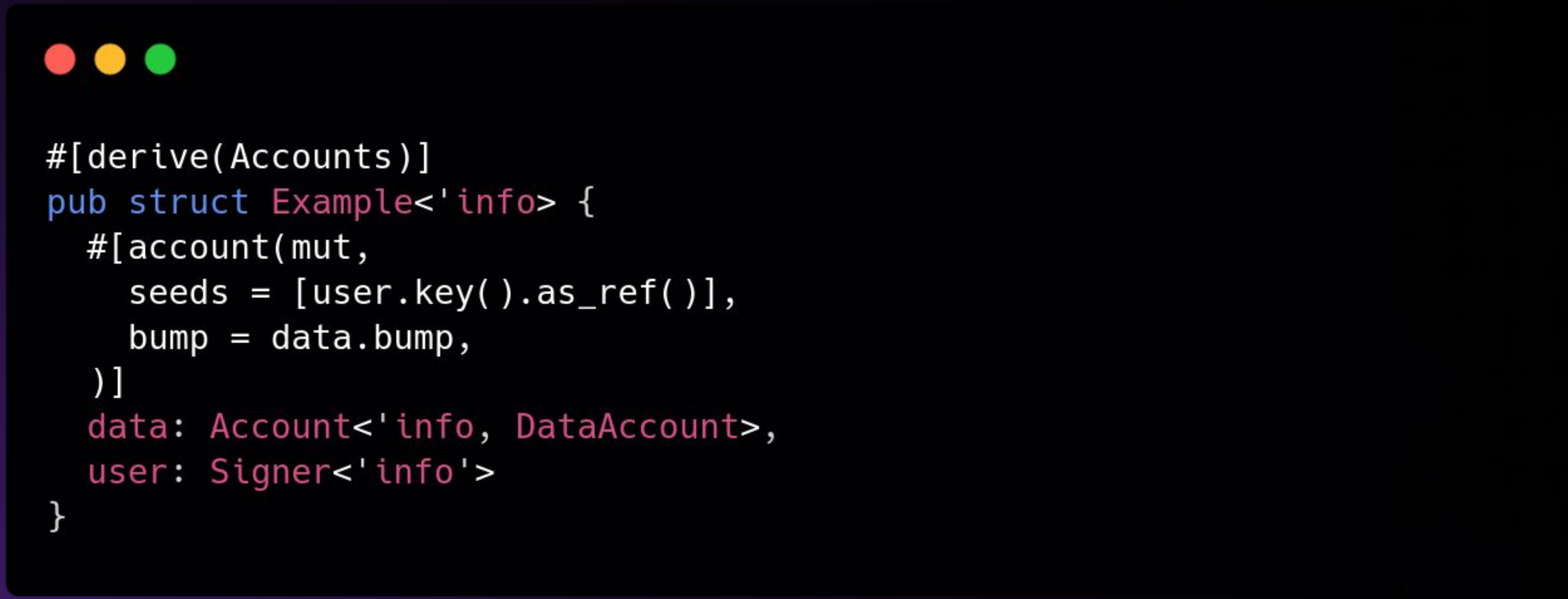
- You don’t want to re-use an already closed account.



```
#[derive(Accounts)]
pub struct CloseAccount {
    #[account(
        mut,
        close = receiver
    )]
    pub data_account: Account<'info, MyData>,
    #[account(mut)]
    pub receiver: SystemAccount<'info>
}
```

Other issues

- Verify account data type to avoid type cosplay.
- Use canonical bump to avoid multiple valid PDAs.
- Do not use shared/global PDA authorities. Use account specific PDAs instead.

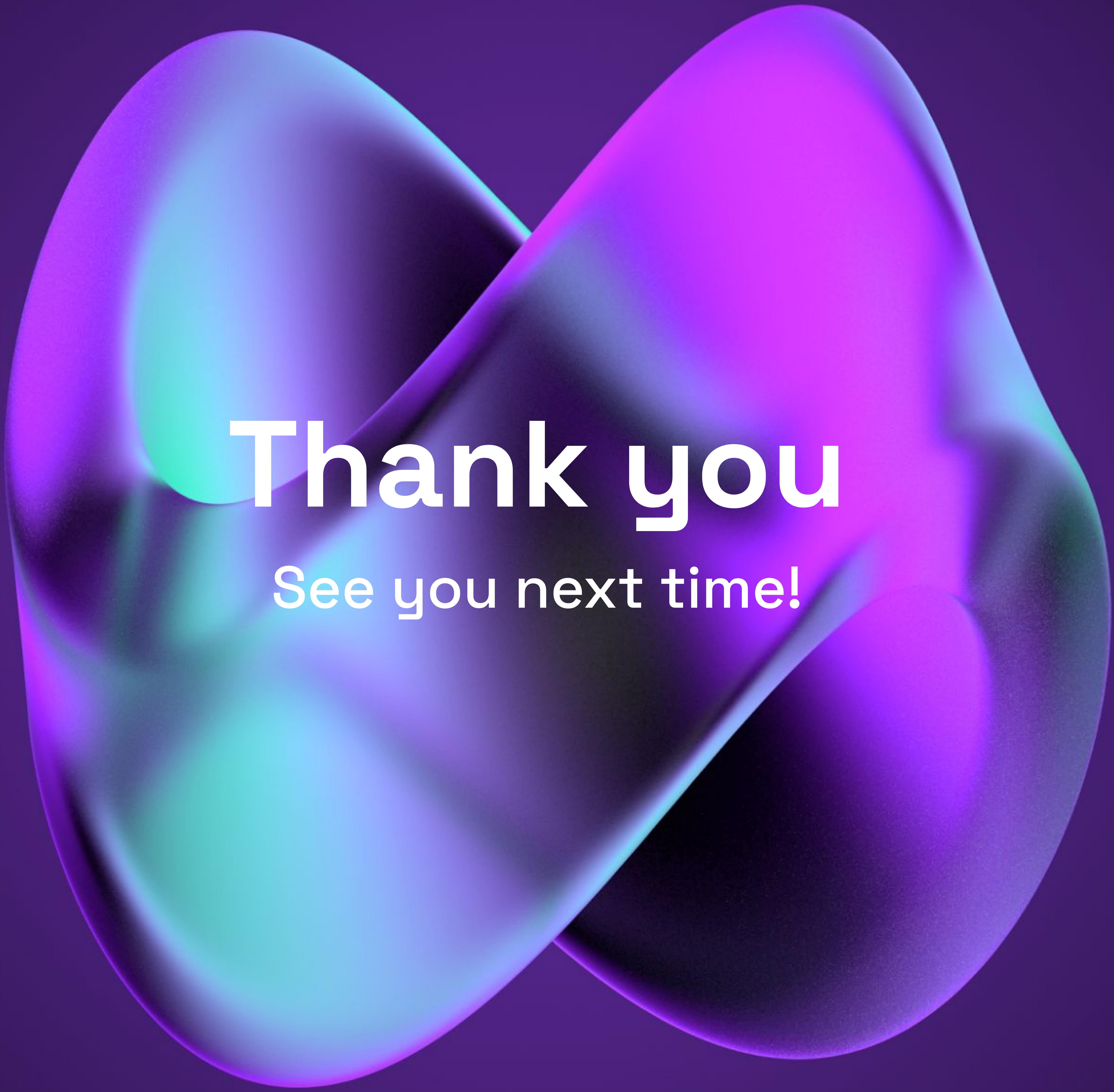


```
#[derive(Accounts)]
pub struct Example<'info> {
  #[account(mut,
    seeds = [user.key().as_ref()],
    bump = data.bump,
  )]
  data: Account<'info, DataAccount>,
  user: Signer<'info'>
}
```

- Recommended resource: Intro to Solana - Module 7: Solana Program Security.
  - https://www.soldev.app/course

Wrap Up

- Use Anchor and it’s built-in security features.
- Test extensively.
- Have your project audited.
- Avoid non-standard code and libraries.
- Compare the list of common exploits with your program.
- An exploit can happen due to combination of multiple vulnerabilities.



Thank you

See you next time!