NEAR Rust SC Security

Course Modules

Rust Security

- Error Handling
  - Recoverable Errors
  - Unrecoverable Errors
    - unwrap & expect
    - Panicking Macros
- Arithmetic Issues
  - Integer Overflow & Underflow
  - Integer Overflow Prevention
    - Checked Maths
    - Saturating Maths
  - Casting Overflow
    - Silent Casting Overflow
    - Panicking Casting Overflow
- Division By 0
- Rounding Direction
- Division Before Multiplication
- Index Out Of Bounds
- Stack Overflow
- OOM (Out Of Memory)
- Crates With Vulnerabilities
- Handy Rust Tools

NEAR Rust Smart Contract Security

- Access Control Issues
  - Missing Access Control
  - Incorrect #[near_bindgen] usage
  - Using env::signer_account_id()
  - Public Callbacks
  - Lack of Separation of Privileges
- DoS (Denial of Service)
  - Usage of unverified accounts
  - Case Study (Appchain Registry)
  - Storage Bloating
  - Prepaid Gas Exceeding
  - Log/Event Bombing
- Race Condition / Reentrancy
- Logical Vulnerabilities
  - Case Study (Custom NEP-141 Token)
  - Case Study (Custom Storage Deposit)
- Tips & Best Practices
  - Redundant state assertion
  - Missing Zero Value Checks
  - Unchecked Attached Deposit
  - Usage of .take()
  - require!()
  - Tautological expressions
  - 2-step ownership transfer process
  - Lack of Pausability
  - Usage of Incorrect JSON Type
  - Two Factor Authentication

Rust Security

Error Handling

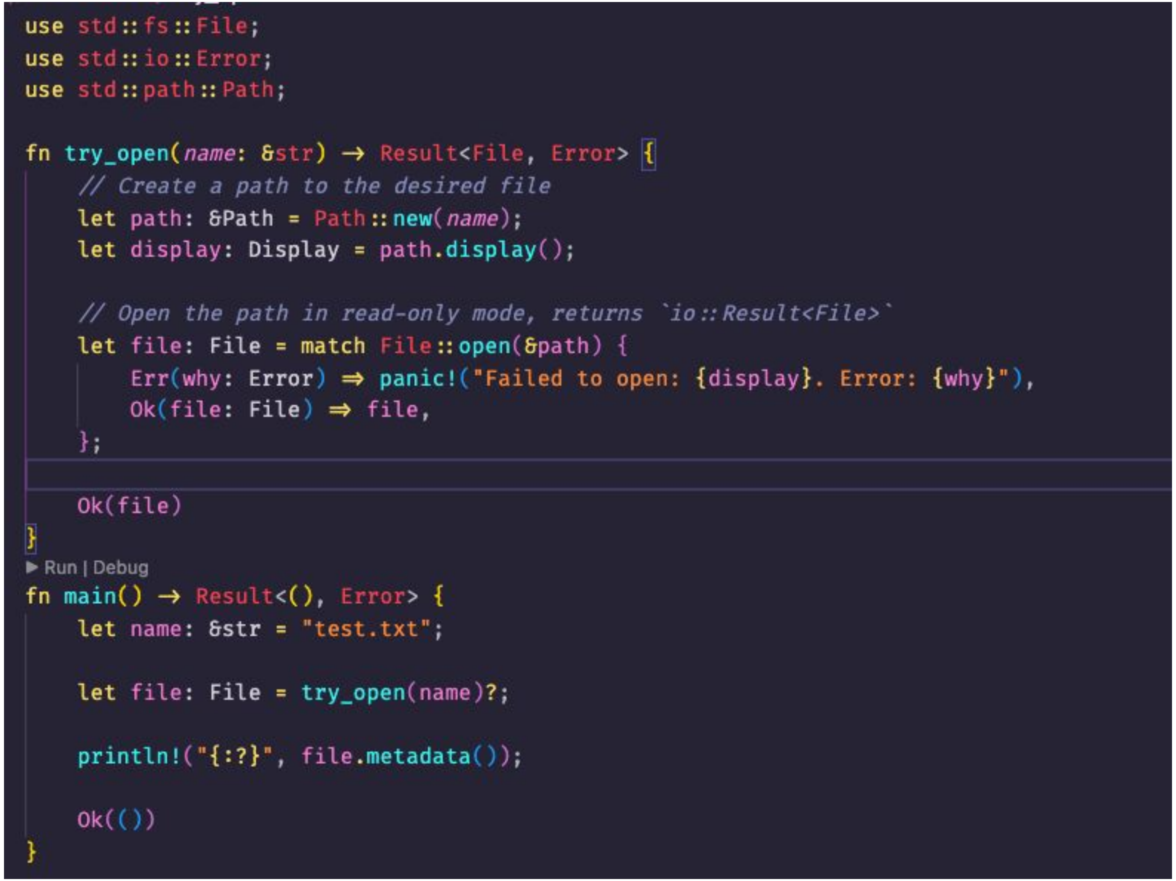
- Recoverable Errors
  - Occur when something goes wrong and can be reasonably handled (displaying error message to a user) => **Result<T , E>**

- Unrecoverable Errors
  - Occur when something goes wrong and cannot be reasonably handled (panic) => **panic!**
  - Might lead to a crash of program

Recoverable Errors

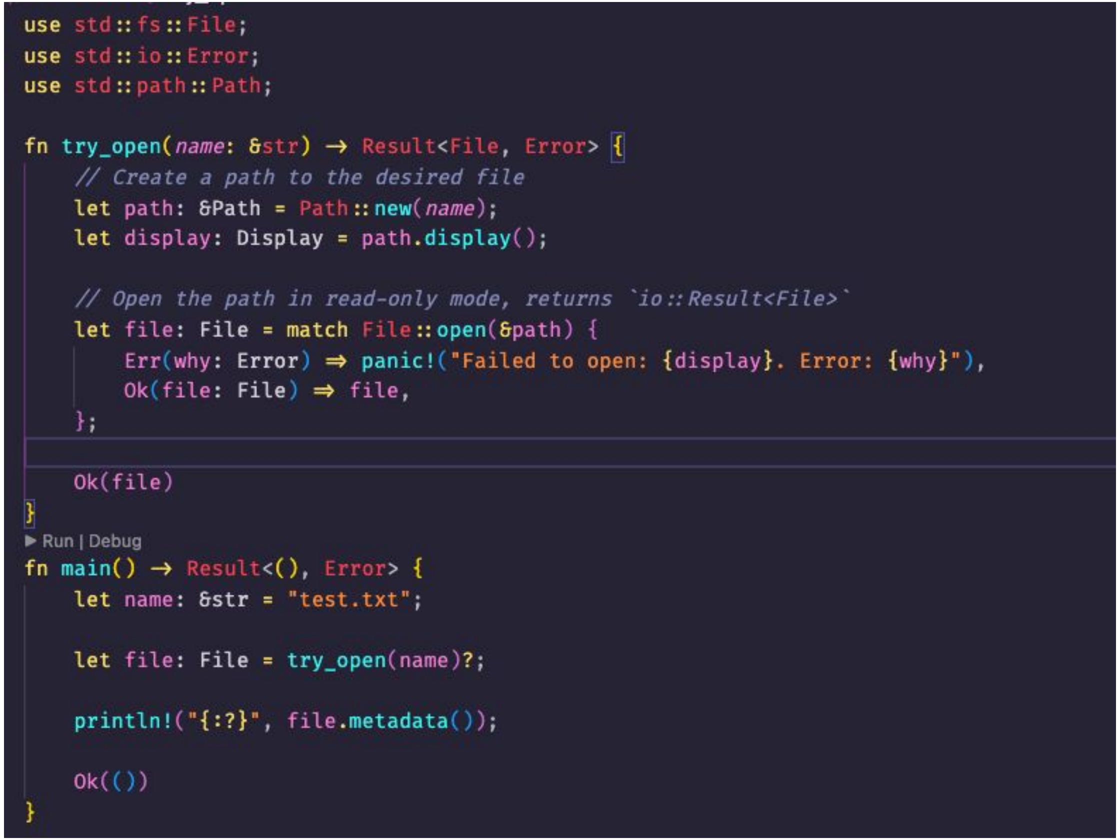
Functions that possess recoverable errors return *Result<T, E>*.

- ➔ **T** represents generic type and denotes a value returned in a success case within **Ok** variant
- ➔ **E** represents generic type and denotes an error returned in a failure case within **Err** variant
- ➔ We can handle those errors by matching over Result type
- ➔ We can decide whether we want to panic on error or gracefully exit by printing a message



```
use std::fs::File;
use std::io::Error;
use std::path::Path;

fn try_open(name: &str) -> Result<File, Error> {
    // Create a path to the desired file
    let path: &Path = Path::new(name);
    let display: Display = path.display();

    // Open the path in read-only mode, returns `io::Result<File>`
    let file: File = match File::open(&path) {
        Err(why: Error) => panic!("Failed to open: {display}. Error: {why}"),
        Ok(file: File) => file,
    };

    Ok(file)
}
```

```
fn main() -> Result<(), Error> {
    let name: &str = "test.txt";

    let file: File = try_open(name)?;

    println!("{:?}", file.metadata());

    Ok(())
}
```



```
use std::fs::File;
use std::io::Error;
use std::path::Path;

fn try_open(name: &str) -> Result<File, Error> {
    // Create a path to the desired file
    let path: &Path = Path::new(name);
    let display: Display = path.display();

    // Open the path in read-only mode, returns `io::Result<File>`
    let file: File = match File::open(&path) {
        Err(why: Error) => panic!("Failed to open: {display}. Error: {why}"),
        Ok(file: File) => file,
    };

    Ok(file)
}
▶ Run | Debug
fn main() -> Result<(), Error> {
    let name: &str = "test.txt";

    let file: File = try_open(name)?;

    println!("{:?}", file.metadata());

    Ok(())
}
```

We can also use **'?'** instead of matching ourselves to propagate error from inner function, if the outer function also returns *Result<T, E>*

Unrecoverable Errors

unwrap() & expect()

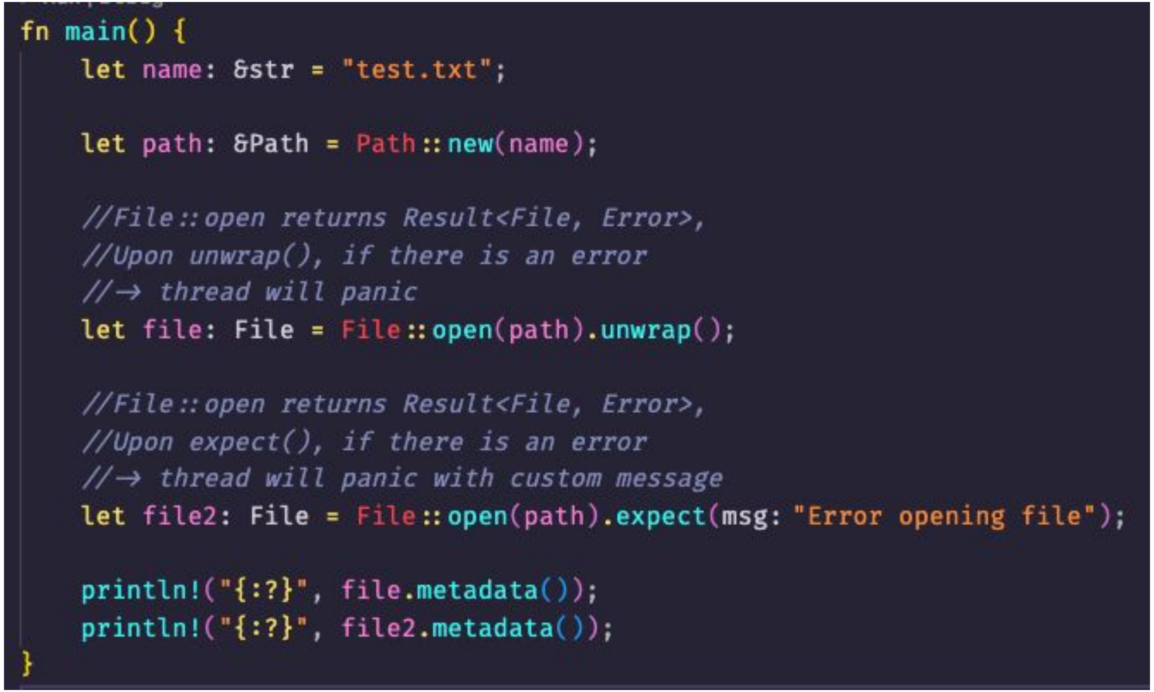
unwrap()

- ➔ Result => Ok(val) => val
- ➔ Result => Err(e) => panic!
- ➔ Option => Some(val) => val
- ➔ Option => None => panic!

expect()

Exactly the same situation as in *unwrap()*, but you can also pass custom error message.

- ➔ .expect("Custom error message")

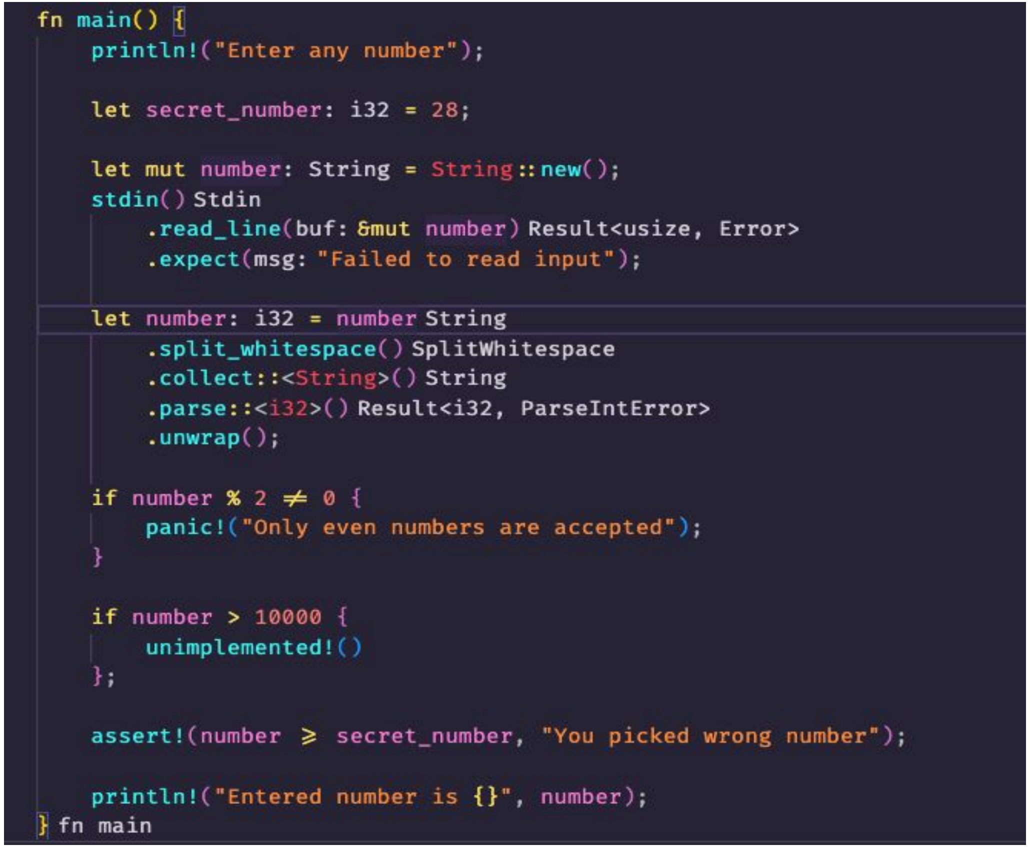


```
fn main() {
    let name: &str = "test.txt";

    let path: &Path = Path::new(name);

    //File::open returns Result<File, Error>,
    //Upon unwrap(), if there is an error
    //→ thread will panic
    let file: File = File::open(path).unwrap();

    //File::open returns Result<File, Error>,
    //Upon expect(), if there is an error
    //→ thread will panic with custom message
    let file2: File = File::open(path).expect(msg: "Error opening file");

    println!("{:?}", file.metadata());
    println!("{:?}", file2.metadata());
}
```

Panicking Macros

Those are macros which trigger panic

- ➔ panic!
- ➔ unreachable!
- ➔ unimplemented!
- ➔ todo!
- ➔ assert! , assert_eq! , assert_ne!
- ➔ debug_assert! , debug_assert_eq! , debug_assert_ne!



```
fn main() {
    println!("Enter any number");

    let secret_number: i32 = 28;

    let mut number: String = String::new();
    stdin() Stdin
        .read_line(buf: &mut number) Result<usize, Error>
        .expect(msg: "Failed to read input");

    let number: i32 = number String
        .split_whitespace() SplitWhitespace
        .collect::<String>() String
        .parse::<i32>() Result<i32, ParseIntError>
        .unwrap();

    if number % 2 ≠ 0 {
        panic!("Only even numbers are accepted");
    }

    if number > 10000 {
        unimplemented!()
    };

    assert!(number ≥ secret_number, "You picked wrong number");

    println!("Entered number is {}", number);
} fn main
```

Arithmetic Issues

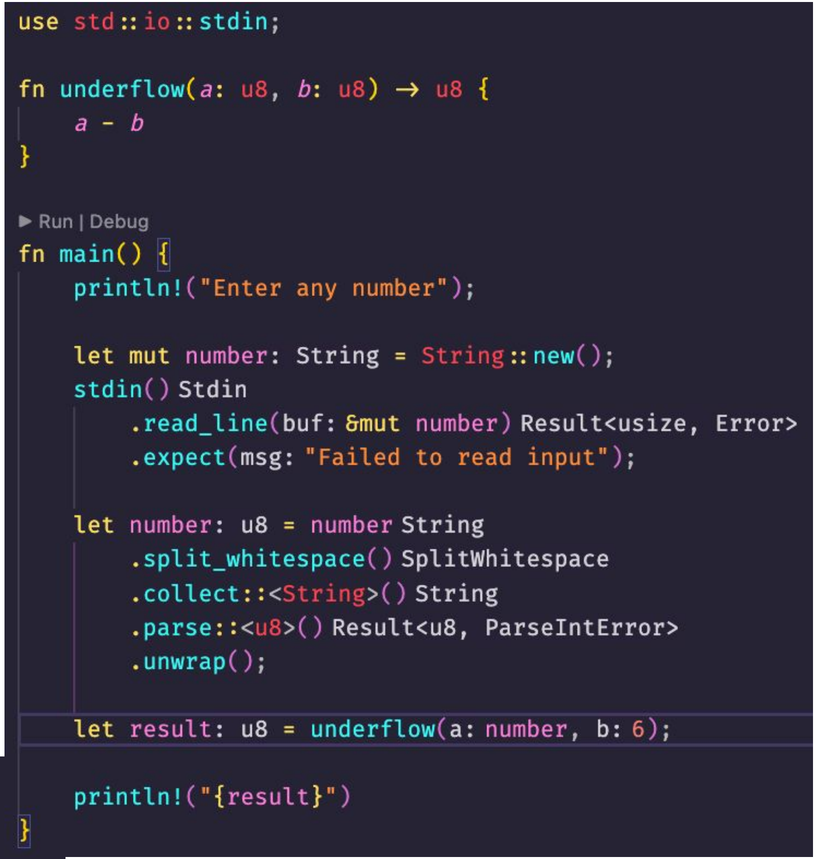
Integer Overflow/Underflow

- In computer programming, an integer overflow occurs when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of digits – either higher than the maximum or lower than the minimum representable value.
- Especially dangerous when compiled in **release** mode
  - In release mode integer overflows are silenced and not catched during runtime

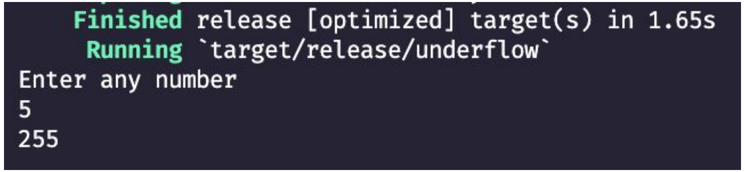
Building and Running Rust program in release mode leads to silencing integer overflow/underflow bugs

To prevent this, release profile in Cargo.toml has to be updated with:

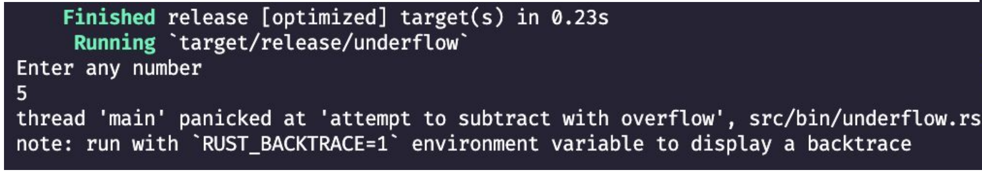
overflow-checks=true



```
use std::io::stdin;

fn underflow(a: u8, b: u8) -> u8 {
    a - b
}

▶ Run | Debug
fn main() {
    println!("Enter any number");

    let mut number: String = String::new();
    stdin() Stdin
        .read_line(buf: &mut number) Result<usize, Error>
        .expect(msg: "Failed to read input");

    let number: u8 = number String
        .split_whitespace() SplitWhitespace
        .collect::<String>() String
        .parse::<u8>() Result<u8, ParseIntError>
        .unwrap();

    let result: u8 = underflow(a: number, b: 6);

    println!("{result}")
}
```



```
    Finished release [optimized] target(s) in 1.65s
     Running `target/release/underflow`
Enter any number
5
255
```



```
    Finished release [optimized] target(s) in 0.23s
     Running `target/release/underflow`
Enter any number
5
thread 'main' panicked at 'attempt to subtract with overflow', src/bin/underflow.rs:4:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Integer Overflow Prevention

Checked Maths

To handle overflows/underflows it in graceful manner - checked maths has to be used

- ➔ **checked_*** (checked_add, checked_sub, checked_div, checked_mul ...)
  - ◆ If *Some(val)* returned => Safe
  - ◆ If *None* returned => Overflow/Underflow

Full list: https://doc.rust-lang.org/std/?search=checked

Saturating Maths

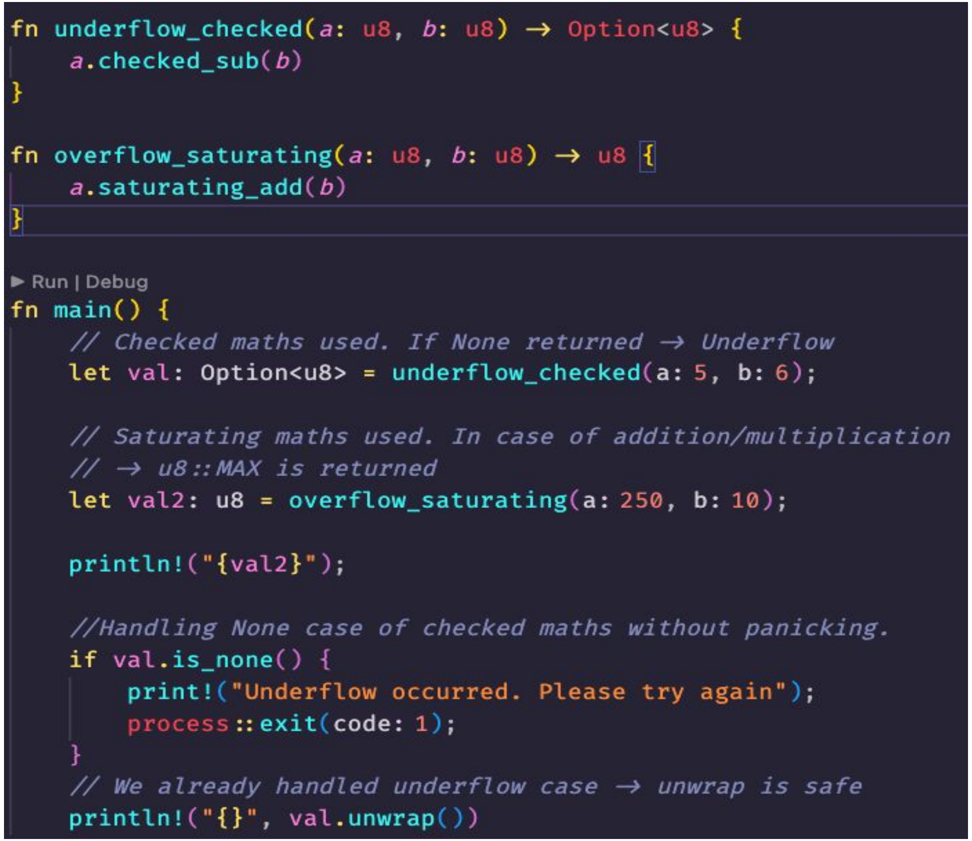
Saturating maths returns a value within numerical bounds instead of overflowing.

- **saturating_***(saturating_add, saturating_sub, saturating_div, saturating_mul, ...)
  - Addition/Multiplication
    - Overflow => type::MAX (uint8::MAX)
  - Subtraction/Division
    - Underflow => type::MIN (uint8::MIN)

Full list: https://doc.rust-lang.org/std/?search=saturating

- By using **checked maths**, we can handle the "None" case ourselves and gracefully exit.

- By using **saturating maths** we can always be sure that the number is within type bounds.



```
fn underflow_checked(a: u8, b: u8) -> Option<u8> {
    a.checked_sub(b)
}

fn overflow_saturating(a: u8, b: u8) -> u8 {
    a.saturating_add(b)
}

▶ Run | Debug
fn main() {
    // Checked maths used. If None returned -> Underflow
    let val: Option<u8> = underflow_checked(a: 5, b: 6);

    // Saturating maths used. In case of addition/multiplication
    // -> u8::MAX is returned
    let val2: u8 = overflow_saturating(a: 250, b: 10);

    println!("{val2}");

    //Handling None case of checked maths without panicking.
    if val.is_none() {
        print!("Underflow occurred. Please try again");
        process::exit(code: 1);
    }
    // We already handled underflow case -> unwrap is safe
    println!("{}", val.unwrap())
```

Casting Overflow

Casting overflow happens when casting is attempted from the larger type that holds the value bigger than the smaller type max value.

Casting **a** to **b**:

➔ *if [Type a > Type b && a(Value) > b::MAX]* => Overflow

➔ *u16(300) to u8* => 300 > u8::MAX(2^8-1) => Overflow

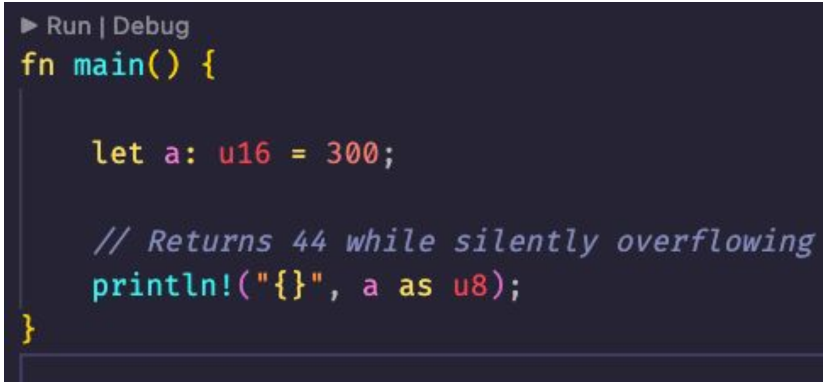
Silent Casting Overflow

Occurs when casting is performed using **“as”** keyword. It does not cause panic. If the overflow happened, value wraps around the type.

It can cause major logical errors if mishandled

Casting **a as b**

➔ If [*Type a* > *Type b* && *a(Value)* > *b::MAX*] = Value - (b::MAX + 1)



```
fn main() {

    let a: u16 = 300;

    // Returns 44 while silently overflowing
    println!("{}", a as u8);
}
```

Panicking Casting Overflow

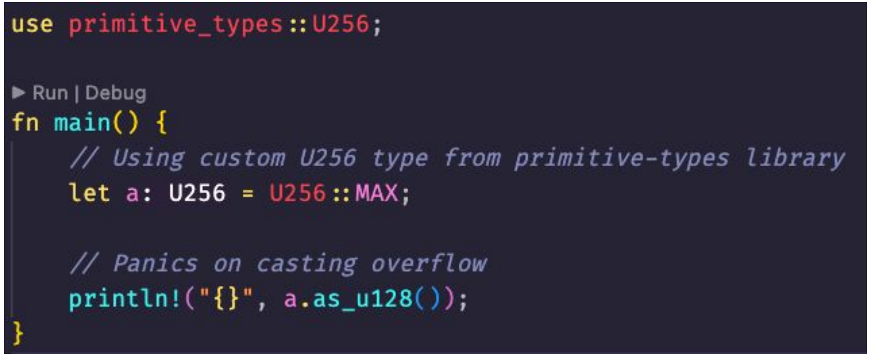
Occurs when casting is performed on custom numerical types from separate crates using methods that have casting with overflow checking.

Casting a to b

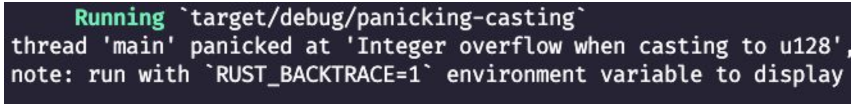
➔ If [*Type a > Type b && a(Value) > b::MAX*] = Panic

primitive-types crates

➔ *U128/256/512.as_u32()*
➔ *U128/256/512.as_64()*
➔ *U128/256/512.as_ u128()*



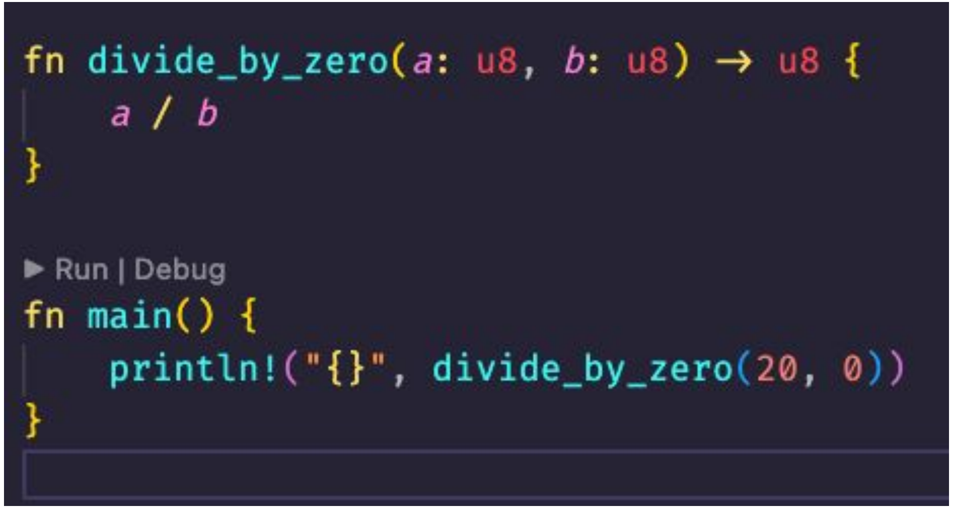
```
use primitive_types::U256;

fn main() {
    // Using custom U256 type from primitive-types library
    let a: U256 = U256::MAX;

    // Panics on casting overflow
    println!("{}", a.as_u128());
}
```



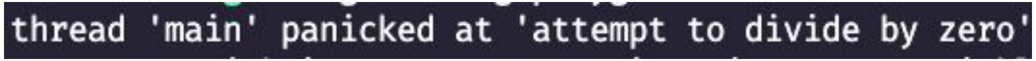
```
     Running `target/debug/panicking-casting`
thread 'main' panicked at 'Integer overflow when casting to u128',
note: run with `RUST_BACKTRACE=1` environment variable to display
```

Division By 0

- ➔ Rust panics when division is performed while denominator is 0. This might lead to a crash of a program
- ➔ To prevent, before performing any division we have to verify that denominator is larger than 0



```rust
fn divide_by_zero(a: u8, b: u8) → u8 {
    a / b
}

fn main() {
    println!("{}", divide_by_zero(20, 0))
}
```

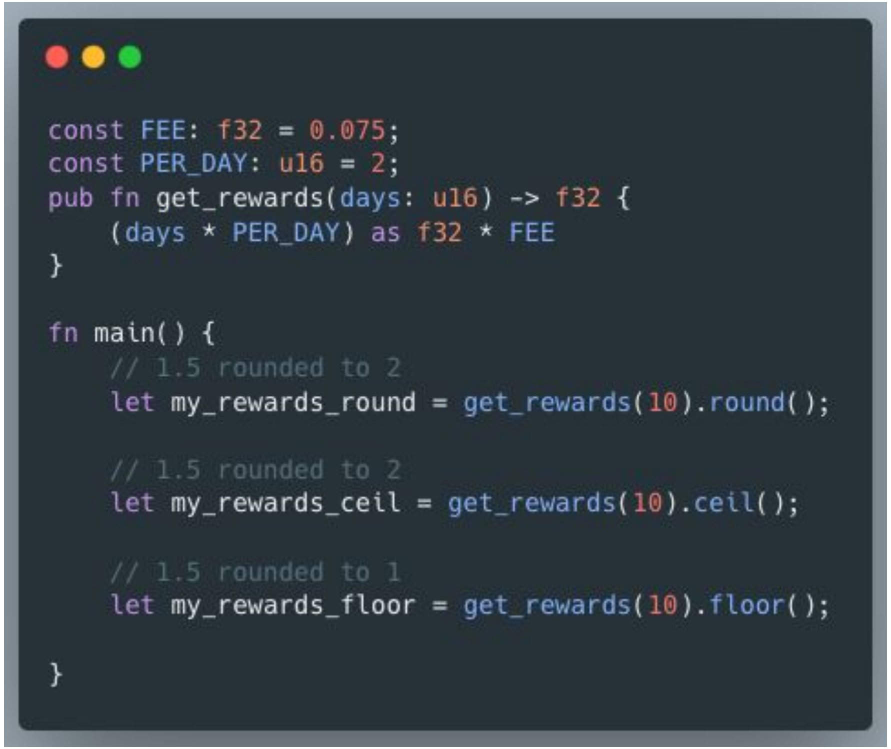


```
thread 'main' panicked at 'attempt to divide by zero'
```

Rounding Direction

Rounding Direction

- In Rust, there are several ways to round floating point number
  - **round()** -> Rounds either up or down to the nearest integer
  - **ceil()** -> Rounds up
  - **floor()** -> Rounds down
- However, it is always crucial to specify rounding direction to avoid logical/calculation mistakes
- Real-world scenario which led to critical vulnerability: How to Become a Millionaire, 0.000001 BTC at a Time

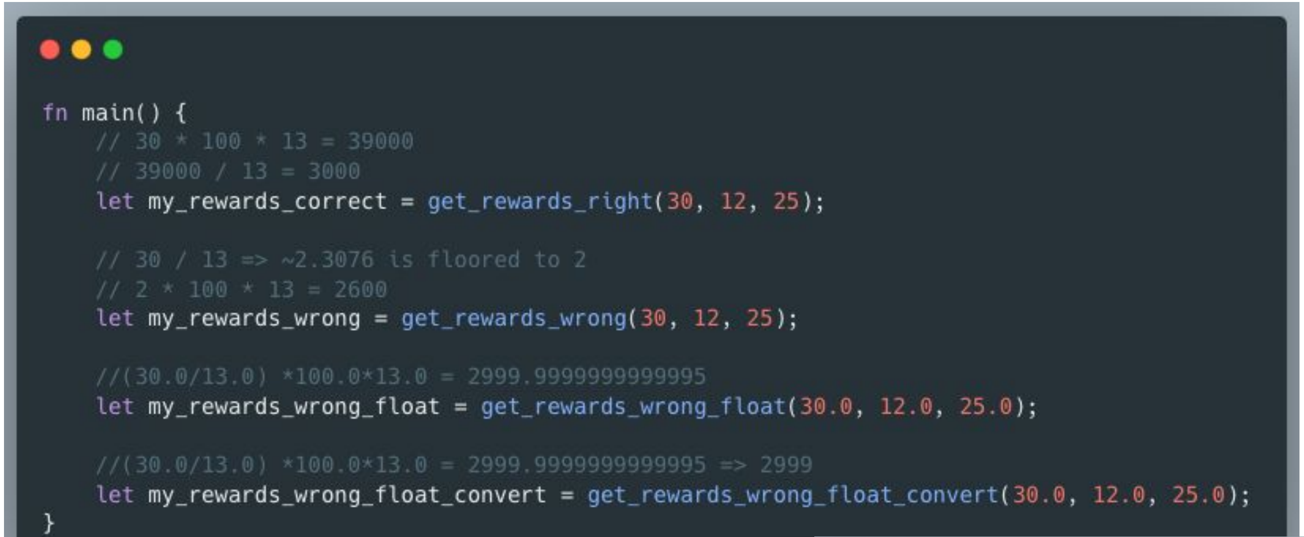


```rust
const FEE: f32 = 0.075;
const PER_DAY: u16 = 2;
pub fn get_rewards(days: u16) -> f32 {
    (days * PER_DAY) as f32 * FEE
}

fn main() {
    // 1.5 rounded to 2
    let my_rewards_round = get_rewards(10).round();

    // 1.5 rounded to 2
    let my_rewards_ceil = get_rewards(10).ceil();

    // 1.5 rounded to 1
    let my_rewards_floor = get_rewards(10).floor();

}
```

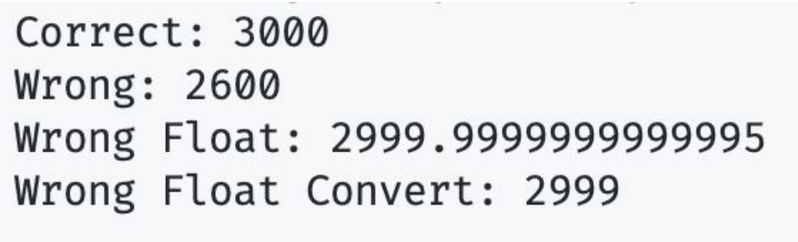
Division Before Multiplication

Division Before Multiplication

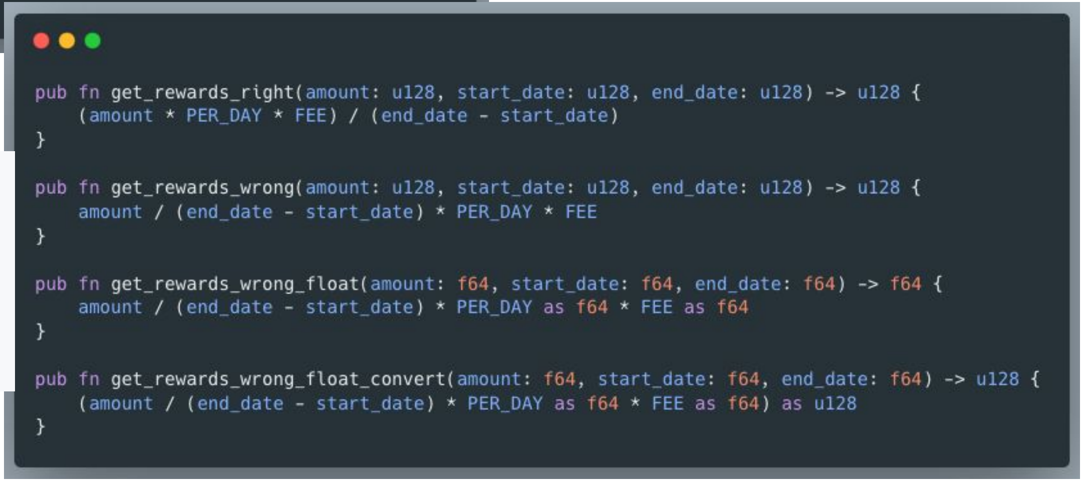
- Order of operations matter
- Dividing before multiplying, depending on the types used, can lead to precision loss or even incorrect value returned
- When we operate with non-floating point numbers, upon division, the value is floored
  - (a/b) * c * d => 30/13 = > ~2.3076 is floored to 2 => 2*100*13 = 2600 (incorrect)
  - (a*c*d) / b => 30 * 100 * 13 => 39000 / 13 => 3000 (correct)
- When we operate with floating point numbers (f64/f32), there will be floating point errors
  - (a/b) * c * d => (30.0/13.0) *100.0*13.0 = 2999.9999999999995 (incorrect)
  - (a*c*d) / b => (30.0 * 100.0 * 13.0) / 13.0 => 3000.0 (correct)
- When we operate with floating point numbers and then convert to non-floating point numbers
  - (a/b) * c * d => (30.0/13.0) *100.0*13.0 = 2999.9999999999995 => 2999 (incorrect)
  - (a*c*d) / b => (30 * 100 * 13) / 13 => 3000 (correct)



```
fn main() {
    // 30 * 100 * 13 = 39000
    // 39000 / 13 = 3000
    let my_rewards_correct = get_rewards_right(30, 12, 25);

    // 30 / 13 => ~2.3076 is floored to 2
    // 2 * 100 * 13 = 2600
    let my_rewards_wrong = get_rewards_wrong(30, 12, 25);

    //(30.0/13.0) *100.0*13.0 = 2999.9999999999995
    let my_rewards_wrong_float = get_rewards_wrong_float(30.0, 12.0, 25.0);

    //(30.0/13.0) *100.0*13.0 = 2999.9999999999995 => 2999
    let my_rewards_wrong_float_convert = get_rewards_wrong_float_convert(30.0, 12.0, 25.0);
}
```



```
Correct: 3000
Wrong: 2600
Wrong Float: 2999.9999999999995
Wrong Float Convert: 2999
```

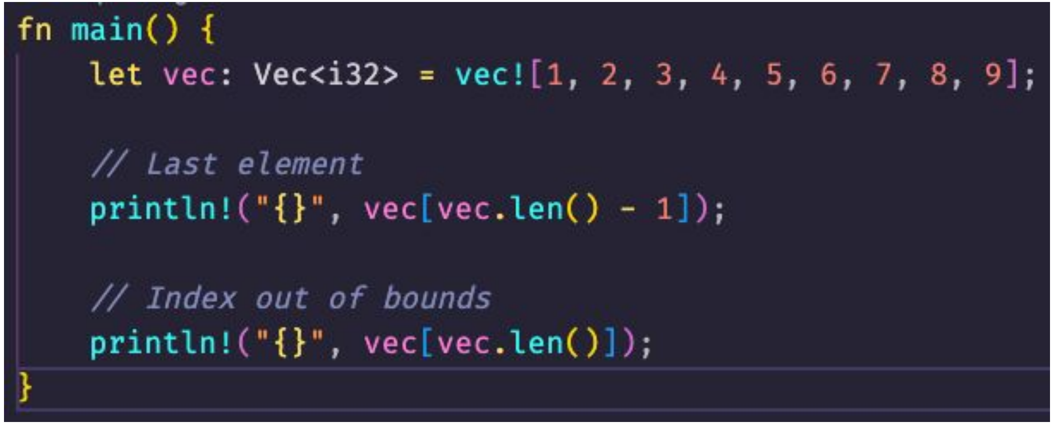


```
pub fn get_rewards_right(amount: u128, start_date: u128, end_date: u128) -> u128 {
    (amount * PER_DAY * FEE) / (end_date - start_date)
}

pub fn get_rewards_wrong(amount: u128, start_date: u128, end_date: u128) -> u128 {
    amount / (end_date - start_date) * PER_DAY * FEE
}

pub fn get_rewards_wrong_float(amount: f64, start_date: f64, end_date: f64) -> f64 {
    amount / (end_date - start_date) * PER_DAY as f64 * FEE as f64
}

pub fn get_rewards_wrong_float_convert(amount: f64, start_date: f64, end_date: f64) -> u128 {
    (amount / (end_date - start_date) * PER_DAY as f64 * FEE as f64) as u128
}
```

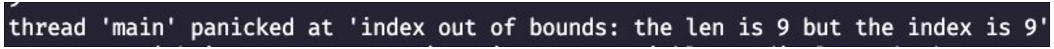
Index Out Of Bounds

Rust panics if we try to access an item of an array via index which is larger than **array.len - 1**.

To prevent use **.get()** method which returns **Option** with **None** if out of bounds



```
fn main() {
    let vec: Vec<i32> = vec![1, 2, 3, 4, 5, 6, 7, 8, 9];

    // Last element
    println!("{}", vec[vec.len() - 1]);

    // Index out of bounds
    println!("{}", vec[vec.len()]);
}
```

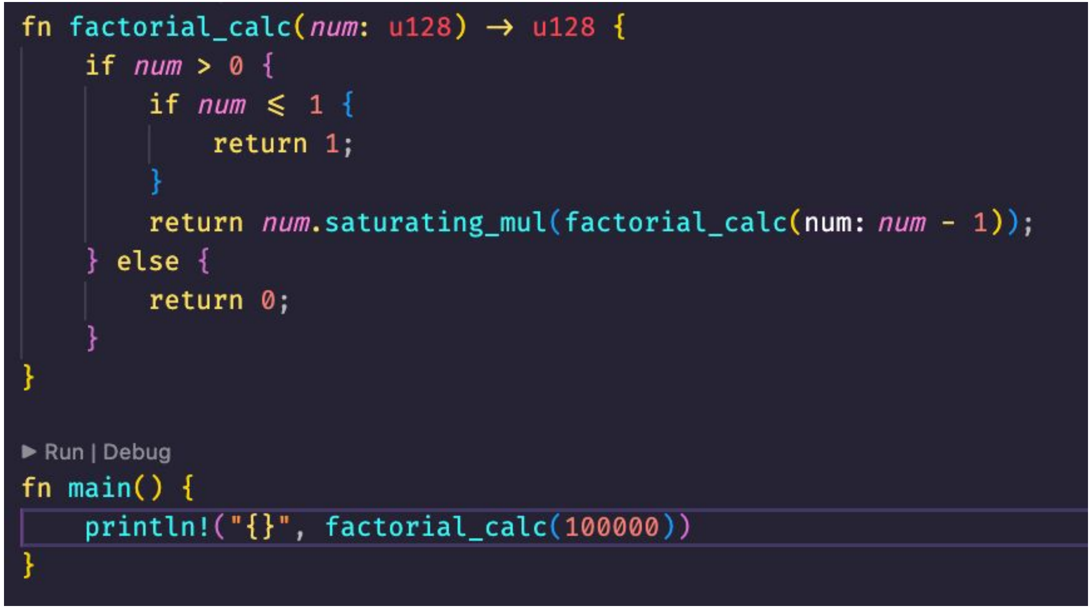


```
thread 'main' panicked at 'index out of bounds: the len is 9 but the index is 9'
```

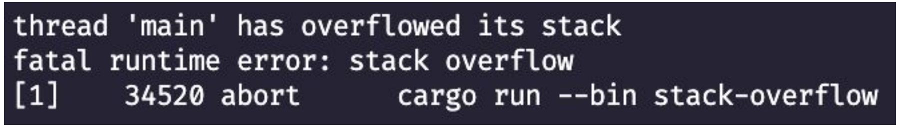
Stack Overflow

Stack overflow occurs when the program consumed more memory than the call stack has available. This leads to program crash.

Usually occurs in recursive implementations of functions.



```
fn factorial_calc(num: u128) → u128 {
    if num > 0 {
        if num ⩽ 1 {
            return 1;
        }
        return num.saturating_mul(factorial_calc(num: num - 1));
    } else {
        return 0;
    }
}

fn main() {
    println!("{}", factorial_calc(100000))
}
```

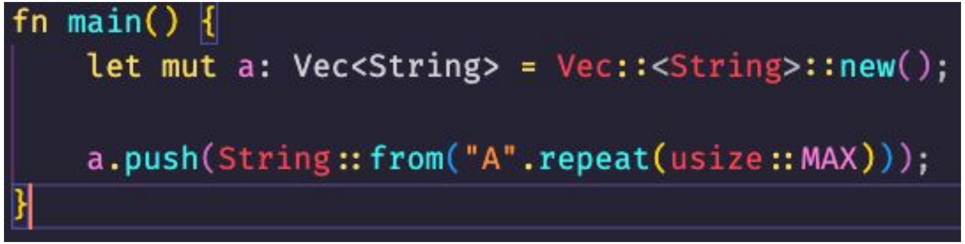


```
thread 'main' has overflowed its stack
fatal runtime error: stack overflow
[1]    34520 abort      cargo run --bin stack-overflow
```

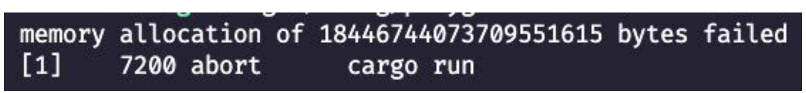
OOM (Out Of Memory)

When there is no enough memory to allocate for a program use OOM error happens. It can lead to denial of service

- ➔ Happens if length of a buffer is not checked.
- ➔ There is no limit on unbounded data types (Arrays, Vectors ...)



```rust
fn main() {
    let mut a: Vec<String> = Vec::<String>::new();

    a.push(String::from("A".repeat(usize::MAX)));
}
```



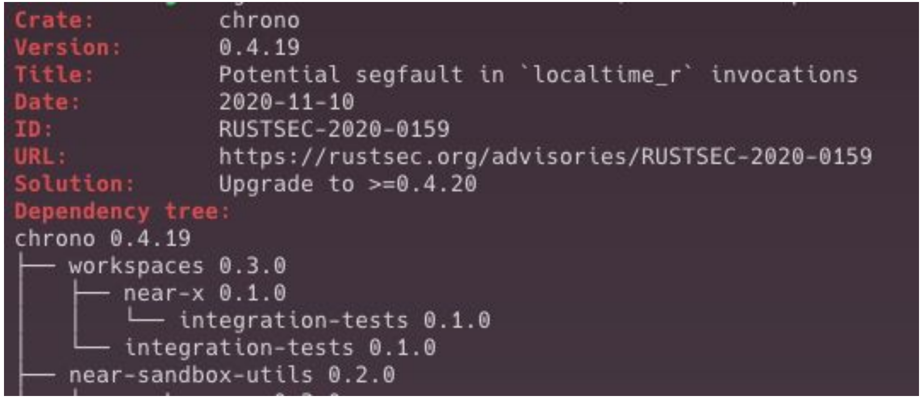
```
memory allocation of 18446744073709551615 bytes failed
[1]    7200 abort      cargo run
```

Crates With Vulnerabilities

All security vulnerabilities discovered in creates are published at:

[Rust Advisory Database]

You can use the tool “***cargo-audit***” to discover vulnerabilities in crates



```
Crate:          chrono
Version:        0.4.19
Title:          Potential segfault in `localtime_r` invocations
Date:           2020-11-10
ID:             RUSTSEC-2020-0159
URL:            https://rustsec.org/advisories/RUSTSEC-2020-0159
Solution:       Upgrade to >=0.4.20
Dependency tree:
chrono 0.4.19
├── workspaces 0.3.0
│   ├── near-x 0.1.0
│   │   └── integration-tests 0.1.0
│   └── integration-tests 0.1.0
├── near-sandbox-utils 0.2.0
```

Handy Rust Tools

- cargo-audit - Discover Vulnerabilities in Crates
- MIRI - Rust’s MIR interpreter to discover wide variety of memory issues
- cargo-geiger - Discover unsafe rust usage within crates
- cargo-tarpaulin - Measure test coverage
- rust-analyzer - VS code extension. It provides features like completion and code checking
- cargo expand - Expand rust macros
- cargo fuzz - Fuzzer
- hongg-fuzz - Fuzzer
- cargo-valgrind - Discover memory leak

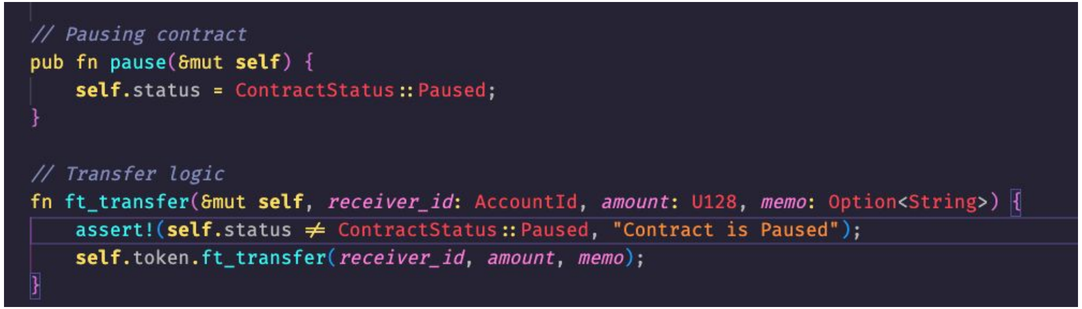
NEAR Rust SC Security

Access Control Issues

Missing Access Control

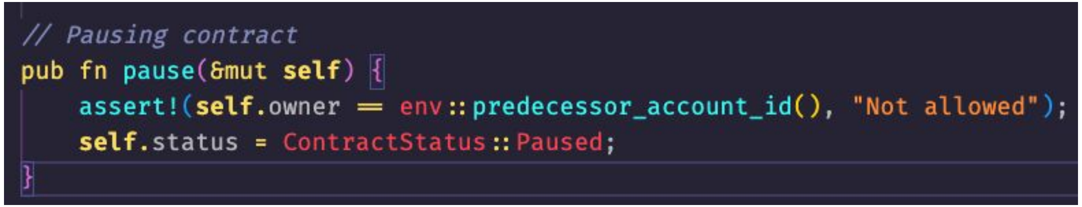
➔ Happens when admin-level/internal functions are exposed for anyone to call

Example: Exposed pausable function allows malicious actors to pause the contract.



```rust
// Pausing contract
pub fn pause(&mut self) {
    self.status = ContractStatus::Paused;
}

// Transfer logic
fn ft_transfer(&mut self, receiver_id: AccountId, amount: U128, memo: Option<String>) {
    assert!(self.status != ContractStatus::Paused, "Contract is Paused");
    self.token.ft_transfer(receiver_id, amount, memo);
}
```

Can be fixed by adding an ownership check

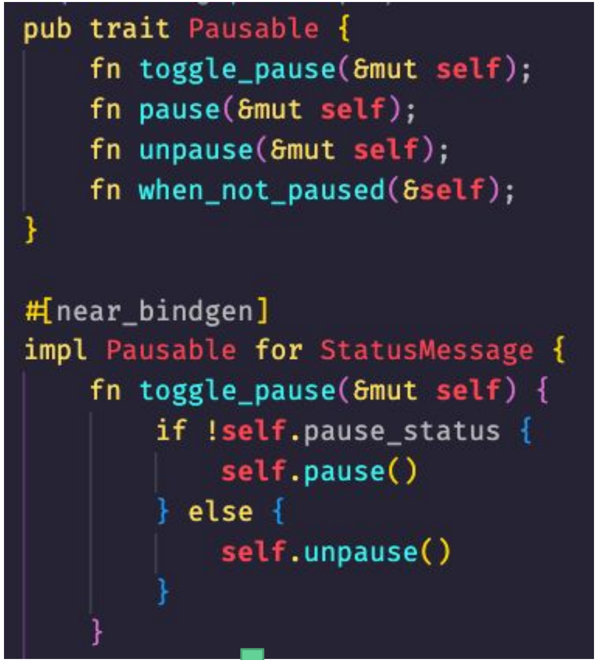


```rust
// Pausing contract
pub fn pause(&mut self) {
    assert!(self.owner == env::predecessor_account_id(), "Not allowed");
    self.status = ContractStatus::Paused;
}
```

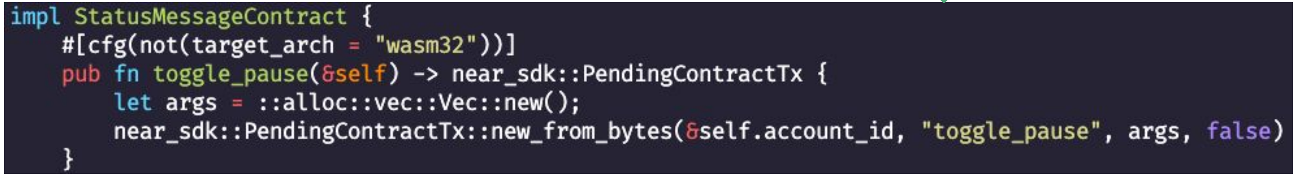
Incorrect #[near_bindgen] Usage

- ➔ #[near_bindgen] is a macro that is used to generate the necessary code to be a valid NEAR contract and expose the intended functions to be able to be called externally.
- ➔ Without knowing how it behaves behind the scenes we may accidentally put it on an internal trait implementation exposing its all internal functions to be called externally

- ➔ When #[near_bindgen] is added to trait implementation for the contract, it exposes all functions in that trait inside impl ContractNameContract{}
- ➔ We can run cargo-expand to see the expanded near_bindgen macro



```rust
pub trait Pausable {
    fn toggle_pause(&mut self);
    fn pause(&mut self);
    fn unpause(&mut self);
    fn when_not_paused(&self);
}

#[near_bindgen]
impl Pausable for StatusMessage {
    fn toggle_pause(&mut self) {
        if !self.pause_status {
            self.pause()
        } else {
            self.unpause()
        }
    }
```

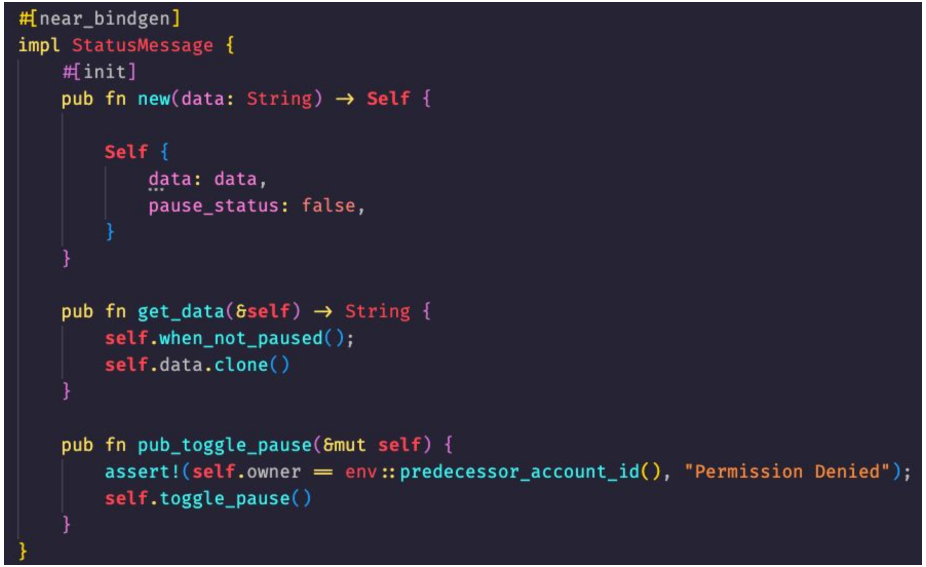


```rust
impl StatusMessageContract {
    #[cfg(not(target_arch = "wasm32"))]
    pub fn toggle_pause(&self) -> near_sdk::PendingContractTx {
        let args = ::alloc::vec::Vec::new();
        near_sdk::PendingContractTx::new_from_bytes(&self.account_id, "toggle_pause", args, false)
    }
```

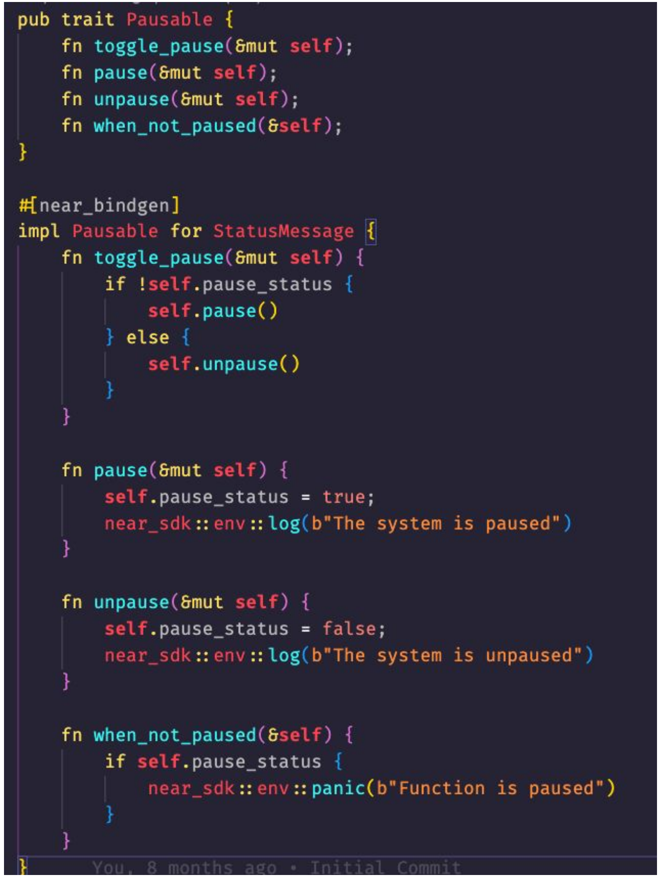
cargo-expand

- In this scenario a malicious actor can circumvent the ownership check in ***pub_toggle_pause*** by simply calling the ***toggle_pause*** function from the Pausable trait implementation.

- That is possible because we put #[near_bindgen] on top of the **Pausable** trait implementation



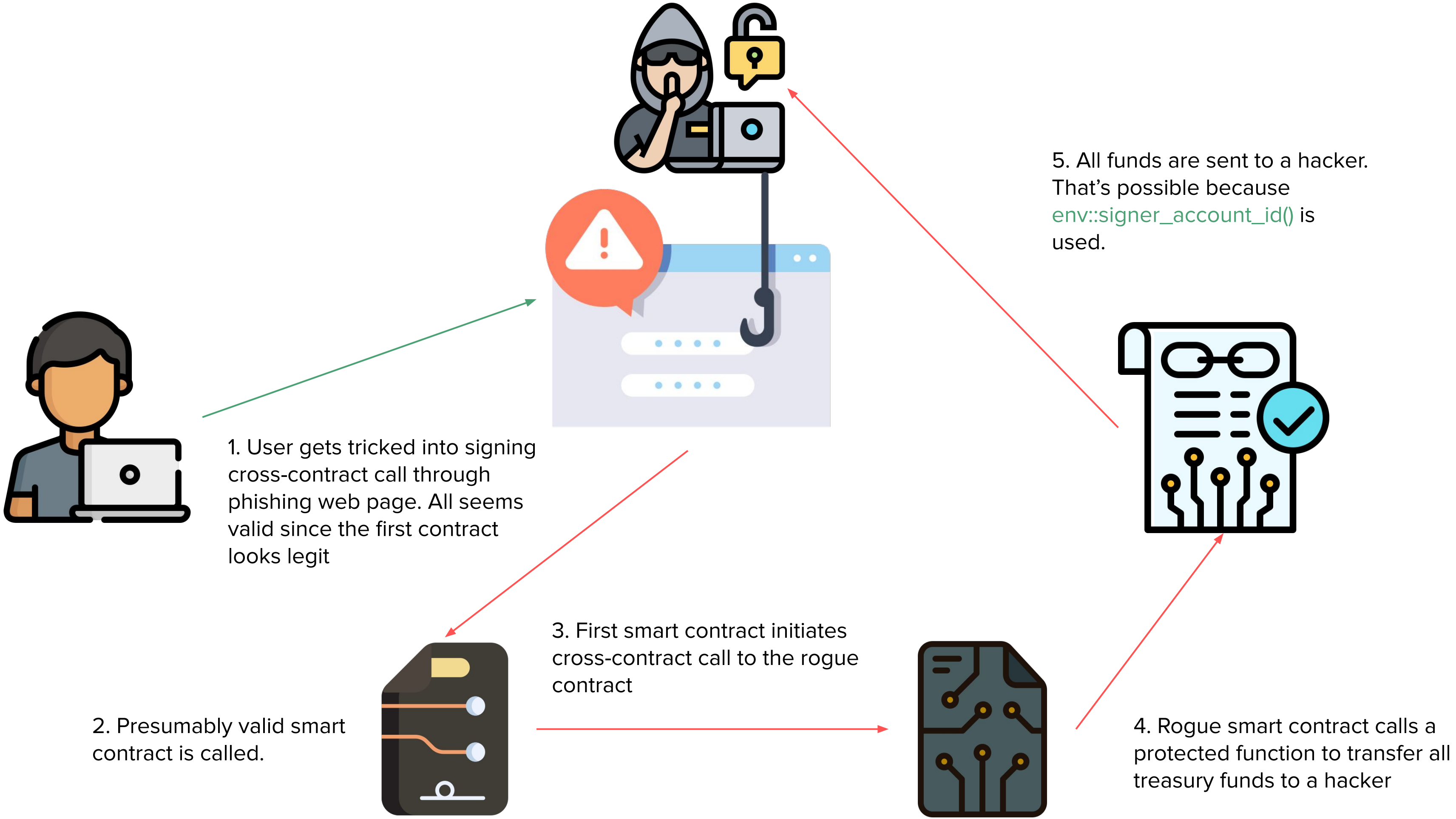
```
#[near_bindgen]
impl StatusMessage {
    #[init]
    pub fn new(data: String) -> Self {

        Self {
            data: data,
            pause_status: false,
        }
    }

    pub fn get_data(&self) -> String {
        self.when_not_paused();
        self.data.clone()
    }

    pub fn pub_toggle_pause(&mut self) {
        assert!(self.owner == env::predecessor_account_id(), "Permission Denied");
        self.toggle_pause()
    }
}
```



```
pub trait Pausable {
    fn toggle_pause(&mut self);
    fn pause(&mut self);
    fn unpause(&mut self);
    fn when_not_paused(&self);
}

#[near_bindgen]
impl Pausable for StatusMessage {
    fn toggle_pause(&mut self) {
        if !self.pause_status {
            self.pause()
        } else {
            self.unpause()
        }
    }

    fn pause(&mut self) {
        self.pause_status = true;
        near_sdk::env::log(b"The system is paused")
    }

    fn unpause(&mut self) {
        self.pause_status = false;
        near_sdk::env::log(b"The system is unpaused")
    }

    fn when_not_paused(&self) {
        if self.pause_status {
            near_sdk::env::panic(b"Function is paused")
        }
    }
}      You, 8 months ago • Initial Commit
```

Usage of env::signer_account_id()

- env::signer_account_id(): The id of the account that either signed the original transaction or issued the initial cross-contract call.
- env::predecessor_account_id(): The id of the account that was the previous contract in the chain of cross-contract calls. If this is the first contract, it is equal to ***signer_account_id***
- Usage of ***signer_account_id*** is risky in access control. It lets malicious actors use phishing attacks on contract owners to bypass ownership checks on protected functions through a chain of cross-contract calls.



1. User gets tricked into signing cross-contract call through phishing web page. All seems valid since the first contract looks legit

2. Presumably valid smart contract is called.

3. First smart contract initiates cross-contract call to the rogue contract

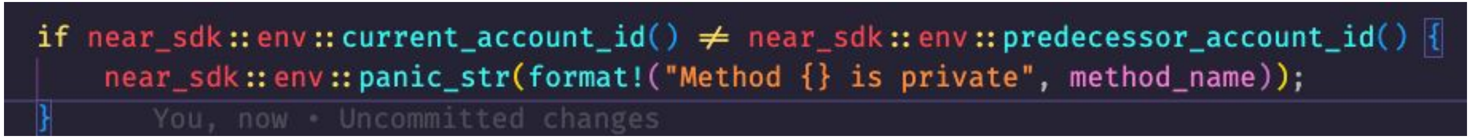
4. Rogue smart contract calls a protected function to transfer all treasury funds to a hacker

5. All funds are sent to a hacker. That’s possible because env::signer_account_id() is used.

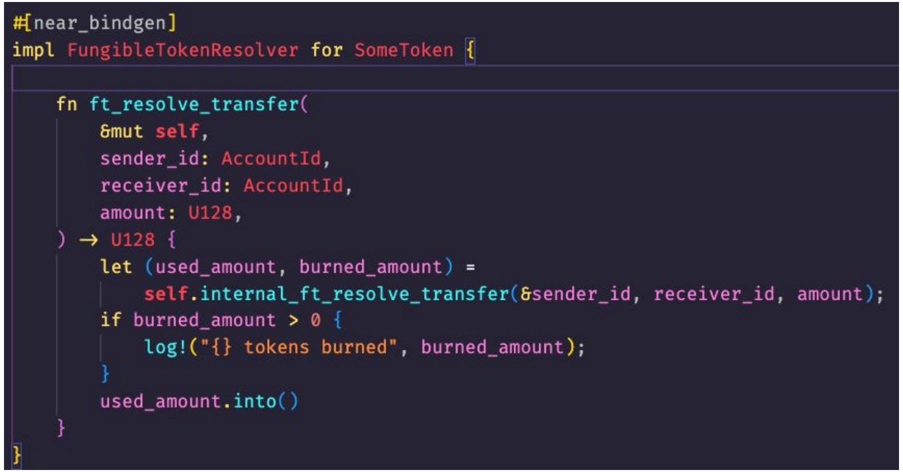
Public Callbacks

- ➔ Callbacks should be accessible on chain. However, only the current contract should have rights to call it
- ➔ Forgetting to put access control on callbacks might lead to devastating consequences
- ➔ To ensure that only current contract can call a callback we can use #[private] macro

It expands to:

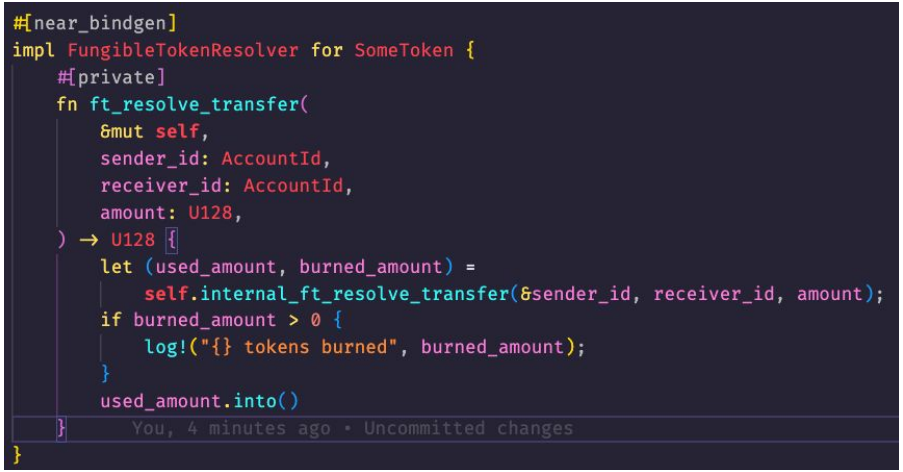


```
if near_sdk::env::current_account_id() ≠ near_sdk::env::predecessor_account_id() {
    near_sdk::env::panic_str(format!("Method {} is private", method_name));
}      You, now • Uncommitted changes
```



```rust
#[near_bindgen]
impl FungibleTokenResolver for SomeToken {

    fn ft_resolve_transfer(
        &mut self,
        sender_id: AccountId,
        receiver_id: AccountId,
        amount: U128,
    ) -> U128 {
        let (used_amount, burned_amount) =
            self.internal_ft_resolve_transfer(&sender_id, receiver_id, amount);
        if burned_amount > 0 {
            log!("{} tokens burned", burned_amount);
        }
        used_amount.into()
    }
}
```

Not Secure



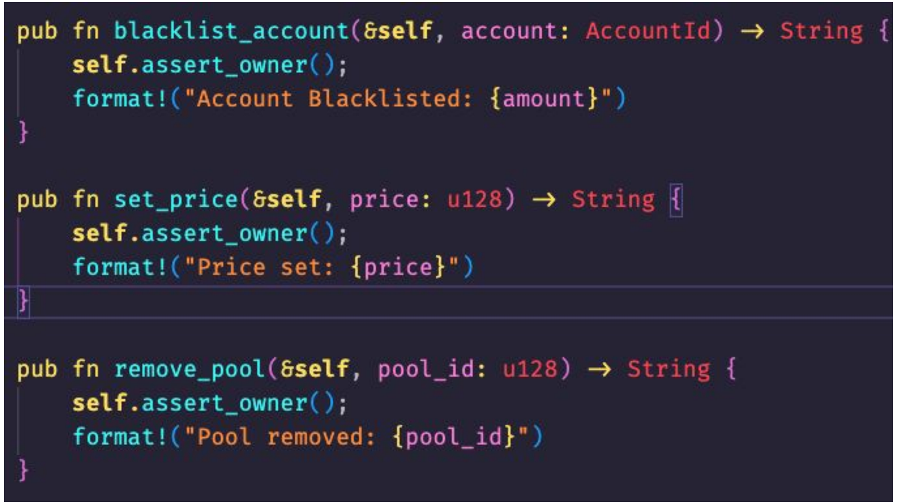
```rust
#[near_bindgen]
impl FungibleTokenResolver for SomeToken {
    #[private]
    fn ft_resolve_transfer(
        &mut self,
        sender_id: AccountId,
        receiver_id: AccountId,
        amount: U128,
    ) -> U128 {
        let (used_amount, burned_amount) =
            self.internal_ft_resolve_transfer(&sender_id, receiver_id, amount);
        if burned_amount > 0 {
            log!("{} tokens burned", burned_amount);
        }
        used_amount.into()
    }       You, 4 minutes ago • Uncommitted changes
}
```

Secure

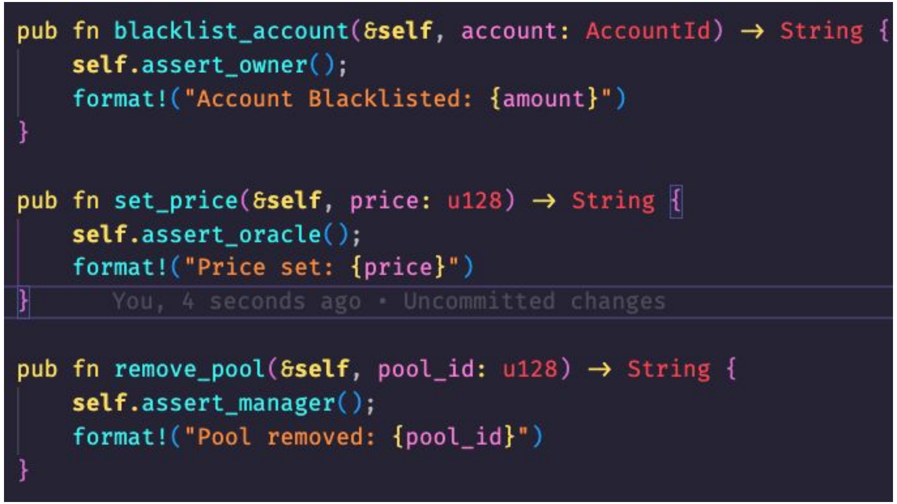
- ➔ In this example, *ft_resolve_transfer* is called as a callback to *ft_transfer call*.
- ➔ If callback is not protected, malicious actor can tamper with *sender_id* & *receiver_id* to drain funds.

Lack of Separation of Privileges

- ➔ One private key ruling inside the protocol harms decentralization and security.
- ➔ The compromise of the private key inevitably leads to the compromise of the whole protocol.
- ➔ To prevent the centralization of privileges and to minimize risk, each protocol has to consider implementing RBAC (Role-Based Access Control) mechanism with a different participant for each role.
- ➔ To harden further, each role must be governed by multisig with different participant sets for each role, or protocol should have a role governed by DAO, which is responsible for crucial operations.



```
pub fn blacklist_account(&self, account: AccountId) → String {
    self.assert_owner();
    format!("Account Blacklisted: {amount}")
}

pub fn set_price(&self, price: u128) → String {
    self.assert_owner();
    format!("Price set: {price}")
}

pub fn remove_pool(&self, pool_id: u128) → String {
    self.assert_owner();
    format!("Pool removed: {pool_id}")
}
```



```
pub fn blacklist_account(&self, account: AccountId) → String {
    self.assert_owner();
    format!("Account Blacklisted: {amount}")
}

pub fn set_price(&self, price: u128) → String {
    self.assert_oracle();
    format!("Price set: {price}")
}       You, 4 seconds ago • Uncommitted changes

pub fn remove_pool(&self, pool_id: u128) → String {
    self.assert_manager();
    format!("Pool removed: {pool_id}")
}
```

More Risk

Less Risk

- ➔ To get an idea of how to architecture robust RBAC, we can have a look at the AccessControl.sol contract in the OpenZepplin github repo
- ➔ NEAR equivalent can be found NRML (NEAR Rust Macros Library) repo. This repo is still a work in progress & unaudited, but you can get a general idea of the RBAC.
  - ◆ Architecture
  - ◆ Example
- ➔ More production ready repo can be found here: NEAR Contract Tools

DoS (Denial of Service)

DoS (Denial of Service)

- DoS happens when the whole or parts of the protocol/smart contract stop to function
- Possible causes include:
  - Logical Vulnerabilities
  - Storage Bloating
  - Prepaid Gas Exceeding
  - Log/Event Bombing

Usage of Unverified Accounts

- Accounts on NEAR follow certain rules described here:
  - Accounts | NEAR Protocol Specification
- If operations such as creation of account are performed on an unverified account string, it will inevitably cause panic
- So it is advised to use AccountId as type when we accept accounts. SDK will enforce validity of AccountId before usage (sdk version 4.0.0+)
- Depending on the protocol's design, this error might lead to DoS

Case Study (Appchain Registry)

- ➔ Logical DoS issue due to unverified account ids (HAL-02) was discovered during the engagement on Octopus Network Appchain Registry
  - ◆ Public report: Octopus Network Appchain Registry Public Audit Report
- ➔ Even though Octopus network team manually verifies the correctness of submitted account ids, from the smart contract perspective it is a critical flaw
- ➔ Appchain registry manages lifecycle of appchains in the Octopus network. Teams provide metadata about their appchain and if selected, it will be registered in the octopus network

Appchain Registration Process (High Level)

- Appchain team submits metadata such as *appchain_id*, *website_url*, *github_address* etc..
- People vote on appchain candidates by either upvoting or downvoting by attaching some amount of tokens
- During certain periods, votes are concluded using *conclude_vote* function.
- The top appchain with the majority of upvotes gets registered as a subaccount to a registry (myappchain.octopusappchainregistry.near)

Root Cause

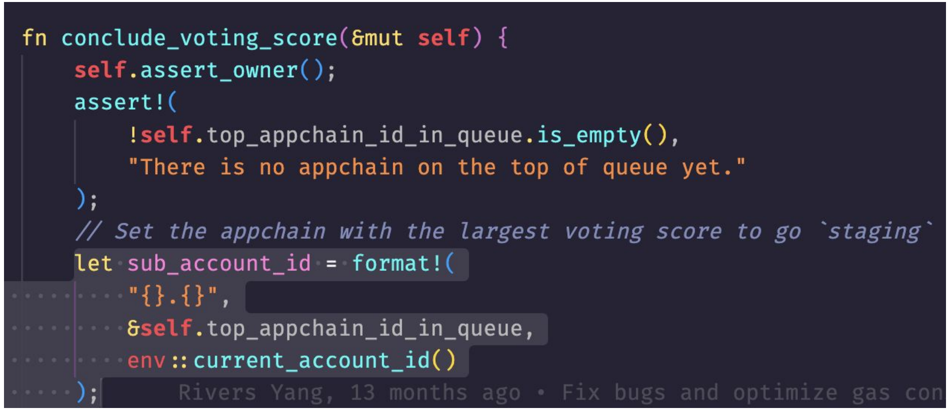
➔ In *conclude_voting_score*, smart contract creates a sub account by appending *appchain_id* of the top appchain to the *current_account_id*

➔ After all actions, at the end of the smart contract, *create_account()* method is called on a generated *sub_account_id*

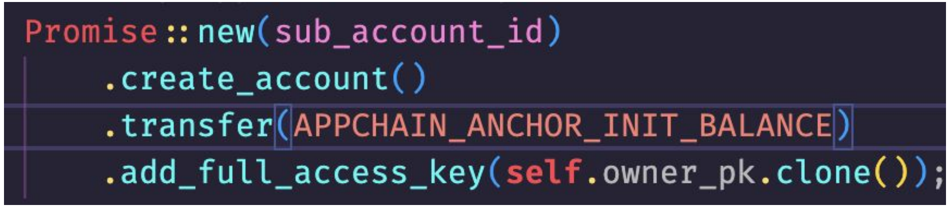
➔ Since appchain *account_id* is not verified during registration, it is possible to create invlaid sub account, which will cause panic during the account creation process

➔ Since there is no way to delete *top_appchain_id_in_queue*, the smart contract won't conclude votes anymore.

➔ The smart contract will get stuck on that *top_appchain_id_in_queue*



```rust
fn conclude_voting_score(&mut self) {
    self.assert_owner();
    assert!(
        !self.top_appchain_id_in_queue.is_empty(),
        "There is no appchain on the top of queue yet."
    );
    // Set the appchain with the largest voting score to go `staging`
    let sub_account_id = format!(
        "{}.{}",
        &self.top_appchain_id_in_queue,
        env::current_account_id()
    );      Rivers Yang, 13 months ago • Fix bugs and optimize gas con
```



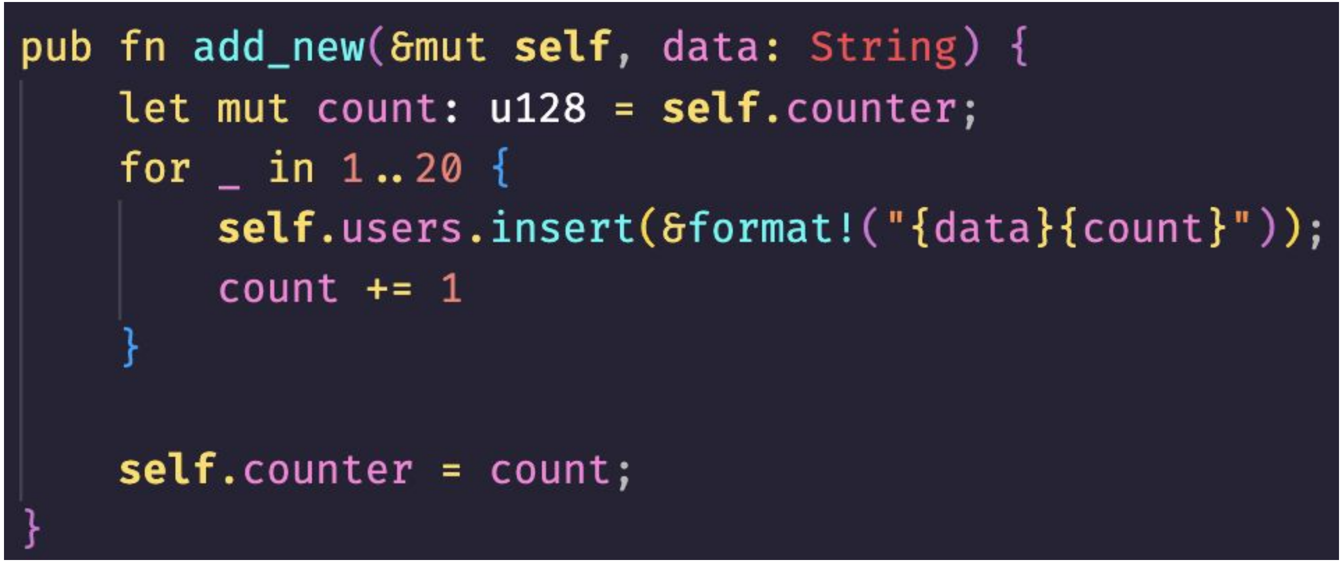
```rust
Promise::new(sub_account_id)
    .create_account()
    .transfer(APPCHAIN_ANCHOR_INIT_BALANCE)
    .add_full_access_key(self.owner_pk.clone());
```

Storage Bloating

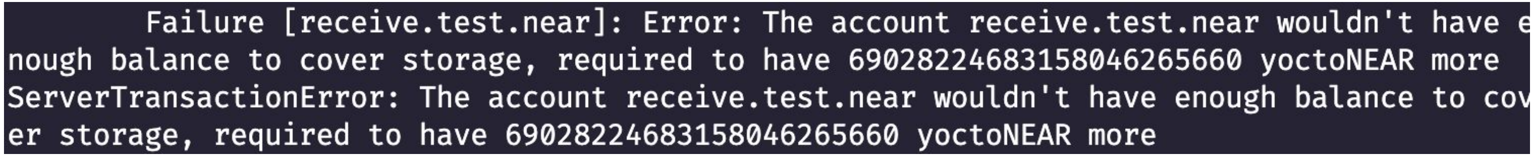
- ➔ Smart contracts pay for their storage
- ➔ Storage and contract’s free balance are inversely proportional. When the storage goes up, the smart contract free balance goes down
- ➔ If the cost per storage write is more expensive than the smart contract gain per storage write (%30 gas fees + deposit), then it opens up a possibility for a malicious actor to bloat the storage with dummy data
- ➔ This will eventually cause DoS since the smart contract won’t be able to accept any new data anymore until there is enough balance to cover the storage cost

Storage Bloating Calculations

- ➔ You could determine whether the particular function is vulnerable by doing these calculations:
  - ◆ **storage_per_write** = storage_contract_after - storage_contract_before (difference in bytes)
  - ◆ **cost_per_storage_write_near** = *storage_per_write* **/** 100000 (100kb of storage = ~1N)
  - ◆ **contract_gain** = contract_balance_after - contract_balance_before
  - ◆ If **contract_gain** < **cost_per_storage_write_near =>** function is vulnerable
- ➔ We can also estimate the cost of an attack on a vulnerable function
  - ◆ **initial_storage_cost** = storage_before_attack / 100000 (100kb of storage = ~1N)
  - ◆ **free_balance**= (contract_balance_before_attack / ONE_NEAR) - *initial_storage_cost*
  - ◆ **attacker_spent_per_write =** (attacker_balance_before - attacker_balance_after) / ONE_NEAR
  - ◆ **cost_of_attack** = Number of writes (Free balance worth of data / storage per write) * attacker spent pet write => ((*free_balance * 100000) / storage_per_write) * attacker_spent_per_write*



```rust
pub fn add_new(&mut self, data: String) {
    let mut count: u128 = self.counter;
    for _ in 1..20 {
        self.users.insert(&format!("{data}{count}"));
        count += 1
    }

    self.counter = count;
}
```

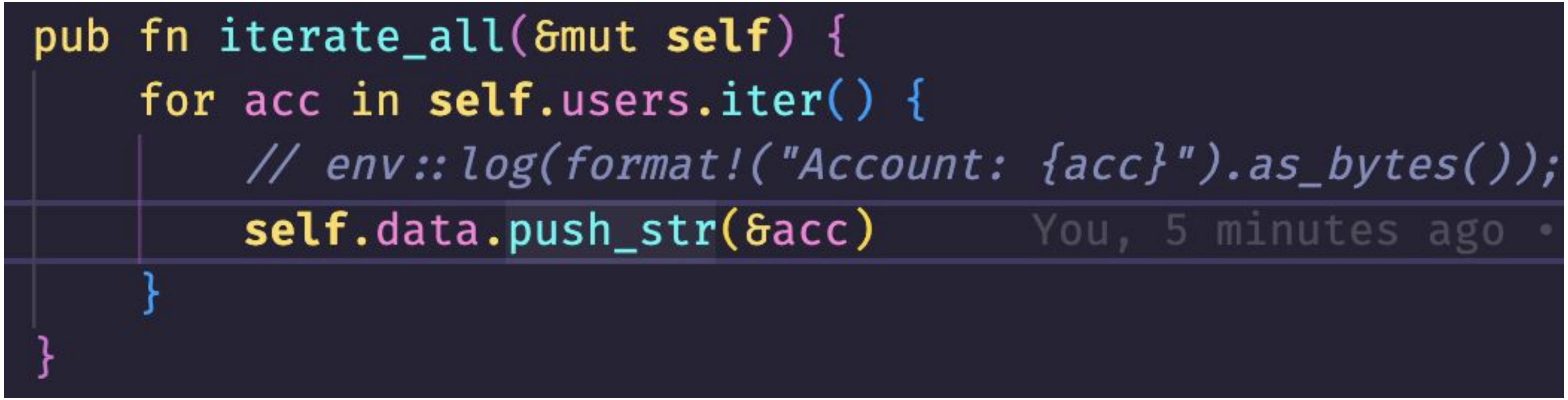


```
        Failure [receive.test.near]: Error: The account receive.test.near wouldn't have e
nough balance to cover storage, required to have 6902822468315804626560 yoctoNEAR more
ServerTransactionError: The account receive.test.near wouldn't have enough balance to cov
er storage, required to have 6902822468315804626560 yoctoNEAR more
```

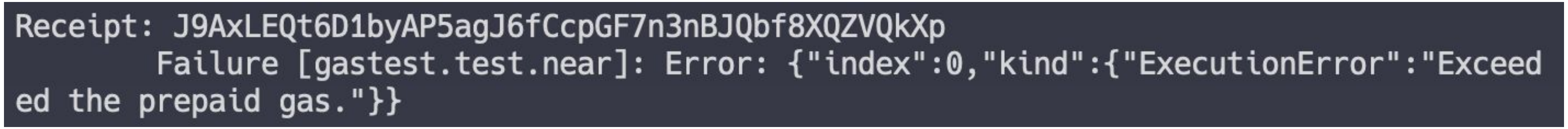
- ➔ In this simple example we have a function that adds data into storage
- ➔ Upon calling this function over and over again malicious actor can cause the smart contract to stop accepting additional storage writes due to lack of funds

Prepaid Gas Exceeding

- Currently, the maximum prepaid gas that can be attached to a smart contract call is ~300Tgas
- Exceeding this will cause the call to fail
- This is especially common in situations where the public function performs some operations by iteration over storage
- In that case, if the storage grows larger, there will be a point where the call to a smart contract will always fail by exceeding 300Tgas.
- Depending on the logic of the application, it might cause DoS



```
pub fn iterate_all(&mut self) {
    for acc in self.users.iter() {
        // env::log(format!("Account: {acc}").as_bytes());
        self.data.push_str(&acc)      You, 5 minutes ago •
    }
}
```

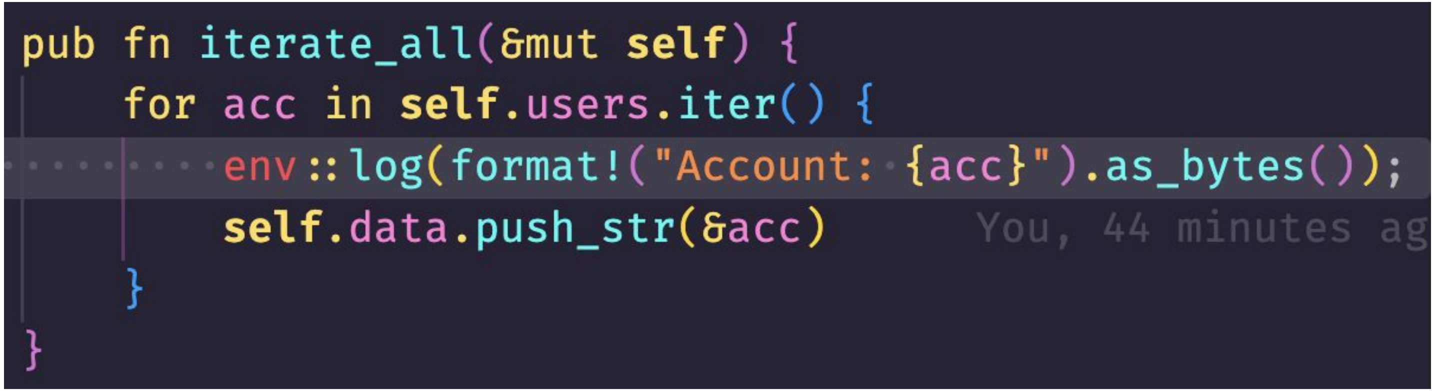


```
Receipt: J9AxLEQt6D1byAP5agJ6fCcpGF7n3nBJQbf8XQZVQkXp
        Failure [gastest.test.near]: Error: {"index":0,"kind":{"ExecutionError":"Exceed
ed the prepaid gas."}}
```

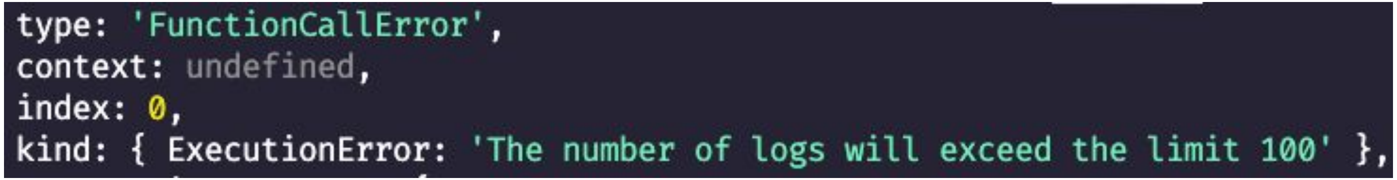
- ➔ In this example, we simply iterate over the set of user accounts
- ➔ However, when the number of accounts grows larger, we get prepaid gas error

Log/Event Bombing

- ➔ The function can emit max 100 events after single transaction
- ➔ Exceeding that limit will cause the call to fail
- ➔ This is especially common in situations where the public function performs some operations by iteration over storage and emitting a log after each action
- ➔ Depending on the logic of the application, it might cause DoS



```
pub fn iterate_all(&mut self) {
    for acc in self.users.iter() {
        env::log(format!("Account: {acc}").as_bytes());
        self.data.push_str(&acc)      You, 44 minutes ag
    }
}
```



```
type: 'FunctionCallError',
context: undefined,
index: 0,
kind: { ExecutionError: 'The number of logs will exceed the limit 100' },
```

- ➔ In this example, we simply iterate over the set of user accounts
- ➔ However, when the number of accounts grows larger, upon reaching the 100th event, we get an error

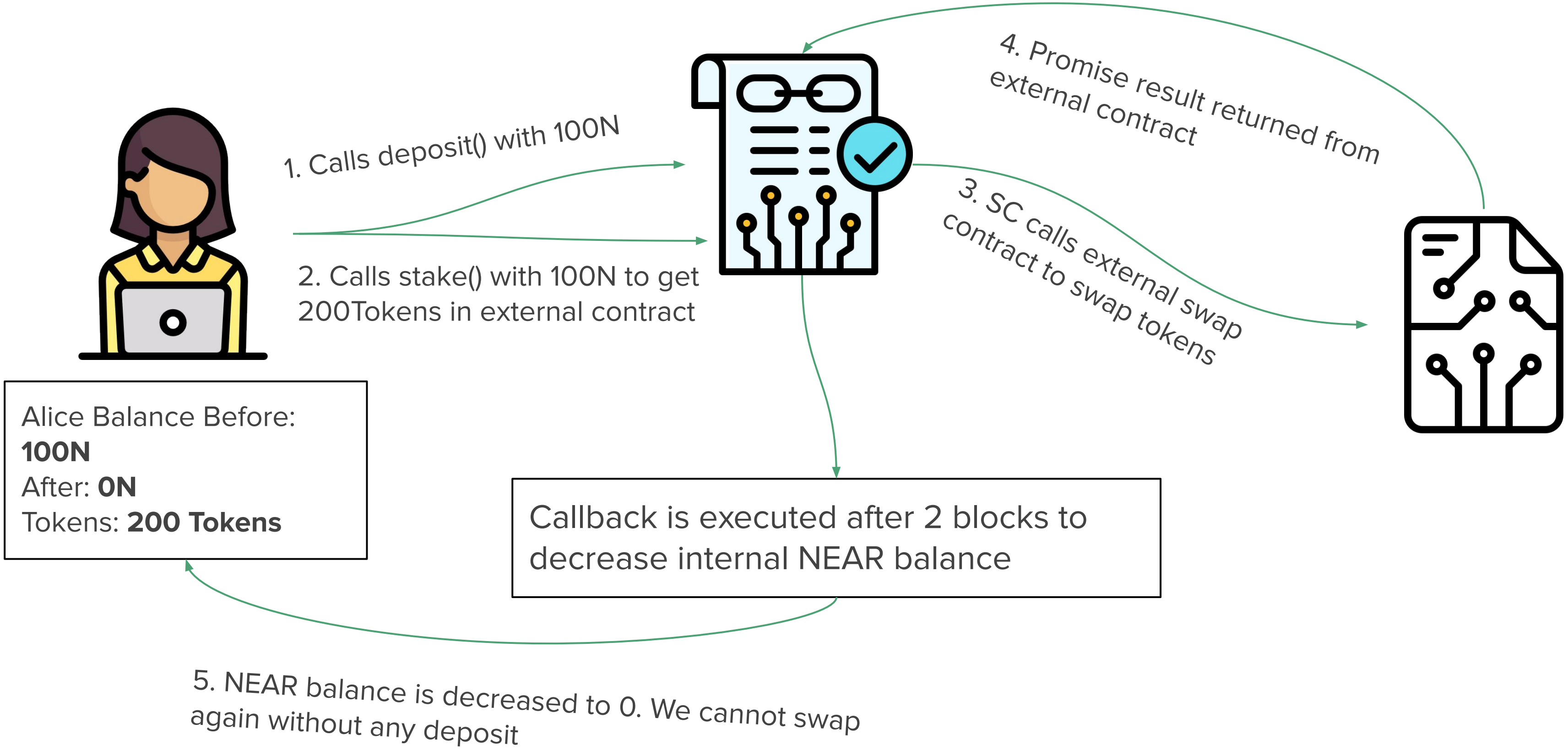
Race Condition/Reentrancy

Race Condition/Reentrancy

- ➔ There is a delay between the external call and callback (1-2 blocks)
- ➔ During that delay, any function can be called before the callback
- ➔ Depending on the logic of the contract, it can lead to serious vulnerabilities
- ➔ We should always handle our promises and revert state manually depending on external call result

Race Condition/Reentrancy

- Imagine a scenario where we have an external swapping contract which swaps NEAR to some token (1N = 2Tokens)
- The user calls the **deposit** function with 100N as an attached deposit
  - User’s internal NEAR balance increases
- The user calls **swap_near** function with 100N
  - SC calls an external contract with the given amount and user account
  - External contract swaps 100N for 200 tokens, increases user’s balance to 200Tokens and returns Promise success
- **resolve_swap** callback is executed after 2 block delay
  - Since promise was success, internal function, **decrease_near_balance** is called with the swapped amount
  - The user’s balance decreases to 0. Now, the user cannot swap NEAR for more tokens without depositing



1. Calls deposit() with 100N

2. Calls stake() with 100N to get 200Tokens in external contract

3. SC calls external swap contract to swap tokens

4. Promise result returned from external contract

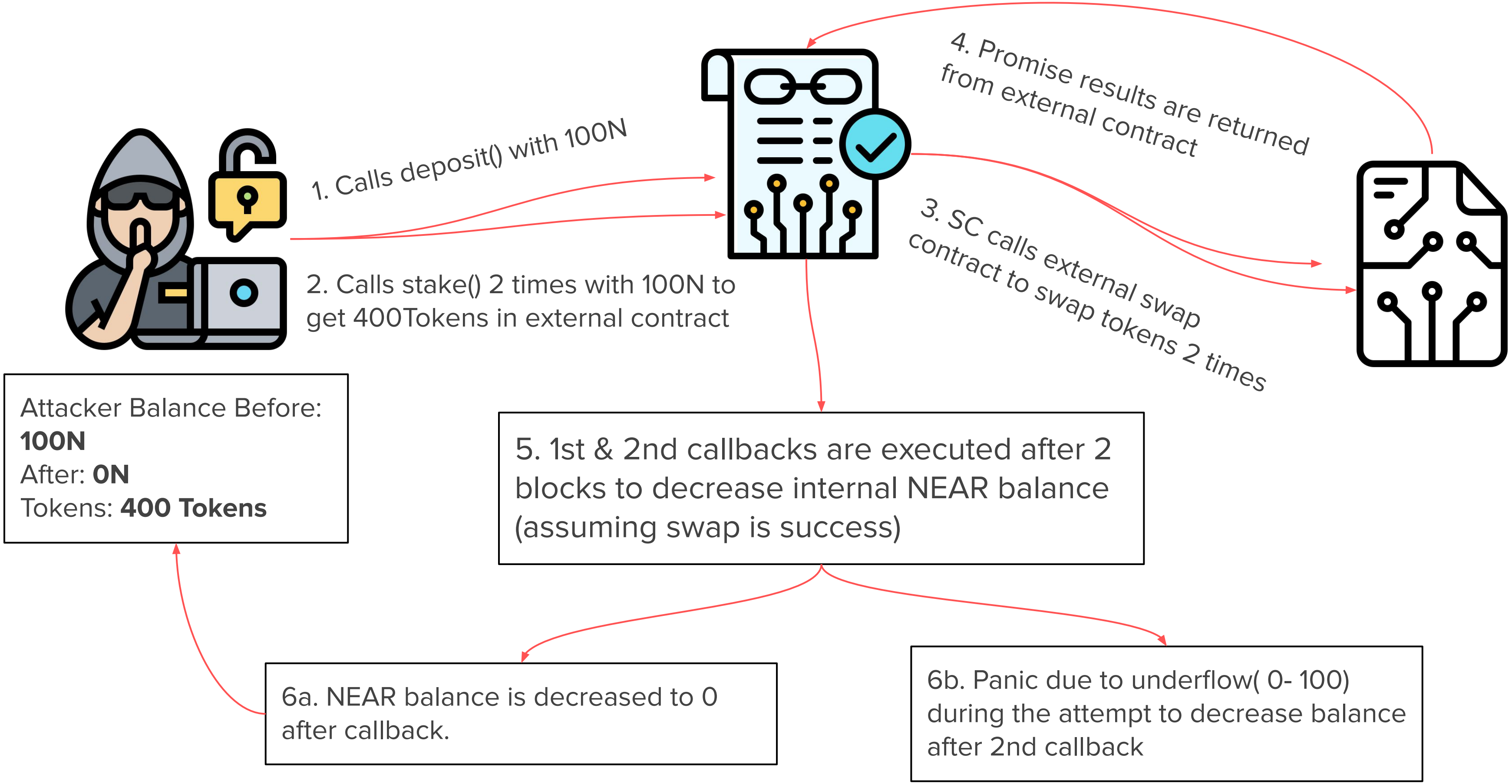
Callback is executed after 2 blocks to decrease internal NEAR balance

5. NEAR balance is decreased to 0. We cannot swap again without any deposit

Alice Balance Before:
100N
After: **0N**
Tokens: **200 Tokens**

Race Condition/Reentrancy

- What if we call the **swap** function twice with **100N**?
- The attacker calls the **swap** function twice with 100N
  - SC calls the external contract twice with the given amount and user account
  - External contract swaps 100N for 200 tokens, increases user’s balance to 200Tokens and returns Promise success
  - External contract swaps 100N for 200 tokens again, increases user’s balance to 400Tokens and returns Promise success
- After 2 block delay first **resolve_swap** callback is called.
  - Since the promise was success, internal function **decrease_near_balance** is called with the swapped amount (100N)
  - The attacker’s balance decreases to 0.
- After the same delay, the second **resolve_swap** callback is called.
  - Since the promise was success, internal function **decrease_near_balance** is called with the swapped amount (100N)
  - Since the attacker’s balance is already 0, we get panic on underflow (0 -100)
- So, due to a delay in the callback, the attacker managed to stake 100N to get 400 Tokens



1. Calls deposit() with 100N

2. Calls stake() 2 times with 100N to get 400Tokens in external contract

3. SC calls external swap contract to swap tokens 2 times

4. Promise results are returned from external contract

5. 1st & 2nd callbacks are executed after 2 blocks to decrease internal NEAR balance (assuming swap is success)

6a. NEAR balance is decreased to 0 after callback.

6b. Panic due to underflow(0- 100) during the attempt to decrease balance after 2nd callback

Attacker Balance Before: **100N**
After: **0N**
Tokens: **400 Tokens**

Logical Vulnerabilities

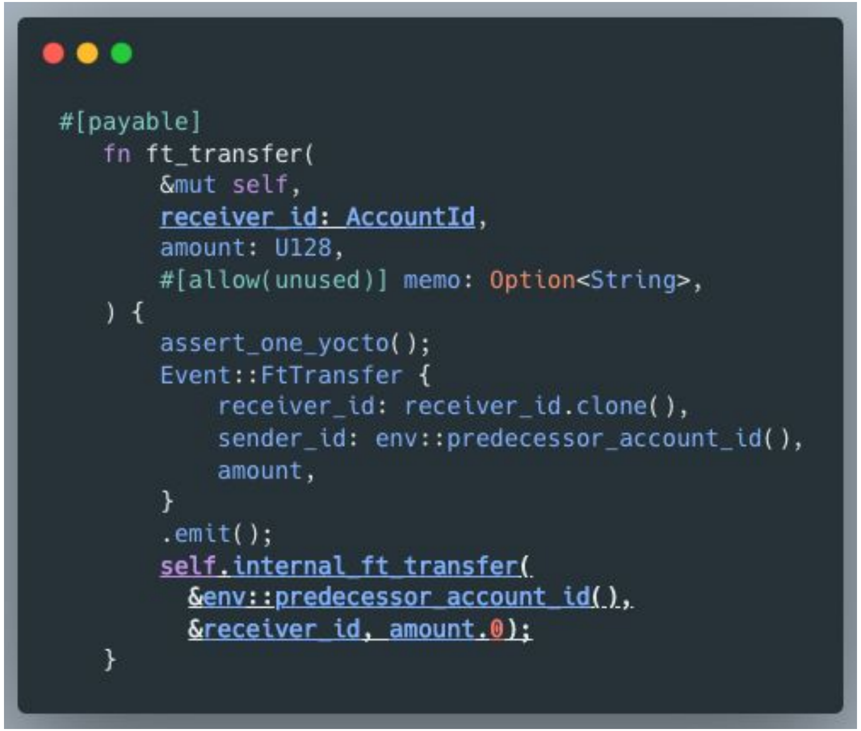
Logical Vulnerabilities

- ➔ Logical vulnerabilities are the hardest to spot. It requires a solid understanding of the project
- ➔ To catch these, it is better to build a threat model with all possible scenarios and then write scenario-based tests in accordance with those
- ➔ Formal verification also helps combat these by ensuring that our program behaves as expected
- ➔ Example scenario: User uses the same account as the sender and receiver during token transfer

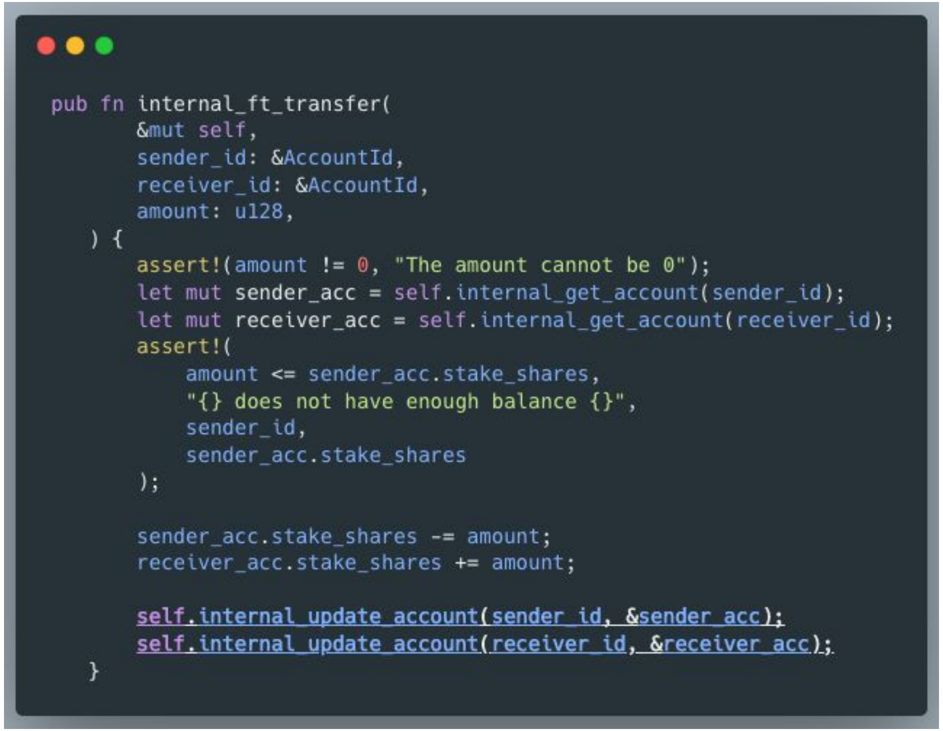
Case Study: Custom NEP-141 Token

- Sometimes projects decide not to inherit prebuilt NEP-141 contract from near-contract-standards but instead build their own or copy the original one with some changes
- This led to an incident in one of the real-world projects
    - StaderLabs Incident
    - Fix - Ensure that sender and receiver are not the same in ft_transfer
- The code was almost identical, but there was a small logical mistake which could have allowed a malicious actor to mint tokens from the contract

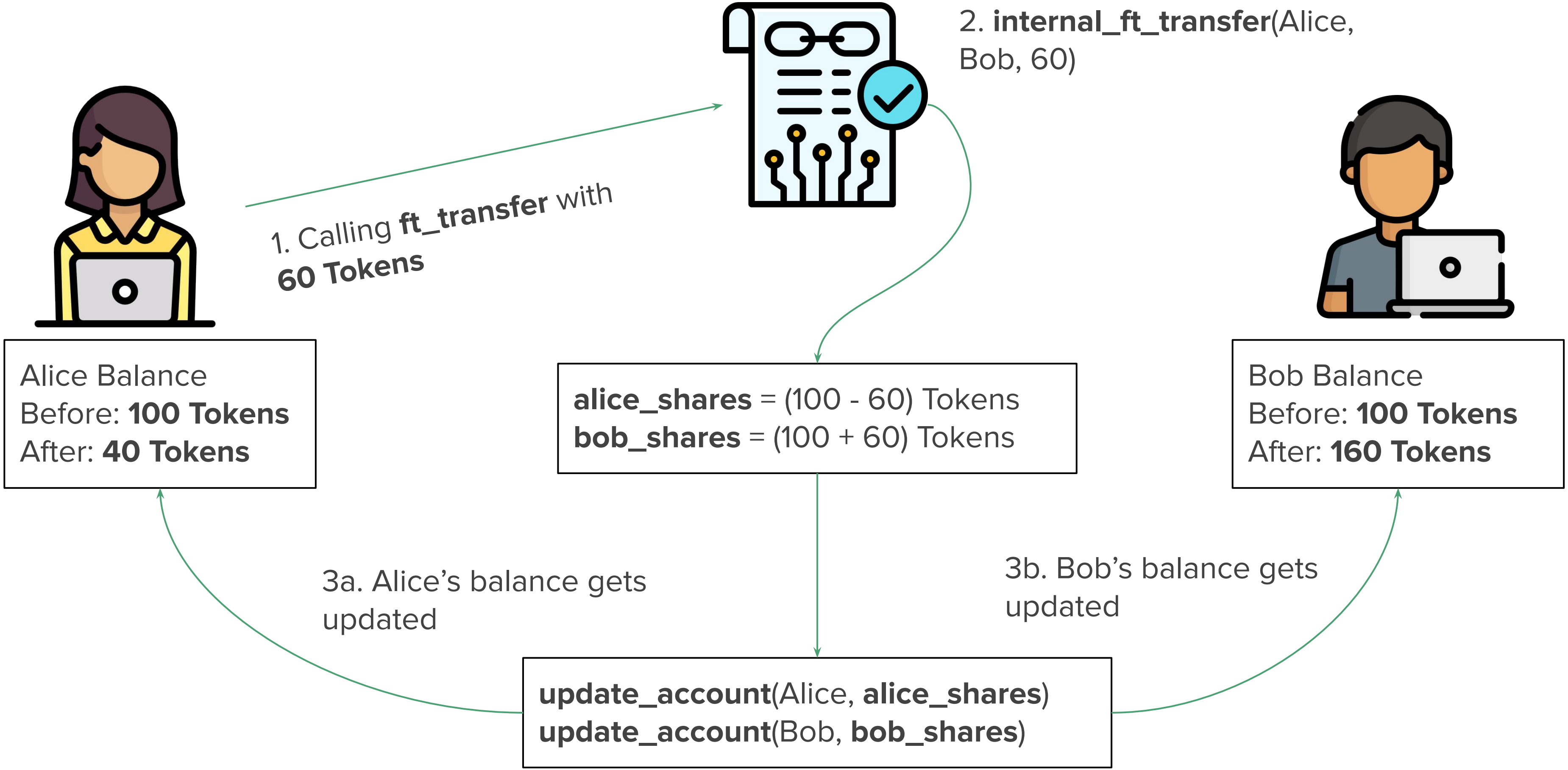
Case Study: Custom NEP-141 Token - Root Cause



```
#[payable]
    fn ft_transfer(
        &mut self,
        receiver_id: AccountId,
        amount: U128,
        #[allow(unused)] memo: Option<String>,
    ) {
        assert_one_yocto();
        Event::FtTransfer {
            receiver_id: receiver_id.clone(),
            sender_id: env::predecessor_account_id(),
            amount,
        }
        .emit();
        self.internal_ft_transfer(
          &env::predecessor_account_id(),
          &receiver_id, amount.0);
    }
```



```
pub fn internal_ft_transfer(
        &mut self,
        sender_id: &AccountId,
        receiver_id: &AccountId,
        amount: u128,
    ) {
        assert!(amount != 0, "The amount cannot be 0");
        let mut sender_acc = self.internal_get_account(sender_id);
        let mut receiver_acc = self.internal_get_account(receiver_id);
        assert!(
            amount <= sender_acc.stake_shares,
            "{} does not have enough balance {}",
            sender_id,
            sender_acc.stake_shares
        );

        sender_acc.stake_shares -= amount;
        receiver_acc.stake_shares += amount;

        self.internal_update_account(sender_id, &sender_acc);
        self.internal_update_account(receiver_id, &receiver_acc);
    }
```



1. Calling **ft_transfer** with **60 Tokens**

2. **internal_ft_transfer**(Alice, Bob, 60)

Alice Balance
Before: **100 Tokens**
After: **40 Tokens**

alice_shares = (100 - 60) Tokens
bob_shares = (100 + 60) Tokens

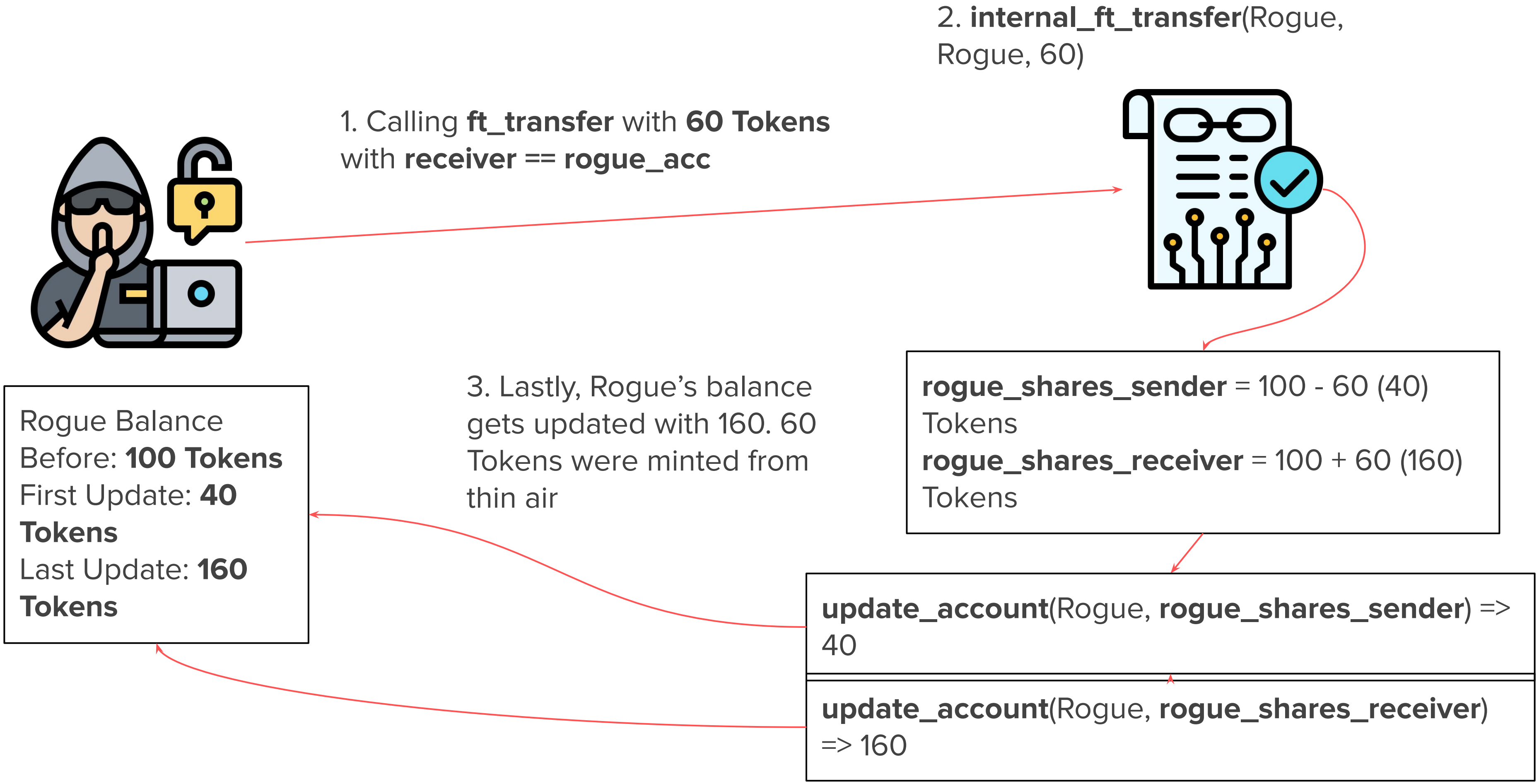
Bob Balance
Before: **100 Tokens**
After: **160 Tokens**

3a. Alice's balance gets updated

3b. Bob's balance gets updated

update_account(Alice, **alice_shares**)
update_account(Bob, **bob_shares**)

What if sender == receiver?



1. Calling **ft_transfer** with **60 Tokens** with **receiver == rogue_acc**

2. **internal_ft_transfer**(Rogue, Rogue, 60)

rogue_shares_sender = 100 - 60 (40) Tokens
rogue_shares_receiver = 100 + 60 (160) Tokens

update_account(Rogue, **rogue_shares_sender**) => 40

update_account(Rogue, **rogue_shares_receiver**) => 160

3. Lastly, Rogue's balance gets updated with 160. 60 Tokens were minted from thin air

Rogue Balance
Before: **100 Tokens**
First Update: **40 Tokens**
Last Update: **160 Tokens**

Case Study: Custom NEP-141 Token - Root Cause

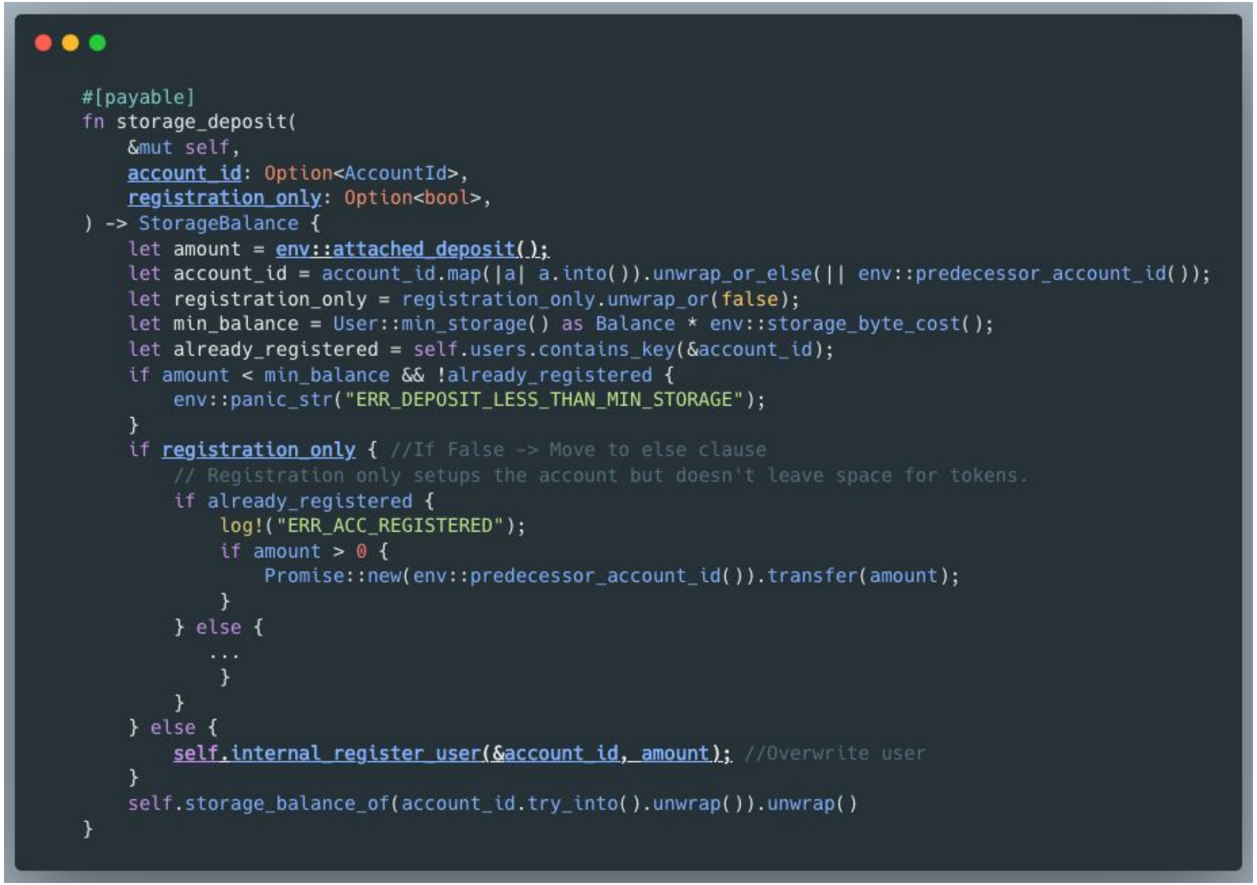
- ➔ In the scenario, since there is no check which ensures that **sender != receiver**, a malicious actor can take advantage of it.
- ➔ In case of two accounts being equal, **internal_update_account** overwrites the previous account update (sender) with the receiver’s account, which has extra tokens after transfer.

Case Study: Custom Storage Deposit

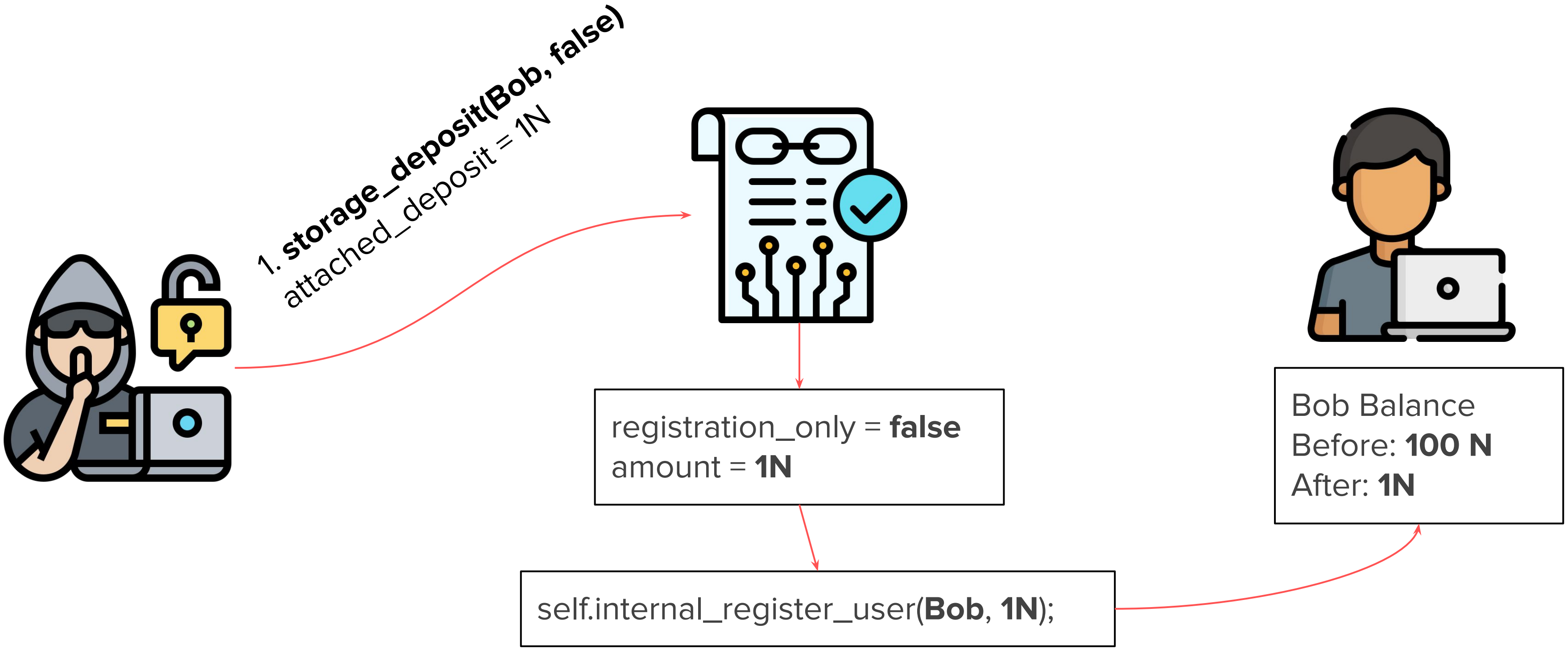
- ➔ Storage is not free, so it makes sense to take storage deposit from users and register them in the system
- ➔ In this real-world scenario, the project did not check whether the specified account was registered or not in the right place, which could lead anyone to overwrite any user’s data

Case Study: Custom Storage Deposit - Root Cause

- ➔ In this scenario, user can deposit a certain amount of coins under the passed account to make it registered
- ➔ The issue is that the check that ensures the account is not registered is inside the **if registration_only** clause.
- ➔ It lets malicious actor overwrite any account with a super small amount by specifying a target through **account_id** and making registration_only as **false**
- ➔ In the else clause, without any check, the passed **account_id** will get overwritten by the attacker



```rust
#[payable]
fn storage_deposit(
    &mut self,
    account_id: Option<AccountId>,
    registration_only: Option<bool>,
) -> StorageBalance {
    let amount = env::attached_deposit();
    let account_id = account_id.map(|a| a.into()).unwrap_or_else(|| env::predecessor_account_id());
    let registration_only = registration_only.unwrap_or(false);
    let min_balance = User::min_storage() as Balance * env::storage_byte_cost();
    let already_registered = self.users.contains_key(&account_id);
    if amount < min_balance && !already_registered {
        env::panic_str("ERR_DEPOSIT_LESS_THAN_MIN_STORAGE");
    }
    if registration_only { //If False -> Move to else clause
        // Registration only setups the account but doesn't leave space for tokens.
        if already_registered {
            log!("ERR_ACC_REGISTERED");
            if amount > 0 {
                Promise::new(env::predecessor_account_id()).transfer(amount);
            }
        } else {
           ...
            }
        }
    } else {
        self.internal_register_user(&account_id, amount); //Overwrite user
    }
    self.storage_balance_of(account_id.try_into().unwrap()).unwrap()
}
```



1. storage_deposit(Bob, false)
attached_deposit = 1N

registration_only = **false**
amount = **1N**

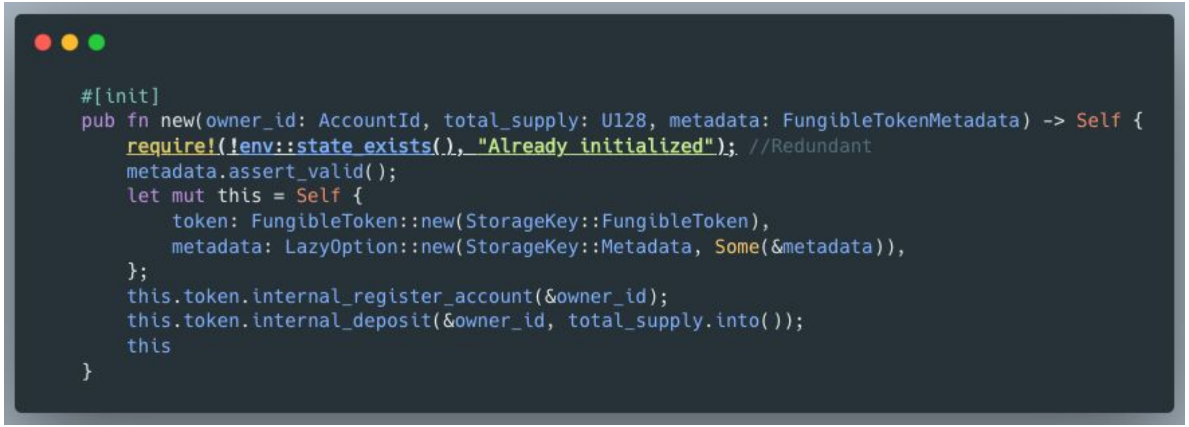
self.internal_register_user(**Bob**, **1N**);

Bob Balance
Before: **100 N**
After: **1N**

Tips & Best Practices

Redundant State Assertion

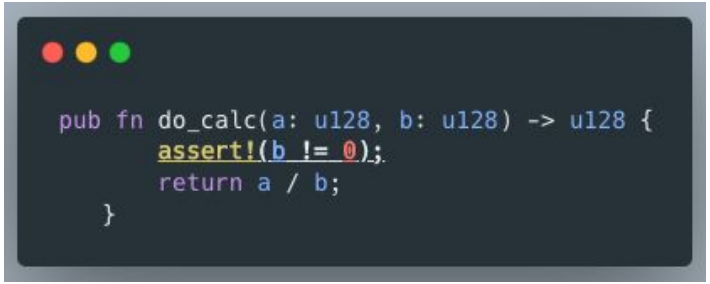
- ➔ **#[init]** macro behind the scenes already checks whether the state exists or not to prevent anyone from calling an initializer again. No need to add an additional check
  - ◆ Source



```
#[init]
pub fn new(owner_id: AccountId, total_supply: U128, metadata: FungibleTokenMetadata) -> Self {
    require!(!env::state_exists(), "Already initialized"); //Redundant
    metadata.assert_valid();
    let mut this = Self {
        token: FungibleToken::new(StorageKey::FungibleToken),
        metadata: LazyOption::new(StorageKey::Metadata, Some(&metadata)),
    };
    this.token.internal_register_account(&owner_id);
    this.token.internal_deposit(&owner_id, total_supply.into());
    this
}
```

Missing Zero Value Checks

- ➔ We always have to make sure that we cover any cases where we get 0 inside of an argument or during the calculation
- ➔ Mainly it is needed for optimization purposes. Usually, operations performed on 0 values are redundant. So if we assert early on it, we could potentially save on some computation
- ➔ Sometimes it also can cause some calculation errors, which might lead to logical bugs
- ➔ In addition, it can also lead to unnecessary panics, for instance, if we attempt to divide by 0.



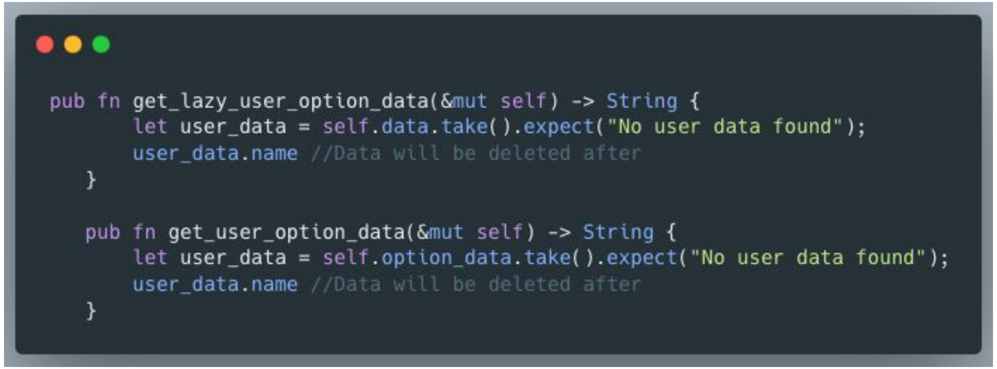
```
pub fn do_calc(a: u128, b: u128) -> u128 {
       assert!(b != 0);
       return a / b;
   }
```

Unchecked Attached Deposit

- Functions marked as **#[payable]** can accept the attached deposit since they do not have a check that prevents it:
  - Source
- In Solidity, there is an optimization trick to mark functions as payable to decrease gas usage. In NEAR this trick does not provide any visible benefit
- It makes sense to use **#[payable]** only if we are planning to use or assert **attached_deposit**
- Otherwise having **#[payable]** is redundant and it only increases possibility of someone accidentally sending money to our contract

Usage of .take()

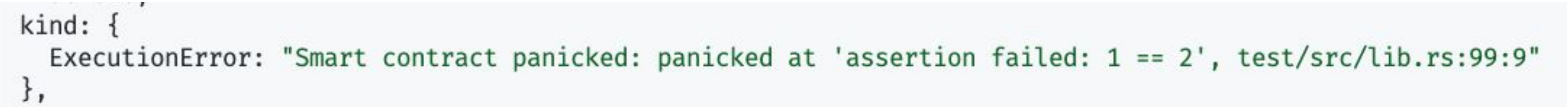
- ➔ When used on an *Option* and *LazyOption* types, **take()** deletes the value and replaces it with **None**
- ➔ We should be careful when we use **take()** since it takes the value and deletes it afterwards
- ➔ For instance, using **take()** to get token metadata will inevitably cause its deletion
- ➔ We should consider using **get()** instead to be sure that our value is preserved



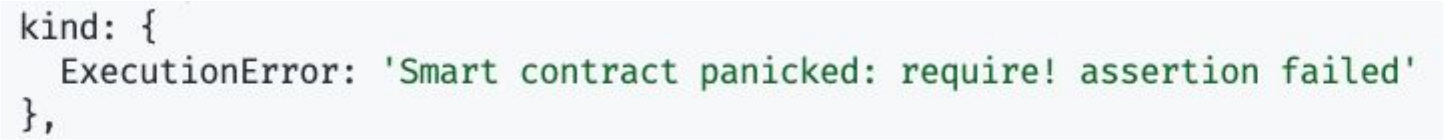
```
  pub fn get_lazy_user_option_data(&mut self) -> String {
        let user_data = self.data.take().expect("No user data found");
        user_data.name //Data will be deleted after
    }

    pub fn get_user_option_data(&mut self) -> String {
        let user_data = self.option_data.take().expect("No user data found");
        user_data.name //Data will be deleted after
    }
```

require!()

- Since SDK version 4.0.0, we have access to **require!()**
- It works the same as **assert!(),** but it does not include files or any rust-specific data in the panic message
- It reduces the code size and also obscures the exact place of error
- Obscuring the exact place of error might be useful if our project is closed source



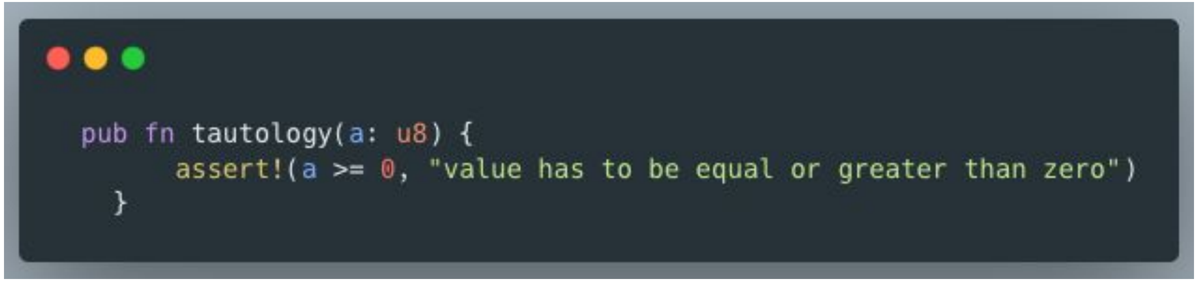
```
kind: {
  ExecutionError: "Smart contract panicked: panicked at 'assertion failed: 1 == 2', test/src/lib.rs:99:9"
},
```



```
kind: {
  ExecutionError: 'Smart contract panicked: require! assertion failed'
},
```

Tautological Expressions

- Those are expressions which always satisfy a specified conditions or are logically redundant
- For instance, checking whether uint is less than 0. Since unit type by definition cannot be less than 0, checking that is redundant. This check will always evaluate as false
- Another example is checking that value is larger than 0. If a value is uint, checking that value is not equal to 0 is enough in this case
- Real-world tautology: near-sdk-rs/core_impl.rs at master



```
pub fn tautology(a: u8) {
        assert!(a >= 0, "value has to be equal or greater than zero")
  }
```

2-step Ownership Transfer Process

- When we want to transfer a role to someone else, it is necessary to take precautions against human error
- Checking the grammatical validity of the account is not enough
- It is still possible to mistakenly transfer ownership to a valid account
- To prevent it, a 2-step ownership process should be implemented
- It can work like this:
  - The current owner proposes a new owner by calling **propose(new_owner)** function
  - The new proposed owner accepts ownership through the **accept_ownership()** function

Lack of Pausability

- ➔ Despite all testing & security measures taken, we cannot be %100 sure that our contracts won’t be hacked or won’t behave unexpectedly
- ➔ In case of those unfortunate events, to have damage control, every protocol should have the ability to pause its functionality
- ➔ It can be implemented by simply:
    - ◆ Having a state variable keeping track of status (*is_paused: bool*)
    - ◆ Creating a function called **toggle_pause()** to flip **is_paused** from **false** to **true** and vice versa
    - ◆ Asserting whether **is_paused** is **false** at the beginning of each function (can be encapsulated in a separate function). Throwing an error if **is_paused** is **true**

Usage of Incorrect JSON Type

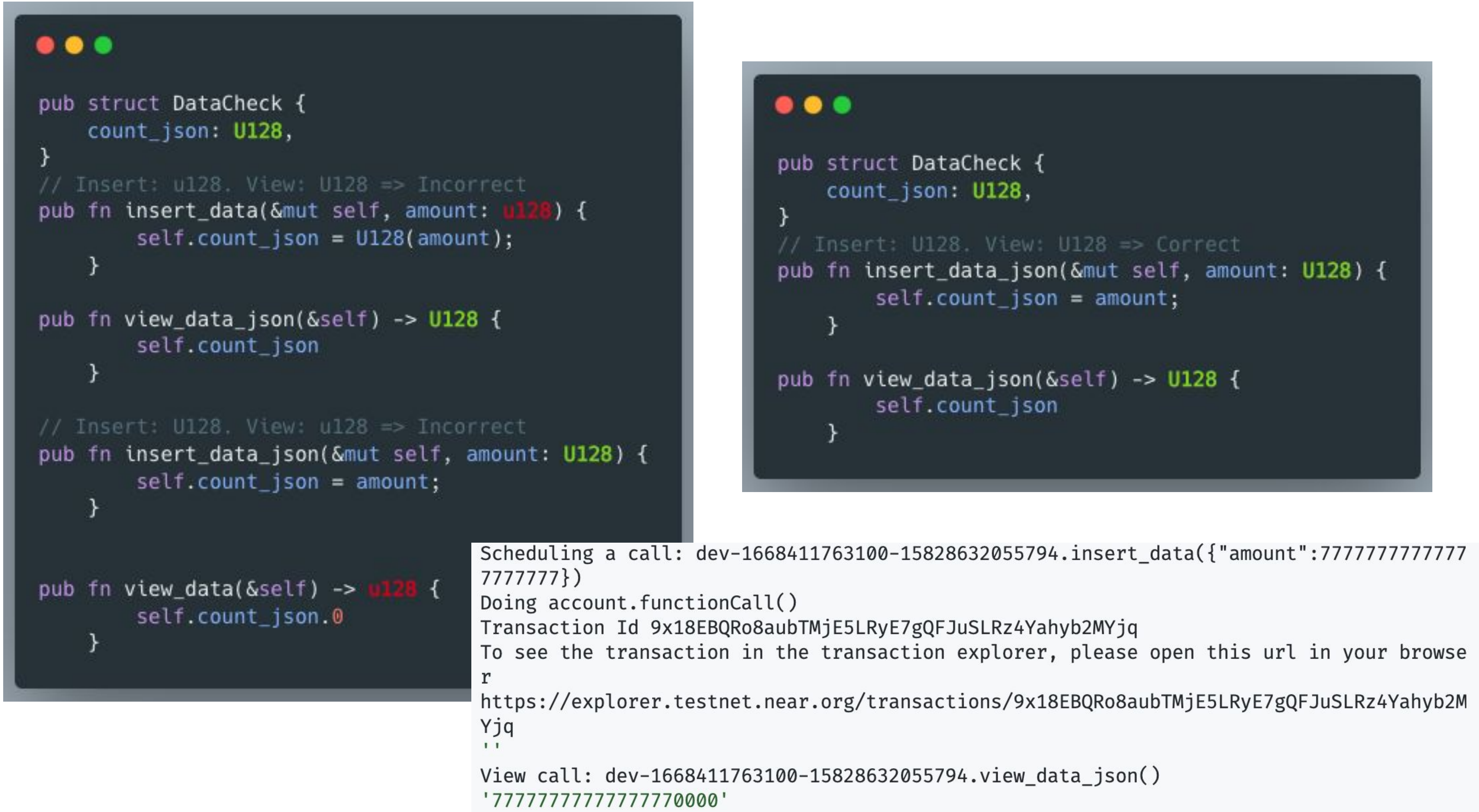
- ➔ In order to preserve interoperability, JSON has a double floating point precision limit in place for numbers. Since double floating-point precision has only 52 bits to represent mantissa (represents significant digits during conversion), the value represented correctly has to be in the range of **[-(2^53)+1, (2^53)-1]** with max safe integer be **9007199254740991**
- ➔ If values larger than **9007199254740991** are used, it might cause behaviors such as:
  - ◆ 9007199254740991 + 1 == 9007199254740991 + 2
  - ◆ 9007199254740991**1111** => 900719925474099**00000** (after parsing)
  - ◆ 7777777777777777**7777** => 7777777777777777**0000** (after parsing)

Usage of Incorrect JSON Type

- ➔ All of this can lead to severe logical & miscalculation bugs
- ➔ To prevent it from happening and still use numbers larger than *2^53-1*, we should input & output data in string format instead
- ➔ NEAR rust SDK provides us with custom JSON types to be used, such as U64 & U128: near_sdk::json_types - Rust
- ➔ If we try to input non-string value to a function that expects U64/U128, we will get an error
- ➔ We should use JSON types as types for values which may be larger than 2^53-1
- ➔ We should accept/return value with a JSON type if it can be larger than 2^53-1

Usage of Incorrect JSON Type

- ➔ If we accept an argument using JSON type but return value using regular types (u64/u128)
  - ◆ During the deserialization of 7777777777777777**7777,** we get the correct value back
  - ◆ During serialization, 7777777777777777**7777** becomes 7777777777777777**0000,** and we return that incorrect value back
- ➔ If we accept an argument using regular types(u64/u128) but return value using JSON types(U64/U128)
  - ◆ During the deserialization 7777777777777777**7777** becomes 7777777777777777**0000** and gets assigned to a variable
  - ◆ During serialization, we return an incorrect value inside of a variable



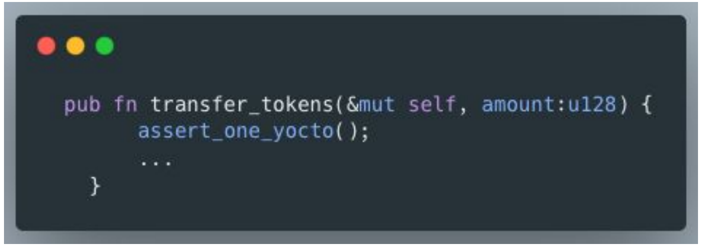
```rust
pub struct DataCheck {
    count_json: U128,
}
// Insert: u128. View: U128 => Incorrect
pub fn insert_data(&mut self, amount: u128) {
        self.count_json = U128(amount);
    }

pub fn view_data_json(&self) -> U128 {
        self.count_json
    }

// Insert: U128. View: u128 => Incorrect
pub fn insert_data_json(&mut self, amount: U128) {
        self.count_json = amount;
    }


pub fn view_data(&self) -> u128 {
        self.count_json.0
    }
```

```rust
pub struct DataCheck {
    count_json: U128,
}
// Insert: U128. View: U128 => Correct
pub fn insert_data_json(&mut self, amount: U128) {
        self.count_json = amount;
    }

pub fn view_data_json(&self) -> U128 {
        self.count_json
    }
```

```
Scheduling a call: dev-1668411763100-15828632055794.insert_data({"amount":7777777777777
7777777})
Doing account.functionCall()
Transaction Id 9x18EBQRo8aubTMjE5LRyE7gQFJuSLRz4Yahyb2MYjq
To see the transaction in the transaction explorer, please open this url in your browse
r
https://explorer.testnet.near.org/transactions/9x18EBQRo8aubTMjE5LRyE7gQFJuSLRz4Yahyb2M
Yjq
''
View call: dev-1668411763100-15828632055794.view_data_json()
'77777777777777770000'
```

Two Factor Authentication

- NEAR has two types of access keys
  - **Full Access Key**: Lets you do anything (call functions, transfer money, create/delete an account)
  - **Function Call Key**: Lets you only call specified smart contract's functions that do not require deposit
- When a user interacts with a smart contract through a website, the website gets a function call key to interact with the smart contract on behalf of the user
- It can be dangerous if a website is compromised. To protect against that, 2FA can be used in crucial functions. 2FA is triggered when we try to attach a deposit to a function call.
- We can assert that by requiring a user to attach one yocto when calling critical functions. It can be done through the prebuilt **assert_one_yocto()** function.
- If one yocto is attached, the user will get prompted to sign a transaction with a private key, which sits in a wallet.



```
pub fn transfer_tokens(&mut self, amount:u128) {
        assert_one_yocto();
        ...
  }
```