Rust Security

Safe Rust Security - Beginner to Intermediate

About

Guvenkaya is a security research firm specializing in Rust security, Web3 security of Rust-based protocols, and Web2 security. With our expertise, we provide both security auditing services and custom security solutions

near Polkadot substrate_ Rust FUEL ink!

Ecosystems We Support

01 NEAR

02 Polkadot

03 Kusama

04 Substrate Based Chains

05 Fuel

06 Any Web2 Industries

Founded by

Timur Guvenkaya founded Guvenkaya with a clear purpose: to fill a critical gap in Web3 security

Recognizing that Web3 isn't just about Ethereum and Solidity, Timur saw a significant risk in the lack of security expertise for Rust-based, Non-EVM ecosystems.

Before starting Guvenkaya, Timur made a mark at Invicti Security, designing a JWT engine for their flagship product used by Fortune 500 companies, banks and government agencies such as Verizon, Ford, and NASA. At Halborn Security, he specialized in Web3 security and led Rust security teams, auditing significant projects like Nodle, Composable Finance, Parallel.fi, and Octopus Network. Timur also created a NEAR Rust Smart Contract Security course and contributed to the SANS SEC-554 course by creating Rust and Substrate security sections.

Socials

Twitter:

- ➔ Timur Guvenkaya🦀 (@timurguvenkaya) / X
- ➔ Guvenkaya (@guvenkaya_sec) / X

LinkedIn:

- ➔ Timur Guvenkaya
- ➔ Guvenkaya

Github

- ➔ timurguvenkaya (Timur Guvenkaya) · GitHub
- ➔ Guvenkaya · GitHub

- 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
- Default Values
- Index Out Of Bounds
- Stack Overflow
- OOM (Out Of Memory)
- Crates With Vulnerabilities
- Handy Rust Tools

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

```rust
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>*

```rust
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(())
}
```

Unrecoverable Errors

- Unwind
  - Let’s application thread to shutdown relatively gracefully. Destructors are called, system resources are reclaimed etc. Instead of killing the whole application process only the panicking thread is stopped
  - Unwinds then can be caught and used to do customized panic handling
    - Web servers instead of only panicking in the command line can return “**500 Internal Server Error**” to users
    - These functions can be used to catch and modify unwinds & panics: **catch_unwind, set_hook, take_hook**

- Abort
  - The whole application process is killed. Directly crashes the program
  - Occurs on catastrophic errors like Stack Overflow or OOM errors
  - We can specify “**panic=abort**” in **Cargo.toml** which will abort the program on every panic. It decreases the binary size

```rust
fn main() {
    panic!("Panicked")
}
```

```
     Finished dev [unoptimized + debuginfo] target(s) in 1.13s
      Running `target/debug/basic`
thread 'main' panicked at 'Panicked', src/bin/basic.rs:2:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

```toml
[profile.release]
panic = "abort"
```

```
      Running `target/release/basic`
thread 'main' panicked at 'Panicked', src/bin/basic.rs:2:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
[1]    4542 abort      cargo run --bin basic --release
```

➔ When compiled in release mode all panics cause abort in the program.

➔ We can use **catch_unwind** to catch unwinding panic and display custom message to a user

```rust
use std::panic::catch_unwind;

▶ Run | Debug
fn main() {
    //Catching unwind
    let result: Result<{unknown}, Box<dyn Any + Send>> = catch_unwind(|| {
        panic!("Some Error!");
    });

    //Checking if result is error
    if result.is_err() {
        //Can be  display on web server as 500 error
        println!("Display Error");
    }
}
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.46s
     Running `target/debug/basic`
thread 'main' panicked at 'Some Error!', src/bin/basic.rs:5:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
Display Error
```

➔ To suppress/modify the default panic behavior for both unwind & abort, **set_hook & take_hook** can be used

```rust
use std::panic::{catch_unwind, set_hook, take_hook};

▶ Run | Debug
fn main() {
    // Setting new panic behavior
    set_hook(Box::new(|_| {
        println!("Custom panic hook");
    }));

    // Custom panic hook message used
    let _ = catch_unwind(|| {
        panic!("Another Error!");
    });

    // Removing custom hook
    let _ = take_hook();

    panic!("Default Panic")
}
```

```
Custom panic hook
thread 'main' panicked at 'Default Panic', src/bin/basic.rs:17:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

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”)

-

```rust
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

```
    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
```

```rust
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}")
}
```

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.

```rust
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

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)

▶ Run | Debug
```rust
fn main() {

    let a: u16 = 300;

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

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()*

```rust
use primitive_types::U256;

▶ Run | Debug
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
}

▶ Run | Debug
fn main() {
    println!("{}", divide_by_zero(20, 0))
}
```

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

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

- 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)

```rust
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
}
```

```rust
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
```

Default Values

- ➔ When you get Option or Result type, you can call **unwrap_or_default** to either unwrap or return a type default value.
- ➔ We should know our default values and know exactly when it is okay to use **unwrap_or_default**
- ➔ Mistakes can be made, especially when dealing with custom types where it is not always clear what default values are
- ➔ We should never use **unwrap_or_default** as a mean to silence compiler or avoid explicitly handling Option or Result types
- ➔ It led to a hack on the Acala Network: Hack Mentioned
- ➔ Patch: https://github.com/AcalaNetwork/Acala/pull/2520/files

```rust
#[derive(Debug)]
struct User {
    name: String,
    age: u8,
}

impl Default for User {
    fn default() -> Self {
        Self {
            name: String::from("Jake"),
            age: 34,
        }
    }
}

fn main() {
    let a: Option<u128> = Some(2);
    let b: Option<u128> = None;
    let c: Option<Option<u128>> = Some(None);
    let d: Option<Option<u128>> = None;
    let e: Result<u128, Error> = Ok(2);
    let f: Result<u128, Error> = Err(Error::from_raw_os_error(2));
    let g: Result<Option<u128>, Error> = Ok(Some(2));
    let h: Result<Option<u128>, Error> = Ok(None);
    let i: Option<User> = Some(User {
        name: String::from("Jane"),
        age: 22,
    });
    let j: Option<User> = None;
    let k: Result<User, Error> = Ok(User {
        name: String::from("Jane"),
        age: 22,
    });
    let l: Result<User, Error> = Err(Error::from_raw_os_error(2));
```

```
a: 2
b: 0
c: None
d: None
e: 2
f: 0
g: Some(2)
h: None
i: User { name: "Jane", age: 22 }
j: User { name: "Jake", age: 34 }
k: User { name: "Jane", age: 22 }
l: User { name: "Jake", age: 34 }
```

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

```rust
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;
    }
}

▶ Run | Debug
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 …)
- ➔ Allocation of large block of memory

```rust
fn main() {
    let mut size: usize = 1024; // Start with 1KB
    while let Some(double_size) = size.checked_mul(2) {
        let _ = Vec::<usize>::with_capacity(double_size);

        println!("Successfully allocated {} bytes", size);

        size = double_size; // Double the allocation size on each iteration
    }

    println!("Failed to allocate memory or reached maximum allocation size.");
}
```

```
Successfully allocated 4398046511104 bytes
memory allocation of 140737488355328 bytes failed
zsh: abort      cargo run --bin oom
```

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
- cargo-clippy - Linter to catch common mistakes in your rust code
- 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 - LSP and IDE 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