1. debug\_assert!() debug\_assert\_eq!() debug\_assert\_ne!()

The debug\_assert macro is used to compare an expression to true or not. This macro gets disappeared when build in release mode (“cargo build –release”)

CODE SNIPPET

debug\_assert!(false);

debug\_assert!(false,"false condition");

debug\_assert\_ne!(true,false);

Remediation: Use assert macros instead of debug\_assert\* macros

1. unsafe {} block

Rust does not check code written under unsafe{} block and thus developer can bypass rust protections and leads to any memory leaks/overflows.

Eg: allows us to execute assembly instructions with asm! macro and this can lead to memory corruptions if not coded properly. also race conditions occur due to unsafe block

CODE SNIPPET

unsafe{

// this allows to execute C functions, Winapi, read write memory locations without any checks, writing the const variable which can lead to race conditions etc. rust analyzer does not check

}

Remediation: Do not use unsafe block unless it’s the last option

1. CVE-2021-29922 – leading zeros does not get truncated in std::net library allows attackers to bypass X-Custom-IP header checks which can lead to SSRF

Link: <https://sick.codes/sick-2021-015/>

use std::net::IpAddr;

“000127.0.0.1”.parse::<IpAddr>().unwrap();

Remediation: Update Rust to the 1.53 or later

1. Command Execution via std::process::Command struct – allows to execute OS commands

Link to code: <https://github.com/winsecurity/Offensive-Rust/tree/main/command_exec>

CODE

// user inputs the url

let url = "127.0.0.1 && whoami";

let cmd = "/c ping -n 2 ".to\_owned()+ &url ;

let cmds:Vec<&str> =cmd.split(" ").collect();

let res =Command::new("cmd.exe").args(&cmds)

.output().unwrap();



Remediation: Sanitize user input or do not pass userinput into command arguments or do not use Command to execute commands

1. Path, open, read\* functions by default accepts any file path. The path should be sanitized before passing on to these functions.

CODE SNIPPET

use std::path::Path;

use std::fs::read\_to\_string;

let basepath =Path::new(r#"D:\rust\_practice\weakcrypto\src"#);

let file = "../../flag.txt";

let fullpath =basepath.join(file);

let content=std::fs::read\_to\_string(&fullpath).unwrap();

println!("{:#?}",fullpath.as\_os\_str());

println!("{}",content);

Remediation: Sanitize the userinput for ../ or whitelist only certain characters

1. CVE-2022-21658 – Race condition in fs::remove\_dir\_all() allows attacker to delete any directory recursively by changing the parameter to a symlink. remove\_dir\_all() checks the parameter if it’s a folder or symlink, if its folder then it will recursively deletes it. After check has been completed, attacker should fastly change this folder to a symlink pointing to another directory. now this function follows this symlink and deletes all

CODE SNIPPET

use std::fs;

fn main(){

fs::remove\_dir\_all(“/directoryname”);

}

Remediation: updating the rust to latest version or 1.5.1 above

1. SQL Injection can be possible in sqlite package which allows string queries instead of ORM. unsanitized input can lead to sql injection.

crates .io link: <https://crates.io/crates/sqlite>

CODE SNIPPET

in cargo.toml file -> sqlite = "0.30.3"

use sqlite;

let filepath = r#"D:\my\_notes\_db\notes.db"#;

let connection =sqlite::open(filepath).unwrap();

let userinput = "Powerview ; UPDATE Powerview SET Name='b' where ID=7-- -";

let query = "SELECT \* FROM ".to\_owned() + userinput;

connection.execute(query).unwrap();

// value gets updated to ‘b’

Remediation: Sanitize all the userinput or use prepared queries

1. Weak Encryption Algorithms.

the following algorithms are considered as weak and can be bruteforced.

RC2, RC4, MD4, MD5, DES, Blowfish, SHA1, ECB algorithm

* AES with ECB is insecure and ecb can be used from block-modes package: block-modes="0.8.1" in cargo.toml file

Link: https://crypto.stackexchange.com/questions/20941/why-shouldnt-i-use-ecb-encryption

* for RSA, private key length of minimum 2048 bits are required.

Link: <https://www.quaxio.com/exploring_three_weaknesses_in_rsa/>

CODE SNIPPETS

MD5

in cargo.toml file -> md5 = "0.7.0",

let hashed =md5::compute(b"testing");

DES

in cargo.toml file -> des = "0.8.1"

ECB Mode

in cargo.toml file -> ecb = "0.1.1"

use ecb;

also

use block\_modes::{Ecb};

use block\_modes::Ecb;

BlowFish algorithm

in cargo.toml file -> blowfish = "0.9.1"

use blowfish;

use blowfish::cipher;

SHA1

in cargo.toml file -> sha1 = "0.10.5"

use sha1;