Sigurnost operacijskih sustava i aplikacija

Laboratorijska vježba

Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva

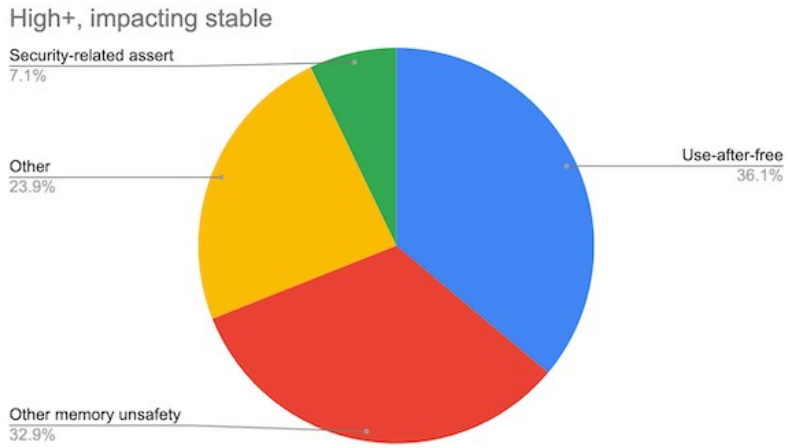
SIGURNO KODIRANJE U PROGRAMSKOM JEZIKU RUST

DANKO DELIMAR

Zagreb, 09. svibanj 2025

Uvod

Prema istraživanjima Chromiuma [1], čak 70% ozbiljnih sigurnosnih pogrešaka (definirano kao High i iznad) je vezano uz greške upravljanja memorijom, specifično memorijskom sigurnošću. Detaljniji pogled u to koje su to vrste memorijskih grešaka moguće je vidjeti na slici 1. Takve greške se događaju jer jezici poput C-a i C++ omogućavaju programerima potpunu kontrolu nad memorijom procesa i time otvaraju brojne mogućnosti za napraviti grešku. Jezici kao Java ili Go su doskočili tom problemu tako da tokom pokretanja programa (*at runtime*) imaju *Garbage collector* koji u pozadini skuplja memoriju koja se više ne koristi, a sami programeri ne mogu nikada pozvati funkciju *free* i time potencijalno izazvati greške (npr. *use-after-free, double free)*. Takvi jezici su dobar odabir za puno aplikacija, ali ponekad su potrebne veće performanse nego takvi jezici mogu ponuditi ili je sustav na kojem radimo izuzetno siromašan memorijom (razna ugradbena računala na primjer). Za takve situacije je razvijen programski jezik Rust koji u trenutku prevođenja (*at compile time)* može garantirati sigurnost od raznih vrsta memorijskih nesigurnosti ako se



Slika 1. Postotak memorijskih ranjivosti

slijedi striktan set pravila.
Cilj ove laboratorijske vježbe biti će proučiti kako Rust radi u usporedbi s programskim jezikom C i kako pravila i mogućnosti Rusta sprječavaju greške koje su inače u C-u moguće.

Zadatak

Vaš zadatak u ovoj vježbi biti će pretvoriti C kod koji je pun sigurnosnih grešaka u ekvivalentan Rust program koji će, zbog Rustovih mogućnosti, biti potpuno siguran. Napomena da će ranjivosti prikazane ovjde biti vrlo jednostavne i ne reprezentativne stvarnim ranjivostima. Ovo je kako ne bi trebalo pisati previše Rust koda koji zahtjeva

poznavanje mogućnosti Rusta koje su kompliciranije kako bi se uštedilo vrijeme čitatelja (npr. neće se pokazivati curenje memorije kako se ne bi objašnjavao Rc tip) Kod koji ćemo razmatrati je dan dolje sa komentiranim greškama koje se događaju u njemu:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void copy_user_input(void) {
    char buf[16];

    printf("Enter something: ");
    gets(buf); // funkcija gets čita dok ne pročita null byte. Ovo može
prouzročiti preljev spremnika.
    printf("You typed: %s\n", buf);
}

void use_after_free(char* data) {
    strcpy(data, "hello");

    free(data); // pozivamo free unutar funkcije, ali pokazavač smo primili
izvana i time bilo koji kod nakon funkcije više ne može koristiti taj
pokazivač, ali C ga neće spriječiti.
}

void logic_bug() {
    int a;
    scanf("%d", &a);

    if (a & 1 == 0) { // ovo je ustvari greška i ovaj kod će uvijek reći
Odd. operator == u C-u ima veći prioritet od operatora & pa će se uvijek
evaluirati prvi.
        printf("Even\n");
    } else {
        printf("Odd\n");
    }
}

void logic_bug2() {
    int a;
    scanf("%d", &a);

    if ((a & 1) == 0) { // Greška je da su sada Even i Odd krivo
postavljeni iako je redoslijed operatora dobar.
        printf("Odd\n");
    } else {
        printf("Even\n");
```

```
    }
}

int main(void) {
    // Jednostavna funkcija koja demonstrira preljev spremnika
    copy_user_input();

    // Vrlo trivijalan prikaz Use-after-free ranjivosti
    char *data = malloc(16);
    use_after_free(data);
    printf("UAF data: %s\n", data);

    // primjer logičke greške koju Rust može riješiti malim promjenama
naspram C-a
    logic_bug();

    // primjer logičke greške koju Rust ne može riješiti
    logic_bug2();
    return 0;
}
```

Teorijski podloga

Kako bismo mogli vidjeti kako Rust osigurava sigurnost programa potrebno je prvo definirati što znače sigurni tj. nesigurni programi. Rust definira memorijsku sigurnost kao izostanak nedefiniranog ponašanja (eng. *undefined behaviour)*. Ako ste programirali prije u C-u onda ste upoznati s konceptom nedefiniranog ponašanja. To su sve stvari koje C dopušta, ali u standardu jezika nije definirano što će se dogoditi ako se to desi. Neki primjeri toga uključuju: pristupanje većem indeksu polja nego je polje veliko, pozivanje funkcije free dvaput nad istom memorijom, dereferenciranje *null* pokazivača i slično. Što će se desiti kada napravimo nešto od navedenih akcija nije poznato i može biti ništa, može program ići dalje, ali dati krivi rezultat, može se desiti neka greška operacijskom sustava (npr. *segmentation fault, stack smashing detected...)* ili u specifičnim situacijama napadač može dizajnirati takav ulaz da preuzme kontrolu nad našim programom i tim potencijalno računalom. Filozofija Rusta je da ako eliminiramo sva ponašanja gore navedena (u trenutku prevođenja), tada ćemo imati sigurne programe koje napadači ne mogu zloupotrebljavati. Bitno je navesti da ovo ne znači da su naši programi bez ikakvih grešaka. Logičke greške koje radimo kao programeri su i dalje prisutne, kao i neke greške kao što je curenje memorije (eng. *memory leak)*. Za te greške je i dalje odgovoran programer kao i u bilo kojem drugom programskom jeziku.
Sada kada znamo što znači memorijska sigurnost možemo krenuti razmatrati kako to Rust postiže. Glavna ideja koju Rust koristi za osiguravanje sigurnosti programa se zove vlasništvo (eng. *ownership).* Vlasništvo je dozvola koju varijabla ima nad

memorijom (slično kao čitanje ili pisanje), a implementirano je unutar Rustovog *borrow checkera*. U svakom trenutku samo jedna varijabla može biti vlasnik neke memorije i ako se desi da se vlasništvo prenese (tako da sada neka druga varijabla također pokazuje na istu memoriju) vlasništvo prve varijable se invalidira. Vlasništvo služi kako bi Rust znao kada treba osloboditi neku memoriju. Za razliku od Jave ili Go-a koji imaju periodičan *garbage collector*, Rustov prevodilj automatski ubacije poziv na funkciju *free* u kod kada varijabla koja je vlasnik neke memorije izađe iz *scopea*. Takvim praćenjem memorije Rust može osigurati optimalne performanse bez da programeru dopusti mogućnost da sam zove *free*, jer sam jezik zna kada treba staviti *free*. Više o vlasništvu moguće je pročitati u Rust knjizi [2] u poglavlju 4.
Ako pogledamo Sliku 1. vidimo da su *Use-after-free* greške daleko naječešći specifičan tip grešaka, a vlasništvo ih potpuno eliminira. No postoje i druge greške za koje Rust ima druge načine osiguravanja sigurnosti. Tako na primjer se za pristupe indeksima polja ubacuju provjere (samo u *debug* verziji programa) koje provjeravaju je li pristup unutar parametara polja.

Postavke za vježbu

Za vježbu je potrebna instalacija Rusta i tekst editor Vašeg odabira.
Kako biste instalirali Rust možete slijediti službene upute da stranici jezika dostupne na: https://www.rust-lang.org/tools/install
Možete kod koji će biti opisan u rješenju isprobati koristeći Visual Studio Code i službenu Rust ekstenziju, možete koristiti JetBrainsov RustRover - https://www.jetbrains.com/rust ili ako želite možete i pisati u notepadu i sve ručno pokretati pomoću Rustovog *package managera – cargo.* Upute kako koristiti *cargo* su opisane u poglavlju 1 sekciji 3 Rust knjige [2].

Rješenje vježbe (do 5000 znakova)

Iako vas ne mogu spriječiti u tome da samo prekopirate Rust kod ispod moja preporuka bi bila da ga barem prepišete jer ćete dobiti puno bolje razumijevanje o tome što kod točno radi. Komentari u kodu su tu da objasne razlike između C i Rust verzije i kako točno Rust štiti od određenih grešaka.

```rust
use std::io::{self, Write};

/*
Funkcija u C-u koja je omogućavala preljev spremnika je bila gets.
Rust nema takve funkcije, ali nema ih iz razloga da direktno
dereferenciranje pokazivača nije moguće pa time nije ni moguće
direktno pisati po memoriji osim kroz sigurne API-je koje daju
Rustove kolekcije i tipovi podataka (String, vec, slice...). Ideja
koju možemo uzeti u naše korištenje C-a ili drugih jezika je da kada
god radimo direktno dereferenciranje memorije, bilo ono direktno
pomoću operatora dereferenciranja ili u pozadini pomoću neke
funkcije koja radi memcpy moramo biti jako oprezni jer je ta
operacija inherentno opasna.
*/
fn copy_user_input() {
    print!("Enter something: ");
    io::stdout().flush().unwrap();

    let mut input = String::new();
    if let Err(e) = io::stdin().read_line(&mut input) {
        eprintln!("I/O error: {e}");
        return;
    }
    println!("You typed: {}", input.trim_end());
}

/*
Postoje dvije verzije use_after_free funkcije koje demonstriraju
kako i kada Rust ubacuje free u naš kod. Prva verzija prima
varijablu baš kao String i time se vlasništvo prebacuje na lokalnu
varijablu data i kada data prestane postojati na kraju funkcije Rust
ubacuje free. No kako je Rust sada ubacio free jer ne postoji više
vlasnik te memorije varijabla data u funkciji main se više ne može
koristiti i ako maknete komentar nakon use_after_free u main vidjet
ćete da se kod više ne može prevesti. Ne može se prevesit jer
pokušavate čitati memoriju pomoću varijable koja nema vlasništvo nad
njome i prevoditelj to prepoznaje i sprječava vas da napravite Use-
after-free.
*/
fn use_after_free(mut data: String) {
    data.push_str("hello");
    // free ubačen ovdje.
}

/*
```

Ova verzija koristi reference umjesto samih varijabli. Kada se koriste reference Rust će samo posuditi vlasništvo nad memorijom lokalnoj varijabli data i zato neće ubaciti free na kraju ove funkcije nego će taj free završiti na kraju funkcije main gdje se nalazi originalni vlasnik. Za vježbu možete probati napisati kod koji koristi ovu verziju funkcije i vidjeti da možete isprintati hello nakon use_after_free2.

Također ekstra napomena je opet da Rust omogućava samo korištenje sigurnih API-ja koje implementiraju njegove kolekcije umjesto da direktno koristimo strcpy ili memcpy kao u C-u koje ako se koriste nad ulazom koje kontrolira korisnik mogu biti prilično katastrofalne.
*/
```
fn use_after_free2(data: &mut String) {
    data.push_str("hello");
    // free nije ubačen jer je vlasništvo samo "posuđeno"
}
```

```
/*
Ovo je samo brzinski primjer toga da Rust zamjenjuje prioritet operatora na način da je teže napraviti glupe greške. Takvih primjera promjena ima još i moguće ih je vidjeti u Rust knjizi [2].
*/
fn logic_bug() {
    let a = read_int("Enter integer: ");
    if a & 1 == 0 {
        println!("Even");
    } else {
        println!("Odd");
    }
}
```

```
/*
Ovo je samo primjer tipične logičke greške koju je programer zeznuo i tu Rust ne može pomoći. Ovo je trivijalan primjer, ali moguće je zamisliti kod koji zbog logičke pogreške dopusit nešto što se ne smije (pristup informacijama bez autentifikacije ili autorizacije na primjer) i takve greške i dalje treba tražiti i pregledati tipičnim načinima traženja grešaka (code review, penetracijski testovi, statička analiza, fuzzing...).
*/
fn logic_bug2() {
    let a = read_int("Enter integer: ");
    if a & 1 == 0 {
```

```
        println!("Odd");
    } else {
        println!("Even");
    }
}

// Pomoćna funkcija za čitanje integera da bude slično scanf("%d")
fn read_int(prompt: &str) -> i32 {
    print!("{prompt}");
    io::stdout().flush().unwrap();
    let mut line = String::new();
    io::stdin().read_line(&mut line).unwrap();
    line.trim().parse().unwrap_or(0)
}

fn main() {
    copy_user_input();

    let data = String::with_capacity(16);
    use_after_free(data);
    //println!("{data}");

    logic_bug();

    logic_bug2();
}
```

Zaključak

Jezici kao C i C++ su vrlo moćni, ali zbog toga i omogućavaju nebrojeni niz grešaka s kojima se kasnije u najboljem slučaju bave programeri, a u najgorem pravosudni sudci. Jezik kao što je Rust omogućava da pišemo sigurniji kod za koji je potrebna visoka razina performansi. Treba paziti da to ne znači da trebamo Rust koristiti za sve niti da sprječava sve vrste grešaka. Primjeri grešaka koje ne sprječava su curenje memorije i logičke greške i toga kao potencijalni budući Rust programeri moramo biti svjesni ako nekoga (ili sebe) savjetujemo oko toga koji programski jezik koristiti. Ono što je sigurno korisno je ako naučimo pravila na koja nas Rust prisiljava, prenijet ćemo ta pravila i u jezike koji nas na to ne prisiljavaju i time postati svjesniji memorijske sigurnosti i nesigurnosti.

Literatura

[1] Chromium project, Memory safety - https://www.chromium.org/Home/chromium-security/memory-safety
[2] The rust book, Brown University edition - https://rust-book.cs.brown.edu