# Ranljivosti v programih zaradi dvojnega sproščanja pomnilnika

Marin Gazvoda de Reggi, Matevž Pesek
Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, Večna pot 113, 1000 Ljubljana
mg4234@student.uni-lj.si, matevz.pesek@fri.uni-lj.si

## Izvleček

V računalništvu je učinkovito upravljanje pomnilnika ključno za delovanje programske opreme. Napake pri tem lahko vodijo do resnih varnostnih ranljivosti, ki omogočajo izvajanje poljubne kode ali krajo občutljivih podatkov. Prispevek obravnava podrobnosti napada na podlagi dvojnega sproščanja pomnilnika (*double free*) in prikazuje primer ranljivosti v programu za upravljanje podatkovne baze. Pojasni, kako lahko napadalec pridobi administratorske pravice brez gesla. Predlagane rešitve vključujejo uporabo pomnilniško varnih jezikov, orodij za statično analizo, omejevanje privilegijev in defenzivno programiranje. Poudarjen je pomen celovitega pristopa k varovanju programske opreme pred takimi napadi.

**Ključne besede:** dvojno sproščanje, napad, upravljanje pomnilnika, varnostne ranljivosti

## BINARY VULNERABILITIES DUE TO DOUBLE FREE

## Abstract

In computer science, effective memory management is crucial for software performance. Mistakes in this area can lead to serious security vulnerabilities, such as arbitrary code execution or theft of sensitive data. This paper discusses the details of a double-free memory attack and illustrates an example of a vulnerability in a database management program. It explains how an attacker can gain administrative rights without a password. Proposed solutions include using memory-safe languages, static analysis tools, privilege restriction, and defensive programming. The importance of a comprehensive approach to protecting software from such attacks is emphasized.

**Key words:** double free, attack, memory management, security vulnerabilities

## 1 UVOD

V sodobni digitalni dobi so računalniški sistemi postali nepogrešljiv del naše družbe, prisotni v vseh vidikih življenja in delovanja. Programska oprema, ki poganja te sisteme, igra ključno vlogo pri upravljanju procesov od osebnih naprav do kompleksne infrastrukture. Ta vseprisotnost pa prinaša nove izzive, predvsem na področju varnosti in zanesljivosti.

Z naraščajočo odvisnostjo od digitalnih sistemov postajajo posledice varnostnih ranljivosti vse resnejše. Vdori lahko vodijo do kraje podatkov, finančnih izgub, ogrožanja zasebnosti in motenj v delovanju kritične infrastrukture. Te grožnje niso več omejene le na specializirane skupine, temveč postajajo dostopne širšemu krogu potencialnih napadalcev zaradi razširjenosti orodij in znanja o izkoriščanju ranljivosti.

Eno ključnih področij, ki zahteva posebno pozornost, je upravljanje pomnilnika. To je še posebej pomembno v jezikih z ročnim upravljanjem, kot sta C in C++, ki se pogosto uporabljajo za razvoj sistemske programske opreme zaradi hitrosti in učinkovitosti. Vendar pa lahko ravno ta fleksibilnost vodi do subtilnih napak z resnimi varnostnimi posledicami.

Med najpogostejšimi napakami pri upravljanju pomnilnika so dvojno sproščanje (*double free*), uporaba blokov po sprostitvi (*use-after-free*) in uhajanje pomnilnika (*memory leak*). Napadalci lahko te napake izkoristijo za izvajanje poljubne kode, pridobivanje občutljivih podatkov ali destabilizacijo sistema. Kljub dolgoletnemu zavedanju o teh težavah ostajajo te ranljivosti razširjene in predstavljajo izziv za varnost programske opreme.

V tem članku se osredotočamo na analizo napada, ki izkorišča ranljivost dvojnega sproščanja pomnilnika. Za ilustracijo predstavljamo primer v programu, ki simulira preprostega upravitelja podatkovne baze, in pokažemo, kako lahko napadalec izkoristi to navidez nedolžno napako za pridobitev administratorskih pravic brez poznavanja gesla.

Razumevanje mehanizma teh napadov je ključno za razvoj varne programske opreme. S poglobljenim vpogledom lahko razvijalci bolje razumejo potencialne grožnje in razvijejo učinkovitejše strategije za njihovo preprečevanje. Takšna analiza pomaga tudi pri oblikovanju boljših orodij za odkrivanje in preprečevanje tovrstnih ranljivosti.

Cilj tega članka je dvojen: prispevati k boljšemu razumevanju mehanizma napadov, ki izkoriščajo ranljivosti pri upravljanju pomnilnika, in spodbuditi razvoj robustnejših pristopov k programiranju in varnostnim praksam. S tem želimo prispevati k razvoju varnejših in zanesljivejših računalniških sistemov, ki bodo kos varnostnim izzivom v vedno bolj digitaliziranem svetu.

## 2 SORODNA DELA

V zadnjih letih je področje varnosti računalniških sistemov doživelo porast raziskav, osredotočenih na ranljivosti pri upravljanju pomnilnika. Te študije se v glavnem ukvarjajo z vzroki, klasifikacijo in zaznavanjem ranljivosti, pri čemer posebno pozornost namenjajo trem ključnim problemom: uhajanju pomnilnika, dvojnemu sproščanju in uporabi pomnilniških blokov po njihovi sprostitvi. V nadaljevanju predstavljamo pregled ključnih prispevkov na tem področju, ki skupaj tvorijo celovito sliko trenutnega stanja raziskav.

Temeljna študija Caballera in sodelavcev je poudarila povezavo med ustvarjanjem in uporabo visečih kazalcev ter ranljivostmi zaradi uhajanja in dvojnega sproščanja pomnilnika. Njihovo orodje "Undangle" omogoča zgodnje odkrivanje teh ranljivosti in je bilo uspešno ovrednoteno na osmih praktičnih primerih, vključno z dvema novima ranljivostma v spletnem brskalniku Firefox [1]. Ta raziskava je spodbudila razvoj sorodnih orodij, kot so FreeSentry [2], DangSan [3], HeapExpo [4] in pSweeper [5], ki so dodatno izboljšala analizo visečih kazalcev.

Na področju odkrivanja varnostnih ranljivosti pri upravljanju pomnilnika so raziskovalci razvili različne pristope, ki združujejo statično analizo in dinamično testiranje. Hu in sodelavci so v tem kontekstu formalno opredelili tovrstne ranljivosti in razvili ogrodje "MRVDAVF" za analizo izvorne kode. Njihov pristop se je izkazal za učinkovitejšega v primerjavi z obstoječimi rešitvami [6]. Komplementarno temu so Rebel in sodelavci predlagali modularni pristop za samodejno izkoriščanje in iskanje ranljivosti pri upravljanju kopice [7], s čimer so razširili nabor orodij za celovito analizo varnosti pomnilnika.

Pomemben napredek na področju dinamičnega testiranja predstavlja orodje "UAFL", ki so ga razvili Wang in sodelavci. To orodje za *fuzz* testiranje temelji na analizi stanja tipov (angl. *typestate analysis*) in se je izkazalo za posebej učinkovito pri odkrivanju ranljivosti zaradi uporabe pomnilniških blokov po sprostitvi [8]. Na tem področju so Yan in sodelavci naredili dodaten korak z integracijo strojnega učenja v statično analizo, kar je privedlo do izboljšane natančnosti pri odkrivanju ranljivosti [9].

V sklopu dinamične analize so Gui in sodelavci razvili odprtokodno orodje "UAFSan", ki vsakemu pomnilniškemu bloku dodeli edinstveno oznako. S preverjanjem konsistence teh oznak so avtorji uspešno identificirali napake zaradi uporabe blokov po sprostitvi [10]. Sorodno temu pristopu so Erdő s in sodelavci predstavili orodje "MineSweeper", ki deluje na principu karantene sproščenih pomnilniških blokov. To orodje preprečuje ponovno alokacijo bloka, dokler se ne potrdi odsotnost kazalcev nanj, s čimer učinkovito preprečuje ranljivosti pri uporabi pomnilniških blokov po sprostitvi [11].

Raziskave so se usmerile tudi v preučevanje tehnik enkratnega dodeljevanja (OTA) istega bloka po-

mnilnika kot preventivnega ukrepa proti omenjenim ranljivostim. V tem kontekstu so Wickman in sodelavci razvili pomnilniški dodeljevalec "FFmalloc", ki optimizira določene pomanjkljivosti enkratnih dodeljevalcev pomnilnika in s tem poveča njihovo praktično uporabnost [12].

Za celovit pregled področja so Gui in sodelavci opravili sistematično analizo, primerjavo in ovrednotenje trenutnih tehnik za odkrivanje in preprečevanje ranljivosti zaradi uporabe pomnilniških blokov po sprostitvi. Njihova študija vključuje primerjavo učinkovitosti izvajanja in porabe pomnilnika različnih tehnik [13], s čimer ponuja dragocen vpogled v kompromise med varnostjo in zmogljivostjo. Nadaljnje raziskave so se usmerile tudi v specifične domene, kot so industrijski nadzorni sistemi (ICS), kjer so Liu in sodelavci opravili pregled ranljivosti v izvršljivih datotekah [14], s čimer so poudarili pomen varnosti pomnilnika v kritičnih infrastrukturnih sistemih.

Ta pregled sorodnih del kaže na kompleksnost in večplastnost problematike varnosti pomnilnika ter poudarja potrebo po celovitem pristopu k odkrivanju in preprečevanju ranljivosti. Raziskave kažejo trend k razvoju vse bolj sofisticiranih orodij in tehnik, ki združujejo statično in dinamično analizo, strojno učenje ter inovativne pristope k upravljanju pomnilnika. Kljub pomembnemu napredku pa ostaja področje varnosti pomnilnika aktualen izziv, ki zahteva nadaljnje raziskave in razvoj.

# 3 METODOLOGIJA

## 3.1 Raziskovalni pristop

V tej raziskavi smo uporabili kvalitativni raziskovalni pristop, ki temelji na kombinaciji študije primera in analize tveganja. Naš metodološki okvir je zasnovan tako, da omogoča celovito razumevanje problematike dvojnega sproščanja pomnilnika, od teoretičnih osnov do praktičnih implikacij in možnih rešitev. Raziskava je potekala v štirih ključnih fazah: priprava, analiza, validacija in sinteza.

V fazi priprave smo razvili preprost program, ki simulira upravitelja podatkovne baze, z namerno vgrajeno ranljivostjo dvojnega sproščanja pomnilnika. Ta namenski primer nam je služil kot osnova za podrobno preučevanje mehanizma ranljivosti v kontroliranem okolju.

Sledila je faza analize, v kateri smo korak za korakom preučili, kako lahko napadalec izkoristi ranljivost za pridobitev nepooblaščenega dostopa. Ta del je vključeval podroben pregled dogajanja v pomnilniku med izvajanjem programa, kar nam je omogočilo globlje razumevanje tehničnih vidikov ranljivosti. Analitična stopnja raziskave je bila ključna za razkritje subtilnih mehanizmov, ki omogočajo izkoriščanje te vrste ranljivosti.

V fazi validacije smo našo teoretično analizo nadgradili s študijo primera iz prakse. Preučili smo konkreten primer ranljivosti v široko uporabljeni mobilni aplikaciji, kar je služilo kot most med našo teoretično analizo in praktičnimi implikacijami. Ta korak je bil ključen za potrditev relevantnosti naše raziskave v kontekstu kompleksnih produkcijskih sistemov.

Zaključna faza sinteze je bila namenjena oblikovanju nabora strokovnih priporočil za preprečevanje in ublažitev tovrstnih ranljivosti. Na podlagi ugotovitev iz predhodnih delov raziskave smo razvili praktične smernice, ki razvijalcem in varnostnim strokovnjakom ponujajo konkretne napotke za izboljšanje varnosti programske opreme.

Naš večstopenjski pristop, ki združuje nadzorovan eksperiment in analizo realnega primera, zagotavlja ravnovesje med natančnostjo laboratorijske analize in relevantnostjo za resnične scenarije v razvoju programske opreme. S tem smo dosegli celovito obravnavo problematike, ki presega zgolj teoretično razumevanje in ponuja praktične vpoglede v varnostne izzive sodobnega razvoja programske opreme.

## 3.2 Ranljivosti v izvršljivih datotekah

Izvršljive datoteke so datoteke, ki jih lahko računalnik izvede (izvrši). V operacijskem sistemu Linux so običajno zapisane v formatu ELF (*Executable and Linkable Format*). Sestavljene so iz več različnih sekcij, ki vsebujejo strojno kodo, podatke, simbole in druge metapodatke [15]. Ko uporabnik zažene izvršljivo datoteko, operacijski sistem prebere metapodatke in naloži kodo v pomnilnik. Koda se po tem začne izvajati (proces).

Proces je osnovna enota izvajanja v operacijskem sistemu. Vsak izmed njih ima svoj ločen prostor v pomnilniku, kjer shranjuje podatke, potrebne za izvajanje. Podatki, ustvarjeni v času izvajanja, so običajno ločeni na sklad (angl. *Stack*), kjer se nahajajo lokalne spremenljivke in naslovi za vrnitve iz funkcij, in kopico (angl. *Heap*), ki se uporablja za dinamično dodeljevanje pomnilnika.

Procesi, ki so ranljivi zaradi napak v programski kodi, so pogosto tarča napadov. Med najpogostejši-

mi ranljivostmi v izvršljivih datotekah so prelivanje medpomnilnika (angl. *Buffer overflow*), ranljivosti v knjižnicah (angl. *Library vulnerabilities*) in ranljivosti pri upravljanju pomnilnika (angl. *Memory management vulnerabilities*).

Prav tako lahko v grobem ločimo napade na sklad in kopico. V nadaljevanju bomo podrobneje obravnavali napade na kopico, ki izkoriščajo ranljivosti pri upravljanju pomnilnika.

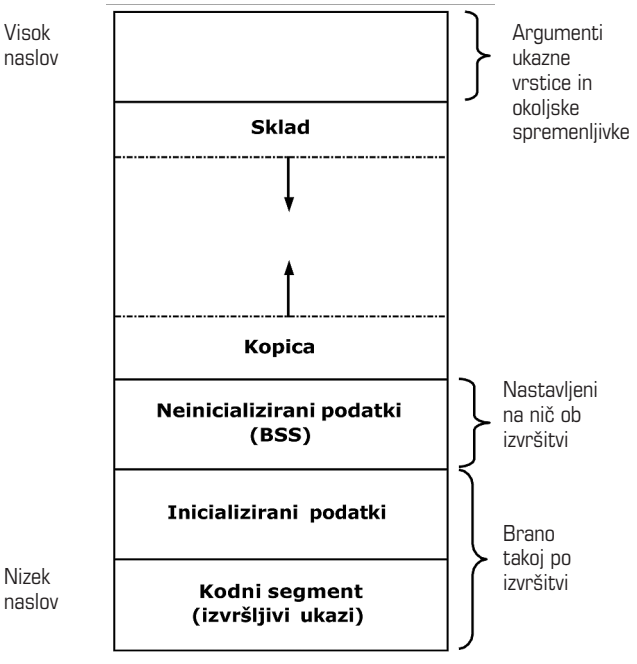
## 3.3 Osnovni opis kopice

Kopica (angl. *Heap*) je prilagodljivo območje pomnilnika za hranjenje večjih podatkovnih struktur in podatkov z dinamično življenjsko dobo. Za razliko od lokalnega pomnilnika, ki se samodejno dodeli in sprosti, je treba s kopico upravljati eksplicitno. V jezikih, kot sta Java ali C++, se pri ustvarjanju struktur ali objektov to običajno izvede z uporabo operatorja new. V programskem jeziku C pa se za dinamično dodelitev pomnilnika uporablja funkcija malloc().

Dodeljen blok pomnilnika (ali objekt) ostane v uporabi, dokler ni eksplicitno sproščen. V nižjenivojskih programskih jezikih to nalogo prevzema programer. Takšen pristop programerju omogoča večji nadzor nad upravljanjem pomnilnika, a hkrati nalaga večjo odgovornost za aktivno skrb zanj. Ena pogostejših napak pri tem je ohranitev reference na pomnilniško lokacijo brez ustreznega sproščanja. Temu rečemo puščanje pomnilnika (angl. *memory leak*).

V mnogih komercialnih programih, napisanih v C ali C++, se pojavlja puščanje pomnilnika, ki povzroči pomanjkanje prostega pomnilnika in sesutje programa. Java in drugi nekoliko višjenivojski jeziki to napako odpravljajo s pomočjo avtomatskega upravljanja oz. čiščenja pomnilnika (angl. *garbage collection*). Slabost tega pristopa pa je, da čiščenje pomnilnika nekoliko upočasni delovanje programa ter se zgodi v nepredvidljivih časih [17].

V operacijskem sistemu Linux je za upravljanje kopice v programskem jeziku C zadolžena knjižnica GNU libc.




Slika 1: **Shematičen prikaz pomnilnika procesa [16].**

## 3.4 Delovanje funkcij malloc() in free()

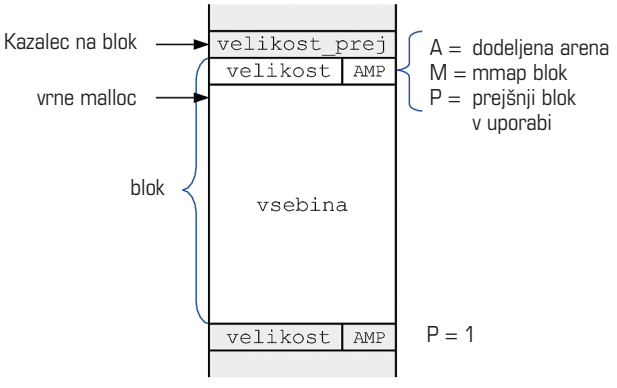
### 3.4.1 Bloki

Knjižnica GNU libc deli kopico na bloke različnih velikosti. Vsak blok vsebuje metapodatke o velikosti in o sosednjih blokih. Ko je blok v uporabi, se v pomnilniku hranita le njegova velikost in zastavice, ko pa je sproščen, pa se poleg tega v pomnilnik zapišeta še kazalca na sosednja bloka [18]. Strukturo dodeljenega in sproščenega bloka prikazujeta sliki 2 in 3.

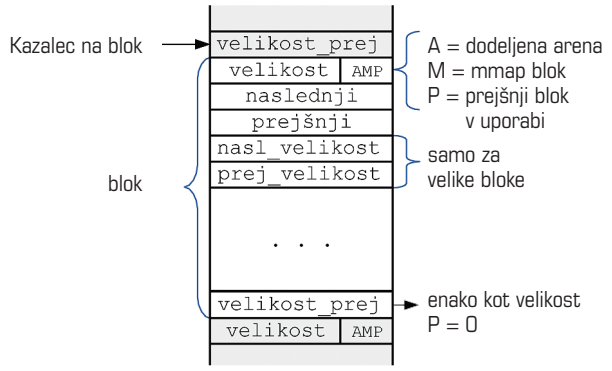
### 3.4.2 Koši

Sproščeni bloki so shranjeni v različnih seznamih – koših (angl. *bins*) glede na velikost in zgodovino, da jih lahko knjižnica učinkovito ponovno dodeli ob novi zahtevi. Koši so štirih vrst: hitri, nerazvrščeni, majhni in veliki.

Pri večnitnih procesih ima vsaka nit svoj lokalni predpomnilnik blokov, dostopnih brez zaklepanja (angl. *tcache*) [18].




Slika 2: **Blok v uporabi [18].**




Slika 3: **Sproščen blok [18].**

### 3.4.3 Zaznavanje okvar kopice

Sistem za alokacijo in dealokacijo pomnilnika sproti preverja za morebitne okvare, vendar je večina pregledov hevrističnih (npr. preverjanje kazalcev) in se jih da pretentati z lažnimi bloki, ki izgledajo resnično. V tem primeru lahko okvara preživi kar nekaj časa, ne da bi bila zaznana [18].

## 3.5 Dvojno sproščanje pomnilnika

Ena izmed pogostih napak, ki se pojavijo pri programiranju v jezikih z ročnim upravljanjem pomnilnika, je dvojno sproščanje pomnilnika (angl. *double free*). Pojavi se, ko program (po pomoti) dvakrat kliče funkcijo free() z istim argumentom (kazalcem na dodeljeni blok pomnilnika). To privede do korupcije podatkovnih struktur za upravljanje pomnilnika, kar lahko povzroči, da program preneha delovati ali pa v nekaterih okoliščinah spremeni tok izvajanja [19]. Napadalec lahko s prepisovanjem pomnilniških prostorov pripravi program, da izvede skorajda poljuben kos kode, kar mu omogoča pridobitev dostopa do lupine.

Poglejmo enostaven primer ranljivosti, ki se pojavi zaradi dvojnega sproščanja pomnilnika:

```
void *ptr = malloc(SIZE);
...
if (some_error) {
    free(ptr);
}
...
free(ptr);
```

Pogosta vzroka ranljivosti sta obravnava napak in drugih izjemnih okoliščin ter nejasnost, kateri del programa je odgovoren za sproščanje pomnilnika. Čeprav nekatere ranljivosti niso veliko bolj zapletene od zgornjega primera, so večinoma razpršene po stotinah vrstic kode ali celo po različnih datotekah, kar močno zmanjšuje preglednost in otežuje sledenje toku izvajanja programa. Ta problem je še posebej izrazit pri uporabi globalnih spremenljivk [19].

# 4 PRIMER IN RAZLAGA NAPADA

V tem delu bomo podrobneje opisali potek in specifike napada na primeru enostavnega programa, napisanega v programskem jeziku C [20].

Program deluje interaktivno v ukazni vrstici po principu REPL (*Read–Eval–Print Loop*) in simulira preprostega upravitelja podatkovne baze. Razlikuje med dvema uporabniškima vlogama: uporabnikom in administratorjem. Uporabnik lahko izvaja poizvedbe na bazi, medtem ko ima administrator poleg te možnosti tudi pravico vnašati nove vnose in brisati vsebino baze.

Za dostop do administratorskih funkcij je potrebna prijava z geslom, ki ga pozna le administrator, zato navadni uporabnik kot tudi napadalec brez gesla ne moreta dostopati do teh funkcij.

Poglejmo si poziv, ki ga vidi uporabnik:

```
Logged in as: user
1) Quit
2) Change <user>
3) Query <something|*>
Enter your choice:
```

Če uporabnik želi z vnosom 2 admin spremeniti uporabniško vlogo in ob tem vpiše napačno geslo, se mu kot pričakovano izpiše sporočilo:

```
Incorrect password!
```

Rezultat veljavne poizvedbe npr. 3 Jack je sledeč:

```
 id | name
----+----------------
  9 | Jack
Found 1 entry.
```

Če pa je poizvedba neveljavna, npr. za ukazom 3 ne podamo argumenta, se izpiše sporočilo:

```
Invalid query.
```

Poglejmo si poenostavljen del kode, ki je odgovoren za obravnavanje poizvedb:

```
void select_from_db(char *line) {
  // line <- "3 Jack"
  DBEntry *e = malloc(sizeof(DBEntry));

  char query[SIZE];
  int args;
  args = sscanf(line, "%*s %s", query);
  if (args != 1) {
    puts("Invalid query.");
    free(line);
    free(e);
    return;
  }
  // query <- "Jack"

  int count = 0;

  ... // print matching entries

  printf("Found %d entries.\n", count);

  free(e);
}
```

Funkcija select_from_db() kot parameter prejme dinamično dodeljen niz znakov line, ki predstavlja vrstico z ukazom. Za dodelitev in sprostitev tega niza sicer skrbi klicatelj funkcije, vendar pa se v primeru neveljavne poizvedbe niz sprosti znotraj funkcije. Zato se ob neveljavni poizvedbi ta blok pomnilnika sprosti dvakrat, kar predstavlja varnostno ranljivost.

V našem primeru so alokacije prikladno enakih velikosti. Poglejmo dogajanje v hitrem košu (*fastbin*) pri sproščanju pomnilnika ob neveljavni poizvedbi:

1. Sprosti se line.

   GLAVA -> line -> REP
2. Sprosti se e.

   GLAVA -> e -> line -> REP
3. Izven funkcije se znova sprosti line.

   GLAVA -> line -> e -> line -> REP

Če bi dvakrat zapored sprostili isti blok, bi tudi starejše verzije knjižnice GNU libc zaradi omenjenih hevristik zaznale napako in prekinile izvajanje programa. V tem primeru pa napaka zaenkrat ostaja nezaznavna.

Analizirajmo še funkcijo za prijavo:

```
void change_user(char *line) {
  // line <- "2 admin"
  char *username = malloc(SIZE);
  ... // set: username <- "admin"
  ... // match username

  char *password = malloc(SIZE);
  int fd = open("passwd.txt", O_RDONLY);
  read(fd, password, SIZE);
  close(fd);

  printf("Enter password: ");
  fgets(line, SIZE, stdin);

  if (!strncmp(line, password, SIZE)) {
    uid = ADMIN_UID;
    puts("Switched to admin.");
  } else
    puts("Incorrect password!");

  free(password);
  free(username);
}
```

Če takoj za neveljavno poizvedbo uporabnik poskusi zamenjati uporabniško vlogo, se pri dodeljevanju pomnilnika zgodi sledeče:

1. Še izven funkcije se dodeli line.

`GLAVA -> e -> line -> REP`

Blok line je zdaj obenem dodeljen kot tudi sproščen.

2. Dodeli se username.

`GLAVA -> line -> REP`

V prejšnji funkciji sproščen blok e se zdaj ponovno dodeli kot username.

3. Dodeli se password.

`GLAVA -> REP`

Hitri koš je zdaj prazen. Blok line je zdaj dodeljen kot password. Vendar je to isti blok, ki je že v uporabi. Trenutno oba kazalca, line in password, kažeta na isti kos pomnilnika.

V funkciji change_user() najprej preberemo geslo iz datoteke passwd.txt in ga shranimo v blok password. Za primere demonstracije se geslo hrani kot čistopis v datoteki, v pravih sistemih pa naj bi bilo geslo seveda ustrezno shranjeno in šifrirano.

Nato uporabnika prosimo za vnos gesla. Če se vnos ujema s prebranim geslom, uporabniku dodelimo administratorske pravice. Prebrano geslo se

shrani v blok line. Ker pa gre za isti blok kot pri kazalcu username, se vsebina bloka enostavno prepiše z uporabniškim vnosom. Posledično se primerja isti niz znakov s samim seboj, kar seveda vedno vrne true, zato se uporabniška vloga uspešno zamenja na administratorsko. Pri tem se izpiše sporočilo:

```
Switched to admin.

Logged in as: admin
1) Quit
2) Change <user>
3) Query <something|*>
4) Insert <entry> into database
5) Wipe database
Enter your choice:
```

## 5 REZULTATI

Analiza primera ranljivosti dvojnega sproščanja pomnilnika v preprostem programu za upravljanje podatkovne baze je razkrila več ključnih ugotovitev:

1. **Uspešno izkoriščanje ranljivosti:**
   Napadalcu je uspelo pridobiti administratorske pravice brez poznavanja gesla. To je bilo doseženo z zaporedjem specifičnih korakov: izvedba neveljavne poizvedbe, ki sproži dvojno sproščanje, in nato poskus zamenjave uporabniške vloge.
2. **Mehanizem napada:**
   Dvojno sproščanje je povzročilo, da sta dva kazalca (line in password) kazala na isti blok pomnilnika. To je omogočilo prepis gesla z uporabniškim vnosom, kar je privedlo do uspešne avtentikacije.
3. **Pogoji za uspešen napad in omejitve zaznavanja:**

- Hevristični varnostni pregledi v knjižnici GNU libc niso zaznali te specifične oblike dvojnega sproščanja, kar kaže na pomanjkljivosti varnostnih mehanizmov pri odkrivanju kompleksnejših vzorcev napačnega upravljanja s pomnilnikom.
- Demonstriran napad je učinkovit na sistemih z GNU libc pred različico 2.26. To vključuje Ubuntu 17.10 in 16.04 LTS, pri čemer je slednja še vedno podprta in prejema razširjene varnostne posodobitve do aprila 2026 [21]. Posledično je prikazan primer napada še vedno aktualen za določene sisteme v uporabi.
- Izdaja GNU libc 2.26 je uvedla podporo za lokalni predpomnilnik blokov za niti (*tcache*), ki onemogoča to specifično obliko napada [22].
- Nadaljnje iteracije knjižnice so izboljšale učinkovitost upravljanja kopice, vendar so hkrati opustile nekatere varnostne mehanizme, kar je odprlo vrata za novejše načine napadov [23].

4. **Širše posledice:**
   Čeprav je bil primer demonstriran na preprostem programu, rezultati kažejo, da lahko podobne ranljivosti v kompleksnejših sistemih vodijo do resnih varnostnih tveganj. Analiza razkriva potrebo po bolj robustnih metodah za preverjanje pravilnosti upravljanja s pomnilnikom v programih, napisanih v jezikih z ročnim upravljanjem pomnilnika.

Za namen demonstracije je bil pripravljen tudi vsebnik Docker, ki simulira okolje z ranljivostjo [20]. Bralec je vabljen, da ga prenese in preizkusi sam.

Ti rezultati poudarjajo pomen natančnega upravljanja s pomnilnikom in potrebo po večslojnih varnostnih pristopih pri razvoju programske opreme. Obenem kažejo na stalno evolucijo varnostnih izzivov in potrebo po nenehnem prilagajanju varnostnih strategij.

## 6 DISKUSIJA

### 6.1 Izraba ranljivosti v praksi

V grobem so napadi na kopico zahtevnejši od napadov na sklad in zahtevajo natančno poznavanje notranje strukture kopice. Dodelitve in sprostitve pomnilnika so v sistemih v praksi običajno manj predvidljive, kot je bilo prikazano v našem primeru, princip napada pa ostaja enak. Običajno je cilj napadalca izvesti poljubno kodo ali pridobiti dostop do lupine ali občutljivih podatkov.

Ker do delitve pravic med izvajanjem programa po navadi ne pride slučajno, kot v našem primeru, se napadalci obenem poslužujejo tudi drugih tehnik, kot so prelivanje medpomnilnika, ranljivosti v knjižnicah in napadi na sklad. Sproščeni in hkrati alocirani blok pa se lahko uporabi za prepisovanje kazalca na naslednji sproščeni blok v košu. S tem se lahko doseže, da se programu in s tem napadalcu ob enem izmed prihodnjih klicev funkcije malloc() (v kolikor uspe blok pretentati hevristične varnostne preglede knjižnice) dodeli dostop do skorajda poljubnega segmenta pomnilnika, tudi do kode, ki se izvaja, ali pa do občutljivih podatkov.

Ko napadalec ugotovi naslov standardne knjižnice v pomnilniku, lahko vrnitveni naslov prepiše z naslovom funkcije, ki jo želi izvesti, npr. system("/bin/sh") in s tem pridobi dostop do lupine. Takemu napadu pravimo "ret2libc".

### 6.2 Primeri ranljivosti v aplikaciji WhatsApp za Android

Leta 2019 so raziskovalci odkrili ranljivost v knjižnici android-gif-drawable [24], ki jo uporablja tudi priljubljena aplikacija WhatsApp za Android. Ranljivost je omogočala izvajanje poljubne kode na daljavo in dostop do lupine na napravi uporabnika, če je ta odprl posebej oblikovan GIF. Napadalec bi lahko izkoristil to ranljivost za krajo občutljivih podatkov, kot so fotografije in sporočila [25].

Ranljivost poteka sledeče:

1. Napadalec pošlje GIF datoteko uporabniku preko kateregakoli kanala. Ena od možnosti je, da pošlje datoteko kot dokument preko WhatsApp-a.
2. Če je napadalec v stiku z uporabnikom (npr. prijatelj), se okvarjena GIF datoteka glede na privzete nastavitve samodejno prenese brez uporabnikovega posredovanja.
3. Uporabnik želi poslati medijsko datoteko kateremu od svojih prijateljev preko WhatsApp-a. Zato pritisne na gumb za pripenjanje datotek in odpre galerijo WhatsApp-a, da izbere medijsko datoteko, ki jo želi poslati prijatelju. Pri tem ni potrebno, da uporabnik dejansko pošlje datoteko, saj že samo odpiranje galerije sproži napako.
4. Ker WhatsApp prikaže predogled vsake medijske datoteke (vključno z GIF datoteko, ki jo je prejel), sproži napako dvojnega sproščanja pomnilnika in omogoči napad.

Ranljivost je znana pod oznako CVE-2019-11932 in je bila odpravljena z izdajo posodobitve aplikacije 2.19.244 [25]. Ta primer iz prakse dodatno poudarja resnost ranljivosti dvojnega sproščanja pomnilnika in potrebo po stalnem posodabljanju programske opreme ter implementaciji robustnih varnostnih mehanizmov. Obenem kaže, da so tovrstne ranljivosti lahko prisotne tudi v zelo razširjenih aplikacijah, kar še povečuje njihov potencialni vpliv.

## 7 PREDLAGANE REŠITVE

Demonstracija je pokazala, da lahko navidezno nedolžna ranljivost dvojnega sproščanja pomnilnika privede do resnih posledic. Za zmanjšanje verjetnosti pojava takšnih napak predlagamo več pristopov:

- **Implementacija načela enkratnega lastništva** pri upravljanju pomnilnika. To načelo določa, da je za vsak blok pomnilnika odgovoren le en del kode. Če je blok sproščen, ga ni več dovoljeno uporabljati. To načelo je še posebej pomembno pri delu z globalnimi spremenljivkami. V primerih, ko striktno upoštevanje tega načela ni mogoče, pa je ključnega pomena, da so posamezne funkcije in deli kode jasni in pregledni ter da ima vsaka funkcija konceptualno en sam namen.
- **Uporaba pomnilniško varnih** (angl. *memory-safe*) **programskih jezikov**, ki vključujejo vgrajene zaščite pred omenjenimi napadi. Med te spadajo jeziki z avtomatskim upravljanjem pomnilnika kot so Python, Swift, C#, Java in Go. Za aplikacije, kjer je kritična učinkovitost izvajanja, pa sta primerni alternativi Rust ali uporaba pametnih kazalcev v C++.
- **Vključitev orodij za statično analizo kode** in odkrivanje napak pri upravljanju pomnilnika v razvojni proces. Primeri takih orodij so Valgrind in AddressSanitizer. Dodatno je koristno izvajanje *fuzz* testiranja, tj. avtomatiziranega testiranja z obsežnim naborom naključnih, nepredvidenih ali neveljavnih vhodov.
- **Implementacija najnovejših varnostnih smernic** in pravil za programiranje. Ključno je tudi redno posodabljanje uporabljenih knjižnic za odpravo znanih ranljivosti. V operacijskih sistemih je priporočljiva aktivacija varnostnih mehanizmov, kot sta ASLR in DEP, ki otežujeta napadalcem napovedovanje naslovov pomnilnika in izvajanje kode v podatkovnih segmentih.
- **Omejitev privilegijev programa** in izvajanje v peskovniku (angl. *sandboxing*) za zmanjšanje potencialnih posledic napadov. Tehnike kot so Seccomp, Landlock, AppArmor in SELinux omogočajo omejevanje dostopa programa do sistemskih virov (npr. branja in pisanja datotek izven predvidenih direktorijev ali dostopa do lupine). Virtualizacija in uporaba vsebnikov dodatno prispevata k izolaciji programa in preprečevanju dostopa do občutljivih podatkov drugih aplikacij.
- **Uveljavitev načel defenzivnega programiranja.** Ta pristop zahteva sistematično predvidevanje potencialnih napak in napadov ter implementacijo ustreznih zaščitnih mehanizmov. To vključuje temeljito preverjanje vhodnih podatkov, validacijo rezultatov funkcijskih klicev in verifikacijo veljavnosti kazalcev pred njihovo uporabo.

## 8 ZAKLJUČEK

V tem članku smo podrobno analizirali ranljivost zaradi dvojnega sproščanja pomnilnika, ki se skupaj s sorodnimi ranljivostmi, kot sta uporaba pomnilniških blokov po sprostitvi in puščanje pomnilnika, uvršča med najpogostejša varnostna tveganja pri upravljanju s kopico. Te ranljivosti se pojavljajo predvsem zaradi napak v programih, napisanih v programskih jezikih z ročnim upravljanjem pomnilnika.

Na praktičnem primeru smo prikazali, kako lahko napadalci izkoristijo omenjeno ranljivost za pridobitev administratorskih pravic v preprostem programu, ki simulira upravitelja podatkovne baze. Prav tako smo opisali, kako se lahko ta ranljivost v praksi izkoristi za izvajanje poljubne kode, kar smo ponazorili s primerom ranljivosti v priljubljeni mobilni aplikaciji.

Kljub naraščajoči priljubljenosti pomnilniško varnih programskih jezikov ostaja uporaba jezikov z ročnim upravljanjem pomnilnika, kot je C, pogosta zaradi njihove hitrosti in praktičnosti v specifičnih razvojnih okoljih. To pomeni, da ostaja nevarnost izkoriščanja opisanih ranljivosti še vedno relevantna.

V priporočilih za zmanjšanje tveganj smo podali več strategij, kako lahko razvijalci in skrbniki sistemov omejijo tovrstne varnostne ranljivosti ter zaščitijo svoje aplikacije pred napadi. To vključuje uporabo sodobnih orodij za statično in dinamično analizo kode, uvedbo strožjih pravil in smernic za upravljanje pomnilnika ter omejitev privilegijev programa. Prav tako smo poudarili pomen defenzivnega programiranja, ki pomaga preprečevati napake že v najzgodnejših fazah razvoja programske opreme.

S tem smo izpostavili kompleksnost varnosti pomnilnika in nujnost celovitega pristopa k varovanju programske opreme pred napadi. Razumevanje teh ranljivosti in uvedba ustreznih zaščitnih ukrepov sta ključna koraka za zagotavljanje varnosti in zanesljivosti aplikacij v vedno bolj povezanem digitalnem okolju.

## LITERATURA


[1] Juan Caballero in sod. “Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities”. V: *Proceedings of the 2012 International Symposium on Software Testing and Analysis.* ISSTA 2012. Minneapolis, MN, USA: Association for Computing Machinery, 2012, str. 133–143. ISBN: 9781450314541. DOI: 10.1145/2338965.2336769.

[2] Yves Younan. “FreeSentry: Protecting Against Use-After--Free Vulnerabilities Due to Dangling Pointers”. V: jan. 2015. DOI: 10.14722/ndss.2015.23190.

[3] Erik van der Kouwe, Vinod Nigade in Cristiano Giuffrida. “DangSan: Scalable Use-after-free Detection”. V: *Proceedings of the Twelfth European Conference on Computer Systems.* EuroSys ’17. Belgrade, Serbia: Association for Computing Machinery, 2017, str. 405–419. ISBN: 9781450349383. DOI: 10.1145/3064176.3064211.

[4] Zekun Shen in Brendan Dolan-Gavitt. “HeapExpo: Pinpointing Promoted Pointers to Prevent Use-After-Free Vulnerabilities”. V: *Proceedings of the 36th Annual Computer Security Applications Conference.* ACSAC ’20. Austin, USA: Association for Computing Machinery, 2020, str. 454–465. ISBN: 9781450388580. DOI: 10.1145/3427228.3427645.

[5] Daiping Liu, Mingwei Zhang in Haining Wang. “A Robust and Efficient Defense against Use-after-Free Exploits via Concurrent Pointer Sweeping”. V: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. Toronto, Canada: Association for Computing Machinery, 2018, str. 1635–1648. ISBN: 9781450356930. DOI: 10.1145/3243734.3243826.

[6] Jinchang Hu in sod. “A memory-related vulnerability detection approach based on vulnerability features”. V: *Tsinghua Science and Technology 25.5 (2020)*, str. 604–613. DOI: 10.26599/TST.2019.9010068.

[7] Dusan Repel, Johannes Kinder in Lorenzo Cavallaro. “Modular Synthesis of Heap Exploits”. V: *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security.* PLAS ’17. Dallas, Texas, USA: Association for Computing Machinery, 2017, str. 25–35. ISBN: 9781450350990. DOI: 10.1145/3139337.3139346.

[8] Haijun Wang in sod. “Typestate-guided fuzzer for discovering use-after-free vulnerabilities”. V: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. ICSE ’20. Seoul, South Korea: Association for Computing Machinery, 2020, str. 999–1010. ISBN: 9781450371216. DOI: 10.1145/3377811.3380386.

[9] Hua Yan in sod. “Machine-Learning-Guided Typestate Analysis for Static Use-After-Free Detection”. V: *Proceedings of the 33rd Annual Computer Security Applications Conference.* ACSAC ’17. Orlando, FL, USA: Association for Computing Machinery, 2017, str. 42–54. ISBN: 9781450353458. DOI: 10.1145/ 3134600.3134620.

[10] Binfa Gui, Wei Song in Jeff Huang. “UAFSan: an object-identifier-based dynamic approach for detecting use-after-free vulnerabilities”. V: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis.* ISSTA 2021. Virtual, Denmark: Association for Computing Machinery, 2021, str. 309–321. ISBN: 9781450384599. DOI: 10.1145/3460319.3464835.

[11] Márton Erdő s, Sam Ainsworth in Timothy M. Jones. “MineSweeper: a “clean sweep” for drop-in use-after-free prevention”. V: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS ’22. Lausanne, Switzerland: Association for Computing Machinery, 2022, str. 212–225. ISBN: 9781450392051. DOI: 10.1145/ 3503222.3507712.

[12] Brian Wickman in sod. “Preventing Use-After-Free Attacks with Fast Forward Allocation”. V: *30th USENIX Security Symposium* (USENIX Security 21). USENIX Association, avg. 2021, str. 2453– 2470. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/ presentation/wickman.

[13] Binfa Gui in sod. “Automated Use-After-Free Detection and Exploit Mitigation: How Far Have We Gone?” V: *IEEE Transactions on Software Engineering 48.11* (2022), str. 4569–4589. DOI: 10.1109/TSE.2021.3121994.

[14] Qi Liu, Kaibin Bao in Veit Hagenmeyer. “Binary Exploitation in Industrial Control Systems: Past, Present and Future”. V: *IEEE Access* 10 (2022), str. 48242–48273. DOI: 10.1109/ACCESS.2022.3171922.
[15] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification.* Ver. 1.2. 1995. URL: https://refspecs.linuxfoundation .org /elf/elf.pdf (pridobljeno 12. 5. 2024).
[16] Yanpas – Wikimedia Commons. *C memory layout of program. bss, stack, heap.* 2015. URL: https: //en.wikipedia.org/wiki/File:C-memlayout.svg (pridobljeno 12. 5. 2024).
[17] OpenDSA. *Heap Memory*. URL: https://opendsa-server.cs.vt.edu/ODSA/Books/CS2/html/ HeapMem.html (pridobljeno 12. 5. 2024).
[18] Malloc Internals. URL: https : / / sourceware . org / glibc / wiki / MallocInternals (pridobljeno 12. 5. 2024).
[19] CWE-415: Double Free. URL: https://cwe.mitre.org/data/definitions/415.html (pridobljeno 12. 5. 2024).
[20] GitHub repozitorij. URL: https://github.com/marindereggi/double-free.
[21] Ubuntu 16.04 LTS transitions to Extended Security Maintenance (ESM). URL: https://canonical. com/blog/ubuntu-16--04-lts-transitions-to-extended-security-maintenance-esm (pridobljeno 12. 5. 2024).
[22] The GNU C Library Repository. Ta potrditev uvede podporo za lokalni predpomnilnik blokov za niti v izvorni kodi. URL: https : / / sourceware . org / git / ?p = glibc . git ; a = commitdiff ; h = d5c3fafc4307c9b7a4c7d5cb381fcdbfad340bcc (pridobljeno 12. 5. 2024).
[23] tukan. thread local caching in glibc malloc. URL: http://tukan.farm/2017/07/08/tcache/ (pridobljeno 12. 5. 2024).
[24] Android GIF Drawable Source Code. URL: https://github.com/koral--/android-gif-drawable/ tree/dev/android-gif-drawable/src/main/c (pridobljeno 12. 5. 2024).
[25] Awakened. How a double-free bug in WhatsApp turns to RCE. URL: https://awakened1712.github. io/hacking/hacking-whatsapp-gif-rce/ (pridobljeno 12. 5. 2024).

▪

**Marin Gazvoda de Reggi** je študent na Fakulteti za računalništvo in informatiko Univerze v Ljubljani. Zanimajo ga področja razvoja programske opreme, kibernetske varnosti in umetne inteligence. Njegovi raziskovalni interesi zajemajo teorijo programskih jezikov in njihovo varnost.

▪

**Matevž Pesek** je docent in raziskovalec na Fakulteti za računalništvo in informatiko Univerze v Ljubljani, kjer je diplomiral (2012) in doktoriral (2018). Od leta 2009 je član Laboratorija za računalniško grafiko in multimedije. Od leta 2024 izvaja predmet Varnost programov.