Analiza delovanja kopice in napadov dvojne sprostitve

Domen Breznik, Mark Novak, Matevž Pesek
Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, Večna pot 113, 1000 Ljubljana
db00709@student.uni-lj.si, mn44954@student.uni-lj.si, matevz.pesek@fri.uni-lj.si

Izvleček

Članek obravnava varnostne ranljivosti, ki nastanejo zaradi nepravilnega upravljanja s pomnilnikom v programih, napisanih v jezikih C/C++. Posvečamo se napadu dvojne sprostitve (angl. double free), ki omogoča napadalcu prevzem nadzora nad pomnilniškim prostorom in potencialno krajo administratorskih privilegijev. Predstavimo pregled sorodnih ranljivosti, kot so prekoračitev kopice (angl. heap overflow) in uporaba po sprostitvi (angl. use-after-free) ter obravnavamo obrambo pred takšnimi napadi. Na osnovi prilagojene aplikacije demonstriramo praktičen napad, prikažemo proces izkoriščanja ranljivosti in analiziramo posledice, ki segajo od nestabilnosti sistema do resnih varnostnih groženj. Članek obravnava obstoječe rešitve, kot so uporaba jezikov z samodejnim upravljanjem s pomnilnikom, alternativnih implementacij funkcije malloc, statične analize kode ter pristopov defenzivnega programiranja.

Ključne besede: dvojna sprostitev, napad, kopica, upravljanje pomnilnika, koši, arene

Analysis of Heap Operation and Double Free Attacks

Abstract

The article examines security vulnerabilities arising from improper memory management in programs written in C/C++. We focus on the double free attack, which enables an attacker to take control of memory space and potentially obtain administrative privileges. A review of related vulnerabilities, such as heap overflow and use-after-free, is presented, along with defenses against such attacks. Based on a customized application, we demonstrate a practical attack, show the process of exploiting the vulnerability, and analyze the consequences, which range from system instability to severe security compromises. The article discusses existing solutions, such as the use of memory-safe programming languages, alternative implementations of the malloc function, static code analysis, and defensive programming approaches.

Keywords: double free, attack, heap, memory management, bins, arenas

1 SEZNAM UPORABLJENIH KRATIC

Kratica	Pomen
CVE	Common Vulnerabilities and Exposures
UAF	Use-After-Free
glibc	GNU C library
tcache	Thread cache
ASLR	Address Space Layout Randomization
LIFO	Last In First Out
WebGL	Web Graphics Library

2 UVOD

Nevarnosti, izvirajoče iz napak pri upravljanju s pomnilnikom, ostajajo ključni izziv v programski opremi, razviti v programskih jezikih C in C++. Ta jezika omogočata neposreden dostop do pomnilnika in izjemno učinkovitost, hkrati pa nalagata odgovornost razvijalcem, da ročno skrbijo za dodeljevanje in sproščanje pomnilniških blokov.

Med najresnejšimi so ranljivosti, ki omogočajo okvaro pomnilnika in izvršitev poljubne kode; po-

sebej izstopa *dvojna sprostitev* (*double free*), ko isti pomnilniški blok neustrezno sprostimo več kot enkrat. Empirični podatki iz zbirke MegaVul (2006–2023), ki vsebuje 17,380 ranljivosti, kažejo, da je 59,27% vseh napak povezanih z upravljanjem pomnilnika, kar potrjuje trajno aktualnost problema in potrebo po sistematičnih pristopih k njegovi obravnavi [21].

Napake pri upravljanju s pomnilnikom imajo dolgo zgodovino v praksi in raziskavah. Rezultati preteklih del so kazali, da lahko prepis kazalcev ali napačno rokovanje z dodeljenimi bloki povzroči napake [1]. Kasnejše študije potrjujejo, da tovrstne ranljivosti niso zgolj teoretične, saj so orodja za odkrivanje prekoračitev kopice razkrila več deset doslej neznanih napak [4]. Takšne napake pogosto vodijo do nedefiniranega vedenja programa, ki se lahko izrazi v obliki sesutja, izgube podatkov ali celo vnosu zlonamerne kode v izvajalni tok.

Kljub različnim poskusom preprečevanja tovrstnih ranljivosti, primeri iz javno dostopne zbirke CVE [15] potrjujejo, da gre za še vedno aktualno in pogosto izrabljeno ranljivost. Napadi izkoriščajo tako stare kot tudi sodobne programske sisteme, kar kaže, da so obstoječi mehanizmi zaščite pogosto nezadostni ali neučinkoviti, še posebej ob kompleksnih scenarijih uporabe. Poleg tega so izkoriščanja teh napak zanimiva za napadalce, saj pogosto omogočajo eskalacijo privilegijev in obid varnostnih mehanizmov na ravni operacijskega sistema. V pričujočem članku v testnem okolju simuliramo in prikažemo izkoriščene ranljivosti dvojnega sproščanja in uporabe po sprostitvi pomnilnika ter demonstriramo metode za zaščito pred napadi tega tipa.

3 PREGLED PODROČJA

3.1 SORODNA DELA

Napadi, kot so *preliv kopice* (angl. heap overflow), *uporaba po sprostitvi* (angl. use-after-free) in *dvojna sprostitev* (angl. double free), so bili obsežno preučeni zaradi njihovega potenciala za okvara pomnilnika in izvajanje poljubne kode.

Temeljna raziskava na tem področju je delo Crispina idr., kjer so razvili metodologijo, imenovano *PointGuard*, za detekcijo zlonamernih programov, s fokusom na virusih. Njihova implementacija je bila efektivna za obrambo pred *prelivom kopice* [1]. Pogosto citiran znanstveni članek je tudi članek avtorja Kouwe in drugih. Osredotočajo se na detekcijo dostopov po sprostitvi s svojo rešitvijo DangSan. Rešitev je skalabilna ter podpira več nitne aplikacije [2].

Preliv kopice je ranljivost, kjer program zapiše več podatkov, kot jih je bilo predvidenih v prostoru na kopici. To lahko povzroči okvaro pomnilnika in ostale varnostne ranljivosti. Heelan idr. z implementacijo *Gollum* avtomatizirajo generacijo ranljivosti v tolmaču. V delu predstavijo moč implementacije z napadi v PHP in Python tolmačih [3]. V drugi smeri so raziskovalci iskali rešitve za detekcijo napadov med delovanjem programa. Pokažejo efektivnost delovanja z 17 resničnimi primeri v realnem svetu ter z najdbo 47 prej ne znanih ranljivosti [4]. Tematika je podrobneje obravnava v članku Gopala idr. [5] in delu He idr. [6]. Taki napadi so prisotni še danes, na primer CVE-2025-27091 [15] in CVE-2025-2531 [15].

Dvojne sprostitve se pojavijo, kadar je isti pomnilniški blok sproščen več kot enkrat, pogosto zaradi semantičnih napak programa. Take napake lahko povzročijo zrušitev programa, okvaro pomnilnika ali izvajanje poljubne kode. Maryam idr. predstavijo *fuzzing* metodo za detekcijo ranljivosti kopice v tej smeri, s čimer nadgradijo podobne programe iz prejšnjih raziskav. Rešitev dosežejo z kalkulacijo simboličnih poti in drugimi omejitvami za izvršljivo datoteko [7]. Na detekcijo tovrstnih napadov so se poglobili tudi Baradaran idr. Osredotočajo se na *detekcijo z enotno simbolično metodo* (angl. unit-based symbolic exection method). Program razdelijo na enote, ki lahko vsebujejo ranljivosti in so statično identificirane, glede na njihove specifikacije [8]. Tematika je podrobneje obravnava v članku Cabballera idr. [9] in članku Novarka idr. [10]. Primeri iz resničnega sveta, kot sta CVE-2025-2027 [15] in CVE-2025-32911 [15], kažejo, da so take ranljivosti prisotne tudi danes.

Dostopi po sprostitvi so tesno povezani z dvojnim sproščanjem in se pojavijo, ko program dostopa do pomnilnika po njegovi sprostitvi. Napadalec lahko nato prevzame nadzor nad tem pomnilniškim prostorom. UAF je vztrajno prisoten problem, zlasti v kompleksnih aplikacijah, kot so spletni brskalniki. Kailong idr. so se osredotočili na detekcijo takih napadov in so naredili prototip imenovan UAFDetector. Detekcijo dosežejo s sledenjem kazalcev ter *function summaries* pristopom [11]. Raziskovalci Quiang idr. so za obrambo pred takimi napadi raziskali rešitev Mpchecker, ki dinamično brani sistem z vmesnimi kazalci, ki jih imenujejo *Multi-Level Pointers*. Reši-

tev omogoča dostop do objektov samo z vmesnimi kazalci, ki jih avtomatsko sprosti ob sprostitvi objektov [12]. Tematika je podrobneje obravnava v članku Shena idr. [13] in delu Feista idr. [14]. Kot pri ostalih napadih, so tudi ti prisotni še danes, na primer CVE-2025-27730 [15]. Za širši pregled računalniških napadov bralcu priporočamo tudi širše preglede, npr. [16], [17].

Dirty COW (CVE-2016-5195) [15] je ranljivost, ki poveča privilegije v Linux jedru. Izkorišča *race condition*, ranljivost, ki se zgodi, ko več niti dostopa do istega podatka pri obravnavi sistemskega klica mmap. S tem lahko napadalec piše v *bralne* (angl. read-only) pomnilniške prostore, kar vodi do povišanja privilegijev. Napad deluje v Linux jedrih od verzije 2.x do 4.8.2.

3.2 DEFINICIJE

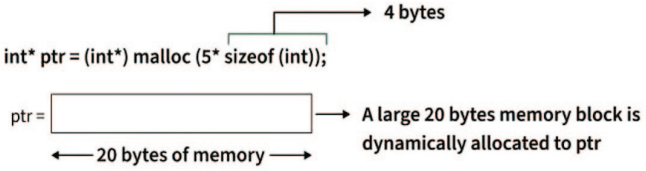
3.2.1 Kopica

Kopica (angl. heap) je prilagodljivo območje pomnilnika za shranjevanje večjih podatkovnih struktur in podatkov z dinamično življenjsko dobo. Od tradicionalnega upravljanja s pomnilnikom se razlikuje po tem, da z njo upravljamo eksplicitno sami, v jezikih kot sta Java in C# ponavadi preko operatorja new, v bolj nizkonivojskih jezikih kot je C, pa preko funkcije malloc(). Pozorni moramo biti na dejstvo da dodeljeni objekt/blok pomnilnika ostane v uporabi, dokler ga eksplicitno ne sprostimo. V nizkonivojskih programskih jezikih kot je C, je za to zadolžen programer z uporabo funkcije free(). Dodatna pozornost mora biti posvečena jezikom, kjer pomnilnik sproščamo sami, saj se hitro zgodi, da pomotoma naredimo preliv (ang. memory leak) ali kakšno drugo napako, ki podvrže naš program ranljivostmi, katero lahko napadalci izkoristijo.

3.2.2 Funkcija malloc()

Funkcija malloc() se uporablja, za dodeljevanje specifičnega števila bajtov pomnilnika. Funkcija pričakuje en argument, ki predstavlja število bajtov. Pozorni moramo biti na to, da malloc() samo rezervira prostor in ga ne inicializira, zato nimamo zagotovila, da bo dodeljeni blok prazen (angl. garbage values).

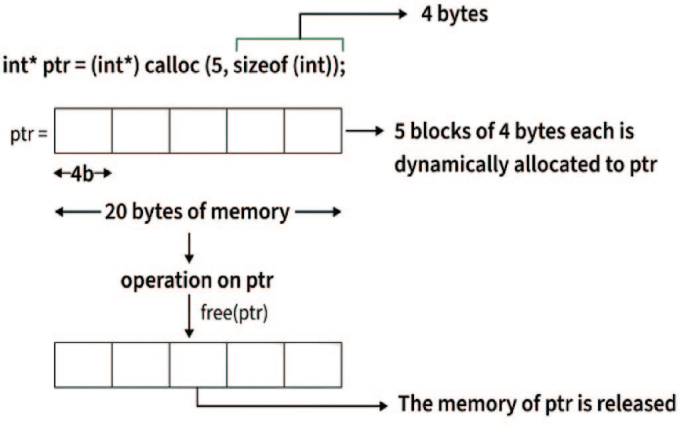
Funkcija malloc() vrne kazalec, ki kaže na dodeljeni blok pomnilnika, sledeči kazalec moramo, če ga želimo uporabiti, pretvoriti v ustrezni tip npr. int, char, itd. V primeru, da nimamo dovolj prostora na pomnilniku, bo funkcija malloc() vrnila prazen kazalec, zato je v praksi dobro preverjati, če je kazalec prazen (angl. NULL).



Slika 1: **Delovanje funkcije malloc [22].**

3.2.3 Funkcija free()

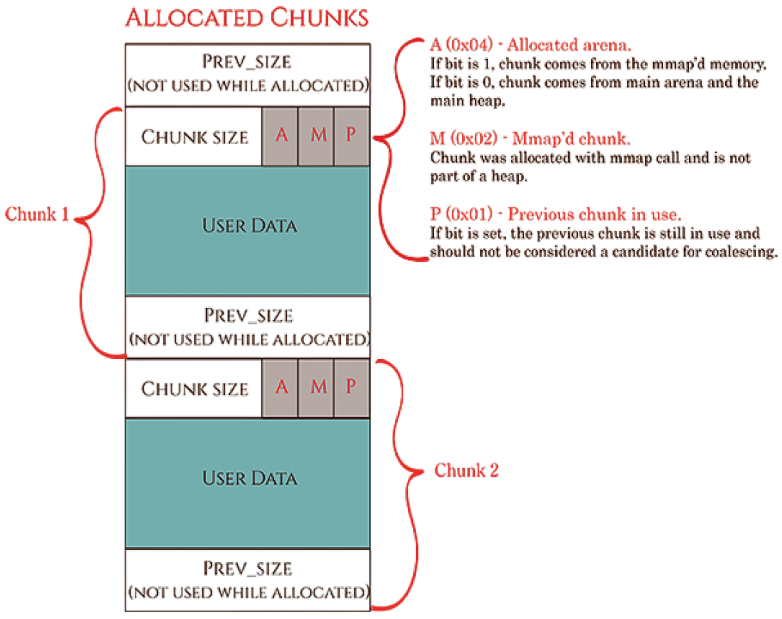
Funkcija free() se uporablja za sprostitev dodeljenih blokov. Njen edini argument je kazalec, ki kaže na lokacijo bloka v pomnilniku, katerega želimo sprostiti. Pri tem obdrži kazalec in samo sprosti blok na katerega je kazalec kazal, tako kazalec zdaj kaže na sproščeni blok. Pozorni moramo biti na dejstvo da funkcija free ne izbriše kazalca niti podatkov, ki so bili shranjeni na blok, ampak samo sporoči sistemu, da je zdaj ta blok prost (angl free) za druge dodelitve.



Slika 2: **Delovanje funkcije free [22].**

3.2.4 Arene

V večnitnih aplikacijah, mora kopični upravljalec (*angl. heap manager*) paziti, da več aplikacij ne dostopa do kopice hkrati. Rešitev so arene, ki delujejo kot različne kopice z svojimi strukturami. Arene so ločeni sklopi pomnilniškega prostora z lastnimi podatkovnimi strukturami za upravljanje dodeljevanje in sproščanje pomnilnika. Pri eno nitnih aplikacijah se uporablja samo glavna arena, ob dodajanju niti pa se ustvarijo sekundarne arene.



Slika 3: **Vizualna predstavitev bloka [23].**

3.2.5 Alokacija blokov

Podatke, ki dodelimo in sprostimo na kopico preko funkcij malloc() in free() se shranijo v enega izmed košev, kjer se rezervira prostor za zaglavje, naš podatek in velikost prejšnjega bloka. Zaglavje sestavljajo:

- trenutna velikost v Bajtih
- 3 zastavice A, M in P.
  - A (*Allocated arena*), je nastavljen na 0, če je blok v glavni areni in nastavljen na 1 sicer
  - M (*Mmap'd chunk*), je nastavljen na 1, če je bil blok alociran z klicem *mmap* in nastavljen na 0 sicer
  - P (*Previous chunk in use*) je nastavljen na 1, če je prejšnji blok še v uporabi, torej ni bil sproščen z funkcijo *free* in nastavljen na 0 sicer
- velikost prejšnjega bloka v Bajtih

Ko dodelimo podatke s funkcijo malloc() dobimo kazalec na prostor za te podatke, torej *user data*. Če hočemo delati z bloki direktno, torej hočemo, da kazalec kaže na zaglavje, lahko uporabimo makro *mem2chunk* ali za obratno makro *chunk2mem*.

3.2.6 Koši

Koši omogočajo učinkovito dodeljevanje in sproščanje pomnilnika. Bloki so vsebovani v koših. Vsaka arena vsebuje 5 tipov košev:

- **64 predpomnilniških košev *(angl. tcache bins)***
  Predpomnilniški koš je en povezan seznam največ 7 majhnih pomnilniških blokov, ki bloke ne združuje po sprostitvi.
- **10 hitrih košev *(angl. fast bins)***
  Hitri koš je namenjen pospeševanju časa alokacije za majhne pomnilniške bloke, tako da hrani predčasno sproščene bloke. Uporablja LIFO pristop, implementiran s povezanimi seznami, kar pomeni, da bo prostor zadnjega sproščenega bloka uporabljen pri novi alokaciji.
- **1 neurejen koš *(angl. unsorted bin)***
  Neurejen koš se uporablja kot medpomnilnik za kopičnega upravljalca, da pohitri dodeljevanje. Ko program sprosti pomnilniški blok, ga kopični upravljalec poskusi združiti s potencialno sproščenimi sosednjimi bloki, da naredi večji sprošče-

```
#define mem2chunk(mem) \\
  ((mchunkptr)tag_at (((char*)(mem) - CHUNK_HDR_SZ)))
#define chunk2mem(p) \\
  ((void*)((char*)(p) + CHUNK_HDR_SZ))
```

ni pomnilniški blok. Ta blok potem vstavi v neurejen koš. Ko program prosi za nov pomnilniški blok, upravljalec najprej pogleda v neurejen koš in kasneje v majhne in velike koše. Če prostora ne najde vse bloke iz neurejenega koša vstavi v primeren majhen ali velik koš.

- **62 majhnih košev** ***(angl. small bins)***
  Majhni koši so hitrejši od velikih, a počasnejši od hitrih. Vsak koš ima bloke enake dolžine: 16 B, 24 B, 32 B, ... Z maksimalno velikostjo 1024 B pri 64 bitnih sistemih, kar pomaga pri iskanju prostega prostora.
- **63 velikih košev** ***(angl. large bins)***
  Veliki koši, za razliko od majhnih, vsebujejo bloke, ki imajo skupno velikost v nekem razponu. Namesto enake velikosti, največji koš vsebuje bloke z velikostjo večjo od 1 MB.

4 METODOLOGIJA

4.1 Raziskovalno okolje

Raziskovalno okolje je sestavljeno iz enostavnega programa napisanega v programskem jeziku C, katerega namen je pisanje zapiskov, ki se shranjujejo na kopico. Program ima namenoma vgrajeno ranljivost kopice, s pomočjo katere lahko napadalec izvede napad dvojne sprostitve. To omogoča analizo in testiranje napada dvojne sprostitve v kontroliranem in znanem okolju.

V programu ločimo med dvema uporabnikoma: navadnim in administratorskim. Razlikujeta se le v tem, da administrator lahko spreminja administratorsko geslo, medtem ko ga navadni uporabnik ne more.

Uporabniki se po programu sprehajajo preko terminala, tako da napišejo pripadajočo številko pred podanimi možnostmi, katere lahko vidimo v sliki spodaj.

```
Welcome to the journal app!
[1] Make entry
[2] Delete entry
[3] Login as admin
[4] Print entries
[5] Exit
```

Slika 4: **Uporabniški vmesnik aplikacije.**

4.2 Struktura programa

Uporabniški vmesnik se vrti v neskončni zanki, katera se konča, ko uporabnik vnese v terminal številko pet, ki prekine delovanje programa. Uporabnik ima na voljo tudi možnost izdelovanja zapiskov, katere lahko tudi briše.

```
typedef struct Entry Entry;
typedef struct Entry {
    char* entry_content;
    Entry* next_entry;
    // ...
} Entry;

Entry* init_entry(char* entry_content,
                  time_t time) {
    // ...
}
void entry_delete(Entry* parent, Entry* entry,
        int entry_number, int current_number) {

    // ...
}
void entry_add(Entry* parent, char* entry_content,
               time_t time) {

    // ...
}
```

Izsek kode 1: **Struktura Entry**

Zapiski se shranjujejo v objekt, ki je poimenovan Entry (Izsek kode 1). Znotraj strukture najdemo 2 kazalca, eden kaže na vsebino zapiska, drugi pa na naslednji zapisek.

Kazalec je spremenljivka, ki kot vrednost shrani pomnilniški naslov dodeljene spremenljivke, v našem primeru bo kazal na uporabniški vnos, ki ga shranimo na kopico oziroma vsebino zapiska.

Funkcija Entry* init_entry(char* entry_content, time_t time) poskrbi za pravilno dodelitev prostora in inicializacijo spremenljivk ustvarjenega zapiska (Izsek kode 1).

Rekurzivni funkciji entry_delete(Entry* entry, int entry_number, int current_number) in entry_add(Entry* parent, char* entry_content, time_t time) se ukvarja z brisanjem zapiskov, oziroma sprostitvijo

```
void handle_action(AppState* app_state) {
    switch (app_state->last_action) {
        // ...
        case 3:
             if(app_state->is_admin == 0){
                printf("Hey admin, \
                    what is your password?\n");
            }
            // logout admin
            else {

                printf(GREEN "successfully \
                    logged out as admin :) \n"
                    COLOR_RESET);
                // ...
            }
            break;
        // ...

    }
}

void admin_login(AppState* app_state) {

  // ...
  if(strcmp(pass, app_state->user_input) == 0){
      printf(GREEN "successfully \
                  logged in as admin :) \n"
                  COLOR_RESET);
      app_state->is_admin = 1;
  }
  else {
      printf("wrong password, \
                  please try again \n");
      app_state->is_admin = 0;
  }
  handle_action(app_state);
}
```

Izsek kode 2: **funkciji za prijavo kot administrator**

dodeljenega prostora, ko uporabnik izbriše specifični zapisek (Izsek kode 1).

Uporabnik ima tudi možnost prijave kot administrator. Analizirajmo še kodo, ki preveri, če je uporabnik vpisal pravilno geslo (Izsek kode 2).

Funkcija handle_action(AppState* app_state) poskrbi za uporabniški vnos, katerega pridobimo preko terminala kot tabelo znakov tipa char (angl. array), na katero kaže kazalec user_input (Izsek kode 2).

Prijavo kot administrator ločimo v dveh primerih. Prva je ko uporabnik izbere tretjo možnost in je že prijavljen kot administrator. V tem primeru ga odjavimo iz sistema nazaj v navadnega uporabnika. Drugi primer je prijava, kot administrator. Takrat se pokliče funkcija admin_login(AppState* app_state), ki od njega zahteva administratorsko geslo (Izsek kode 2).

Najprej se bo uporabniku izpisalo ustrezno besedilo za vnos gesla:

Hey admin, whats your password?

Program potem čaka na vnos uporabnika in v primeru, da je vpisano geslo napačno izpiše sporočilo:

Wrong password, please try again!

Če je geslo pravilno, bo program uporabnika ustrezno prijavil v sistem kot administratorja.

Zaradi lažje predstave gesla namenoma ne šifriramo. Tukaj se opazi prva ranljivost našega programa. V realnem svetu, bi geslo, preden bi ga shranili v datoteko, ustrezno šifrirali.

4.3 Demonstracija napada

Preko programa opisanega v 4.1 bomo prikazali napad dvojne sprostitve. Napadalec lahko program napade z uporabo dvojne sprostitve. Najprej ustvari 2 zapiska, potem prvega sprosti, nato sprosti še drugega in nato ponovno prvega. Medtem mora sprostiti še en zapisek, da se izogne napaki zaporedne dvojne sprostitve, ki bi program izključila.

```
Welcome to the journal app!
[1] Make entry
[2] Delete entry
[3] Login as admin
[4] Print entries
[5] Exit
1
Entry:
aaa
--------------------
ENTRIES:
[1][freed: NO][Fri Apr 25 16:14:28 2025] aaa
--------------------
[1] Make entry
[2] Delete entry
[3] Login as admin
[4] Print entries
[5] Exit
1
Entry:
bbb
--------------------
ENTRIES:
[1][freed: NO][Fri Apr 25 16:14:28 2025] aaa
[2][freed: NO][Fri Apr 25 16:14:29 2025] bbb
--------------------
```

Slika 5: **Začetno ustvarjanje dveh zapiskov.**

```
[1] Make entry
[2] Delete entry
[3] Login as admin
[4] Print entries
[5] Exit
2
--------------------
ENTRIES:
[1][freed: NO][Fri Apr 25 16:14:28 2025] aaa
[2][freed: NO][Fri Apr 25 16:14:29 2025] bbb
--------------------
What entry to delete?
1
--------------------
ENTRIES:
[1][freed: YES]
[2][freed: NO][Fri Apr 25 16:14:29 2025] bbb
--------------------
```

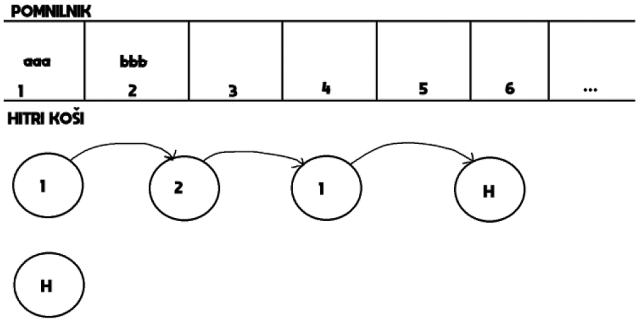
Slika 6: **Sproščanje prvega zapiska.**

```
[1] Make entry
[2] Delete entry
[3] Login as admin
[4] Print entries
[5] Exit
2
--------------------
ENTRIES:
[1][freed: YES]
[2][freed: NO][Fri Apr 25 16:14:29 2025] bbb
--------------------
What entry to delete?
2
--------------------
ENTRIES:
[1][freed: YES]
[2][freed: YES]
--------------------
```

Slika 7: **Sproščanje drugega zapiska.**

```
[1] Make entry
[2] Delete entry
[3] Login as admin
[4] Print entries
[5] Exit
2
--------------------
ENTRIES:
[1][freed: YES]
[2][freed: YES]
--------------------
What entry to delete?
1
--------------------
ENTRIES:
[1][freed: YES]
[2][freed: YES]
--------------------
```

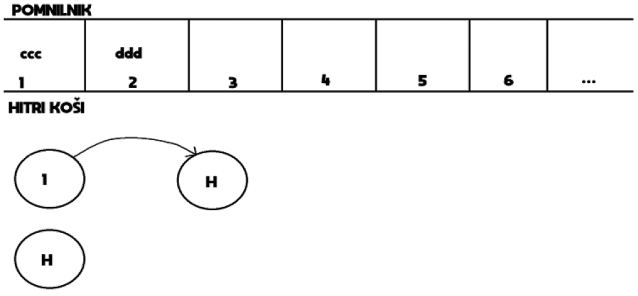
Slika 8: **Ponovno sproščanje prvega zapiska.**



Slika 9: **Stanje pomnilnika in hitrih košev po dvojni sprostitvi.**

```
[1] Make entry
[2] Delete entry
[3] Login as admin
[4] Print entries
[5] Exit
1
Entry:
ccc
--------------------
ENTRIES:
[1][freed: YES]
[2][freed: YES]
[3][freed: NO][Fri Apr 25 16:14:35 2025] ccc
--------------------
[1] Make entry
[2] Delete entry
[3] Login as admin
[4] Print entries
[5] Exit
1
Entry:
ddd
--------------------
ENTRIES:
[1][freed: YES]
[2][freed: YES]
[3][freed: NO][Fri Apr 25 16:14:35 2025] ccc
[4][freed: NO][Fri Apr 25 16:14:36 2025] ddd
--------------------
```

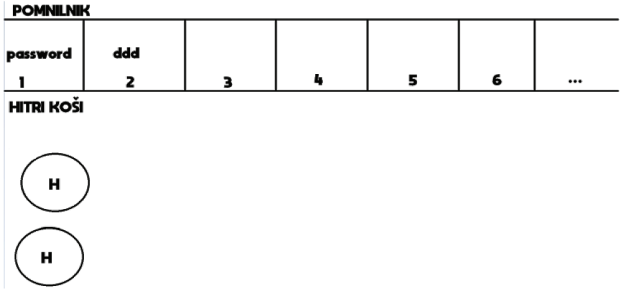
Slika 10: **Ustvarjanje še dveh zapiskov.**



Slika 11: **Stanje pomnilnika in hitrih košev po ustvaritvi še dveh zapiskov.**

```
[1] Make entry
[2] Delete entry
[3] Login as admin
[4] Print entries
[5] Exit
3
Hey adming ;), whats your password?
???
wrong passowrd, please try again
[1] Make entry
[2] Delete entry
[3] Login as admin
[4] Print entries
[5] Exit
4
--------------------
ENTRIES:
[1][freed: YES]
[2][freed: YES]
[3][freed: NO][Fri Apr 25 16:14:35 2025] password
[4][freed: NO][Fri Apr 25 16:14:36 2025] ddd
--------------------
[1] Make entry
[2] Delete entry
[3] Login as admin
[4] Print entries
[5] Exit
5
EXITING with exit status 0
```

Slika 12: **Neuspešna prijava in branje gesla.**



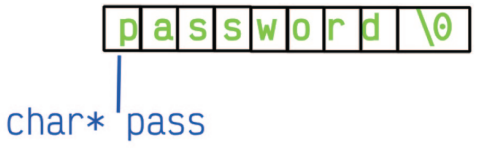
Slika 13: **Stanje pomnilnika in hitrih koševpo napadu.**

5 ANALIZA NAPADA

Po ponovnem sproščanju (Slika 8) vidimo, da sta na mestu 1 in 2 v pomnilniku dodeljeni vrednosti naših nizov “aaa” in “bbb”. Ob sprostitvi pomnilniške lokacije se dodeljeni nizi uvrstijo v isti koš, ker so podobne velikosti. Napadalec mora paziti na velikost njegovih zapiskov, saj se morajo ob sprostitvi zapiski uvrstiti v isti koš iz katerega kasneje dobi prostor geslo (Slika 10).

Ustvarimo še 2 zapiska (Slika 10). Ker ustvarjamo zapiske podobne velikosti, bomo dobili prostor od istega koša v katerega sta se sprostili pomnilniški lokaciji 1 in 2 (Slika 11). V hitrem košu je ostala še ena pomnilniška lokacija 1. Napadalec lahko to izkoristi tako, da se poskusi prijaviti kot administrator. V ozadju se pokliče funkcija admin_login (Izsek kode 3).

Funkcija admin_login() prebere geslo iz datoteke pass.txt in zanj rezervira ustrezno število bajtov. Prebrano geslo se shrani na kopico, tukaj moramo biti pozorni, da dodelimo en znak več kot je dolgo geslo, saj se v programskem jeziku kot je C tabele znakov končajo z “null terminatorjem”. Če za primer vzamemo besedo password, kljub temu da je beseda dolga 8 znakov, bi morali zanjo rezervirati 9 znakov, dodatni znak za “null terminator” (Izsek kode 3).



Slika 14: **Proces shranjevanja tabele znakov v programskem jeziku C**.

Funkcija dodeli prostor za geslo. Če je PASS_SIZE podobne velikosti, bo za dodelitev geslo dobilo prostor iz koša v katerem je pomnilniška lokacija 1. Posledično bo funkcija v ta pomnilniški prostor, nad katerim imamo nadzor, zapisala geslo. Vse kar mora napadalec narediti je, da se prijavi kot administrator in po neuspehu prebrati njegove zapiske. V primeru, da je PASS_SIZE primerne velikosti, bo funkcija

```
void admin_login(AppState* app_state) {
  char* pass = malloc(PASS_SIZE);
FILE* file = fopen("pass.txt", "r");
// ...
int size =
      fread(pass, sizeof(char), PASS_SIZE, file);
pass[size - 1] = '\0';

// ...
}
```

Izsek kode 3: **funkcija za prijavo kot administrator**

```
void entry_delete(Entry* parent, Entry* entry,
                  int entry_number,
                  int current_number) {

    if (entry_number == current_number) {
        // ...
        entry->is_freed = 1;
        free(entry->entry_content);
    }
    else {
    // ...
```

Izsek kode 4: funkcija za izbris zapiska

admin_login v ta prostor zapisala geslo, ki jo bo napadalec lahko prebral in se prijavil v aplikacijo kot administrator.

Napadalec to zlorabi in prebere geslo ter se prijavi kot administrator (Slika 12).

7 DISKUSIJA

Demonstracija napada dvojne sprostitve je razkrila številne posledice, ki jih napad ima kot so:

Kraja administratorskih privilegijev: Napadalec lahko prebere administratorjevo geslo in se prijavi kot administrator. Po prijavi lahko napadalec spremeni administratorjevo geslo in mu s tem prepreči dostop.

Nestabilnost sistema: Zaradi nepravilnega ravnanja s pomnilnikom (dvojna sprostitev) lahko pride do nepredvidenih vedenj aplikacije, vključno z zrušitvami, nedeterminističnim vedenjem in odkritjem dodatnih ranljivosti.

Širše posledice takih napadov: Takšni napadi ne vplivajo samo na delovanje aplikacije, ampak tudi na zaupanje uporabnikov, finančne izgube ter dolgoročni ugled.

Ugotovili smo, da so napadi na kopico zahtevni, saj napadalec potrebuje temeljito znanje in razumevanje njene strukture. V našem primeru je napadalec moral pazljivo dodeljevati prostor na pomnilniku preko ustvarjanja zapiskov, da so se shranili v enak koš, da je potem lahko pravilno prepisal shranjeno geslo.

Ni nujno, da je ranljivost prisotna znotraj našega programa, ampak lahko izvira iz zunanjih knjižnic ali drugih zunanjih virov, ki jih naš program potrebuje za delovanje. V praksi so te ranljivosti bolj pogoste, kot npr. ranljivost knjižnice WebGL za Chromium brskalnike [15].

Ostane nam še vprašanje kako se pred napadi zaščiti in kako jih rešiti? Vrnimo se v funkcijo entry_delete, ki skrbi za brisanje zapiskov (Izsek kode 4).

V funkciji opazimo uporabo spremenljivke *is_freed*, katero naš program uporablja za preverjanje, ali je bila vsebina zapiska sproščena ali ne. Gre za eno-

```
void entry_delete(Entry* parent, Entry* entry,
                  int entry_number,
                  int current_number) {
    //...
        if(entry->is_freed == 0){
            free(entry->entry_content);
            entry->is_freed = 1;
        }
        else {
            printf("DOUBLE FREE DETECTED ABORTING");
        }
    // ...
```

Izsek kode 5: posodobljena funkcija za izbris zapiska

stavno spremenljivko tipa int, katera ima lahko samo vrednosti 0, ki izraža vrednost false ali 1, ki izraža vrednost true (Izsek kode 4).

Rešitev je precej enostavna. Spremenljivko nastavimo na vrednost 1 šele potem, ko dejansko sprostimo vsebino s klicem funkcije free. S tem smo že na polovici rešitve. Kar nam še ostane je, da napišemo preprosti if stavek, ki preveri ali je zapisek že sproščen. Če to drži izpišemo opozorilo, da tega zapiska ni mogoče sprostiti, saj je že sproščen (Izsek kode 5).

Čeprav uvedba zastavice is_freed zmanjša možnost nenamerne dvojne sprostitve je ta pristop v praksi precej omejen. Zastavica ne preprečuje zlonamernega prepisovanja metapodatkov ali manipulacije z drugimi podatkovnimi strukturami na kopici. V večjih projektih je takšna rešitev pogosto nezanesljiva, saj se stanje objekta lahko spremeni na več mestih in zastavica ne zagotavlja dejanske zaščite pred logičnim napadom

Druga možna rešitev bi bila posodobitev verzije glibc, katera bi potem preko validacij uspešno zaznala dvojno sprostitev in tako prekinila izvajanje programa. Rešitev le preloži problem, ne pa nujno odpravi temeljne pomankljivosti, saj imajo tudi nove verzije glibc lahko varnostne ranljivosti.

Poglejmo si še bolj praktične rešitve, ki se uporabljajo v produkciji:

- **Uporaba pomnilniško varnih programskih jezikov:** Predvsem tisti, ki vključujejo samodejni sistem za upravljanje s pomnilnikom, na primer jeziki kot so Python, Java, C#, JavaScript in drugi visoko-nivojski jeziki. Za dodeljevanje pomnilnika poskrbi sam jezik in odgovornost ne leži več na programerju. Slabost tega je slabša hitrost in učinkovitost programa.
- **Uporaba alternativnih in varnejših implementacij funkcije malloc:** Kot alternative nam bolj varne implementacije omogočajo varnejši program, brez da bi zato žrtvovali hitrost in učinkovitost našega programa. Implementacije ponavadi funkcijo malloc popolnoma spremenijo kakor tudi delovanje dodeljevanja blokov.
- **Upoštevanje defenzivnega programiranja/dobrih praks:** Preko testiranja in igranja v peskovniku lahko odkrijemo potencialne napade in ranljivosti našega programa, kar omogoča zaznavanje napadov in hitro odzivanje programa na njih. Npr. blokiranje dostopa nepooblaščeni osebi. Tako poskrbimo, da kljub napadu, program še vedno deluje. To je predvsem pomembno v aplikacijah, kjer je pomembna celodnevna dostopnost, visoka varnost in hitrost.

Uporaba orodij za statično analizo kode: to nam omogoča odkrivanje ranljivosti že v razvojnem procesu, uporaba tako imenovanega fuzz testiranja, ki avtomatično generira nepredvidljive, naključne in neveljavne vnose za testiranje funkcionalnosti programa z namenom, da ga pokvari.

8 ZAKLJUČEK

V članku smo raziskali ranljivosti, povezane z napačnim upravljanjem s pomnilnikom ter predstavili praktično demonstracijo napada dvojne sprostitve. Poudarili smo, kako pomembno je razumevanje delovanja kopice, saj lahko napadalci s tem znanjem, izkoristijo tovrstne ranljivosti.

Analiza je pokazala, da lahko že preproste napake v kodi, npr. nepravilno ravnanje s kazalci, vodijo do varnostnih posledic. Prav tako je tudi opozorila, da je pomembna previdna uporaba zunanjih knjižnic.

Predstavili smo tudi potencialne rešitve, med glavnimi so preverjanje stanja kazalca pred sprostitvijo, uporaba novejših verzij knjižnic, uporaba orodij za statično analizo kode ter uporaba jezikov z samodejnim upravljanjem s pomnilnikom.

Predstavljeni ukrepi so koristni, vendar imajo omejitve. Preverjanje kazalcev pred sprostitvijo prepreči le enostavne napake, ne pa logičnih zlorab v kompleksnih sistemih. Posodabljanje knjižnic zmanjša tveganje, vendar ne izključi novih ranljivosti in prinaša težave z združljivostjo. Statična analiza pogosto daje lažne pozitivne rezultate ter ne zazna vseh napak med izvajanjem. Tudi jeziki z samodejnim upravljanjem pomnilnika odpravljajo le del težav, saj ranljivosti pogosto obstajajo v logiki aplikacije ali integraciji zunanjih modulov.

Obstajajo tudi kompleksnejši in nevarnejši napadi na kopico, ki jih nismo obravnavali. Primer takih so:

- **Tcache poisoning:** zloraba tcache koša za ponovno uporabo prostih blokov pomnilnika.
- **House of einherjar** in **House of force:** manipulacija z metapodatki kopice za prevzem nadzora nad dodeljevanjem pomnilnika.
- **Heap spraying:** množično polnjenje kopice s predvidljivimi podatki, kar poveča verjetnost uspeha napada.

Ti napadi so nevarnejši, saj zahtevajo poglobljeno znanje o notranjih mehanizmih upravljanja s pomnilnikom in so pogosto odporni na osnovne zaščitne ukrepe.

Smer razvoja se nagiba k rešitvam, ki temeljijo na ASLR tehnologiji, kar nakazujejo tudi novejše raziskave *Oreo: Protecting ASLR Against Microarchitectural Attacks (Extended Version)* [20].

Zaključimo lahko, da je razumevanje delovanje kopice in preprečevanje kopičnih ranljivosti ključno za zagotavljanje varnosti sistemov.

LITERATURA

[1] Cowan, C., Beattie, S., Johansen, J., & Wagle, P. (2003). PointGuardTM: Protecting pointers from buffer overflow vulnerabilities. In Proceedings of the 12th USENIX Security Symposium (pp. 91–104). USENIX Association.

[2] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-free Detection. In Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17). Association for Computing Machinery, New York, NY, USA, 405–419. https://doi.org/10.1145/3064176.3064211

[3] Heelan, S., Melham, T., & Kroening, D. (2019). Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In Proceedings of the ACM Conference on Computer and Communications Security (pp. 1689–1706). Association for Computing Machinery. https://doi.org/10.1145/3319535.3354224

[4] Jia, X., Zhang, C., Su, P., Yang, Y., Huang, H., & Feng, D. (2017). Towards efficient heap overflow discovery. In Proceedings of the 26th USENIX Security Symposium (pp. 989–1006). USENIX Association.

[5] Gopal, A. U. S., Soori, R., Ferdman, M., & Lee, D. (2023). TAILCHECK: A Lightweight Heap Overflow Detection Mechanism with Page Protection and Tagged Pointers. 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), 535–552. https://www.usenix.org/conference/osdi23/presentation/gopal

[6] L. He et al., "Automatically assessing crashes from heap overflows,"2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana, IL, USA, 2017, pp. 274-279, doi: 10.1109/ASE.2017.8115640.

[7] Mouzarani, M., Sadeghiyan, B., & Zolfaghari, M. (2016). A smart fuzzing method for detecting heap-based vulnerabilities in executable codes. Security and Communication Networks, 9(18), 5098–5115. https://doi.org/10.1002/sec.1681

[8] Baradaran, S., Heidari, M., Kamali, A., & Mouzarani, M. (2023). A unit-based symbolic execution method for detecting memory corruption vulnerabilities in executable codes. International Journal of Information Security, 22(5), 1277–1290. https://doi.org/10.1007/s10207-023-00691-1

[9] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012). Association for Computing Machinery, New York, NY, USA, 133–143. https://doi.org/10.1145/2338965.2336769

[10] Gene Novark and Emery D. Berger. 2010. DieHarder: securing the heap. In Proceedings of the 17th ACM conference on Computer and communications security (CCS '10). Association for Computing Machinery, New York, NY, USA, 573–584. https://doi.org/10.1145/1866307.1866371

[11] Zhu, K., Lu, Y., & Huang, H. (2020). Scalable static detection of use-after-free vulnerabilities in binary code. IEEE Access, 8, 78713–78725. https://doi.org/10.1109/ACCESS.2020.2990197

[12] Qiang, W., Li, W., Jin, H., & Surbiryala, J. (2019). Mpchecker: Use-After-Free Vulnerabilities Protection Based on Multi-Level Pointers. IEEE Access, 7, 45961–45977. https://doi.org/10.1109/ACCESS. 2019.2908022

[13] Zekun Shen and Brendan Dolan-Gavitt. 2020. HeapExpo: Pinpointing Promoted Pointers to Prevent Use-After-Free Vulnerabilities. In Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC '20). Association for Computing Machinery, New York, NY, USA, 454–465. https://doi.org/10.1145/3427228.3427645

[14] Josselin Feist, Laurent Mounier, Sébastien Bardin, Robin David, and Marie-Laure Potet. 2016. Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free. In Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering (SSPREW '16). Association for Computing Machinery, New York, NY, USA, Article 2, 1–12. https://doi.org/10.1145/3015135.3015137

[15] cve.org "CVE". [Online]. Available: https://www.cve.org/

[16] Simon Hansman, Ray Hunt, A taxonomy of network and computer attacks, Computers & Security, Volume 24, Issue 1, 2005, Pages 31-43, ISSN 0167-4048, https://doi.org/10.1016/j.cose.2004.06.011. (https://www.sciencedirect.com/science/article/pii/S0167404804001804)

[17] Mohan V. Pawar, J. Anuradha, Network Security and Types of Attacks in Network, Procedia Computer Science, Volume 48, 2015, Pages 503-506, ISSN 1877-0509, https://doi.org/10.1016/j.procs.2015.04.126. (https://www.sciencedirect.com/science/article/pii/S1877050915006353)

[18] Liu, B., Olivier, P., & Ravindran, B. (2019). Slimguard: A secure and memory-efficient heap allocator. In Middleware 2019 - Proceedings of the 2019 20th International Middleware Conference (pp. 1–13). Association for Computing Machinery, Inc. https://doi.org/10.1145/3361525.3361532

[19] J. Ahn, K. Lee, C. Park, H. Moon and Y. Kwon, ŠwiftSweeper: Defeating Use-After-Free Bugs Using Memory Sweeper Without Stop-the-World,"in 2025 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2025, pp. 755-771, https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00131

[20] Song, S., Zhang, J., & Yan, M. (2024). Oreo: Protecting ASLR Against Microarchitectural Attacks (Extended Version). https://arxiv.org/abs/2412.07135

[21] Chao Ni, Liyu Shen, Xiaohu Yang, Yan Zhu, and Shaohua Wang. 2024. MegaVul: A C/C++ Vulnerability Dataset with Comprehensive Code Representations. In Proceedings of the 21st International Conference on Mining Software Repositories (MSR '24). Association for Computing Machinery, New York, NY, USA, 738–742. https://doi.org/10.1145/3643991.3644886

[22] Geeks for geeks. Dynamic Memory Allocation in C. URL: https://www.geeksforgeeks.org/c/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

[23] Azeria labs. Arm Heap Exploitation. URL: https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/

▪

Domen Breznik je študent 3. letnika 1. stopnje univerzitetnega študija na Fakulteti za računalništvo in informatiko Univerze v Ljubljani. Posebej ga zanimajo področja programske opreme in računalniške varnosti.

▪

Mark Novak je študent 2. letnika 1. stopnje univerzitetnega študija na Fakulteti za računalništvo in informatiko Univerze v Ljubljani. Posebej ga zanimajo področja razvoja videoiger, operacijski sistemi in računalniške varnosti.

▪

Matevž Pesek je izredni profesor 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 predmeta Varnost programov in Varnost sistemov, kjer se raziskovalno ukvarja s poučevanjem konceptov in organizacijo dogodkov s področja računalniške varnosti.