

# Brez gesla in brez problemov s prelivom spomina

Maks Kolman

April 2021

## 1 Uvod

Bolj kot se poglobimo v računalništvo, bolj spoznamo, da teži modernega sveta držijo le desetletja stara koda, lepilni trak in upanje. To negotovost najboljše prikažejo varnostne luknje v sistemih, ki so razširjeni po celem svetu. Primer take ranljivosti je *Heartbleed*, ki je bil zaznan leta 2014 in je omogočal javen vpogled v sisteme z ranljivo verzijo knjižnice OpenSSL. Prva ranljiva verzija je bila izdana leta 2012 do odkritja napake pa je bila ta prisotna že v več kot dveh tretjinah vseh spletnih strežnikov.

V tem članku se bomo poglobili v napako (poimenovano *Baron Sameedit*), ki so jo Januarja letos odkrili na sistemih Linux in macOS. Ranljivost so našli v programu *sudo* in pokazali da lahko vsak uporabnik dobi administratorske (root) pravice na sistemu ne da bi imel za to pravice ali poznal primerna gesla.

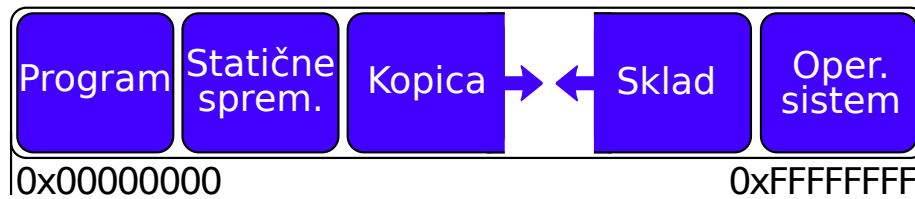
Tako Heartbleed kot Baron Sameedit sta ranljivosti, ki jih je povzročil prekomeren in neželen dostop do pomnilnika. Takšne vrste napak so med najbolj pogostimi in so tipične za programe napisane v jezikih C/C++. Da bomo lahko razumeli zakaj pride do takih ranljivost in zakaj lahko ostanejo tako dolgo skrite moramo najprej razumeti kako programi sploh uporabljajo spomin.

## 2 Model računalniškega pomnilnika

Računalniški pomnilnik (angl. Random Access Memory - RAM) opravlja pomembno vlogo pri delovanju računalnika. Vsakič, ko zaženemo katerikoli program, mu operacijski sistem dodeli del pomnilnika in vsak bajt spomina označi z unikatnim naslovom. Na 32-bitnih sistemih je vsak naslov sestavljen iz 32 bitov, ki zapisani v šestnajstiškem sistemu segajo od 0x00000000 do 0xFFFFFFFF.

Pomnilnik je razdeljen na predele, ki opravljajo različne naloge (Slika 1). Ob zagonu programa se v najmanjše naslove pomnilnika naloži koda programa, ki smo ga zagnali, za tem pa je prostor za vse statično definirane spremenljivke v našem programu. Na drugem koncu pomnilnika so programu na voljo knjižnice operacijskega sistema, ki jih uporablja za dostop do priklopljenih naprav npr. ekran, miška, tiskalnik, ali celo trdi disk. Med obema ekstremoma pomnilnika pa se nahajata dve vrsti dinamičnega spomina: kopica in sklad.

Sklad se nahaja pri višjih naslovih pomnilnika, v njem pa so shranjeni trenutno aktivni klici funkcij ter lokalne spremenljivke. Sklad deluje hitro, vendar je njegova velikost omejena na nekaj megabajtov (točna omejitev je odvisna od operacijskega sistema). Vse večje objekte mora program zato shraniti v kopico. Ta se nahaja pri manjših naslovih in raste proti večjim. Velikost mu omejuje



Slika 1: Model računalniškega pomnilnika ob delovanju programa.

le skupno število naslovov, ki jih ima program na voljo, ter velikost praznega spomina v računalniku.

### 3 Enostaven primer ranljivosti

Kako pa ranljiv program zgleda v praksi? V tem delu bomo pregledali konkreten primer napake. Spodaj je napisano v jeziku C++, ki skriva kritično napako. Jo lahko opaziš?

```

1  #include <stdio.h>
2  #include <string.h>
3
4  const char* const GESLO = "geslo123";
5  int main() {
6      int geslo_je_pravilno = 0;
7      char prebrano_geslo[100];
8
9      printf("Vpiši geslo: ");
10     gets(prebrano_geslo);
11
12     if (strcmp(prebrano_geslo, GESLO)) {
13         printf("Napačno geslo.\n");
14     } else {
15         printf("Pravilno geslo.\n");
16         geslo_je_pravilno = 1;
17     }
18
19     if (geslo_je_pravilno) {
20         // Uporabnik se je uspešno prijavil
21         printf("Dobrodošli v vaš račun.\n");
22     }
23     return 0;
24 }
```

Preden poskušamo razumeti napako se posvetimo delovanju programa. V prvih dveh vrsticah z uvozom knjižnic poskrbimo da bomo lahko brali iz standardnega vhoda, pisali na standardni izhod in primerjali nize znakov. Naša prva spremenljivka, `GESLO`, je enaka "geslo123" in predstavlja skrivno geslo, ki ga uporabnik potrebuje za vstop v svoj račun. Kot smo se naučili v prejšnjem odseku so statični podatki shranjeni pri nizkih naslovih.

Program se začne izvajati na peti vrstici s funkcijo `main()`. Najprej si pripravimo spremenljivko, ki nam bo povedala ali je to geslo pravilno, in jo nastavimo na ničelno vrednost takoj zatem rezerviramo 100 bajtov prostora za geslo, ki ga bo vpisal uporabnik. Obe spremenljivki se nahajata na skladu ena za drugo.

Uporabnika prosimo, da vnese geslo, ga preberemo in shranimo v `prebrano_geslo`. Naslednji blok kode primerja prebrano geslo in `GESLO` ter obvesti uporabnika ali je vnešeno geslo pravilno. V primeru pravilnega gesla si to zabeležimo v `geslo_je_pravilno`. Če se je uporabnik uspešno vpisal ga na koncu pozdravimo v njegov račun. V primeru resničnega programa bi imeli uporabniki tu dostop do podatkov ali funkcionalnosti namenjeno le njim.

Kako pa izvedba programa zgleda v praksi? Poglejmo si dva primera:

Vpiši geslo:	Vpiši geslo:
geslo123	123456
Pravilno geslo.	Napačno geslo.
Dobrodošli v vaš račun.	

Levo: Uspešen vpis v sistem. Desno: Neuspešen vpis v sistem.

Vidimo, da program očitno deluje. Kje je torej problem? Poglejmo kaj se zgodi, če poskusimo vnesti daljše geslo:

```
Vpiši geslo: iiii
iiii
iiii
iiii
Napačno geslo.
Dobrodošli v vaš račun.
[1] 692249 segmentation fault
```

Zanimivo. Program zazna, da je geslo napačno, vendar nas na koncu vseeno spusti v račun preden se sesuje z napako `segmentation fault`. Kako je to mogoče? Kaj se je zgodilo?

Problem je v delovanju funkcije `gets`. Ta se namreč ne meni za to koliko spomina ima na voljo, ampak je veselo napisala vse dobljene znake v spomin, četudi je to pomenilo, da je ob tem prepisala druge spremenljivke. Kot smo že omenili je spremenljivka `prebrano_geslo` shranjena v skladu. Tabela 1 prikazuje strukturo spomina v treh primerih.

	prebrano_geslo												geslo_je_pravilno				
...	'1'	'2'	'3'	'4'	'5'	'6'					...		0	0	0	0	...
...	'g'	'e'	's'	'l'	'o'	'1'	'2'	'3'			...		1	0	0	0	...
...	'i'	'i'	'i'	'i'	'i'	'i'	'i'	'i'	'i'	'i'	...	'i'	105	105	105	105	...

Tabela 1: Vsaka celica tabele predstavlja en bajt spomina v skladu. Na levi so naslovi z manjšim spominom, na desni pa z večjim. Tri vrstice ponazarjajo tri različne primere, ki smo jih poizkusili. Vidimo, da je v tretjem primeru vneseno besedilo preraslo 100 bajtov dodeljenih spremenljivki `prebrano_geslo` in se prelilo v naslednje celice spomina.

Zanima nas predvsem tretji primer, kjer smo z besedilo prepisali število `geslo_je_pravilno`. Bajt ki smo ga ponavljajoče pisali je  $01101001_2$ . Kot znak se to prevede v 'i', kot število pa v 105. Vrednost spremenljivke `geslo_je_pravilno` je tako neničelna, kar nam dovoli vstop v račun. Bolj natančno je vrednost štirikrat ponovljena vrednost 105 v binarnem kar je enako

$$01101001\ 01101001\ 01101001\ 01101001_2 = 105 + 105 \cdot 2^8 + 105 \cdot 2^{16} + 105 \cdot 2^{24} = 1768515945.$$

Zaradi prekomerne količine i-jev se te nadaljujejo še naprej po spominu. Za potrebe našega razumevanja je pomembno, da nekoč naletijo na vrnitveni naslov (return address). To je spremenljivka v kateri je shranjen naslov spomina, kjer bo program nadaljeval izvajanje, ko se trenutna funkcija zaključi. Ker smo ta naslov prepisali z `0x69696969` (105 v šestnajstiškem sistemu je  $69_{16}$ ) je program poskusil dostopati do neveljavnega spomina, kar je povzročilo napako **segmentation fault**. Iz tega smo se naučili, da ne rabimo pretiravati z dolžino gesla, ki ga vpišemo. Če je geslo daljše od 100 znakov vendar ne predolgo, lahko dobimo dostop do sistema ne da bi se potem sesul.

Tako previdno ravnanje podcenjuje moč dejstva, da imamo dostop do vrnitvenega naslova in do razmeroma velikega dela spomina, ki ga lahko prepišemo po svoji volji. Vrnitveni naslov bi lahko spremenili, tako da bi bil usmerjen v spomin znotraj spremenljivke `prebrano_geslo`. To pomeni, da bi računalnik začel interpretirati bajte kot ukaze programa. V našem primeru bi ponavljal ukaz `0x69`, kar je na 32-bitnih Intel procesorjih ukaz za množenje.

V principu lahko v ta del spomina napišemo kakršenkoli program si zaželim. Potrebno je samo, da je program manjši od 100 bajtov in da se lahko izvede v takih okoliščinah. Najbolj zanimiv program, ki ga lahko zaženemo je ukazna vrstica (shell), saj lahko z njeno pomočjo nato zaženemo poljubne druge programe. Točna priprava in umestitev programa je zunaj sklopa tega članka lahko pa najdete več učnih virov na to temo na spletu.

## 4 Ranljivost Baron Samedit

Ukaz *sudo* (*angl.* SuperUser DO) dovoli izbranim uporabnikom sistema, da zaženejo druge programe s pravicami superuporabnika. Podobno kot administrator na sistemu Windows ima superuporabnik neomejeno moč na sistemu. Lahko namesti nove programe, prebere vse datoteke na sistemu - tudi tiste, ki so last drugih uporabnikov - ali, če želi, iz računalnika izbriše celoten operacijski sistem. Superuporabnik je vsemogočen.

Torej si lahko mislite, kako nevarno bi bilo, če bi imeli do tega ukaza dostop vsi uporabniki. Prav takšno ranljivost so letos odkrili raziskovalci pri Qualys Research Labs. Vsak z dostopom do uporabniškega računa na sistemu lahko dobi pravice superuporabnika. Tudi če ta račun sicer nima dovoljenja za uporabo *sudo*. Tudi če uporabnik ne pozna gesla računa, ki ga upravlja. Najbolj presenetljivo je, da je ta napaka v *sudo* prisotna že od Julija leta 2011.

Dejansko napako so našli pri uporabi ukaza *sudedit*, ki je točna kopija ali celo povezava programa *sudo*. Ukaz je namenjen urejanju tekstovnih datotek s pravicami superuporabnika. Namesto, da s *sudo* odpreš urejevalnik besedila in datoteko, lahko s *sudedit* neposredno odpreš datoteko in ti program sam odpre urejevalnik besedila. V obeh primerih program preveri, da ima uporabniški

račun dovoljenje za uporabo sudo in zahteva da uporabnik vnese geslo računa, ki ga uporablja.

Ranljivost je prisotna pri uporabi argumenta `-s`, saj bo, v primeru da se njegov argument konča z `\` prekoračil meje svoje spremenljivke in začel nenadzorovano pisati po spominu. Za razliko od našega primera v prejšnjem odseku se vse to dogaja v kopici, ne v skladu. Če vas zanima ali je vaš sistem ranljiv lahko poizkusite pognati spodnji ukaz.

```
sudoedit -s '\ ' 'To besedilo se bo prelilo po spominu'
```

Če dobite napako **Segmentation Fault**, je vaš sistem še vedno ranljiv in bi bilo pametno, da ga posodobite. V nasprotnem primeru bi se vam morala na zaslon izpisati navodila za uporabo, začenši z `usage: sudoedit`. Ker je napaka prisotna v kopici in ne v skladu se moramo za uporabo te ranljivosti boriti še z randomizacijo spomina (*angl.* Address space layout randomization - ASLR). Naloga ASLR je da naključno spreminja na katerem naslovu pomnilnika se nahaja kakšna spremenljivka, kar močno oteži napade, kjer je potreben natančen dostop do spremenljivk.

V tem primeru so problem rešili tako, da so preizkusili vse možnosti. Ker ranljivost preverijo hitro, in ker je možnosti razmeroma malo so lahko raziskovalci dobili neomejen dostop do sistema v manj kot 20 sekundah. Po tem, ko so raziskovalci našli ranljivost so 13. Januarja o tem skrivaj obvestili razvijalce programa sudo. Nato so skupaj 19. Januarja poslali popravek vsem distribucijam Linuxa, ki so ga vsi skupaj izdali teden dni kasneje. Pri takšni ranljivosti je zelo pomembno, da ni prehitro razkrita. To bi pomenilo, da bi bilo veliko ranljivih sistemov brez obrambe proti vsem ki bi znali uporabiti to za napad.

Več podrobnosti o raziskavi in ranljivosti si lahko preberete na njihovi spletni strani [blog.qualys.com](http://blog.qualys.com).

## 5 Zaključek

Naslednjič, ko boste programirali, raje dvakrat pomislite ali ima vaš sistem trde temelje in ne uporabljate ranljivih funkcij. Na primer uporaba funkcije `gets` nam v novejših prevajalnikih kode prikaže opozorilo, da njena uporaba ni varna. Od verzije `c++14` naprej pa `gets` sploh ni več na voljo. Varnost programov in programskih jezikov se izboljšuje, vendar so jim varnostni raziskovalci vselej za petami.

Če od tega članka odnesete le eno stvar, naj bo to, da morate redno in skrbno posodabljeni svoje računalnike in še posebej sisteme, do katerih ima dostop veliko ljudi.