Brez gesla in brez omejitev s prelivom spomina

Maks Kolman

Uvod

Ko se poglobimo v računalništvo, spoznamo, da težo modernega sveta držijo le desetletja stara koda, lepilni trak in upanje. To negotovost najbolje prikažejo varnostne luknje v sistemih, ki so razširjeni po celem svetu. Primer take ranljivosti je *Heartbleed*, ki je bil zaznan leta 2014. Heartbleed je omogočal javen vpogled v sisteme z ranljivo verzijo knjižnice OpenSSL, ki jo uporabljamo za vzpostavljanje varne povezave do strežnikov. Prvo ranljivo verzijo so izdali leta 2012, do odkritja napake pa je bila ta prisotna že v več kot dveh tretjinah vseh spletnih strežnikov.

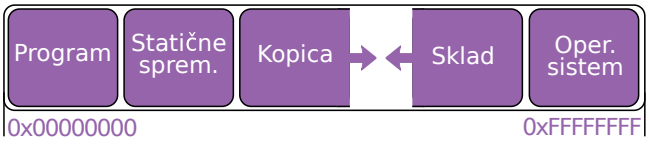
V tem članku se bomo poglobili v napako (poimenovano *Baron Samedit*), ki so jo januarja letos odkrili na sistemih Linux in macOS. Ranljivost so našli v programu *sudo* in pokazali, da lahko dobi vsak uporabnik administratorske (root) pravice na sistemu, ne da bi imel za to pravice in brez poznavanja kakršnih koli gesel.

Tako Heartbleed kot Baron Sameedit sta ranljivosti, povzročeni s prekomernim in neželenim dostopom 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 uporabljajo spomin.

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. Ko jih zapišemo 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 zagnanega programa. Poleg 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 povezanih naprav, kot so ekran, miška, tiskalnik ali trdi disk. Med dvema skrajnostma pomnilnika se nahajata dve vrsti dinamičnega spomina: kopica in sklad.



SLIKA 1.
Model računalniškega pomnilnika ob delovanju programa

Sklad se nahaja pri višjih naslovih pomnilnika. V njem 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 posledično shraniti v kopico. Kopica se nahaja pri manjših naslovih in raste proti večjim. Velikost mu omejuje le skupno število naslovov, ki jih ima program na voljo, ter količina prostega spomina v računalniku.

Preprost primer ranljivosti

Kako zgleda ranljiv program v praksi? V tem delu bomo pregledali konkretni primer napake. Spodaj je program, napisan v jeziku C++ in skriva kritično napako. Jo opazimo?

```
#include <cstdio>
#include <cstring>

const char* const GESLO = "geslo123";
int main() {
    int geslo_je_pravilno = 0;
    char prebrano_geslo[100];

    printf("Vpiši geslo: ");
    gets(prebrano_geslo);

    if (strcmp(prebrano_geslo, GESLO)) {
        printf("Napačno geslo.\n");
    } else {
        printf("Pravilno geslo.\n");
        geslo_je_pravilno = 1;
    }

    if (geslo_je_pravilno) {
      // Uporabnik se je uspešno prijavil
      printf("Dobrodošli v vaš račun.\n");
    }
    return 0;
}
```

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.

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. Potem 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 vneseno geslo pravilno. Če je geslo pravilno, si to zabeležimo v `geslo_je_pravilno`. Če se je uporabnik uspešno vpisal, ga na koncu pozdravimo v njegovem računu. V primeru resničnega programa bi imeli uporabniki tu dostop do podatkov ali funkcionalnosti, ki je namenjena le njim.

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

```
Vpiši geslo:
geslo123
Pravilno geslo.
Dobrodošli v vaš
račun.
```

```
Vpiši geslo:
123456
Napačno geslo.
```

PRIMER.
Levo: uspešen vpis v sistem. Desno: neuspešen vpis v sistem.

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

```
Vpiši geslo:  iiiiiiiiiiiiiiiiiiiiiiii
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
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 veselo napiše vse dobljene znake v spomin, četudi to pomeni, da ob tem prepiše druge spremenljivke. Kot smo že omenili, je spremenljivka `prebrano_geslo` shranjena na skladu. Tabela 1 prikazuje strukturo spomina v treh primerih.

Zanima nas predvsem tretji primer, kjer smo z besedilom prepisali število `geslo_je_pravilno`. Ponavljajoče zapisan bajt 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 siste-

	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 je spomin z manjšim naslovom, na desni pa z večjim. Tri vrstice ponazarjajo tri različne primere, ki smo jih poizkusili. Vidimo, da je v tretjem primeru vneseno besedilo preseglo 100 bajtov, dodeljenih spremenljivki `prebrano_geslo`, in se prelilo v naslednje celice spomina.

mu, 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 ti nadaljujejo še naprej po spominu. Za potrebe našega razumevanja je pomembno, da nekoč naletijo na vrnitveni naslov funkcije (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 ni potrebno 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 ta sesul.

Takšna previdnost podcenjuje dejstvo, 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 zaželimo. 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 zaženemo poljubne druge programe. Točna priprava in umestitev programa presega namen tega članka. Več učnih virov na to temo pa lahko najdete na spletu.

Ranljivi 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. Namesti lahko nove programe, prebere vse datoteke na sistemu, tudi tiste, ki so last drugih uporabnikov. Če želi, lahko iz računalnika celo izbriše operacijski sistem. Superuporabnik je vsemogočen.

Torej si lahko mislite, kako nevarno bi bilo, če bi imeli do tega ukaza nenadzorovan 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*, ali če uporabnik ne pozna gesla računa, ki ga upravlja. Najbolj presenetljivo je, da je ta napaka v *sudo* prisotna že od julija 2011.

Dejansko napako so našli pri uporabi ukaza *sudoedit*, ki je točna kopija ali celo povezava programa *sudo*. Ukaz je namenjen urejanju tekstovnih datotek s pravicami superuporabnika. Sudoedit avtomatično odpre urejevalnik besedila in v njem datoteko, ki jo hoče uporabnik urediti. Pred tem program preveri, da ima uporabniški račun dovoljenje za uporabo sudo, in zahteva, da vnese uporabnik geslo računa, ki ga uporablja.

Ranljivost je prisotna pri uporabi argumenta `-s`. V primeru, da se njegov argument konča s poševnico (\), bo 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 ga morate posodobiti. 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, nam izkoriščanje te ranljivosti preprečuje še randomizacija spomina (*angl.* Address space layout randomization – ASLR). Naloga ASLR je, da naključno spreminja, na katerem naslovu pomnilnika se nahaja kakšna spremenljivka. S tem zaščitimo programe pred napadi, za katere je potreben natančen dostop do spremenljivk.

V tem primeru so problem rešili tako, da so napad pognali 128 tisočkrat na različnih naslovih. Ker ranljivost preverijo hitro in je možnosti razmeroma malo, lahko raziskovalci dobijo neomejen dostop do sistema v manj kot 20-ih sekundah. Po tem, ko so raziskovalci našli ranljivost, so 13. januarja o tem skrivaj obvestili razvijalce programa sudo. Nato so 19. januarja poslali popravek vzdrževalcem vseh večjih distribucij Linuxa, ki so teden dni kasneje istočasno izdali popravljene verzije. Pri takšni ranljivosti je zelo pomembno, da ni prehitro javno razkrita. Z etičnim razkritjem kritičnih napak raziskovalci poskrbijo, da lahko vsak zaščiti svoje naprave, preden navodila za napad pristanejo v rokah javnosti.

Več podrobnosti o raziskavi in ranljivosti si lahko preberete na njihovi spletni strani `blog.qualys.com`.

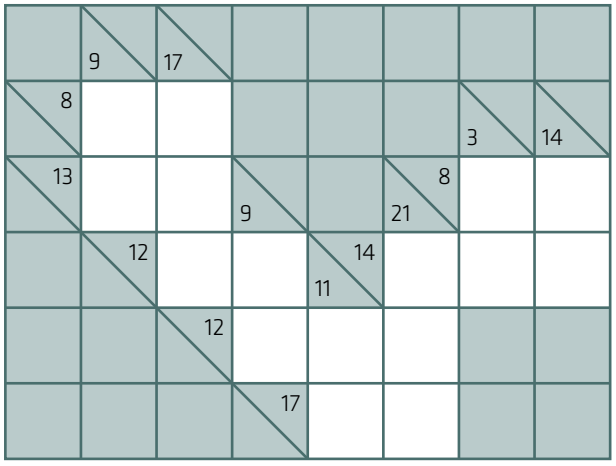
Zaključek

Naslednjič, ko boste programirali, raje dvakrat preverite, ali ima vaš sistem trdne temelje in ali uporabljate ranljive funkcije. Uporaba funkcije `gets` nam na primer 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, da najdejo vse, še tako skrite, napake.

Če vas ta članek nauči le eno stvar, naj bo to: redno in skrbno posodabljajte svoje računalnike, še posebej sisteme, do katerih ima dostop veliko ljudi.

Križne vsote

→ Naloga reševalca je, da izpolni bele kvadratke s številkami od 1 do 9 tako, da bo vsota števk v zaporednih belih kvadratkih po vrsticah in po stolpcih enaka številu, ki je zapisano v sivem kvadratku na začetku vrstice (stolpca) nad (pod) diagonalo. Pri tem morajo biti vse številke v posamezni vrstici (stolpcu) različne.



REŠITEV KRIŽNE VSOTE