HRVATSKA AKADEMSKA I ISTRAŽIVAČKA MREŽA
CROATIAN ACADEMIC AND RESEARCH NETWORK

Propusti prepisivanja spremnika

CCERT-PUBDOC-2007-08-202

CARNet CERT u suradnji s LS&S

Sigurnosni problemi u računalnim programima i operativnim sustavima područje je na kojem CARNet CERT kontinuirano radi.

Rezultat toga rada ovaj je dokument, koji je nastao suradnjom CARNet CERT-a i LS&S-a, a za koji se nadamo se da će Vam koristiti u poboljšanju sigurnosti Vašeg sustava.

CARNet CERT, www.cert.hr - nacionalno središte za **sigurnost računalnih mreža** i sustava.

LS&S, www.lss.hr - laboratorij za sustave i signale pri Zavodu za elektroničke sustave i obradbu informacija Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

Sadržaj

1. Uvod

Prepisivanje spremnika (eng. *buffer overflow*) je oblik nepravilnog ponašanja programa koji karakterizira pokušaj upisivanja određene količine podataka u memorijski spremnik koji ju ne može primiti. Ovakva situacija dovodi do upisa dijela podataka u spremnik, dok se preostali dio podataka zapisuje u okolni memorijski prostor, prepisujući tako podatke koji su se tamo prije nalazili.

Opisani scenarij predstavlja velik problem jer prepisivanje okolnog memorijskog prostora dovodi do nepredvidljivih rezultata. Naime, ovisno o sadržaju kojim je spremnik prepisan, moguće su posljedice rušenja aplikacije ili njen nepravilni rad. Ukoliko se u obzir uzmu zlonamjerni korisnici i napadači, slika postaje daleko gora. Posebnim oblikovanjem podataka kojima će se prepisati sadržaj spremnika napadači, osim dva već spomenuta scenarija, mogu i pokrenuti izvođenje proizvoljnog programskog koda koje ih može dovesti do potpunog preuzimanja nadzora nad ranjivim sustavom. Upravo zbog toga, kao i zbog relativne jednostavnosti i pouzdanosti zlouporaba temeljenih na iskorištavanju propusta prepisivanja spremnika, ove ranjivosti su najčešće iskorištavana vrsta propusta, a njihove zlouporabe obično imaju i najozbiljnije posljedice.

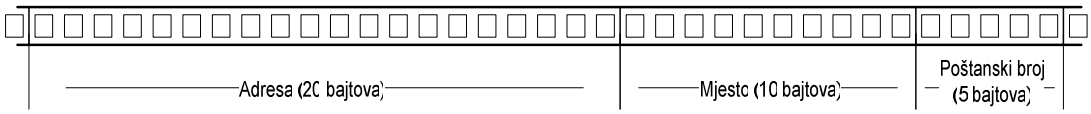
U ostatku dokumenta opisane su temeljne značajke programskog stoga i programske gomile te principi iskorištavanja propusta prepisivanja spremnika pohranjenih na njima. Posebna je pažnja posvećena opisu tehnika odgovarajuće zaštite i analizi njihovih kako pozitivnih, tako i negativnih strana.

2. Općenito o prepisivanju spremnika

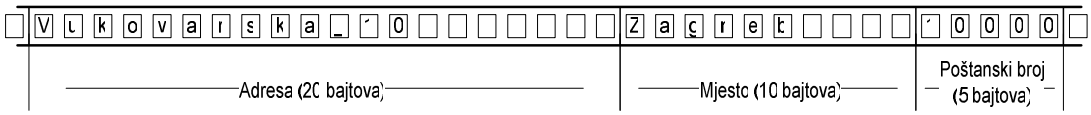
Spremnik (eng. *buffer*) je dio memorijskog prostora namijenjen privremenoj pohrani podataka. Spremnici mogu imati različitu veličinu kako bi mogli pohraniti različite vrste podataka koji mogu biti cijeli brojevi, realni brojevi, znakovni nizovi, ali i tzv. sirovi (eng. *raw*) podaci proizvoljne duljine. Spremnici se koriste i za pohranu korisnički unesenih ulaznih podataka, ali i za obradu podataka dobivenih od udaljenih računala.

Svim je spremnicima zajedničko ograničenje njihove veličine. To znači da svaki spremnik ima svoju fiksnu veličinu te se u njega ne može pohraniti više podataka bez da ga se ponovno definira. Ukoliko se u spremnik ipak pokuša upisati više podataka nego što u njega stane, dolazi do pojave prepisivanja spremnika (eng. *buffer overflow*). Pritom se suvišak podataka pohranjuje u memorijski prostor koji ne pripada spremniku te se, ovisno o tome koji se dio memorijskog prostora prepisuje, može izazvati ponašanje programa drukčije od onog koje je programer inicijalno želio.

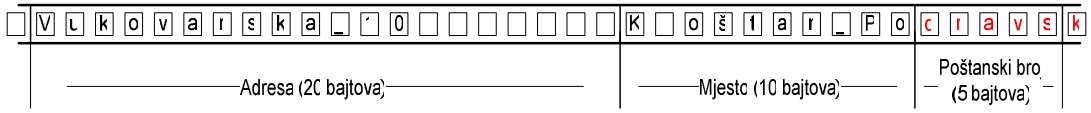
Na slijedećoj slici prikazana je pojava prepisivanja spremnika. U a) dijelu slike prikazan je raspored triju spremnika koji se nalaze jedan iza drugog u memorijskom prostoru. Spremnici su namijenjeni pohrani ulice, mjesta i poštanskog broja stanovanja neke osobe. Na b) dijelu slike vidljiv je raspored podataka u memoriji u situaciji kada ne dolazi do pojave prepisivanja spremnika. Svaki od dijelova adrese "Vukovarska 10, 10000 Zagreb" kraći je od odgovarajućeg spremnika te je pohrana uredna. U c) dijelu slike podatak o mjestu u adresi "Vukovarska 10, 48362 Kloštar Podravski" je dulji od predviđenih 10 bajtova te prepisuje cijeli spremnik za pohranu poštanskog broja, ali i slijedeći nezauzeti memorijski prostor.



a) spremnici u memorijskom prostoru



b) izgled memorije prilikom unosa bez prepisivanja spremnika



c) izgled memorije kod pojave prepisivanja spremnika

Slika 1. Izgled memorije prilikom pojave prepisivanja spremnika

Općenito je vrlo teško predvidjeti što se nalazi u memorijskom prostoru iza nekog spremnika, odnosno što će biti prepisano u slučaju njegovog prepisivanja. Iznimno, to je nešto jednostavnije ukoliko je spremnik pohranjen na stogu.

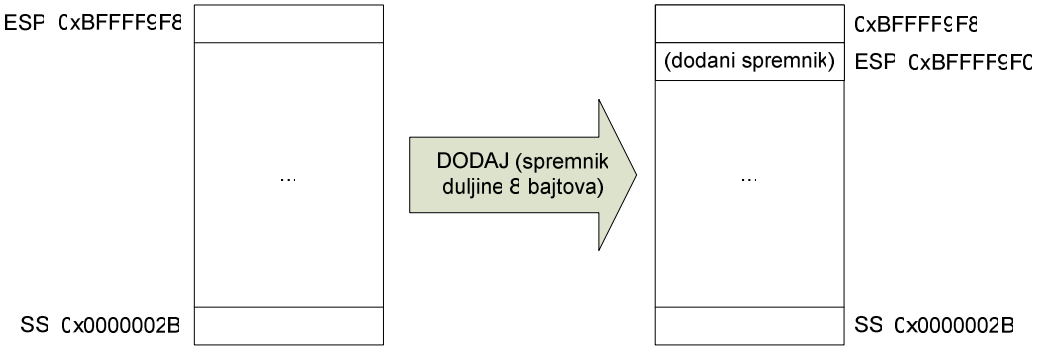
2.1. Osnovno o stogu

Stog je struktura podataka koja se koristi po principu zadnji-unutra-prvi-van (eng. *Last In first Out - LIFO*). Funkcionalno ga se može zamisliti kao hrpu podataka u kojoj elementi stoje jedan na drugome i koja može odgovoriti na dva zahtjeva:

- SKINI (eng. POP) i
- POSTAVI (eng. PUSH).

Akcija POSTAVI pohranjuje novi podatak na vrh stoga, dok akcija SKINI dohvaća podatak s vrha stoga. U programiranju se struktura stoga koristi za ostvarenje funkcionalnosti pozivanja procedura i za pohranu privremenih podataka tijekom njihovog izvođenja.

32-bitni procesori Intel arhitekture stog implementiraju kao neprekinuti dio memorijskog prostora za čije upravljanje koriste dva procesorska registra SS i ESP. SS (eng. *Stack Segment*) pohranjuje adresu baze stoga, tj. najvišu adresu do koje stog svojim rastom može doći. ESP (eng. *Extended Stack Pointer*) sadržava adresu trenutnog vrha stoga, te se njegova vrijednost mijenja kako se podaci dodaju, odnosno brišu sa stoga. Sam procesor ima sklopovski implementirane funkcije SKINI i POSTAVI.
Slijedeća slika prikazuje izgled praznog stoga i stoga nakon što mu je dodan jedan podatak.



Slika 2. Dodavanje elementa na stog

Sa slike je vidljivo da stog raste prema dolje, odnosno prema nižim adresama, primičući se tako adresi pohranjenoj u spremniku SS. Budući da je veličina stoga konačna i ograničena, moguće je u njega pokušati staviti više podataka nego što on može prihvatiti. U tom slučaju dolazi do pojave prepisivanja stoga (eng. *stack overflow*).

2.2. Osnovno o pozivanju procedura

Većina programskih jezika nudi mogućnost jednokratnog pisanja odsječka programskog koda koji se kasnije može pozivati iz više mjesta u programu. Različiti jezici nude različite nazive za ove odsječke te ih nazivaju funkcijama, metodama ili procedurama. U ovom dokumentu koristi se jedinstven termin procedure.
Procesor procedure podržava tzv. CALL instrukcijom. Prilikom njenog izvođenja na stog se pohranjuje adresa instrukcije koja se nalazi neposredno iza CALL, a daljnje izvršavanje programa nastavlja se od prve instrukcije procedure. Procedura nakon svog završetka glavnom programu vraća kontrolu pozivom RET instrukcije koja dohvaća adresu pohranjenu na stogu i nastavlja izvršavanje programa od te adrese. Primjer rada ovog mehanizma prikazan je slijedećim ispisom programskog koda.

```
Adresa    Sadržaj          Asembler
8048080:  e8 0c 00 00 00   call 0x8048091
8048085:  bb 00 00 00 00   mov $0x0,%ebx
804808a:  b8 01 00 00 00   mov $0x1,%eax
804808f:  cd 80            int $0x80
8048091:  31 c0            xor %eax,%eax
8048093:  c3               ret
```

Kada program u svom izvođenju dođe do adrese 8048080, naredba CALL na stog će pohraniti podatak 8048085 (adresu slijedeće instrukcije) te će izvršavanje programa nastaviti od adrese (8048091) koja predstavlja adresu prve instrukcije u proceduri. Nakon izvršavanja instrukcije s adrese 8048091 na red dolazi instrukcija na adresi 8048093. Budući da se radi o RET instrukciji, ona sa stoga uzima pohranjeni podatak (broj 8048085 koji je pohranila instrukcija CALL) te ga interpretira kao adresu od koje treba nastaviti daljnje izvršavanje programa.
Adresa koju instrukcija CALL postavlja na stog uobičajeno se naziva povratnom adresom (eng. *return address*). Ukoliko napadač uspije promijeniti povratnu adresu na stogu u adresu početka zlonamjerno oblikovanog programskog koda, danu situaciju može iskoristiti za preuzimanje kontrole nad sustavom. To je ujedno i cilj napada prepisivanjem spremnika.

2.3. Konvencija pozivanja procedura

Da bi se posve razumjelo prepisivanje spremnika, potrebno je znati kako se stog koristi u stvarnim aplikacijama prilikom pozivanja procedura. Iako su CALL i RET instrukcije važan dio rukovanja procedurama, u modernim je aplikacijama funkcionalnost koju nude nedovoljna.
Pravila koja propisuju korištenje stoga i registara prilikom pozivanja procedura poznata su pod nazivom Konvencija standardnog pozivanja (eng. *Standard Calling Convention*). Svrha ove konvencije je omogućiti interakciju i simultani rad programskog koda dobivenog korištenjem različitih programskih jezika i različitih prevoditelja programskih jezika.
Konvencija standardnog pozivanja nije službeni standard i razlikuje se kod različitih računalnih arhitektura, kao i kod različitih operacijskih sustava. Konvencija opisana u ovom dokumentu koristi se u Slackware Linux distribuciji inačice 9.1 za IA-32 platforme.

2.3.1. Konvencija standardnog pozivanja IA-32 arhitekture

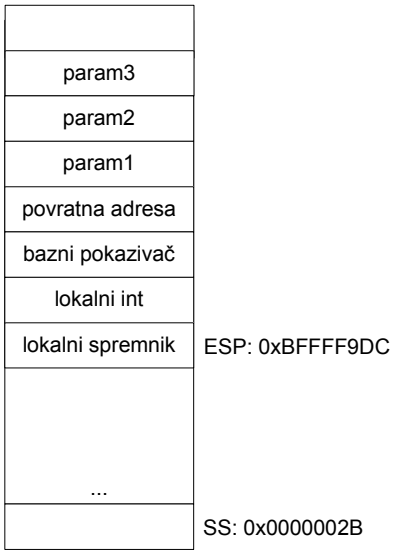
Prvi dio Konvencije bavi se pitanjem pohrane parametara procedure i utvrđuje da se parametri na stog pohranjuju obrnutim poretkom (s desna nalijevo).
Nakon što su parametri postavljeni, poziva se CALL instrukcija kako bi se pokrenula procedura. Kao što je prethodno opisano, CALL instrukcija ujedno postavlja i povratnu adresu na vrh stoga.
Jednom pozvana, procedura pozivateljev bazni pokazivač (eng. *Caller's Base Pointer*) koji se nalazi u EBP registru seli na stog te u EBP registar postavlja svoj bazni pokazivač (tj. kopira vrijednost registra ESP).
Bazni pokazivač je koncept koji omogućuje jednostavan pristup parametrima i ostalim podacima na stogu bez poznavanja njihove apsolutne adrese.
Procedura često treba memorijski prostor kako bi radila sa svojim podacima, tj. podacima koji za vanjski svijet nemaju nikakvo značenje. Ti podaci se pohranjuju na stog i poznati su pod nazivom lokalne varijable. Konvencija za njihovu pohranu predviđa prostor neposredno iza pohranjenog baznog pokazivača.
Nakon završetka procedure, memorija zauzeta na stogu za pohranu lokalnih varijabli vraća se kopiranjem vrijednosti EBP registra u ESP. Nakon toga se obnavlja pozivateljev bazni pokazivač skidanjem vrijednosti sa stoga u registar EBP. Na koncu slijedi poziv instrukcije RET koja vraća kontrolu pozivajućem programu.
Kako bi se bolje prikazala Konvencija standardnog pozivanja, može se razmotriti primjer slijedeće funkcije:

```
void proc( int param1, int param2, int param3 );
```

Procedura `proc` ima tri parametra, te ih pozivajući program prije pozivanja mora postaviti na stog u obratnom poretku.
Nakon izvršenja instrukcije CALL, na stog se pohranjuje povratna adresa te se započinje s izvođenjem procedure `proc`. Ona na samom početku radi tri stvari:

1. na stog pohranjuje pozivateljev bazni pokazivač,
2. incijalizira svoj bazni pokazivač na vrijednost adrese trenutnog vrha stoga te
3. pravi mjesto za svoje lokalne varijable umanjujući vrijednost ESP pokazivača.

Za potrebe ovog primjera pretpostavlja se da procedura ima dvije lokalne varijable: jednu cjelobrojnog tipa i jednu koja služi kao 8-bajtni spremnik.
Slijedeća slika prikazuje izgled stoga u trenutku kad procedura završava sa inicijalizacijskim radnjama i započinje izvođenje svog zadatka.



Slika 3. Izgled stoga prilikom poziva procedure

Kada procedura završi sa svojim izvođenjem u spremnik ESP pohranjuje vrijednost spremnika EBP, što je ekvivalentno skidanju gornjih dvaju elemenata sa stoga (podataka "lokalni int" i "lokalni spremnik"). Nakon toga skida još jedan element ("bazni pokazivač") sa stoga i pohranjuje ga u spremnik EBP. Procedura i formalno završava pozivom RET instrukcije koja sa stoga skida povratnu adresu (gornji element) i nastavlja izvođenje glavnog programa od te adrese. Na pozivajućem programu je da sa stoga ukloni parametre koje je na njega i stavio ("param1", "param2" i "param3").

3. Vrste prepisivanja spremnika

3.1. Prepisivanje spremnika pohranjenog na stogu (eng. *stack-based buffer overflow*)

Tehnički potkovan zlonamjeran korisnik prepisivanje spremnika pohranjenog na stogu može iskoristiti na nekoliko načina:

- prepisivanjem vrijednosti lokalne varijable koja se nalazi u blizini lokalnog spremnika može izmijeniti ponašanje programa sebi u korist,
- prepisivanjem povratne adrese na stogu, što će dovesti do pokretanja programskog koda koji se nalazi na upisanoj adresi, te
- prepisivanjem pokazivača na funkciju, što dovodi do pokretanja druge funkcije u trenutku kad je trebala biti pokrenuta izvorna.

Od sva tri pobrojana načina najčešće se pod prepisivanjem spremnika pohranjenog na stogu podrazumijeva drugi. Kako bi se bolje razumjela tehnička pozadina ove tehnike, potrebno je razmotriti slijedeći primjer:

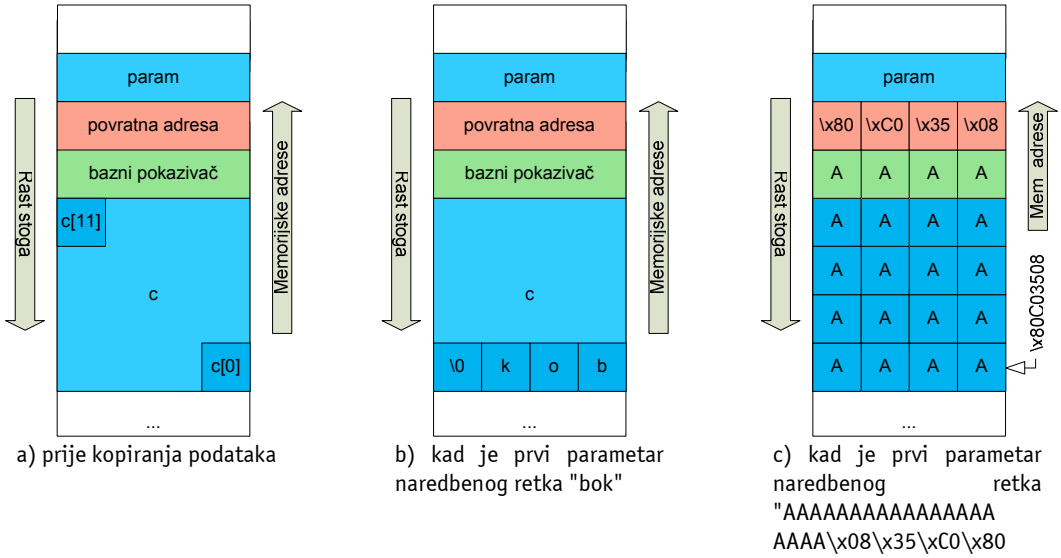
```
void procedura (char *param) {
  char  c[12];

  strcpy(c, param);  // nema provjere veličine...
}

int main (int br_argumenata, char **argumenti) {
  procedura(argumenti[1]);
}
```

Ovaj odsječak programskog koda parametar dobiven iz naredbenog retka prosljeđuje proceduri `procedura(...)` koja ga kopira u stogovni spremnik `c`. Opisani slijed događaja je sasvim uredan

sve dok se programu prosljeđuju ulazni parametri duljine do 12 znakova. Svaki uneseni znakovni niz dulji od 11 znakova uzrokovat će prepisivanje spremnika (Najveći broj znakova koji je sigurno moguće koristiti je za jedan manji od duljine spremnika jer se u programskom jeziku C znakovni nizovi završavaju tzv. nul-znakom, odnosno bajtom čija je vrijednost 0).



Slika 4. Izgled stoga prilikom pokretanja procedure s različitim ulaznim parametrima

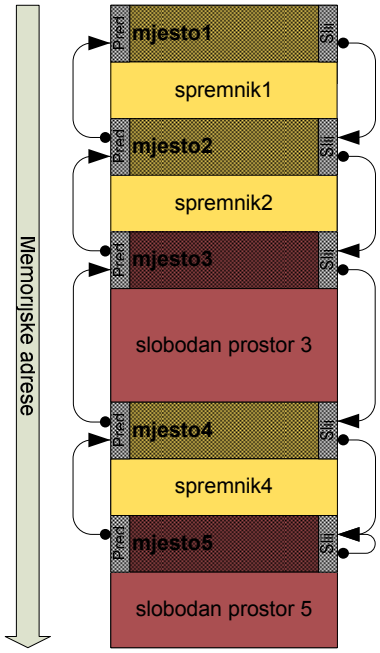
U c) dijelu gornje slike vidljivo je kako uneseni podatak prepisuje bazni pokazivač i, što je još važnije, povratnu adresu. Kada procedura završi sa svojim izvođenjem, ona sa stoga preuzima pohranjenu povratnu adresu i nastavlja izvođenje programa od te adrese u memorijskom prostoru računala. Kao što se na slici može vidjeti, napadač je prepisao povratnu adresu sa adresom početka spremnika `c`. To znači da će se nakon povratka iz procedure započeti s izvođenjem izmijenjenog sadržaja spremnika `c`. U slučaju stvarne zlouporabe napadač ne bi spremnik prepisao nizom znakova A, već bi to učinio proizvoljnim strojnim kodom, a ako je ranjivi program bio pokrenut s administratorskim ovlastima, to bi značilo i mogućnost preuzimanja potpunog nadzora nad ranjivim računalom.

3.2. Prepisivanje spremnika pohranjenog na programskoj gomili (eng. *Heap-based buffer overflow*)

Prepisivanje spremnika pohranjenog na gomili (eng. *heap*) sasvim je drukčije od prepisivanja onog koji je pohranjen na stogu. Programska gomila služi kao dinamičko spremište podataka u kome se prostor zauzima i oslobađa bez prethodno utvrđenog poretka.

Za razliku od stoga čija je organizacija manje-više uniformna na većini platformi, organizacija gomile je vrlo različita ovisno o operacijskom sustavu, sklopovlju, ali i alatu kojim je aplikacija razvijena. Upravo je zbog toga zlouporaba propusta prepisivanja spremnika pohranjenog na gomili daleko složenija od prepisivanja spremnika pohranjenog na stogu.

Kako bi se razumjela tehnička pozadina rada programske gomile, na slijedećoj je slici, više shematski nego realistično, prikazan njen uobičajen izgled.



Slika 5. Organizacija memorijskog prostora programske gomile

Vidljivo je da se svaki od unosa gomile sastoji od bloka metapodataka (na slici označeni šrafurom) i samog spremnika. Ukoliko unos sadržava neke podatke obojen je žuto, a ako označava prazan prostor, obojen je smeđe. Na slici su, dakle vidljiva tri unosa s podacima i dva unosa koja nemaju podatke. Važno je primijetiti kako svaki blok metapodataka sadrži dva pokazivača:

- pokazivač na prethodni blok i
- pokazivač na slijedeći blok.

Osim pokazivača blok metapodataka sadržava i informacije o veličini bloka kojeg opisuje, njegovom stanju (je li zauzet ili slobodan) i sl. Sve te informacije koristi programski kod namijenjen upravljanju programskom gomilom.

Napadač koji prepisivanjem spremnika programske gomile želi preuzeti nadzor nad napadnutim računalom u stvari želi prepisati metapodatke, odnosno, preciznije pokazivače u metapodacima. Ti pokazivači koriste se kao parametri funkcija za upravljanje memorijom i ako se na njihovo mjesto upiše primjerice adresa početka stoga, rezultat će biti prepisivanje podataka na stogu, što se može iskoristiti za pokretanje proizvoljnog koda, odnosno preuzimanje nadzora nad ranjivim računalom.

Budući da se po memoriji ne može pisati unatrag, već samo unaprijed napadač ne može izmijeniti metapodatke prepisanog spremnika, već to može učiniti jedino s metapodacima nekog od slijedećih unosa. Zahvaljujući dinamičkoj organizaciji gomile, vrlo mu je teško predvidjeti koji će to točno biti spremnici, što mu dodatno otežava zlouporabu.

4. Tehnike zlouporabe pojave prepisivanja spremnika

U realnim primjerima zlouporabe napadači moraju svladati cijeli niz problema kako bi njihova rješenja radila pouzdano. Tu se prije svega misli na probleme:

- tzv. *NULL* bajtova u adresama,
- promjenjivosti lokacije umetnutog koda,
- razlike među različitim okruženjima te
- na različite obrambene mehanizme koji se moraju zaobići.

Kako bi se riješili ti problemi vremenom su razvijene različite tehnike zlouporabe. Danas ih je poznat čitav niz, a opisivanje svih njih prelazi granice ovog dokumenta. Zbog toga su u ovom poglavlju opisane samo dvije najznačajnije među njima.

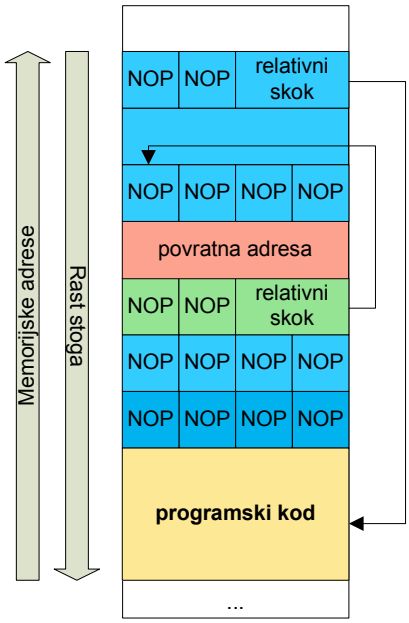
4.1. *NOP sled* tehnika

NOP sled je najpoznatija i najstarija tehnika za iskorištavanje propusta prepisivanja spremnika. *NOP sled* (od NOP *slide*) označava niz NOP instrukcija u memoriji računala, a NOP (eng. *No OPeration*) je instrukcija koja ne obavlja nikakvu radnju.

Ukoliko netko želi pokrenuti izvođenje nekog programskog koda na procesoru, on neophodno mora znati adresu od koje počinje zapis toga koda u programskoj memoriji. Ipak, ukoliko programski kod započinje nizom od, primjerice, nekoliko tisuća NOP instrukcije, tada je dovoljno samo otprilike poznavati adresu početka koda, jer će pokretanje izvođenja sa svake od tih nekoliko tisuća adresa dati jednak rezultat, odnosno rezultirat će pokretanjem zadanog koda kojem će prethoditi izvršavanje određenog broja NOP instrukcija što je ionako beznačajno.

Korištenjem ove tehnike napadač si može olakšati zlouporabu, jer više ne mora pogoditi jednu instrukciju u memoriji, već je samo bitno da pogodi u određeni raspon adresa.

Izgled programskog stoga prilikom izvođenja napada ovom tehnikom prikazan je na slijedećoj slici.



Slika 6. Izgled programskog stoga pri izvođenju *NOP sled* tehnike

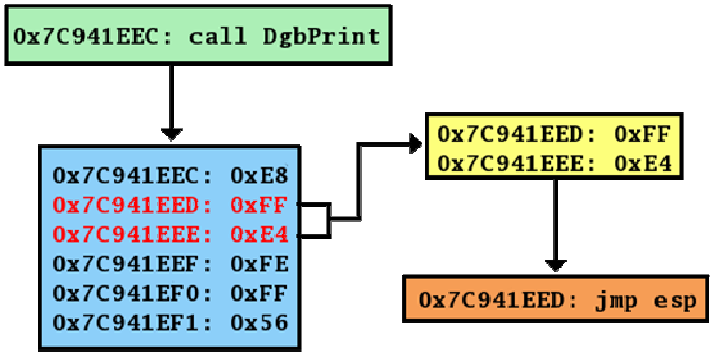
Zbog popularnosti ove tehnike mnogi su proizvođači IPS (eng. *Intrusion Prevention Systems*) sustava ugradili funkcionalnost traženja uzorka NOP instrukcija u modulima namijenjenim pronalaženju umetnutog programskog koda odnosno detekciji napada. Ovdje je važno primijetiti kako niz NOP instrukcija u stvari ne mora nužno uključivati samo NOP instrukcije. Tu može biti postavljena svaka instrukcija koja neće narušiti stanje registara za izvođenje umetnutog programskog koda. Zbog ove spoznaje sve se češće pojavljuju zlouporabe u kojima se NOP instrukcije zamjenjuju proizvoljnim nizovima instrukcija koje je daleko teže detektirati.

Iako opisana tehnika značajno povećava vjerojatnost uspjeha zlouporabe, ona ne rješava sve probleme. Korištenjem ove tehnike napadač se ipak mora osloniti na određenu dozu sreće pri pogađanju dijela memorije ispunjenog NOP instrukcijama. Pogreška u ovom pogađanju obično dovodi do rušenja napadnute aplikacije, što može skrenuti pažnju administratora na napadačeve aktivnosti. Drugi problem je potreba za relativno velikom količinom dostupne memorije koja će biti ispunjena NOP instrukcijama. Ovaj zahtjev posebno je problematičan ukoliko je stog u trenutku zlouporabe plitak, odnosno ako je udaljenost od početka stoga u memoriji do njegovog trenutnog vrha relativno mala.

Unatoč opisanim nedostacima, *NOP sled* tehnika je često jedina tehnika koja može dati očekivane rezultate neovisno o danoj situaciji, platformi ili okruženju.

4.2. Tehnika preskakivanja na registar

Ova tehnika omogućuje pouzdano iskorištavanje propusta prepisivanja spremnika pohranjenog na stogu i pri tome ne zahtjeva dodatni prostor za slijed NOP instrukcija kao ni pogađanje adrese početka umetnutog programskog koda.
Strategija se sastoji od prepisivanja povratnog pokazivača vrijednošću koja će uzrokovati skok programa na adresu pohranjenu u nekom od registara koji pokazuje na prepisani spremnik, odnosno na umetnuti programski kod. Primjerice, ako registar A sadržava adresu početka spremnika, tada se svaki skok ili poziv procedure koji kao parametar uzimaju registar A mogu iskoristiti za preuzimanje kontrole nad tokom izvođenja programa.
U praksi program uglavnom ne sadržava instrukciju za skok na određeni registar. Tradicionalno rješenje ovog problema je traženje primjerka prikladnog operacijskog koda (eng. *opcode* – binarni zapis instrukcije u memoriji) koji stoji na fiksnom mjestu unutar memorije dodijeljene programu. Na slijedećoj slici može se vidjeti primjer pronalaženja instrukcije JMP ESP.



Slika 7. Primjer pronalaženja nenamjerno ostavljene instrukcije JMP ESP kod i386 procesora

Operacijski kod instrukcije JMP ESP je 0xFFE4 (na procesoru i386). Ova dva bajta mogu se pronaći kao zadnja dva u trobajtnoj instrukciji CALL DbgPrint koja se nalazi na adresi 0x7C941EEC. Ukoliko napadač prepiše povratnu adresu s adresom 0x7C941EED, program će nastaviti s izvođenjem od te adrese. S nje će učitati instrukciju JMP ESP te će početi izvršavati programski kod koji se nalazi na vrhu stoga, tj. kod koji je umetnuo napadač.
Ranjivosti koje omogućuju primjenu ove tehnike pružaju vrlo dobru osnovu za zlouporabu i predstavljaju ogroman rizik za sustav na kome se nalaze. Ova činjenica posljedica je vrlo velike pouzdanosti zlouporabe koja vrlo malo ovisi o okolini i koja se stoga može automatizirati. Baš zbog te mogućnosti automatizacije, ovu tehniku često koriste zlonamjerno oblikovani programi poznati pod nazivom Internet crvi (eng. *Internet worms*).

5. Zaštita od prepisivanja spremnika

Kako bi se programi zaštitili od pojave prepisivanja spremnika koriste se različite tehnike koje, svaka za sebe, nose i određene nedostatke. Najpouzdaniji način zaštite je sprečavanje prepisivanja spremnika na razini programskog jezika. Ova vrsta zaštite, ipak se ne može primijeniti uvijek. Neka tehnička, poslovna ili druga ograničenja mogu uvjetovati korištenje "nesigurnog" programskog jezika. U slijedećim je poglavljima prikazan pregled mogućnosti zaštite koje se stavljaju pred programere i razvojne timove.

5.1. Izbor programskog jezika

Izbor programskog jezika ima ogroman utjecaj na mogućnost pojave prepisivanja spremnika. Iz popisa najčešće korištenih programskih jezika izdvajaju se jezik C i njegov derivat jezik C++. Velika većina današnje programske potpore napisana je upravo tim jezicima. Ova dva jezika nemaju ugrađen nikakav oblik nadzora nad pojavom prepisivanja spremnika. Čak štoviše, oni ne provode nikakav oblik kontrole nad strukturama podataka niti nad poljima čije je pravilno korištenje prepušteno volji programera.

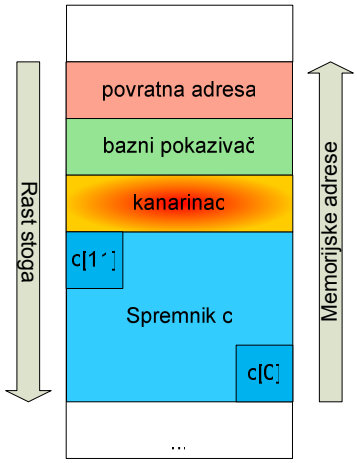
S druge strane valja primijetiti da je C++ programerima pružena mogućnost sigurnog korištenja memorijskih objekata kroz biblioteku standardnih predložaka (eng. *Standard Template Library* - STL). Sličnu mogućnost imaju i C programeri. U svakom slučaju odluka o tome je li vrijedno žrtvovati dio performansi kako bi se postila određena sigurnost ostaje na programerima.
Mnogi drugi programski jezici nude provjeru prepisivanja spremnika u realnom vremenu. Među njima se nalaze danas vrlo popularni Python, Lisp, Java, .NET i ostali. Jezici koji nude nadzor nad pojavom prepisivanja spremnika često imaju postavku koja omogućuje njeno uključivanje, odnosno isključivanje.
Dizajneri programske potpore trebaju pažljivo odvagnuti štetu lošijih performansi i dobitak veće sigurnosti te uravnotežiti te dvije veličine pažljivim odabirom programskog jezika i postavki njegovog prevoditelja.

5.2. Korištenje sigurnih biblioteka

Problem prepisivanja spremnika čest je kod programske potpore napisane programskim jezicima C, odnosno C++. Ovdje je on uzrokovan izlaganjem vrlo niske razine programske apstrakcije prema programeru koji, unatoč boljem uvidu u rad aplikacije, lako gubi nadzor nad svim aspektima toga rada. Stoga se pojava prepisivanja spremnika mora sprečavati programskim kodom koji mora imati vrlo visok stupanj točnosti.
Ovo rješenje ipak nije sasvim dobro jer zahtjeva velik trud, a sasvim mala pogreška u fazi razvoja može donijeti vrlo mnogo dodatnog posla u fazi održavanja programskog rješenje. Kako bi se to izbjeglo, dobro je pribjeći kvalitetno napisanim bibliotekama koje centraliziraju i automatiziraju proces upravljanja spremnicima.
Dva glavna mjesta na kojima se događa prepisivanje spremnika su mjesta uporabe podatkovnih polja (eng. *arrays*) i znakovnih nizova (eng. *strings*). Zbog toga uporaba "sigurnih" biblioteka može pokriti najveći dio potencijalnih sigurnosnih rupa.
Ipak, neodgovarajuće korištenje sigurnih biblioteka, jednako kao i eventualno postojanje nedostataka u samoj njihovoj implementaciji može dovesti do velikog broja nedostataka koji bi bili izbjegnuti da biblioteka nije korištena. Unatoč tim problemima, programerima uobičajenih aplikacija kojima gubitak performanse od desetak posto nije od velikog značaja, preporuča se korištenje "sigurnih" biblioteka uz temeljito proučavanje pripadne dokumentacije.

5.3. Zaštita razbijanjem stoga (eng. *Stack-smashing Protection*)

Zaštita razbijanjem stoga temelji se na izmjeni organizacije podataka pohranjenih na stogu tako da se u svaki stogovni okvir procedure (eng. *procedure stack frame*) doda po jedan ili više tzv. "kanarinaca". Kanarinac je posebna vrijednost uključena u stog koja, ukoliko je uništena, indicira pojavu prepisivanja spremnika koji se nalazi neposredno ispred nje. Na slijedećoj slici prikazan je primjer korištenja kanarinca.



Slika 8. Korištenje kanarinca u zaštiti stoga

Ukoliko se dogodi prepisivanje nekog od spremnika unutar pozvane procedure (primjerice spremnika c) i to prepisivanje obuhvati i promjenu povratne adrese na stogu, dogodit će se i prepisivanje kanarinca. Prije nego procedura završi sa svojim radom provjerava se je li vrijednost kanarinca ostala sačuvana. Ukoliko je to istina program nastavlja s normalnim radom, a u protivnom se ruši, sprečavajući tako izvođenje eventualno umetnutog programskog koda.
Termin kanarinac uzet je zbog povijesne prakse držanja kanarinaca u rudarskim oknima. Naime, kanarinci su tamo služili kao indikator nedostatka kisika, jer su na njega mnogo osjetljiviji od čovjeka.
Prema nekim izvorima tehnika primjene kanarinaca zanemarivo utječe na performanse, što je čini vrlo zanimljivom.
Postoje tri podvrste ove tehnike:

- terminirajući kanarinci (eng. *terminating canaries*),
- slučajni kanarinci (eng. *random canaries*) i
- slučajni XOR kanarinci (eng. *random XOR canaries*).

5.4. Zaštita izvršnog prostora (eng. *Executable Space Protection*)

Zaštita izvršnog prostora je oblik zaštite koji sprečava pokretanje programskog koda kako na stogu, tako i na programskoj gomili. Napadač, unatoč postojanju ove zaštite može prepisati spremnik, ali ne može pokrenuti izvođenje eventualno umetnutog koda jer će se u tom slučaju dogoditi iznimka.
Neki procesori podržavaju funkcionalnost nazvanu NX (eng. *No eXecute*), odnosno XD (eng. *eXecute Disabled*) bita koja se u suradnji s programskom potporom može koristiti za označavanje memorijskih stranica čiji se sadržaj može čitati, ali se ne može i izvršavati.
Neki Unix operacijski sustavi (primjerice OpenBSD i Mac OS X) isporučuju se s ugrađenom zaštitom izvršnog prostora. Ostalima na raspolaganju stoje opcionalni paketi poput paketa:

- PaX,
- Exec Shield ili
- Openwall.

Novije inačice Microsoft Windows operacijskog sustava također podržavaju zaštitu izvršnog prostora koja je kod njih nazvana *Data Execution Prevention*.
Kod zaštite izvršnog prostora važno je primijetiti da ona štiti samo od napada koji uključuju pokretanje programskog koda. Ostale zlouporabe, primjerice zlouporaba kojom se mijenja sadržaj neke lokalne varijable i time narušava pravilan rad programa, neće biti spriječena jer ona ne uključuje pokretanje umetnutog memorijskog sadržaja.

5.5. Raštrkavanje adresnog prostora (eng. *Address Space Layout Randomization*)

Raštrkavanje adresnog prostora (ASLR) je tehnika kojom se početak ključnih dijelova memorijskog prostora određuje slučajno. Pod ključnim dijelovima memorijskog prostora ovdje se podrazumijevaju slijedeće adrese:

- adresa početka izvršnog programskog koda,
- adresa početka programskog koda učitanih biblioteka,
- adresa početka programske gomile (eng. *heap*) i
- adresa početka programskog stoga (eng. *stack*).

Slučajnim razmještanjem ključnih elemenata napadaču postaje vrlo teško sinkronizirati pojedine korake zlouporabe. Primjerice, ukoliko uspije prepisati spremnik na programskoj gomili, mora pronaći programski stog kako bi pokrenuo njegovo izvođenje i sl. S druge strane, svaka pogreška u ovom traženju obično dovodi do rušenja napadnute aplikacije i gubitka dotad uspješno obavljenih koraka napada.

5.6. Opsežna analiza mrežnih paketa (eng. *Deep Packet Inspection* - DPI)

Analiza mrežnih paketa je oblik zaštite koji uključuje pregledavanje podatkovnog dijela dolazećeg mrežnog paketa. Prilikom pregleda posebna se pažnja posvećuje nepodudarnostima sa specifikacijama korištenog protokola kao i određenim preddefiniranim kriterijima. Ovdje je potrebno uočiti razliku o odnosu na uobičajeni pregled mrežnih paketa koji, primjerice, obavljaju vatrozidi. Kod takvih pregleda promatra se samo zaglavlje mrežnog paketa te je ova tehnika bitno jednostavnija od DPI pristupa.

Ukoliko se DPI koristi za zaštitu od pokušaja prepisivanja programskog spremnika, u dolazećim mrežnim paketima traži se uzorak koji odgovara određenom napadu. U ovoj analizi često se koristi i heuristika, a primjer uzorka koji se može tražiti u mrežnim paketima je duga sekvenca tzv. NOP instrukcija.

Ovaj oblik zaštite može štititi samo od poznatih zlouporaba i samo od onih kojima poznaje izgled poslanog mrežnog paketa. Spomenuta ograničenja bitno smanjuju mogućnost zaštite koja je još manja kada se uzme u obzir tendencija napadača ka pisanju samoizmjenjivog programskog koda, kojeg je gotovo nemoguće pouzdano raspoznati čak i uz primjenu heuristike.

6. Zaključak

Ranjivosti prepisivanja spremnika danas su najviše iskorištavane i ranjivosti koje imaju najgore posljedice na rad cijelog sustava. Rijetke su vrste ranjivosti koje je moguće iskoristiti za preuzimanje potpunog nadzora nad ranjivim računalom, a ranjivost prepisivanja spremnika, pogotovo ako je on pohranjen na stogu, je upravo takva.

Budući da se zlouporaba propusta prepisivanja spremnika temelji na iskorištavanju temeljnih značajki rada računalnog sustava, vrlo ju je teško sustavno i pouzdano spriječiti, bez da to sprečavanje donese i značajne promjene, kako u radu samog sustava, tako i u njegovim performansama.

Unatoč tome, razrađeno je nekoliko takvih strategija. Svaka od njih ima svoje nedostatke i rijetke su među njima one koje potpuno rješavaju prisutan problem. Upravo zbog toga glavna je odgovornost na dizajnerima i programerima aplikacija koji trebaju pravilno odvagnuti korist dobivenu ugrađivanjem ili implementacijom određene zaštite i gubitak u performansama odnosno konačnoj cijeni gotove aplikacije. Pri tom vaganju vrlo je bitno poznavati parametre poput okoline u kojoj se aplikacija pokreće, korisnika koji će ju rabiti i osjetljivosti zadatka kojem je namijenjena, a od svega toga još je važnije osloboditi se toliko čestih predrasuda (poput "to ne zna nitko osim mene" predrasude) koje su uzrok velikoj većini svih današnjih sigurnosnih propusta.

7. Reference

[1] Prepisivanje spremnika , http://en.wikipedia.org/wiki/Buffer_overflow , kolovoz 2007.

[2] Deckard J. Defeating Overflow Attacks, 2004.

[3] Next Generation Security Software: Buffer Overflows for Beginners, 2004.

[4] Kay R., Prepisivanje spremnika, http://www.computerworld.com/securitytopics/security/story/0,10801,82920,00.html, kolovoz 2007.

[5] SPI Dynamics: Buffer Overflows in Ten Easy Steps , 2006.