SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ARHITEKTURA RAČUNALA 2 - SEMINAR

Napadi temeljeni na preljevu meduspremnika i metode zaštite

Katja Malvoni
Voditelj: *Prof. dr. sc. Siniša Šegvić*

Zagreb, siječanj 2012.

SADRŽAJ

1. Uvod

Prvi poznati napad temeljen na preljevu međuspremnika dogodio se 1988. godine (Morris worm). Od tada se dogodilo nekoliko značajnih napada i dosta onih manje značajnih. Najznačajniji su crvi Code red (2001.), Blaster (2003.) i Sasser (2004.). Karakteristika navedenih napada bila je u brzom širenju i Denial of Service napadu. S obzirom da je korištenjem napada temeljenog na preljevu međuspremnika moguće dobiti kontrolu nad zaraženim računalom počeo je razvoj zaštite od takvih napada. Hardverska zaštita je nx bit, a mnogi operacijski sustavi podržavaju i softversku simulaciju nx bita. Kroz ovaj rad bit će objašnjeni osnovni principi napada temeljenih na preljevu međuspremnika i kratki opisi zaštita od takvih napada. Uz objašnjenje pojma nx bit, dan je i pregled obrane od napada temeljenih na preljevu međuspremnika po operacijskim sustavima, a oni najzanimljiviji i najviše korišteni su detaljnije objašnjeni.

2. Buffer overflow attacks

Izraz buffer overflow odnosi se na pokušaj programa da zapiše podatke izvan granice međuspremnika i s time prepiše potencijalno korisne podatke. Napadi temeljeni na preljevu međuspremnika iskorištavaju programe koji koriste međuspremnike, a ne provjeravaju odgovara li veličina podataka koja se sprema u međuspremnik veličini međuspremnika. Napadi se mogu dogoditi na stogu (stack overflow) ili na gomili (heap overflow). Primjeri u nastavku su vezani uz preljeve međuspremnika na stogu. Napadi se temelje na ubacivanju koda (asemblerskih instrukcija) u međuspremnik, a izvođenjem tog koda pokušava se preusmjeriti tok programa i dobiti potpuna kontrola nad zaraženim računalom. Kod koji se ubacuje naziva se shellcode zato što izvođenje tog koda pokreće komandnu liniju (command shell) i daje napadaču pristup komandnoj liniji udaljenog računala. Dijelovi koda koji su izloženi napadima s preljevom međuspremnika su polja, formatirani ispisi (funkcije: printf(), fprintf() i sprintf ()), funkcije koje čitaju iz međuspremnika (sscanf(), vscanf() i fscanf()) i funkcije koje barataju sa stringovima (gets(), puts(), strcpy() i strcat()).

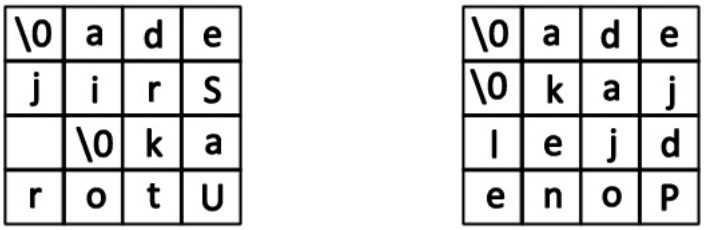
U jeziku C, primjer ugrađenog spremnika je polje. Za preljev međuspremnika je pogodno polje znakova (char array). Ako se u polje zapisuje string:

```
char p[]="String";
```

primjećuje se da duljina polja nije unaprijed poznata. Polje p određeno je s pokazivačem na prvi bajt u polju, a kraj polja označen je s \0 (null byte). Osim nepoznate veličine polja, problem predstavlja i činjenica da C nema ugrađene funkcije koje bi provjeravale da li podaci koji se pokušavaju upisati u polje, stanu u to polje. Zbog navedenoga, lako se može dogoditi sljedeći jednostavan primjer preljeva međuspremnika:

```
int main()
{
  char a[]="Utorak";
  char b[]="Srijeda";
  strcpy(b, "Ponedjeljak");
  printf("%s\n", a);
  return 0;
}
```

Riječ Utorak sastoji se od 6 znakova. Duljina polja a je 7 znakova zbog toga što svaki string završava s \0. Analogno, duljina polja b je 8 znakova. Funkcija strcpy() ne provjerava odgovara li duljina niza koji se kopira duljini polja u koji se kopira. Zbog toga se dogodi da Ponedjeljak izađe iz memorijskog prostora koji je predviđen za polje b i prepiše dio polja a, kao što je prikazano na Slici 2.1. Ispis dobiven nakon izvođenja



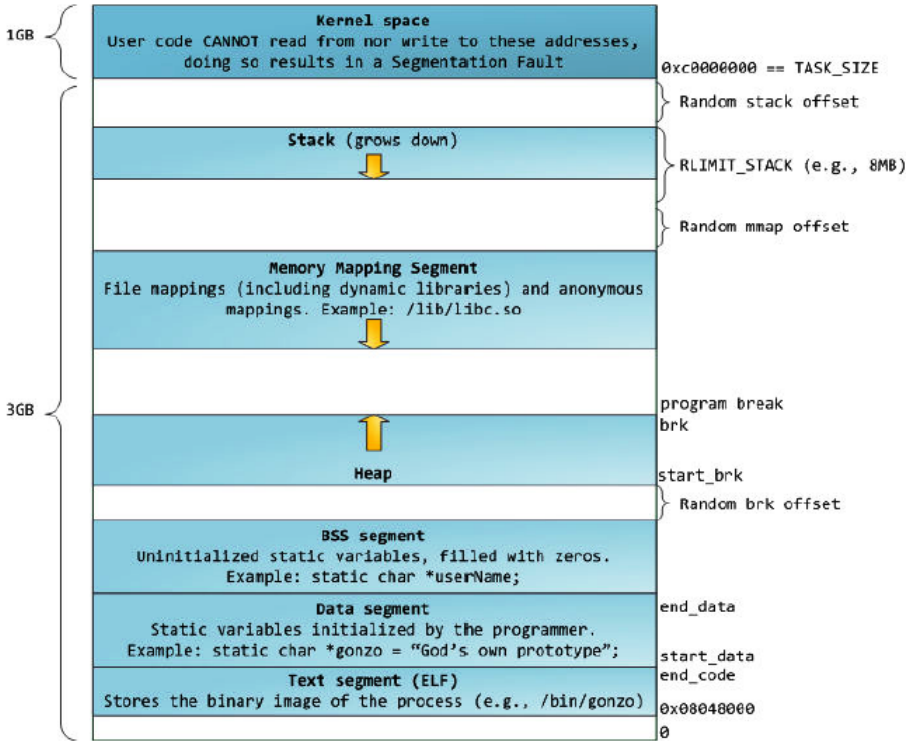
Slika 2.1: Preljev međuspremnika

programa je: jak. U ovom primjeru, preljev međuspremnika se dogodio na stogu.

Nešto složeniji primjer jest BO koji se dogodi prilikom pozivanja funkcije koja kao argument prima polje. I ovaj preljev međuspremnika se dogodi na stogu.

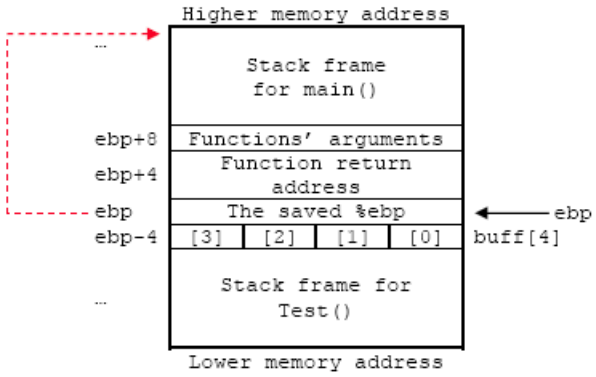
```
void Test()
{
  char buff[4];
  printf("Some input: ");
  gets(buff);
  puts(buff);
}
int main(int argc, char *argv[ ])
{
  Test();
  return 0;
}
```

Prije opisa samog preljeva međuspremnika i njegovih posljedica, pogledajmo kako izgleda program nakon što se učita u memoriju (Slika 2.2). U dijelu označenom s



Slika 2.2: Položaj programa u memoriji

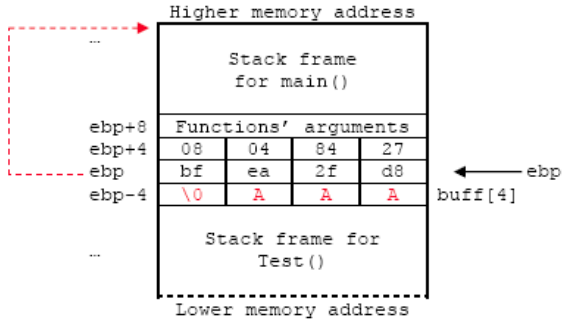
Text nalazi se prevedeni binarni program. Data segment sadrži statičke varijable čija veličina i adrese su unaprijed poznati. Stog (stack) sadrži lokalne varijable, argumente za pozive funkcija i povratne adrese. Raste prema nižim adresama, a na zadnji podatak postavljen na stog pokazuje ESP (extended stack pointer). Gomila (heap) se odnosi na dinamički alociranu memoriju. Ovaj primjer prikazuje preljev međusprem-



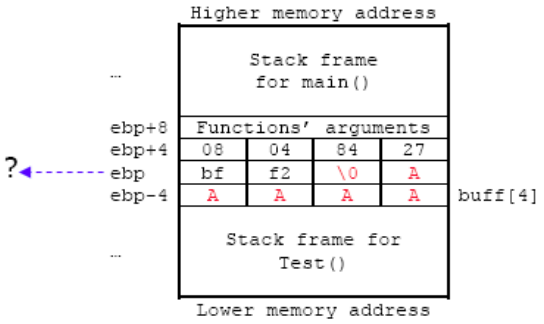
Slika 2.3: Prikaz stoga

nika na stogu, pa se slijedeća objašnjenja koncentriraju samo na stog. Prilikom pozi-

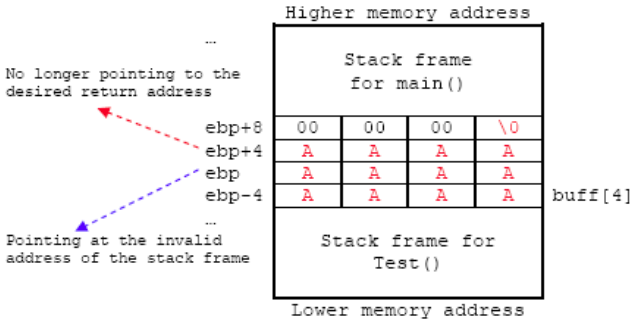
vanja funkcije, na stog se stavljaju argumenti i povratna adresa, a iznad njih, stavljaju se lokalne varijable funkcije. Položaj varijabli na stogu prikazan je na Slici 2.3. U polje buff stane 3 znaka pa će unos više od tri znaka rezultirati preljevom i pisanjem po memoriji u kojoj se nalaze neki drugi podaci. Unos više od 3 znaka rezultira prepisivanjem okvira stoga funkcije, povratne adrese i argumenata funkcije. Na sljedećem slikama (Slike 2.4, 2.5 i 2.6) prikazano je stanje stoga nakon unosa sljedećih nizova: AAA, AAAAA i AAAAAAAAAAAA. Ako unesemo dovoljno znakova, moguće je



Slika 2.4: Unos niza AAA

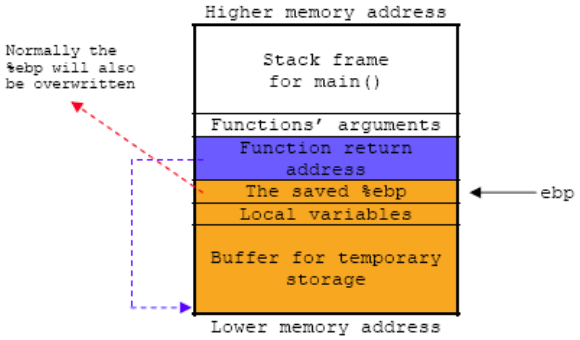


Slika 2.5: Unos niza AAAAA

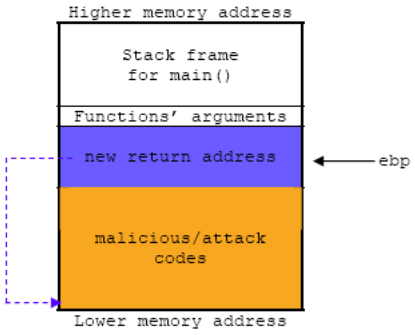


Slika 2.6: Unos niza AAAAAAAAAAAA

prepisati povratnu adresu. Cilj napadača je prepisati povratnu adresu (Slika 2.7) i preusmjeriti izvođenje programa na neki drugi dio stoga u kojem je zapisan kod (shellcode) koji preuzima računalo (Slika 2.8). Kod se može umetnuti u prostor okvira stoga funkcije, tj. u memoriju rezerviranu za lokalne varijable funkcije. Prepisivanjem povratne adrese u instrukcijski registar EIP može se staviti adresa bilo koje druge instrukcije koja se želi izvršiti. Da bi to bilo moguće potrebno je znati na kojoj točno adresi se nalazi kod koji se želi izvršiti. Postoje razne metode otkrivanja povratne adrese na kojoj se nalazi umetnuti kod. Jedna metoda sastoji se od toga da se u umetnutom kodu na početku nalaze NOP (no operation) instrukcije, a nakon njih kod koji se želi izvršiti. Takav kod ne zahtjeva točnu adresu na kojoj počinju korisne instruk-



Slika 2.7: Prepisivanje povratne adrese



Slika 2.8: Umetanje koda

cije. Adresu je moguće otprilike izračunati ispitivanjem vlastitog operacijskog sustava. Naime, stog na istom operacijskom sustavu se nalazi na približno jednakim adresama. Potrebno je jednom prepisati povratnu adresu s neupotrebljivom vrijednosti. Prilikom izvođenja takvog programa dogodit će se segmentation fault. Ako analiziramo taj program koristeći gdb, moguće je dobiti adresu zapisanu u esp. Na približno toj adresi će se nalaziti umetnuti kod. Zbog toga što se ovom metodom ne može sa stopostotnom sigurnošću odrediti adresa umetnutog koda, ubacuju se NOP instrukcije koje se nalaze ispred shellodea. Ako znamo točnu adresu na kojoj se nalazi umetnuti kod, moguće je spremiti tu adresu u registar (npr. edx) i izvršiti instrukciju call edx. Ta instrukcija će u registar EIP ubaciti adresu na kojoj se nalazi umetnuti kod. Još jedna metoda je pogađanje. Na Unix-u je stog gotovo uvijek na istom mjestu. Moguće je ispitivanjem svog sustava saznati adresu na kojoj je umetnuti kod. U slučaju da kod ne bude na očekivanoj adresi napad neće uspjeti. Sad kada je moguće preusmjeriti tok programa po želji javlja se još nekoliko problema. Preljev međuspremnika se događa zbog toga što funkcije koje koriste polja znakova ne provjeravaju prelaze li ulazni podaci granice. Kraj polja označen je s \0 i znakovi nakon \0 se ne uzimaju u obzir. Iz toga se vidi da umetnuti kod ne smije imati niti jedan znak \0 prije kraja, inače neće sav kod biti zapisan u memoriju. Posebnu pažnju potrebno je obratiti prilikom pohranjivanja malih cijelih brojeva u memoriju. Ako npr. pokušamo na stog staviti broj 10, čiji binarni zapis je 1010, ostalih 28 bita bit će popunjeno nulama. Pogledajmo sljedeću instrukciju:

```
push 10
```

Ovakva instrukcija će na stog pohraniti 3 bajta vrijednosti 0 i jedan bajt u kojem je zapisana vrijednost 10. Dobit će se upravo ono što se pokušava izbjeći, a to je znak \0 usred koda. Da bi se to izbjeglo potrebno je baratati s bajtovima. Instrukcija u tom slučaju izgleda ovako:

```
push byte 10
```

Kako bi se izbjegli znakovi \0 u samom kodu, čitav kod se terminira samo jednom i to tako da u jednom registru postoji adresa koja pokazuje na početak koda. Znak \0 se doda na kraj koda. Primjer za terminiranje koda duljine 14 bajtova:

```
xor eax,eax
mov byte [ebx + 14],al
```

Prva linija služi da bi se u registar eax pohranila vrijednost 0, a ebx pokazuje na početak koda. Pohranjivanjem najnižeg bajta registra eax iza umetnutog koda dodan je znak \0 na kraj niza.

Sljedeći problem vezan je uz adresiranje. Većina napada nakon preusmjeravanja koda poziva neku sistemsku funkciju, npr. execve u Linuxu. Sistemske funkcije zahtijevaju određene argumente. Argumenti su najčešće pokazivači na polja znakova. Da bi došlo do ostvarivanja sistemskog poziva, potrebno je predati ispravne adrese. Trik s kojim je to moguće koristi jmp i call instrukcije. Pogledajmo sljedeći kod (Foster (2005)):

```
global _start
_start:
    jmp short data
code:
    pop ebx
    ...
data:
    call code
    db 'Hello!'
```

Na početku programa jmp instrukcija skače na labelu data na kojoj se nalazi samo instrukcija call code. Instrukcija call radi tako da na stog pohranjuje adresu iduće instrukcije i u EIP stavlja adresu labele na koju skače (u ovom slučaju code). Pošto se nakon instrukcije call nalazi niz 'Hello!', na stog je pohranjena adresa stringa Hello! koja se sprema u registar ebx.

Osim preusmjeravanja na drugi dio stoga, moguće je preusmjeriti tok programa na sistemske biblioteke ili na neke druge funkcije koje su pogodne za napad. Takve funkcije se nalaze na fiksnim adresama i učitane su u memoriju prije implementacije bilo kakve zaštite od preljeva međuspremnika. Primjer je standardna biblioteka libc. Prototip funkcije execve je sljedeći:

```
int execve(const char *path, char *const argv[],
      char *const envp[]);
```

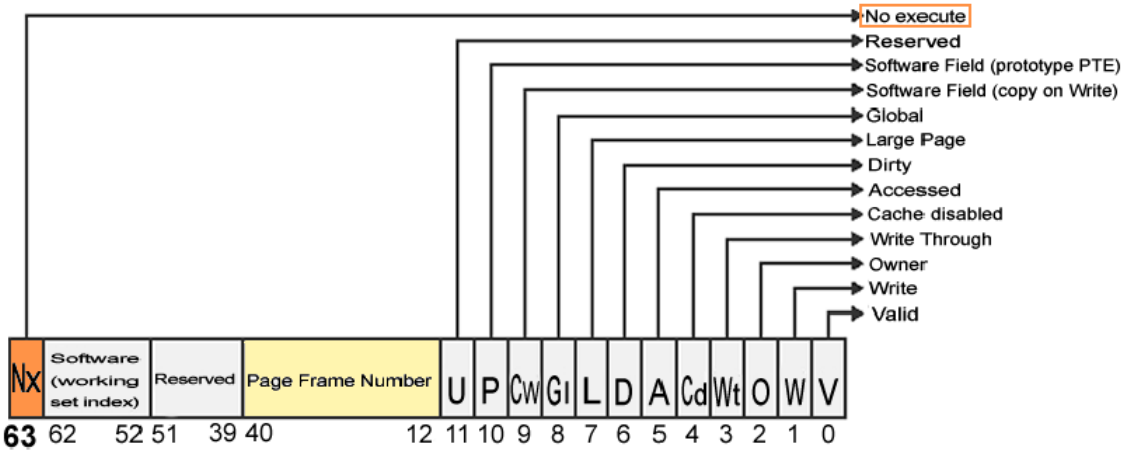
Prvi argument funkcije predstavlja putanju do izvršne datoteke koja se želi pokrenuti. Drugi argument predstavlja argumente koje želimo predati programu prilikom pokretanja. Treći argument je također polje pokazivača na stringove. Primjer umetnutog koda (Foster (2005)) koji poziva sistemsku funkciju execve i pokreće /bin/sh na Linuxu:

```
global _start
_start:
    BITS 32
    xor eax,eax
    cdq
    push eax
    push long 0x68732f6e
    push long 0x69622f2f
    mov ebx,esp
    push eax
    push ebx
    mov ecx,esp
    mov al, 0x0b
    int 0x80
```

Na početku je potrebno u registru eax dobiti nulu. Instrukcija cdq postavlja vrijednost registra edx na 0. Nije korištena instrukcija xor edx, edx jer je operacijski kod te instrukcije duljine 2 bajta, dok je operacijski kod instrukcije cdq duljine jedan bajt. Slijedi postavljanje niza '//bin/sh\0' na stog. Na početak je dodana jedna kosa crta kako bi niz bio veličine 8 bajtova. Ta duljina je praktična jer ga je na stog moguće pohraniti sa dvije push instrukcije. \0 na kraju niza dobiven je instrukcijom push eax. Pošto stog raste prema nižim adresama, string '//bin/sh' pohranjuje se obrnutim redoslijedom. Prvo se pohranjuje hs/n pa onda ib//. Nakon toga pohranjujemo esp u ebx kako bi dobili pokazivač na početak prvog argumenta. Drugi argument dobivamo s push eax i push ebx. Kao i ranije, push eax služi za oznaku kraja niza. U ecx se pohranjuje pokazivač na drugi argument. Instrukcija mov al. 0x0b u donjih 8 bitova registra eax pohranjuje vrijednost koja odgovara pozivu funkcije execve. Instrukcija int 0x80 predstavlja poziv sistemtske funkcije. Prvi argument je pohranjen u registar ebx, drugi u register ecx, a treći u registar edx.

3. Nx bit - No eXecute

Nx bit u x86 arhitekturi, odnosi se na najviši bit u jednom zapisu tablice stranica (Slika 3.1 (Argento Daniele (2004))). Ako je nx bit postavljen u 0 onda se iz stranice može izvršavati kod, ako je 1 onda se pretpostavlja da se u stranici nalaze samo podaci i onemogućeno je izvršavanje koda. Zapisi u tablici stranica kod x86 arhitekture su 32-bitni pa je očito da nemaju 63. bit. Korištenje nx bita u x86 arhitekturi omogućuje



Slika 3.1: Jedan zapis tablice stranica

PAE – Physical Adress Extension. PAE omogućuje 32-bitnim procesorima adresiranje memorijskog prostora većeg od 4 GB. Arhitektura procesora je proširena sa dodatnim adresnim linijama pa se za adresiranje koristi 36 bitova umjesto prijašnjih 32 bita. To procesoru omogućuje adresiranje do 64 GB adresnog prostora. Operacijski sustav koristi tablice stranica kako bi mapirao 64 GB fizičkog prostora u 4 GB logičkog prostora. Zapisi u takvoj tablici stranica su duljine 64 bita, a na najvišem mjestu se nalazi nx bit.

Nx bit sprječava dio napada temeljenih na preljevu međuspremnika. Napad koji se temelji na prepisivanju povratne adrese tako da ona pokazuje na kod koji se nalazi na stogu bit će zaustavljen. Ako se stog označi kao non executable, izvršavanje koda koji se nalazi na stogu neće biti dozvoljeno. Ovakav pristup ne sprječava napade koji

preusmjeravaju tok programa na neki već postojeći dio izvršnog koda. Tehnike koje omogućuju su RTL (Return-To-Libc) i ROP (Return-oriented Programming). RTL tehnika se temelji na preusmjeravanju toka programa na dio koda koji je već učitan u memoriju, tj. na standardne C biblioteke. Korištenjem funkcije mprotect() moguće je dio memorije u kojem se nalazi umetnuti kod označiti kao izvršnu. Osim mprotect(), moguće je preusmjeriti tok programa na funkciju execv() koja učitava program i izvršava ga. Kod ovakvih napada povratna adresa je prepisana i pokazuje na već učitani kod, dok se argumenti koji se predaju funkciji pohranjuju na stog. ROP je nešto složeniji i temelji se na vraćanju na dijelove instrukcija već učitanih u memoriju, iza kojih slijedi nova ret instrukcija. Tako je moguće dobiti različite instrukcije izvođenjem više manjih instrukcija. Nizovi instrukcija formiraju se u gadgete koji obavljaju operacije kao što su: zapisivanje točno određene 32 – bitne vrijednosti na točno određenu memorijsku lokaciju, aritmetičko - logičke operacije između vrijednosti na memorijskoj lokaciji i neke druge neposredne vrijednosti, pozivanje funkcija iz biblioteka i sl.

Iz navedenog se vidi da uporaba nx bita ne sprječava sve napade temeljene na preljevu međuspremnika. Razlika je u tome što je s uporabom nx bita iskorištavanje preljeva međuspremnika postalo nešto teže. Da bi se spriječilo i ostale vrste napada temeljenih na preljevu međuspremnika, potrebno je kombinirati još neke metode zaštite uz nx bit. Neke od metoda su Stack cookie i ASLR (Address Space Layout Randomization). Stack cookie se temelji na umetanju slučajne vrijednosti između povratne adrese i lokalnih varijabli. Prije izvođenja instrukcije ret i pohranjivanja povratne adrese u EIP, provjerava se slučajna vrijednost zapisana na stog. Ako je vrijednost pročitana sa stoga jednaka vrijednosti koja je bila zapisana, izvođenje programa se nastavlja. Ako nije tako, riječ je o preljevu međuspremnika i izvođenje programa se zaustavlja. Uspješnost napada temeljenih na preljevu međuspremnika se temelji na poznavanju adresa na kojima se nalazi stog i adrese na kojoj se nalazi standardna biblioteka. ASLR postavlja adrese stoga i standardne biblioteke na slučajne vrijednosti pa nije moguće znati na kojim adresama će se nalaziti. Iako se čini da ASLR sprječava napade temeljene na preljevu međuspremnika, to nije uvijek tako. Postavljanje standardne biblioteke na slučajnu adresu postaje beskorisno ako napadač može izvesti napad više puta. Naime, standardne biblioteke nalaze se u određenom dijelu memorije i moguće ih je učitati samo u taj dio. Ako napadač pokuša dovoljno puta, može naći adrese koje mu trebaju i tako izvesti napad temeljen na preljevanju međuspremnika.

4. Izvedbe

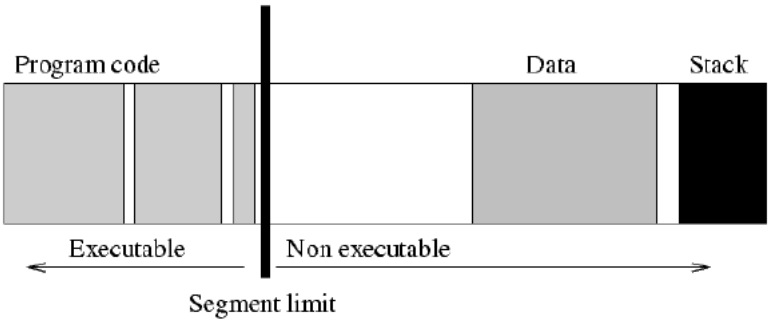
Izvedba zaštite pomoću nx bita može biti hardverska ili softverska. Hardverska izvedba oslanja se na prisutnost nx bita u procesoru. Softverska izvedba emulira nx bit. Izvedbe se razlikuju ovisno o operacijskom sustavu. U nastavku slijedi pregled izvedbi s njihovim dobrim i lošim stranama.

4.1. Linux

Linux podržava nx bit na x86-64 procesorima i na x86 procesorima koji imaju implementiran nx bit. Najpoznatije tehnologije koje omogućavaju zaštitu s nx bitom su Exec Shield i PaX.

4.1.1. Exec Shield

Exec Shield je sigurnosna zakrpa za linux jezgru koja emulira nx bit na x86 procesorima koji nemaju implementiran nx bit. Emulacija nx bita izvedena je podjelom memorije na dva dijela, izvršni dio, i dio iz kojeg nije moguće izvršavati kod. Slika 4.1 (van de Ven (2004)) prikazuje tu podjelu. Niže adrese, na kojima se nalazi izvršni



Slika 4.1: Podjela memorije kod Exec Shielda

kod programa, označene su kao izvršne, dok su više adrese, na kojima su stog i gomila označene kao ne izvršne. Kod ovakve podjele operacijski sustav mora paziti da je

sav izvršni kod učitan ispod granice, a da se stog i gomila nalaze iznad granice. Ako se pokuša napad na temelju preljeva međuspremnika, i to takav da se kod umetne na stog, program će završiti izvođenje i dogodit će se segmentation fault. Osim emulacije nx bita, Exec Shield nudi i ASLR. ASLR postavlja adrese stoga, gomile i standardnih biblioteka na slučajnu vrijednost pa je jako teško pogoditi točnu povratnu adresu s kojom bi se preusmjerio tok programa. Uz to, adresa stoga je različita i za svaki proces. Exec Shield omogućuje i postojanje Position Independent Executables (PIE). PIE je tehnologija koja omogućuje programeru da se izvršni kod u memoriju učita na različitom mjestu prilikom svakog pokretanja. Ne poznavanje adrese na kojoj se nalazi izvršni kod napadaču znatno otežava izvršavanje napada. Nedostatak PIE - a je processing overhead pa su zbog toga samo neke standardne biblioteke kompajliranje kao PIE.

4.1.2. Pax

Pax se pojavljuje 2000. godine i ima sličnu funkcionalnost kao i Exec Shield. PaX također označava određene dijelove memorije kao ne izvršne i randomizira položaj stoga, gomile i standardnih biblioteka. Pax može emulirati nx bit ili koristiti hardversku podršku za nx bit, ako takva postoji. Detaljniji opis ove tehnologije bit će dan u nastavku.

4.2. Ostali UNIX-i

4.2.1. OpenBSD

Tehnologija koja omogućuje označavanje dijelova memorije kao ne izvršnih na operacijskom sustavu OpenBSD naziva se Write XOR Execute. Naziv se odnosi na to da se u jednu stranicu u memoriji može zapisivati podatke ili se iz nje može izvršavati kod, ali ne oboje. S ovakvim načinom zaštite Write XOR Execute sprječava napade temeljene na preljevu međuspremnika koji se događaju na stogu tako da stog označi kao dio memorije iz koje se ne može izvršavati kod. To ne sprječava RTL i ROP napade. Write XOR Execute podržava samo hardverski nx bit. U slučaju da procesor nema nx bit, zaštita je izvedena tako da je samo dio adresnog prostora označen kao ne izvršni da bi postojala barem nekakva zaštita.

4.2.2. Solaris

Solaris dosta dugo podržava onemogućavanje izvođenja koda sa stoga na procesorima SPARC. Podrška za nx bit je dodana nešto kasnije. Ako procesor ima nx bit, korištenje nx bita je automatski omogućeno.

4.3. Windows

Na operacijskom sustavu Windows podrška za NX bit pojavljuje se sa Windows XP Service Pack 2. Podrška za NX bit implementirana je za x86 procesore i zove se Data Execution Prevention (DEP). Ranije verzije imale su podršku samo za NX bit, dok je podrška za ASLR implementirana kasnije. Na Windowsima postoje 4 načina rada DEP-a, a to su OptIn, OptOut, AlwaysOn i AlwaysOff. OptIn je podrazumijevani način rada u kojem je DEP omogućen samo za određene Windows aplikacije. U koliko neka druga aplikacija želi koristiti DEP mora to zahtijevati od operacijskog sustava. Iznimka su 64-bitne verzije Windowsa kod kojih sve 64-bitne aplikacija koriste DEP. U načinu rada OptOut sve aplikacije koriste DEP. U slučaju da se želi koristiti neka aplikacija koja ne podržava DEP potrebno ju je dodati na listu aplikacija koje ne koriste DEP. U načinima rada AlwaysOn i AlwaysOff DEP je omogućen ili onemogućen za sve aplikacije bez iznimaka.

5. PaX

Temeljne funkcionalnosti PaX - a već su navedene. Slijede opisi funkcionalnosti koje PaX nudi.

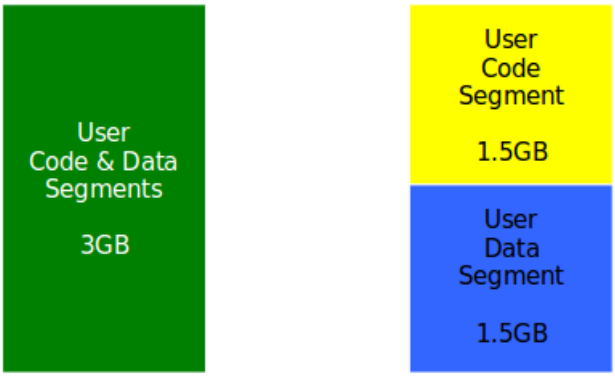
5.1. PAGEEXEC

PAGEEXEC omogućava da se iz određenog dijela memorije ne može izvršavati kod. Ako procesor ima implementiran nx bit, on se koristi, a ako nema, onda se emulira. Emulacija nx bita smanjuje performanse sustava, dok u slučaju hardverske implementacije nema utjecaja na performanse. Emulacija nx bita izvedena je korištenjem bita Supervisor. ako prilikom izvođenja programa dođe do pristupa stranici koja se ne nalazi u translacijskom spremniku (TLB) dogodit će se povreda prava. Današnji procesori koriste dva translacijska spremnika, instrukcijski (ITLB) i podatkovni (DTLB). Operacijski sustav provjerava u koji translacijski spremnik se pokušava učitati stranica. Ako se pokuša učitati u instrukcijski, doći će do pogreške i proces koji je pokušao pristupiti stranici koja se ne nalazi u instrukcijskom translacijskom spremniku bit će terminiran. Ako se radi o pristupu stranici koja bi se trebala učitati u DTLB omogući se učitavanje i proces se nastavlja normalno izvoditi. Iz prethodno opisane situacije vidi se loš utjecaj na performanse računala. Kod svakog promašaja koji se dogodi, operacijski sustav mora provjeravati da li proces pokušava izvršiti kod sa stranice koja se ne nalazi u ITLB.

5.2. SEGMEXEC

SEGMEXEC emulira nx bit na način da virtualni adresni prostor dijeli na dva jednaka dijela. SEGMEXEC je namijenjen 32 - bitnim procesorima, pa je veličina virtualnog prostora 3 GB. Donja polovica (0 - 1,5 GB) je podatkovni segment, dok je gornja polovica (1,5 - 3 GB) namijenjena instrukcijama. Ovakva implementacija temelji se

na zrcaljenju adresa na kojima se nalaze instrukcije. Podaci i instrukcije se pohranjuju u rasponu od 0 - 1,5 GB.



Slika 5.1: SEGMEXEC - podijela memorije

Instrukcije iz donje polovice se zrcale u gornju polovicu. Ako proces pokuša izvršiti instrukciju koja se nalazi u podatkovnom segmentu, a koja nije zrcaljena u instrukcijski segment, doći će do pogreške i proces će biti terminiran. Nedostatak SEGMEXEC - a je u podijeli memorije. Zbog takve podijele, programu je dostupno samo 1,5 GB. Prednost je u manjem utjecaju na performanse računala.

Osim navedenih emulacija nx bita, PaX koristi već spomenuti ASLR. Podržana je randomizacija stoga, gomile i standardnih biblioteka. Osim toga, PaX podržava mogućnost mapiranja koda koji nije PIC bilo gdje u RAM - u. Ovakav pristup loše utječe na performanse sustava. Osim toga, može izazvati i lažne uzbune pa PaX terminira procese iako nisu pokušali izvesti napad.

6. Zaključak

Napadi temeljeni na preljevu međuspremnika čine znatno manji postotak napada nego što je bio slučaj prije desetak godina. Opisane metode zaštite nisu u potpunosti zaustavile napade, ali su ih otežale. Jednostavne metode poput umetanja koda na stog ili prepisivanja povratne adrese više ne uspijevaju, ali zato napadi koji koriste ROP su i dalje opasnost. Na windowsima je moguće zaobići DEP i ASLR koristeći ROP. Isto tako, prijetnja su napadi koji se događaju na gomili. Očito je da razvoj metoda zaštite nije završen.

7. Literatura

Del Basso Luca Argento Daniele, Boschi Patrizio. *NX bit - A hardware-enforced BOF protection*, 2004. URL `http://ivanlef0u.fr/repo/todo/NX-bit.pdf`.

James C. Foster. *Buffer Overflow Attacks: detect, exploit, prevent*. Syngress Publishing, Inc., 2005.

Vincent Glaume Pierre-Alain Fayolle. *A Buffer Overflow Study Attacks & Defenses*. ENSEIRB - Networks and Distributed Systems, 2002. URL `http://www.shell-storm.org/papers/files/539.pdf`.

Arjan van de Ven. *New Security Enhancements in Red Hat Enterprise Linux v.3, update 3*. RedHat, 2004. URL `http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`.

`http://en.wikipedia.org/wiki/Buffer_overflow`
`http://en.wikipedia.org/wiki/NX_bit`
`http://en.wikipedia.org/wiki/Exec_Shield`
`http://en.wikipedia.org/wiki/Data_Execution_Prevention`
`http://en.wikipedia.org/wiki/PaX`
`http://pageexec.virtualave.net`
`http://pax.grsecurity.net/docs/`
`http://www.tenouk.com/Bufferoverflowc/Bufferoverflow4.html`