

## Laboratório 2 – Felipe Melo – Thalles Nonato

DRE Felipe: 119093752

DRE Thalles: 119058809

### Função Inicializa:

Antes de chamar as funções de cada bomba, a função main chama uma outra, chamada inicializa. Ela simplesmente espera um argumento a ser passado após o ./bomb (primeiraSenha, segundaSenha...). Caso nenhum argumento seja passado, ela detona a bomba, colocando puts("Explodiu!") e fecha o arquivo em seguida com exit. Isso também se aplica para o comando run no GDB, que usamos após um breakpoint para poder explorar melhor o código.

```
--Type <RET> for more, q to quit, c to continue without paging--
0x000012f2 <+151>:  sub    $0xc,%esp
0x000012f5 <+154>:  lea     -0x1fb8(%ebx),%eax
0x000012fb <+160>:  push   %eax
0x000012fc <+161>:  call   0x10a0 <puts@plt>
0x00001301 <+166>:  add     $0x10,%esp
0x00001304 <+169>:  sub     $0xc,%esp
0x00001307 <+172>:  pushl   -0x810(%ebp)
0x0000130d <+178>:  call   0x1060 <fclose@plt>
0x00001312 <+183>:  add     $0x10,%esp
0x00001315 <+186>:  sub     $0xc,%esp
0x00001318 <+189>:  push    $0x1
0x0000131a <+191>:  call   0x10b0 <exit@plt>
0x0000131f <+196>:  call   0x1820 <__stack_chk_fail_local>
0x00001324 <+201>:  mov     -0x4(%ebp),%ebx
0x00001327 <+204>:  leave
0x00001328 <+205>:  ret
End of assembler dump.
```

Acima está o trecho do código responsável por fechar o arquivo caso não digitemos uma entrada.

## Primeira Bomba:

O primeiro desafio foi o mais fácil dos quatro. Ao digitarmos o comando `disas primeiraBomba`, logo nos atentamos para a grande quantidade de instruções `cmp` que o código tinha. Os operandos dessas instruções eram sempre um número em hexadecimal comparado com o byte menos significativo do registrador `%eax`, que é `%al`. Tivemos a ideia de verificar se esses hexadecimais tinham códigos correspondentes na tabela ASCII (e tinham). Depois disso só precisamos verificar o restante dos códigos na tabela para decifrar a primeira senha

`Mab - 353 - 2020.2` ou `mab - 353 - 2020.2`

Inicialmente, há uma verificação se estamos digitando o caractere correspondente a `0x4D`, que é a letra “M”. Caso sim, ocorre um desvio, que salta a verificação da letra “m” (`0x6D`) direto para a da letra “a” (`0x61`). A partir daí, em todas as verificações há um `jne 0x144C <primeiraBomba+291>`, que detona a bomba chamando a função boom caso o caractere lido seja diferente do esperado.

```
(gdb) disas primeiraBomba
Dump of assembler code for function primeiraBomba:
0x00001329 <+0>:    push    %ebp
0x0000132a <+1>:    mov     %esp,%ebp
0x0000132c <+3>:    push    %ebx
0x0000132d <+4>:    sub     $0x4,%esp
0x00001330 <+7>:    call    0x1130 <__x86.get_pc_thunk.bx>
0x00001335 <+12>:   add     $0x2ccb,%ebx
0x0000133b <+18>:   mov     0x8(%ebp),%eax
0x0000133e <+21>:   movzbl (%eax),%eax
0x00001341 <+24>:   cmp     $0x4d,%al
0x00001343 <+26>:   je      0x1353 <primeiraBomba+42>
0x00001345 <+28>:   mov     0x8(%ebp),%eax
0x00001348 <+31>:   movzbl (%eax),%eax
0x0000134b <+34>:   cmp     $0x6d,%al
0x0000134d <+36>:   jne     0x144c <primeiraBomba+291>
0x00001353 <+42>:   mov     0x8(%ebp),%eax
0x00001356 <+45>:   add     $0x1,%eax
0x00001359 <+48>:   movzbl (%eax),%eax
0x0000135c <+51>:   cmp     $0x61,%al
0x0000135e <+53>:   jne     0x144c <primeiraBomba+291>
0x00001364 <+59>:   mov     0x8(%ebp),%eax
0x00001367 <+62>:   add     $0x2,%eax
0x0000136a <+65>:   movzbl (%eax),%eax
0x0000136d <+68>:   cmp     $0x62,%al
0x0000136f <+70>:   jne     0x144c <primeiraBomba+291>
```

Destacados na imagem acima estão, respectivamente:

- Verificação de “M” e possível salto para verificação de “a”
- Verificação de “m” e possível salto para explosão
- Verificação de “a” e possível salto para explosão
- Verificação de “b” e possível salto para explosão

## Segunda Bomba:

A segunda bomba foi consideravelmente mais difícil que a primeira. Inicialmente, notamos que não há instruções `cmp` no código e, por isso, tivemos de pensar em outra maneira para descobrir a senha. Percebemos depois, que novamente existia uma grande quantidade de instruções de mesmo nome (`movl`). Três delas destoavam do restante, pois copiavam valores diferentes de zero para posições da pilha, enquanto as outras copiavam o valor zero. Ao analisar os operandos dessas instruções, percebemos que haviam sempre pares de números em hexadecimal sendo copiados, o que mais uma vez nos fez pensar na tabela ASCII. Verificamos os caracteres correspondentes dos seguintes números:

0x20 0x62 0x61 0x4C → 0x42 0x20 0x3A 0x32 → 0x62 0x6D 0x6F

A partir disso, obtivemos a seguinte sequência de caracteres: [espaço]baL B :2 bmo

Invertendo cada bloco, chegamos a

Lab 2: Bomb,

que é exatamente a segunda senha.

Tivemos de inverter pois:

$\%eax = (\%al + 3) :: (\%al + 2) :: (\%al + 1) :: (\%al + 0)$

```
(gdb) disas segundaBomba
Dump of assembler code for function segundaBomba:
0x00001469 <+0>:      push    %ebp
0x0000146a <+1>:      mov     %esp,%ebp
0x0000146c <+3>:      push    %ebx
0x0000146d <+4>:      sub     $0x64,%esp
0x00001470 <+7>:      call   0x1130 <__x86.get_pc_thunk.bx>
0x00001475 <+12>:     add     $0x2b8b,%ebx
0x0000147b <+18>:     mov     0x8(%ebp),%eax
0x0000147e <+21>:     mov     %eax,-0x5c(%ebp)
0x00001481 <+24>:     mov     %gs:0x14,%eax
0x00001487 <+30>:     mov     %eax,-0xc(%ebp)
0x0000148a <+33>:     xor     %eax,%eax
0x0000148c <+35>:     movl    $0x2062614c,-0x4c(%ebp)
0x00001493 <+42>:     movl    $0x42203a32,-0x48(%ebp)
0x0000149a <+49>:     movl    $0x626d6f,-0x44(%ebp)
0x000014a1 <+56>:     movl    $0x0,-0x40(%ebp)
0x000014a8 <+63>:     movl    $0x0,-0x3c(%ebp)
0x000014af <+70>:     movl    $0x0,-0x38(%ebp)
0x000014b6 <+77>:     movl    $0x0,-0x34(%ebp)
0x000014bd <+84>:     movl    $0x0,-0x30(%ebp)
0x000014c4 <+91>:     movl    $0x0,-0x2c(%ebp)
0x000014cb <+98>:     movl    $0x0,-0x28(%ebp)
0x000014d2 <+105>:    movl    $0x0,-0x24(%ebp)
0x000014d9 <+112>:    movl    $0x0,-0x20(%ebp)
0x000014e0 <+119>:    movl    $0x0,-0x1c(%ebp)
0x000014e7 <+126>:    movl    $0x0,-0x18(%ebp)
0x000014ee <+133>:    movl    $0x0,-0x14(%ebp)
0x000014f5 <+140>:    movl    $0x0,-0x10(%ebp)
```

## Terceira Bomba:

Nessa bomba notamos que haviam diversas funções `strcmp` no código. Com isso, nosso raciocínio partiu do princípio de que o a última delas é que faria a comparação se o que digitamos é equivalente à senha esperada. Por isso, utilizamos o comando `break *0x565565C3` (endereço da instrução imediatamente anterior ao último `strcmp`) para criar um breakpoint nessa instrução. Com isso, imprimimos o conteúdo de `%eax` para descobrir a senha.

```
Dump of assembler code for function terceiraBomba:
0x56556540 <+0>:  push    %ebp
0x56556541 <+1>:  mov     %esp,%ebp
0x56556543 <+3>:  push    %ebx
0x56556544 <+4>:  sub     $0x14,%esp
0x56556547 <+7>:  call    0x56556130 <_x86.get_pc_thunk.bx>
0x5655654c <+12>: add     $0x2ab4,%ebx
0x56556552 <+18>: lea     -0x1f00(%ebx),%eax
0x56556558 <+24>: mov     %eax,-0xc(%ebp)
0x5655655b <+27>: sub     $0x8,%esp
0x5655655e <+30>: pushl   0x8(%ebp)
0x56556561 <+33>: pushl   -0xc(%ebp)
0x56556564 <+36>: call    0x56556040 <strcmp@plt>
0x56556569 <+41>: add     $0x10,%esp
0x5655656c <+44>: test    %eax,%eax
0x5655656e <+46>: jne     0x56556577 <terceiraBomba+55>
0x56556570 <+48>: call    0x5655622d <boom>
0x56556575 <+53>: jmp     0x565565e9 <terceiraBomba+169>
0x56556577 <+55>: sub     $0x8,%esp
0x5655657a <+58>: pushl   0x8(%ebp)
0x5655657d <+61>: lea     -0x1efa(%ebx),%eax
0x56556583 <+67>: push    %eax
0x56556584 <+68>: call    0x56556040 <strcmp@plt>
0x56556589 <+73>: add     $0x10,%esp
0x5655658c <+76>: test    %eax,%eax
0x5655658e <+78>: jne     0x56556597 <terceiraBomba+87>
0x56556590 <+80>: call    0x5655622d <boom>
0x56556595 <+85>: jmp     0x565565e9 <terceiraBomba+169>
0x56556597 <+87>: sub     $0x8,%esp
0x5655659a <+90>: pushl   0x8(%ebp)
0x5655659d <+93>: lea     -0x1eec(%ebx),%eax
0x565565a3 <+99>: push    %eax
0x565565a4 <+100>: call    0x56556040 <strcmp@plt>
0x565565a9 <+105>: add     $0x10,%esp
0x565565ac <+108>: test    %eax,%eax
0x565565ae <+110>: jne     0x565565b7 <terceiraBomba+119>
0x565565b0 <+112>: call    0x5655622d <boom>
0x565565b5 <+117>: jmp     0x565565e9 <terceiraBomba+169>
0x565565b7 <+119>: sub     $0x8,%esp
0x565565ba <+122>: pushl   0x8(%ebp)
0x565565bd <+125>: lea     -0x1edc(%ebx),%eax
=> 0x565565c3 <+131>: push    %eax
--Type <RET> for more, q to quit, c to continue without paging--
0x565565c4 <+132>: call    0x56556040 <strcmp@plt>
```

Em destaque na imagem acima está a instrução na qual queremos visualizar o conteúdo. Após isso, digitamos

```
run 'Mab - 353 - 2020.2' 'Lab 2: Bomb'c
```

e em seguida

```
print /x *(int *) ($eax)
```

para de fato visualizar os conteúdos em hexadecimal desejados.

```

0x565565c3 in terceiraBomba ()
(gdb) print /x *(int *) ($eax)
$3 = 0x676f7250
(gdb) print /x *(int *) ($eax+4)
$4 = 0x616d6172
(gdb) print /x *(int *) ($eax+8)
$5 = 0xa3c3a7c3
(gdb) print /x *(int *) ($eax+12)
$6 = 0x6f43206f
(gdb) print /x *(int *) ($eax+16)
$7 = 0x7475706d
(gdb) print /x *(int *) ($eax+20)
$8 = 0x726f6461
(gdb) print /x *(int *) ($eax+24)
$9 = 0x24097365
(gdb) print /x *(int *) ($eax+28)
$10 = 0x21

```

Traduzindo 0x67, 0x6F, 0x72, 0x50, 0x61, 0x6D e assim por diante, até 0x21 para caracteres da tabela ASCII, percebemos que alguns códigos faziam parte da tabela estendida. Então obtemos:

gorP amar oãç [espaço]oC tupm roda \$[tab]se !

Paramos em (%eax + 28) pois o conteúdo de (%eax + 32) deixava de fazer sentido. Além disso, pela mesma razão da segunda bomba, ao inverter, chegamos à senha:

**Programação Computadores\t\$!**

Vale ressaltar que para o computador entender \t como um tab horizontal, devemos colocar o caractere \$ antes da senha (que fica entre aspas).

```

felipe@Felipe-VirtualBox:~/Documents/CompProg/mab353-labs/02-bomb$ ./bomb 'Mab - 353 - 2020.2' 'Lab 2: Bomb' '$Programação Computadores\t$!'
Primeira bomba defusada
Segunda bomba defusada
Terceira bomba defusada
Explodiu!
felipe@Felipe-VirtualBox:~/Documents/CompProg/mab353-labs/02-bomb$ █

```



## Quarta Bomba:

A última bomba nos deixou bastante confusos, pois a senha que descobrimos não era exatamente uma palavra ou frase que fazia sentido. O processo para descobrir a senha partiu do princípio de que o programa estava lendo algo de um arquivo, pois haviam várias funções que lidavam com manipulações de arquivos, tais como `fopen`, `fgets` e `fclose`. A partir dessa constatação, utilizamos o comando `break *0x5655665a` para criar um breakpoint imediatamente após a chamada da função `fgets` na execução do programa (uma vez que ela é responsável por “pegar” algo de um arquivo), para assim, podermos ver os conteúdos de `%eax`. Fizemos isso pois notamos uma mudança no conteúdo de `%eax` de antes da chamada para depois dela e, como ele é o registrador acumulador, suspeitamos que a senha poderia estar nele.

```
(gdb) disas quartaBomba
Dump of assembler code for function quartaBomba:
0x565565ef <+0>:    push    %ebp
0x565565f0 <+1>:    mov     %esp,%ebp
0x565565f2 <+3>:    push    %ebx
0x565565f3 <+4>:    sub     $0x824,%esp
0x565565f9 <+10>:   call    0x56556130 <__x86.get_pc_thunk.bx>
0x565565fe <+15>:   add     $0x2a02,%ebx
0x56556604 <+21>:   mov     0x8(%ebp),%eax
0x56556607 <+24>:   mov     %eax,-0x81c(%ebp)
0x5655660d <+30>:   mov     %gs:0x14,%eax
0x56556613 <+36>:   mov     %eax,-0xc(%ebp)
0x56556616 <+39>:   xor     %eax,%eax
0x56556618 <+41>:   sub     $0x8,%esp
0x5655661b <+44>:   lea     -0x1e9b(%ebx),%eax
0x56556621 <+50>:   push    %eax
0x56556622 <+51>:   lea     -0x1fe3(%ebx),%eax
0x56556628 <+57>:   push    %eax
0x56556629 <+58>:   call    0x565560e0 <fopen@plt>
0x5655662e <+63>:   add     $0x10,%esp
0x56556631 <+66>:   mov     %eax,-0x810(%ebp)
0x56556637 <+72>:   cmpl    $0x0,-0x810(%ebp)
0x5655663e <+79>:   je      0x5655667a <quartaBomba+139>
0x56556640 <+81>:   sub     $0x4,%esp
0x56556643 <+84>:   pushl   -0x810(%ebp)
0x56556649 <+90>:   push    $0x800
0x5655664e <+95>:   lea     -0x80c(%ebp),%eax
0x56556654 <+101>:  push    %eax
0x56556655 <+102>:  call    0x56556050 <fgets@plt>
0x5655665a <+107>:  add     $0x10,%esp
0x5655665d <+110>:  sub     $0xc,%esp
0x56556660 <+113>:  lea     -0x80c(%ebp),%eax
0x56556666 <+119>:  push    %eax
0x56556667 <+120>:  call    0x565560c0 <strlen@plt>
0x5655666c <+125>:  add     $0x10,%esp
```

Em destaque na imagem acima, está a instrução imediatamente após a função `fgets`, de onde podemos visualizar os conteúdos desejados. Após isso, utilizamos o comando

```
run 'Mab - 353 - 2020.2' 'Lab 2: Bomb' '$'Programação Computadores\t$!' d
```

para de fato podermos prosseguir. Depois bastou verificar os conteúdos de %eax até (%eax + 24) para descobrir os números que em ASCII nos dariam a quarta senha.

```
Breakpoint 2, 0x5655665a in quartaBomba ()
(gdb) print /x *(int *) ($eax)
$12 = 0x31395765
(gdb) print /x *(int *) ($eax+4)
$13 = 0x75554864
(gdb) print /x *(int *) ($eax+8)
$14 = 0x76556d59
(gdb) print /x *(int *) ($eax+12)
$15 = 0x346c3164
(gdb) print /x *(int *) ($eax+16)
$16 = 0x31594865
(gdb) print /x *(int *) ($eax+20)
$17 = 0x464a6a62
(gdb) print /x *(int *) ($eax+24)
$18 = 0x4b736d64
(gdb) print /x *(int *) ($eax+28)
$19 = 0xa
(gdb) print /x *(int *) ($eax+32)
$20 = 0x0
(gdb) █
```

De cima para baixo, verificamos os correspondentes na tabela para 0x31, 0x 39, 0x57, 0x65, 0x75, 0x55 e assim por diante, até 0x64 (paramos aqui pois em (%eax + 28) já não tinha uma sequência). Fazendo isso, chegamos à seguinte sequência de caracteres:

19We uUHd vUmY 4l1d 1YHe FJjb Ksmd

Novamente, invertendo cada bloco pela razão explicada na segunda bomba, finalmente obtemos a senha

eW9IdHUuYmUvd1l4eHY1bjJFdmsK

que decodificada de base64 nos leva para um maravilhoso vídeo do YouTube.

```
felipe@Felipe-VirtualBox:~/Documents/CompProg/mab353-labs/02-bomb$ ./bomb 'Mab - 353 - 2020.2' 'Lab 2: Bomb' '$Programação Computadores\t$!'
'eW9IdHUuYmUvd1l4eHY1bjJFdmsK'
Primeira bomba defusada
Segunda bomba defusada
Terceira bomba defusada
Quarta bomba defusada
Todas as bombas defusadas. Parabéns!
felipe@Felipe-VirtualBox:~/Documents/CompProg/mab353-labs/02-bomb$ █
```