

TPC8 e Guião Laboratorial

Dicas para validação da execução do trabalho

Este problema cobre uma gama variada de tópicos: quadros de funções na *stack* (*stack frames*), representação de *strings*, código ASCII, e ordenação de *bytes*. Este trabalho usa uma versão compilada sem otimização (para melhor compreender o processo de compilação), em que apenas o registo `%ebp` é salvaguardado antes de se reservar espaço para o vetor `buf`. Os valores de endereços indicados aqui poderão não corresponder aos que foram analisados nas aulas.

Há 2 pistas relevantes que deverão ser consideradas:

- (i) a sugestão dada no enunciado para testarem o código com uma *string* longa, e
(ii) a indicação do compilador de que a função `gets` é "perigosa" (por ter "cadastro criminal"...). Assim, é de desconfiar que a anomalia poderá estar no uso do espaço de memória alocado à *string* (no quadro da função `getline`, na *stack*) e na utilização da função `gets`.

2. Código desmontado e devidamente anotado (sem anotações na fase de arranque):

```
(gdb) disas getline
Dump of assembler code for function getline:
0x08048400 <getline+0>: push %ebp
0x08048401 <getline+1>: mov %esp,%ebp
0x08048403 <getline+3>: sub $0x18,%esp      # reserva espaço na stack p/ 24 bytes
0x08048406 <getline+6>: sub $0xc,%esp       # reserva espaço para mais 12 bytes
0x08048409 <getline+9>: lea 0xfffffffff8(%ebp),%eax  # %eax=buf
0x0804840c <getline+12>: push %eax          # coloca na stack argumento p/ gets
0x0804840d <getline+13>: call 0x8048304      # invoca a função gets c/ argum buf

Reparar onde o gcc decidiu colocar a string local buf, em (%ebp-8)
```

Reparar onde o gcc
decidiu colocar a *string*
local buf, em (%ebp-8)

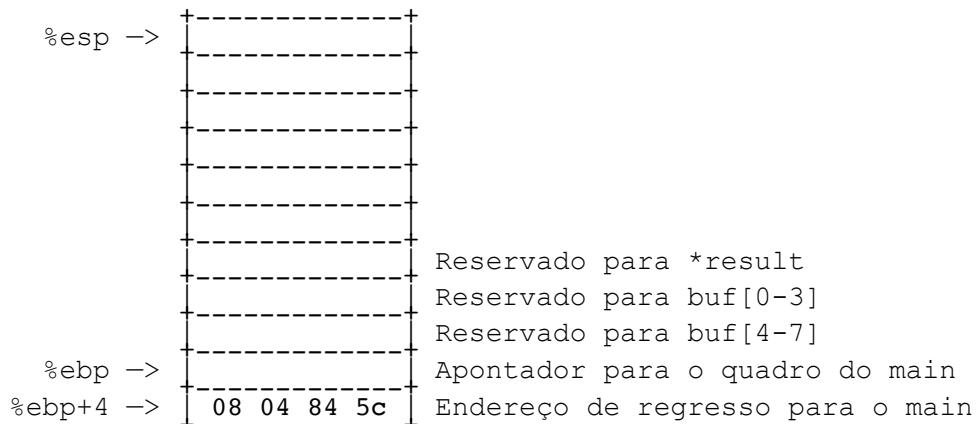
3. Análise do que aconteceu quando se mandou executar o código:

```
[docente@sc TPC8]$ ./main-ensaio  
123456789012  
  
Segmentation fault  
[docente@sc TPC8]$
```

4. Na construção do quadro de getline na stack antes de chamar a função gets (a stack cresce para cima), o endereço de regresso está no código da main: é o endereço da instrução a seguir à instrução de call getline. Veja em baixo:

```
0x08048457 <main+16>:    call   0x8048400 <getline>
0x0804845c <main+21>:    mov    %eax, 0xfffffffxfc(%ebp)
```

O valor do apontador para o quadro da `main` (o *frame pointer* do `main`) obtém-se lendo o valor em `%ebp` após uma paragem logo no início de `getline`.



5. Após paragem no *breakpoint* indicado, obterá os valores do *stack pointer* e do *frame pointer* de `getline` (estes deverão ser valores do género `0xbffffxxxx` para o código desmontado nesta resolução, pois este quadro da função deverá estar empilhado por cima do quadro do `main`). E estes são os valores que deverão ser colocados no diagrama em cima à esquerda das caixas.

```
Breakpoint 1, 0x08048409 in getline ()
(gdb) info registers
...
esp          0xbffff9054      0xbffff9054
ebp          0xbffff9078      0xbffff9078
```

E para confirmar os valores no quadro, "examinar" em hexadecimal as 2 *words* (32 bits na terminologia do `gdb`) a partir da posição apontada por `$ebp`:

```
(gdb) x/2xw $ebp
0xbffff9078: 0xbffff9088
```

6. Vamos analisar o estado do quadro de `getline` após regressar de `gets` e assumindo que foi introduzida a *string* 123456789012.

A função `gets` limita-se a ler uma linha do *standard input* até encontrar o carácter `newline` ou uma condição de erro, terminando a seguir a escrita da *string* com o carácter `null` (ASCII `0x00`); neste caso, vai ler os 12 caracteres da *string* e acrescentar o `null`; mas como apenas tem reservado para a *string* um *array* de 8 elementos, veja o que acontece...

Visualize o conteúdo das células de memória do quadro da função com o comando do `gdb x/12xw $esp`. Este comando mostra as 12 “palavras” (`w, words` – que em IA32 são 4 *bytes*, cada equivalente a uma “célula” do diagrama da *stack*) em hexadecimal (`x, hex`) a partir do endereço de memória contido no registo `%esp` (cujo valor aponta para o topo da *stack*).

34 33 32 31	buf[0-3]
38 37 36 35	buf[4-7]
32 31 30 39	Apontador para o quadro do main
08 04 84 00	Endereço de regresso para o main

Observe o quadro da função, agora apenas representado na parte mais relevante:

- os caracteres 1, 2, 3, ... estão lá claramente identificados (em ASCII, 31, 32, 33, ...) bem como o carácter `null` (em ASCII 00);
- destacam-se (com cor diferente e a *bold*) as 5 células de memória que foram indevidamente alteradas.

7. (e 8.) Analise o que aconteceu no quadro da função:

- o byte menos significativo do endereço de regresso da função foi indevidamente alterado; este programa está a tentar regressar ao endereço `0x08048400`, uma vez que o byte menos significativo **foi modificado** (*overwritten*) pelo carácter terminador (`null`);
- o valor guardado do apontador para o quadro do `main` foi modificado para `0x32313039`, e este valor será o “recuperado” para `%ebp` antes do regresso de `getline`.

Notas finais:

A chamada de `malloc` deveria ter como argumento `strlen(buf) + 1`, e deveria também verificar que o valor a devolver é *non-null*.

Para evitar este tipo de problema usar `fgets` ou `scanf` em vez de `gets` (que não é suportado desde 2011).

Se tivesse compilado com otimização `-O2`, seria preciso introduzir bastantes mais valores para ter o mesmo efeito (quantos mais, e porquê?).

Adicionalmente, todos os registos salvaguardados pela função também seriam alterados (porquê?).

Adenda

Nota 1

O processo de submissão de trabalhos TPC8 continua a ser feita com algumas incorreções, as quais ainda não irão ter penalizações. Contudo, chamamos a atenção para algumas dessas incorreções, pois já serão penalizadas no próximo TPC (ver Adenda no TPC7r):



Há, no entanto, submissões **que ignoraram por completo as regras**, nem juntaram a resolução num único ficheiro nem colocaram o nº de aluno nem o turno a que pertence! Essas submissões irão sofrer já uma penalização de 50%. Eis um desses casos:



Pior ainda, foram apresentados 3 trabalhos realizados na mesma conta do sistema remoto; esta situação foi considerada **FRAUDE** pela equipa docente e todos ficaram com classificação **-1**.

Nota 2

Continua a haver alunos que não sabem ler/interpretar o que se pede num enunciado. Eis alguns exemplos:

```
]$ gcc -o2 getline.c -o get
```

Neste caso, para além de ignorar que o enunciado pedia explicitamente

Compile o código C sem qualquer otimização (com -O0)

nem sequer sabe distinguir um comando do compilador para indicar o nome do ficheiro que vai ficar com o executável, o ficheiro de output “o”, de um comando de otimização “O”!

Mas infelizmente este não foi o único aluno que não descobriu ainda esta diferença...

Outra situação anómala ocorre quando se pede **Replique aqui tudo que apareceu no monitor**.

E a resposta que se obtém é a seguinte

A handwritten screenshot of a terminal window. The text inside the window reads: '/get\n1,2,3,4,5,6,7,8,9,0,12,13\nSegmentation fault'

totalmente incongruente com o quadro da função mostrado adiante, pois, por ex., no quadro adiante não estão lá representadas as vírgulas codificadas em ASCII nem esta sequência de algarismos.

Esta é mais uma pista indicadora de fraude: este trabalho sofreu uma penalização de 50%.

Uma outra deficiência detetada na interpretação das questões e respetivas respostas (que poderiam ser consideradas fraudulentas) é quando se pede para que estimem os valores e endereços que se encontram na *stack* (por ex., na questão 4) analisando apenas o código, **sem o executar**.

E aparecem resultados como este:

<code>0xbffffe460</code>	<code>0xbffffe464</code>	<code>0xbffffe468</code>	<code>char * result</code>	<code>buf[0-3]</code>	<code>buf[4-7]</code>	espaço para função getline
	<code>0xbffffe464 ← 0xbffffe028</code>		<code>%ebp antigo</code>			
			<code>0x08048458</code>			endereço de regresso para a main

Apenas com base na análise do código, é possível saber:

- para onde estão a apontar os registos *stack pointer* (%esp) e o *frame pointer* (%ebp) da função `getline`, mas não o conteúdo do *frame pointer* da `main` que foi salvaguardado na *stack*, assumindo que foi esta a função que invocou `getline`;
- o endereço de regresso, analisando o código gerado pelo compilador para o `main`.

Contudo, sem executar o código não é possível saber o conteúdo dos registos %esp e %ebp, nem o resto dos conteúdos no quadro da função.

Nota 3

Uma análise dos trabalhos submetidos mostra que há ainda alunos que mostram deficiências no manuseamento da *stack*, quer por falta de estudo das matérias teóricas, quer ainda por alguma leviandade na resolução dos TPCs. São de destacar:

- ignorância na utilização que o compilador faz da *stack* no âmbito do quadro da função, não distinguindo entre salvaguarda/recuperação de registos e utilização da *stack* para passar argumentos na chamada de outras funções;
- não serem capazes de localizar para onde o %esp (o registo que contém o *stack pointer*, i.e., o apontador para o topo da pilha) está a apontar em cada uma das partes do código;
- não saberem examinar os conteúdos do quadro da função na *stack* (com o comando `x`, de `examine`), preferindo usar um outro comando do `gdb` contendo menos informação útil (o `info frame`) e de leitura mais difícil.

Seguem vários exemplos de anomalias encontradas nas resoluções apresentadas, começando com uma deficiência que ainda persiste: a **anotação de código**, i.e., a introdução de comentários no código *assembly* que permitem explicar o que está a acontecer nesse código de baixo nível e sua ligação ao código fonte, normalmente em C. Eis um exemplo:

<code>lea 0xfffffff8(%r.ebp),%eax</code>	- O <code>fffffff8</code> (%.ebp) é colocado no registo %eax.
<code>push %eax</code>	- O registo %eax é colocado no topo da stack
<code>call 0x8048328</code>	- O endereço <code>0x8048328</code> é colocado no stack

Infelizmente este tipo de anotação inútil é ainda repetido em vários trabalhos submetidos; inútil por não trazer nenhuma informação adicional relativamente ao programa fonte e ao objetivo dessas instruções. Eis mais um exemplo com as mesmas instruções:

<code>(5) coloca 0xffffffff8 (%0.0sp) no resultado</code>
<code>%0.0sp</code>
<code>(6) coloca o resultado %0.004 no topo da stack</code>
<code>(7) Coloca o endereço 0x8048308 na stack</code>

Neste exemplo confunde uma reserva de espaço na *stack* com “salvaguarda de memória”!!!

`sub $0x18,%esp` → Salvaguarda memória na stack
`sub $0xc,%esp` → endereço (%ebp)-8 é guardado

O termo “salvaguarda” no contexto desta UC significa guardar com segurança para ser depois recuperado. É o que normalmente acontece quando se quer preservar o conteúdo dum registo.

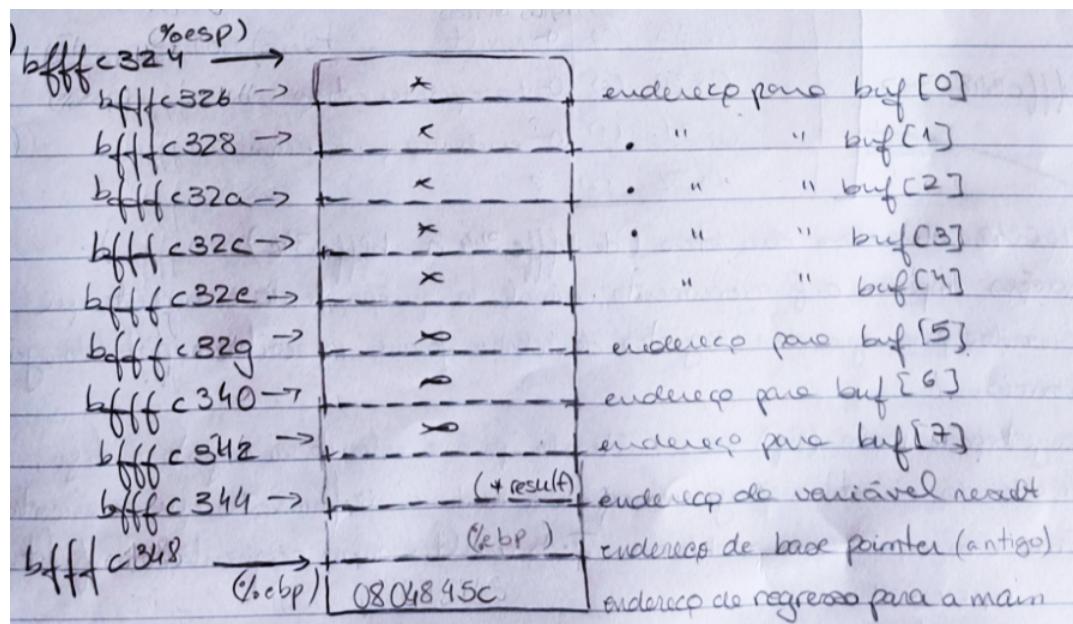
Ou então estes exemplos com anotações incorretas:

<code>mov %esp, %ebp</code>	;	reserva espaço na stack para array buf
<code>sub \$0x15, %esp</code>	;	reserva espaço na stack para result;
<code>sub \$One, %esp</code>	;	coloca o registo %eax no começo da array
<code>lea 0xFFFFFFF8(%ebp), %eax</code>	;	coloca o registo %eax no topo da stack
<code>push %eax</code>	;	chama o gots
<code>call 0x8048304</code>		

(2)		
<code>sub \$0x18, %esp</code>	;	12 (0x8) bytes mem. útil
<code>sub \$0xc, %esp</code>	;	espaço p/ buf [8]
<code>ba 0x1FFFFFF8(%ebp), %eax</code>	;	salvo, regis. %eax
<code>push %eax</code>	;	início da função gots
<code>call 0x804834e</code>		

Este caso a seguir é gritante e é mais um caso de ignorância para além de fraude na resposta à questão 6 (aqui pretendia-se avaliar a capacidade dos alunos em usar o `gdb` para examinar porções de memória e, neste caso, memória contendo o quadro da função):

- ignorância por considerar que a codificação de um carácter ocupa 2 células de memória; houve outros casos similares, em que os alunos consideraram que cada carácter ocupava 4 células de memória;
- fraude ao apresentar uma resolução como sendo a confirmação destes valores, conforme pedido nesta questão, quando de facto não usou o `gdb` para verificar o que estava armazenado na `stack`.



Um outro caso de ignorância:

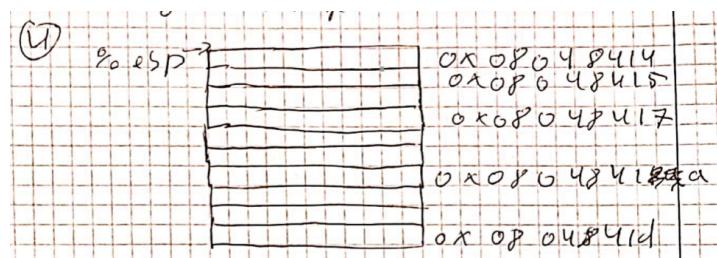
- não distingue um *array* de 4 bytes de um endereço de 4 bytes; o 1º é um conjunto de 4 valores escalares, colocados na memória ordenados pelo índice do *array*, o 2º é um único valor escalar de 32 bits, sendo este guardado na memória na ordem *little endian*;

- não considerou que a *string* é um caso particular de um *array* de caracteres, pois termina sempre com o carácter *null* ("0" em ASCII).

Adicionalmente esta resolução é também mais uma fraude na resposta à questão 6, já que não se usou o *gdb*, pela ignorância referida em cima:

0(%buf?) →	?? ?? ?? ??		}	gets escreve nos 8
2(%ebp) →	31 32 33 34 (1) (2) (3) (4)		}	bytes reservados para buf
28(%ebp) →	35 36 37 38 (5) (6) (7) (8)		}	e escreve por cima do buf? ?
2c(%ebp) →	39 30 31 32 (9) (10) (11) (12)		}	endereço de regresso para a main
	08 04 84 5e			

Mais um caso espantoso de ignorância: o quadro da função contém o código da função!!!



Agora um exemplo diferente:

~~lea 0xffffffff8(%ebp), %eax~~

Esta instrução, equivalente a `lea -0x8 (%ebp), %eax`, subtrai 8 ao valor contido no `%ebp` e coloca o resultado em `%eax`. Aqui apenas está a ser calculado um endereço e colocado no `%eax`, pelo que não existe nenhuma reserva de memória (!).

Esta instrução é precedida por dois `subl`, mas vamos considerar para esta explicação a existência apenas da 2ª subtração `subl 0xc,%esp` (a 1ª subtração tem a ver com a gestão eficiente das linhas da cache).

Esta instrução subtrai 12 ao valor contido no `%esp` (que aponta para o topo da pilha), fazendo com que o topo da pilha suba 3 "células-do-diagrama" (não esquecer que nos diagramas deste exercício cada "célula-do-diagrama" representa 4 células-de-memória ou 4 bytes), e assim reservando espaço na *stack* mas não colocando lá nada.

Esse espaço reservado, que é a área toda entre os endereços contidos em `%ebp` e `%esp`, pode agora ser utilizado pelo resto da função. A instrução neste exemplo calcula o endereço de uma posição na pilha 8 bytes acima de `%ebp` e coloca esse endereço em `%eax`.

O endereço em `%eax` é depois passado como argumento à função `gets`, indicado pelo `push %eax` imediatamente antes do `call` da `gets`, o que indica que estes 8 bytes reservados serão o espaço usado pelo `buf` e que a `gets` irá colocar a partir desse endereço a *string* que passarmos à função.

Assim podemos ver que a "reserva de memória" não é feita apenas pelo `lea`:

- os `subs` reservam efetivamente a memória necessária na *stack*;
- o `lea` calcula um endereço dentro dessa memória reservada, para o `gets` poder usar;
- esse endereço é passado como argumento para a função `gets`, e para tal é usado a instrução `push %eax`, já que os argumentos de funções são colocados na *stack* no IA-32.