Assembly do IA-32 em ambiente Linux

Exercícios de Programação 3

Resolução

Exercício 3.1 (*Transferência de controlo na invocação de funções*):

Este é mais um exemplo de uma utilização estranha do código *assembly*, também designada por um "idioma": parece estranho haver uma instrução de `call` para a linha seguinte (!) e sem a correspondente instrução de `ret` ! Só então é que nos apercebemos que afinal isto não é uma invocação de função.

a) `%eax` fica com o endereço da instrução `popl`.

b) Isto não é uma verdadeira invocação de subrotina, uma vez que o PC/IP fica a apontar para a instrução seguinte, e o endereço de retorno é explicitamente retirado da *stack*.

c) Esta é a única maneira, no IA-32, de se conseguir copiar o valor do PC/IP para um registo de inteiros.

Exercício 3.2 (*Convenção*):

Este é um problema típico relativo à convenção da utilização dos registos na invocação de funções: a responsabilidade de preservar os conteúdos dos registos `%edi`, `%esi`, e `%ebx` é da função chamada (salvaguardando-os na stack antes de os utilizar, e recuperando-os antes de regressar da função), enquanto a responsabilidade de preservar os conteúdos dos outros registos é da função chamadora (fazendo o `push` para a *stack* antes da invocação da subrotina, e o `pop` logo após o retorno).

Exercício 3.3 (*Funções*):

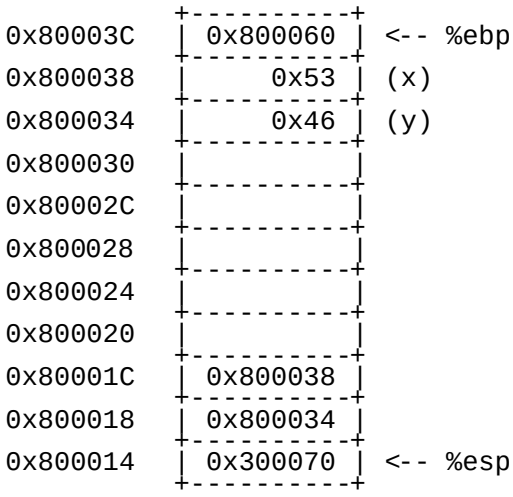
Entender o modo como as funções usam a *stack* é das componentes mais críticas para se ser capaz de ler e compreender o código gerado pelos compiladores. Reparem como, neste exemplo, o compilador reserva um área significativa na *stack*, que nunca será usada.

a) Começa-se com o valor `0x800040` em `%esp`. A linha 2 decrementa-o por 4, dando `0x80003C`, e este passa a ser o novo valor de `%ebp`.

b) Podemos ver como é que as 2 instruções `leal` calculam o valor dos argumentos a passar a `scanf`. Uma vez que os argumentos são "empurrados" para a *stack* pela ordem inversa, podemos ver que `x` se encontra deslocado de –4 células relativo a `%ebp` e que `y` está com um deslocamento de –8. Os endereços de memória são então `0x800038` e `0x800034`.

c) Começando com o valor original de `0x800040`, a linha 2 decrementou o valor de SP de 4 unidades, a linha 4 de 24, e a linha 5 de 4, e as 3 instruções de `push` decrementaram de 12,; no total houve uma variação de 44. Assim, na linha 11 `%esp` tem o valor `0x800014`.

d) A *stack frame* tem a seguinte estrutura e conteúdo:

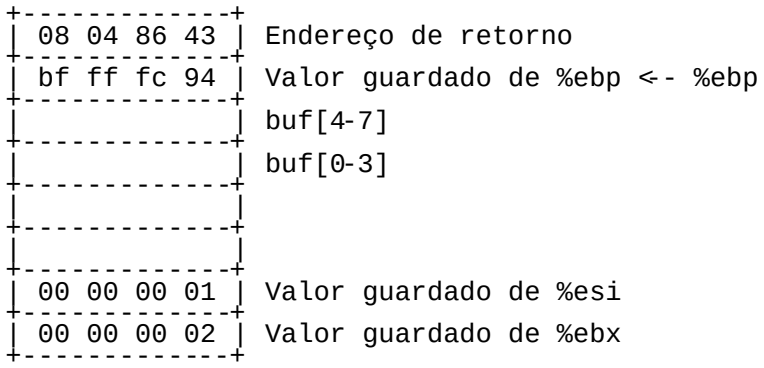


e) As células com os endereços `0x800020` a `0x800033` não são usadas.

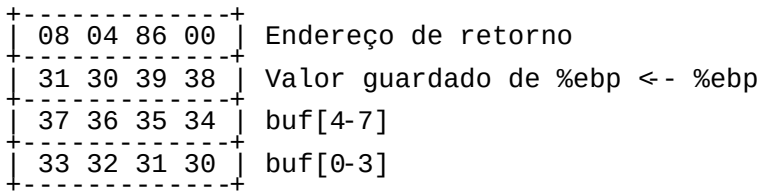
Exercício 3.4 (*Buffer overflow*):

Este problema cobre uma gama variada de tópicos: *stack frames*, representação de *strings*, código ASCII, e ordenação de *bytes*. Mostra ainda os perigos de referenciar a memória fora de limites previstos (*out-of-bounds memory references*), e as ideias básicas relativas a *buffer overflow*

a) Estado da *stack* na linha 7.



b) Estado da *stack* depois da linha 10 (mostrando apenas as palavras que foram alteradas).



c) Este programa está a tentar regressar ao endereço `0x08048600`, uma vez que o *byte* menos significativo foi modificado (*overwritten*) pelo caracter terminador (*null character*).

d) O valor guardado de `%ebp` foi modificado para `0x31303938`, e este valor será o "recuperado" para `%ebp` antes do regresso de `getline`. Os valores guardados dos outros registos não são afectados, uma vez que eles estão guardados na *stack* em endereços mais baixos que `buf`.

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