Theoretical Exercises

Exercise 1

Функция использует комбинации SCAS и STOS для работы. Сначала объясните, какого типа `[ebp+8], [ebp+C]` в строках 1 и 8. Затем объяснить, что делает этот фрагмент кода

```
01: 8B 7D 08          mov   edi, [ebp+8]
02: 8B D7             mov   edx, edi
03: 33 C0             xor   eax, eax
04: 83 C9 FF          or    ecx, 0FFFFFFFFh
05: F2 AE             repne scasb
06: 83 C1 02          add   ecx, 2
07: F7 D9             neg   ecx
08: 8A 45 0C          mov   al, [ebp+0Ch]
09: 8B FA             mov   edi, edx
10: F3 AA             rep stosb
11: 8B C2             mov   eax, edx
```

Ответ

В инструкциях `mov edi, [ebp+8]` и `mov al, [ebp+0Ch]` происходит перемещение значения из адреса/чтение по адресу `[ebp+8]` и `[ebp+0Ch]` соответственно. В случае `[ebp+8]` читается 4 байта, так как регистр 32-разрядный, а в `[ebp+0Ch]` - 1 байт, так как `al` - младший байт регистра `eax`.

После чтения четырех байт из адреса `[ebp+8]` это значение записывается в регистр edx - для сохранения и дальнейшего использования. С помощью `xor eax, eax` обнуляется значение регистра `eax`. Затем происходит логическое сложение значения регистра `ecx` и `0FFFFFFFFh` - в результате в регистре `ecx` будет находится результат этой операции - максимальное значение для размерности регистра.

`SCASB` осуществляет поиск байтов в строке, которая должна находиться в `al` - для совпадения размерности операндов - так как в eax лежит 0, то ищется символ конца строки. Адрес для поиска находится в edi - сейчас это значение по адресу `[ebp+8]`. В ecx находится количество элементов, которые нужно обработать. Префикс `repne` устанавливает поиск первого элемента, равного значению в аккумуляторе `(eax/al)`

Затем происходит увеличение ecx на 2. После чего с помощью инструкции `neg ecx` это значение умножается на -1.

В регистр al записывается значение по адресу `ebp+0Ch`, в `edi` - ранее сохраненное значение этого регистра до выполнения операции `scasb` из edx, так как при её выполнении увеличивается на размер искомого элемента.

И вызывается `rep stosb`, который ecx раз записывает в edi значение из al. Затем сохраненный адрес передается в регистр eax.

Можно предположить, что этот фрагмент кода представляет собой функцию для работы со строками. Сначала ищется нулевой элемент для определения размера строки, а затем она заполняется определенным символом, включая последний, нуль-символ конца строки.

На C\C++ функция может выглядеть следующим образом:

```
char* fill_string(char* s, char c)
{
    size_t len = strlen(s)+1;
    memset(s,c,len);
    return s;
}
```

Exercise 2

Учитывая то, что вы узнали о CALL и RET, объясните, как можно считать значение EIP? Почему нельзя просто выполнить команду MOV EAX, EIP?

Регистр eip не доступен ни по чтению, ни по записи. Так как адрес возврата загружается в стек сразу после вызова функции, можно сделать функцию, которая будет считывать значение из `[esp]` в `eax` и возвращать его.

```
get_eip: mov eax, [esp]
ret
```

Exercise 3

Придумайте как минимум две кодовые последовательности, чтобы установить EIP в значение 0xAABBCCDD.
Ответ
Как минимум, для того, чтобы поместить в EIP нужное значение, нужно поместить его на верхушку стека до момента выполнения инструкции `ret`.
В первом способе мы можем записать значение 0xAABBCCDD в `eax`, а затем записать его по адресу верхушки стека `[esp]`

```
set_eip_1:
mov eax, 0xAABBCCDD
mov [esp], eax
ret
```

Во втором способе можно просто запушить значение в стек перед выполнением `ret`

```
set_eip_2:
push 0xAABBCCDD
ret
```

Exercise 4

Что произойдет в функции addme из примера, если указатель стека не будет восстановлен должным образом перед выполнением RET?

```
int
__cdecl addme(short a, short b)
{
    return a+b;
}
```

01: 004113A0 55 push ebp
02: 004113A1 8B EC mov ebp, esp
03: ...
04: 004113BE 0F BF 45 08 movsx eax, word ptr [ebp+8]
05: 004113C2 0F BF 4D 0C movsx ecx, word ptr [ebp+0Ch]
06: 004113C6 03 C1 add eax, ecx
07: ...
08: 004113CB 8B E5 mov esp, ebp
09: 004113CD 5D pop ebp
10: 004113CE C3 retn

The function is invoked with the following code:

C

```
sum = addme(x, y);
```

Assembly

```
01: 004129F3 50              push  eax
02: ...
03: 004129F8 51              push  ecx
04: 004129F9 E8 F1 E7 FF FF  call  addme
05: 004129FE 83 C4 08        add   esp, 8
```

Ответ

Если указатель стека не будет восстановлен перед выполнением RET, то управление передастся по другому адресу - то есть значению, которое будет лежать в верхушке стека `esp` перед выполнением RET

Exercise 5

Во всех описанных соглашениях о вызовах возвращаемое значение сохраняется в 32-битном регистре (EAX). Что происходит, если возвращаемое значение не помещается в 32-битный регистр? Напишите программу, чтобы проверить свой ответ. Меняется ли этот механизм в зависимости от компилятора?

Ответ

Если возвращаемое значение не помещается в 32-битный регистр EAX, оно будет помещено в комбинированный регистр `EDX:EAX` (для 64-бит, например). Компилятор может передать указатель на значение через стек или регистр. Механизм меняется в зависимости от соглашения, компилятора, платформы.

Exercise 6

В некоторых листингах ассемблера, имя функции имеет префикс `@`, за которым идёт число. Когда и почему используется такое оформление?

Ответ

(из документации Microsoft о декорации имён) В соглашении о вызове `__stdcall` префикс с десятичным числом байтов в списке параметров, в `__fastcall` префикс и постфикс с десятичным числом байтов в списке параметров. В `__vectorcall` используется `@@` с десятичным числом байтов с писке параметров

Exercise 7

Если текущий уровень привилегий закодирован в регистре CS, который может быть изменен кодом пользовательского режима, то почему код пользовательского режима не может изменить регистр CS для изменения уровня привилегий?

Ответ

Пользовательский процесс не может изменить значение поля привилегий в дескрипторе - у него нет доступа к таким инструкциям. Кроме того, содержимое регистра может измениться как результат выполнения инструкции, управляющей потоком выполнения. Ещё уровень привилегий может измениться в случае передачи управления кодовом сегменту с помощью использования дескриптора-шлюза. Процессор ограничивает передачу управления в пределах одного кольца защиты.

Exercise 8

Прочитайте главу «Виртуальная память» в «Руководстве разработчика программного обеспечения Intel», том 3, и «Руководстве программиста архитектуры AMD64», том 2: «Системное программирование». Выполните несколько преобразований виртуальных адресов в физические и проверьте результат с помощью отладчика ядра. Объясните, как

работает защита от выполнения данных (DEP, Data Execution Prevention).

Ответ

Запустим WinDbg в режиме local kernel debugging.

Возьмем список процессов

В самом верху будет процесс System

```
PROCESS ffff890b1e664080
    SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 001ad000  ObjectTable: ffffb80077e60e40  HandleCount: 2224.
    Image: System
```

DirBase - значение CR3 при запуске процесса - адрес, который запускает преобразование виртуальных адресов в физические. CR3 - управляющий регистр в x86-64, который хранит физический адрес каталога страниц или таблицы указателей каталогов страниц

Возьмем адрес ядра

```
lkd> lm
start             end                 module name
fffff800`6ce00000 fffff800`6de46000   nt         (pdb symbols)         c:\symbols\ntkrnlmp.pdb\68A17FAF3012B7846079AEECDBE0A5831\ntkrnlmp.pdb

Unloaded modules:
fffff800`70700000 fffff800`70710000   dump_storport.sys
fffff800`70740000 fffff800`7076e000   dump_stornvme.sys
fffff800`70790000 fffff800`707ae000   dump_dumpfve.sys
fffff800`71800000 fffff800`7181c000   dam.sys
fffff800`6f8e0000 fffff800`6f8f1000   hwpolicy.sys
```

Адрес nt - fffff8006ce00000 - виртуальный адрес

CR3 = 001ad000

PLM4E = 001adf80

```
lkd> ? 001adf80 + ((fffff8006ce00000 >> 0x1e & 0x1ff) * 8)
Evaluate expression: 1761160 = 00000000`001adf88
lkd> !dq 1adf80
#  1adf80 00000000`04109063 00000000`00000000
#  1adf90 00000000`00000000 00000000`00000000
#  1adfa0 00000000`00000000 00000000`00000000
#  1adfb0 00000000`00000000 00000000`00000000
#  1adfc0 00000000`00000000 00000000`00000000
#  1adfd0 00000000`00000000 80000000`001ad063
#  1adfe0 00000000`00000000 00000000`00000000
#  1adff0 00000000`00000000 00000000`04022063
```

PDPE = 04109008

```
lkd> ? (0000000004109063 & fffffffffffff000) + ((fffff8006ce00000 >> 0x1e & 0x1ff) * 8)
Evaluate expression: 68194312 = 00000000`04109008
lkd> !dq 04109008
# 4109008 00000000`0420a063 0a000001`1b60f863
# 4109018 0a000000`0cb31863 0a000000`a7755863
# 4109028 00000000`00000000 00000000`00000000
# 4109038 00000000`00000000 00000000`00000000
# 4109048 00000000`00000000 00000000`00000000
# 4109058 00000000`00000000 00000000`00000000
# 4109068 00000000`00000000 00000000`00000000
# 4109078 00000000`00000000 00000000`00000000
```

PDE = 0420ab38

```
lkd> ? (0000000004109063 & fffffffffffff000) + ((fffff8006ce00000 >> 0x1e & 0x1ff) * 8)
Evaluate expression: 68194312 = 00000000`04109008
lkd> !dq 04109008
# 4109008 00000000`0420a063 0a000001`1b60f863
# 4109018 0a000000`0cb31863 0a000000`a7755863
# 4109028 00000000`00000000 00000000`00000000
# 4109038 00000000`00000000 00000000`00000000
# 4109048 00000000`00000000 00000000`00000000
# 4109058 00000000`00000000 00000000`00000000
# 4109068 00000000`00000000 00000000`00000000
# 4109078 00000000`00000000 00000000`00000000
```

Physical Address

Least 8 bytes, &0x000

Physical address = 2000000

Проверим

```
lkd> !vtop 0001ad000 fffff8006ce00000
Amd64VtoP: Virt fffff8006ce00000, pagedir 00000000001ad000
Amd64VtoP: PML4E 00000000001adf80
Amd64VtoP: PDPE 0000000004109008
Amd64VtoP: PDE 000000000420ab38
Amd64VtoP: Large page mapped phys 0000000002000000
Virtual address fffff8006ce00000 translates to physical address 2000000.
```