Лекция 11. Совместное использование функций на языках ассемблера и C

Вызов функции на языке ассемблера из функции на языке C

Предупреждения редактора связей ld о потенциальных уязвимостях

Примеры.

Простой пример вызова функции на языке C из функции на языке ассемблера

Пример вызова функции Read_Sym на языке C из функции на языке ассемблера

Вызов функций printf и scanf, входящих в библиотеку glibc, из функции на языке ассемблера

Вариант предыдущей программы с использованием макроопределений

Рассмотрим программу, решающая ту же задачу, что и рассмотренная в предыдущей лекции программа примера раздельной трансляции. Отличие в том, что вызывающая функция теперь реализована на языке C в файле call-as.c а не в файле main.S, как это было в предыдущем примере.

Модули Read_sym.S и Trans_sym.S задаются как в предыдущей лекции. Связь между модулями осуществляется с помощью внешних имен по соглашениям ABI.

Рассмотрим исходный текст call-as.c без вводного комментария.

```
#include <stdio.h>
/* прототип внешней функции.
  Параметр 1 - количество символов, кот. надо обработать
  Параметр 2 - адрес первого элемента массива
*/
extern void Read_Sym(int, char*);
/* массив результатов*/
int Numbers[10];
int main()
{
  char Symbols[14] = "91A23B456C789"; /* исходный массив*/
    int i;
    printf ("%s\n",Symbols);
    Read_Sym(8, Symbols);          /* вызов as функции */
    /* печать результата */
    for (i = 0; i < 8; i++)
        printf("%d\t", Numbers[i]);
    return 0;
}
```

NB. В функциях на C следует задавать прототипы внешних функций, написанных на языке ассемблера согласно правилам C. Это необходимо для формирования правильных команд передачи их параметров, вызова и очистки стека после возврата.

В нашем случае нужно дополнительно описать прототип функции `Read_sym` с помощью ключевого слова `extern` языка C, которое эквивалентно директиве `.global` языка ассемблера, то есть определяет символьное имя `Read_sym` как имя внешней функции, доступное редактору связей. Прототип имеет вид:

```
extern void Read_Sym(int, char*);
```

Описание параметров в прототипе соответствует их определениям,

использованным в функции Read_Sym. Первый параметр описан как int (4-х байтовое целое) — число обрабатываемых символов, его значение при вызове будет помещено в стек.

NB. Второй параметр — char* - указатель (адрес) — в нашем случае адрес массива символов. Напомним, что в языке C по умолчанию имя массива является указателем на его первый элемент — в терминах ассемблера — адресом этого элемента. Поэтому при вызове Read_Sym(8,Symbols) в стек сперва будет помещен этот адрес. А затем первый параметр — количество обрабатываемых символов.

NBNB. При использовании модулей на языке C целесообразно вместо вызова для сборки редактора связей ld вызывать компилятор gcc, как это сделано в Makefile для рассматриваемого примера:

```
c-as:   call-as.c Read_Sym.o Trans_Sym.o
  gcc -m32 -gstabs+ -o c-as call-as.c Read_Sym.o Trans_Sym.o

Trans_Sym.o: Trans_Sym.S
as -ahlsm=Trans_Sym.lst --32 -gstabs+ -o Trans_Sym.o Trans_Sym.S

Read_Sym.o: Read_Sym.S
   as -ahlsm=Read_Sym.lst --32 -gstabs+ -o Read_Sym.o Read_Sym.S
```

Это удобно т. к. программа на языке C требует на этапе редактирования связей подключения объектных модулей среды выполнения C. Компилятор gcc автоматически сформирует правильную команду ld, обеспечивающую такое подключение.

Вообще говоря можно получать исполняемый файл c-as даже

командой:

```
gcc -m32 -gstabs+ -o c-as call-as.c Read_Sym.S Trans_Sym.S
```

что, однако, нецелесообразно с точки зрения раздельной правки исходных текстов функций.

Таким образом, по данной в Makefile команде компилятор gcc:

- **Для функции на языке C выполнит три стандартные фазы:**
  - **препроцессирование (например добавление в исходный код заголовочных файлов по директивам include и обработку макро);**
  - **компиляцию (получение файла на языке ассемблера);**
  - **ассемблирование (получение объектного файла из ассемблерного).**
- **Для всех объектных файлов сформирует и запустит правильную команду ld, обеспечивающую совместное редактирование связей полученных объектных модулей и необходимых объектных модулей из стандартных библиотек языка C.**

Предупреждения редактора связей ld о потенциальных уязвимостях

Редактор связей ld при генерации исполняемого файл создает в нем секции данных, кода, архитектурного стека, возможно другие, присваивая им атрибуты, определяющие разрешения операций с содержимым этих секций в оперативной памяти. Это атрибуты:

- **доступна для чтения;**

- доступна для записи;
- доступна для выполнения команд.

Секция, имеющая все три атрибута позволяет злоумышленнику вставлять в нее команды, выполняющие зловредный действия, и является потенциально уязвимой для атак. Защиту можно обеспечить убедившись, что секции данных и архитектурного стека не имеют атрибута «доступна для выполнения команд» и что секция кода не имеет атрибута «доступна для записи».

Такие потенциально опасные секции могут появиться в оперативной памяти при работе исполняемого файла за счет:

- явного указания параметра `-z execstack` редактора связей `ld`;
- по запросу компилятора (например `gcc` — при использовании вложенных функций);
- при использовании объектных файлы созданных либо из кода на языке `ассемблера`, либо более старыми компиляторами, не имеющими механизмов передачи требований к `ld`;
- для некоторых режимов по умолчанию компиляторов и `ld`.

Для информирования программиста в редактор связей `ld`, входящий в набор утилит `Binutils` выпуска `2.39` от `05.08.2022` были добавлены предупреждения о наличии у секций атрибутов, делающих их потенциально опасными. Детальное описание этих предупреждений и их использования дано в статье: Nick Clifton The linker’s warnings about executable stacks and segments [Электронный ресурс] — URL: https://www.redhat.com/en/blog/linkers-warnings-about-executable-stacks-and-segments (20.08.2024).

Там же отмечено, что эти предупреждения могут появиться при

использовании `ld` выпуска `2.39` и старше при сборке программ, которые до этого предупреждений не давали.

Ниже мы опишем эти предупреждения и способы управление атрибутами архитектурного стека для рассматриваемого в этой лекции случая использования объектных файлов, полученных из кода на языке ассемблера.

Прежде всего отметим, что команда `execstack` позволяет установить или удалить атрибут «доступна для выполнения команд» секции архитектурного стека выполняемого файла формата `ELF`. Также команда:

```
execstack -q <имя ELF файла>
```

выдаст статус этого атрибут для стека:

- **– не установлен;**
- **`X` установлен;**
- **? нельзя определить статус.**

Примеры.

Приведенные ниже протоколы сборки рассмотренного выше примера вызова функции на языке `C` из функции на языке `ассемблера` получены для версий `gcc версия 7.5.0 (SUSE Linux)` и `GNU ld (GNU Binutils; SUSE Linux Enterprise 15) 2.39.0.20220810-150100.7.40`

Выполнение `Makefile`, данного на стр.3 дает:

```
ybgv@kappa:~/GnuAS/Labs/func-as-from-c> make -B
as -ahlsm=Read_Sym.lst --32 -gstabs+ -o Read_Sym.o Read_Sym.S
```

```
as -ahlsm=Trans_Sym.lst --32 -gstabs+ -o Trans_Sym.o Trans_Sym.S
gcc -m32 -gstabs+ -o c-as call-as.c Read_Sym.o Trans_Sym.o
/usr/lib64/gcc/x86_64-suse-linux/7/../../../../x86_64-suse-
linux/bin/ld: warning: Trans_Sym.o: missing .note.GNU-stack
section implies executable stack
/usr/lib64/gcc/x86_64-suse-linux/7/../../../../x86_64-suse-
linux/bin/ld: NOTE: This behaviour is deprecated and will be
removed in a future version of the linker

ybgv@kappa:~/GnuAS/Labs/func-as-from-c> execstack -q c-as
X c-as
ybgv@kappa:~/GnuAS/Labs/func-as-from-c>
```

Т.е. `ld` **выдал предупреждения и команда** `execstack` **подтверждает, что выполнение в стеке разрешено.**

На предупреждение возможно две реакции — отменить вывод предупреждения и явно управлять выполнение в стеке с помощью директив языка ассемблера.

Для отмены следует при вызове `gcc` **передать редактору связей** `ld` **его ключ:** `--no-warn-execstack` **с помощью ключа** `gcc -Wl`**, т. е. команду запуска** `gcc` **из** `Makefile` **дать в виде:**

```
gcc -m32 -gstabs+ -Wl,--no-warn-execstack -o c-as call-as.c
Read_Sym.o Trans_Sym.o
```

Выполнение модифицированного `Makefile` **дает:**

```
ybgv@kappa:~/GnuAS/Labs/func-as-from-c> make -B -f MakefileW
as -ahlsm=Read_Sym.lst --32 -gstabs+ -o Read_Sym.o Read_Sym.S
as -ahlsm=Trans_Sym.lst --32 -gstabs+ -o Trans_Sym.o Trans_Sym.S
gcc -m32 -gstabs+ -Wl,--no-warn-execstack -o c-as call-as.c
Read_Sym.o Trans_Sym.o
```

```
ybgv@kappa:~/GnuAS/Labs/func-as-from-c> execstack -q c-as
X c-as
ybgv@kappa:~/GnuAS/Labs/func-as-from-c>
```

Т.е. ld **не выдал предупреждения но команда** execstack **подтверждает, что выполнение в стеке по прежнему разрешено.**

Программист может явно управлять наличием исполняемого кода в стеке задавая в исходном модуле на языке ассемблера **директивы:**

- `.section .note.GNU-stack,"",@progbits` — **запретить исполняемый код в стеке;**
- `.section .note.GNU-stack,"x",@progbits` — **разрешить исполняемый код в стеке;**

После размещения директивы запрета в файлах Read_Sym.S **и** Trans_Sym.S **выполнение** Makefile**, данного на стр.3 дает:**

```
ybgv@kappa:~/GnuAS/Labs/func-as-from-c> make -B
as -ahlsm=Read_Sym.lst --32 -gstabs+ -o Read_Sym.o Read_Sym.S
as -ahlsm=Trans_Sym.lst --32 -gstabs+ -o Trans_Sym.o Trans_Sym.S
gcc -m32 -gstabs+ -o c-as call-as.c Read_Sym.o Trans_Sym.o
ybgv@kappa:~/GnuAS/Labs/func-as-from-c> execstack -q c-as
- c-as
ybgv@kappa:~/GnuAS/Labs/func-as-from-c>
```

Т.е. ld **НЕ выдал предупреждения и команда** execstack **подтверждает, что выполнение в стеке НЕ разрешено.**

<u>NB NB.</u> Подчеркнем еще раз что ld **будет выполнять это контроль**

для любых сочетаний вызывающих и вызываемых модулей на языках ассемблера и C.

Мы благодарим Георгия Романовича Захарова, который обратил наше внимание на эти предупреждения весной 2024 г. будучи студентом гр 22107.

Рассмотрим выполнение файла c-as в отладчике.

Простой пример вызова функции на языке C из функции на языке ассемблера

Рассмотрим тексты функций

Вызывающая функция main.S	**Вызываемая функция** func.c

.text
 	
# Функция main 	
.globl main
.type main, @function
main:
  # Пролог функции
  pushl %ebp
  movl %esp, %ebp
  # Вызов функции на C
  call func
 	
  # Вернуть 0 из main
  movl $0, %eax
 	
  # Эпилог
  movl %ebp, %esp
	#include <stdio.h>
	
	void func() {
	  printf("Hello!\n");
	}

popl %ebp ret	

NB. Отметим, что в этом случае стандартно вызывающая функция на ассемблере должна иметь внешнее имя `main`**.**

Внимательный студент видимо уже заметил, что в функции `main.S` **символьное имя** `func` **не определено как внешнее. Как же тогда разрешаются внешние имена при построении исполняемого файла?**

NB. **Оказывается, что** `gcc` **по умолчанию определяет имена функций на языке** C **как внешние. Чтобы убедиться в этом выполним команду, создающую файл** `func.s`**, содержащий результат трансляции** `func.c` **на язык ассемблера,:**

```
gcc -m32 -c -fno-asynchronous-unwind-tables -S func.c
```

Функции ключей:

- `-c` **— отмена фазы редактирования связей;**
- `-fno-asynchronous-unwind-tables` **— удаление из ассемблерного файла кодов некоторых псевдоопераций;**
- `-S` **— не выполнять фазу ассемблирования.**

Получаемый файл `func.s` **имеет вид:**

```
        .file   "func.c"
        .text
        .section        .rodata
.LC0:
        .string "Hello!"
        .text
        .globl  func
        .type   func, @function
func:
        pushl   %ebp
```

```
        movl    %esp, %ebp
        subl    $8, %esp
        subl    $12, %esp
        pushl   $.LC0
        call    puts
        addl    $16, %esp
        nop
        leave
        ret
        .size   func, .-func
        .ident  "GCC: (SUSE Linux) 7.5.0"
        .section        .note.GNU-stack,"",@progbits
```

Мы видим, что в этом файле имя `func` **описано как** `.globl`**, т. е. является внешним.**

Исполняемый файл `main` **можно получить с помощью** `Makefile`**:**

```
main: main.o func.o
        gcc -m32 -o main main.o func.o
# ключ -c не выполнять редактирование связей
func.o: func.c
        gcc -m32 -c func.c

main.o: main.S
        as --32 -o main.o main.S
```

В данном случае использование редактора связей `ld` **также нецелесообразно , т. к. требуется подключение дополнительных объектных модулей среды выполнения C, в частности объектных файлов** `crt0.o` **или** `crt1.o`**, содержащих символьное имя точки входа - метку** `_start` **и обеспечивающих вызов функции** `main` **(метку** `_start` **мы сами, в отличие от чисто ассемблерной программы, не определяем). Для этого подключения нужно задавать** `ld` **аргументы,**

определяющие местонахождение в файловой системе модулей среды выполнения C.

Компилятор gcc сам будет формировать вызов ld с необходимыми аргументами. Реально используемые при таком вызове gcc аргументы ld (включая пути к дополнительным объектным файлам) можно увидеть, если выполнять команду

```
gcc -m32 -o main main.o func.o
```

с дополнительным ключом -v.

Пример вызова функции Read_Sym на языке C из функции на языке ассемблера

В предыдущей лекции в примере раздельной трансляции все три функции - main, Read_Sym и Trans_Sym были реализованы на языке ассемблера. Рассмотрим пример в котором функция Read_Sym реализована на языке C.

Файл main.S

```
#       ABI соглашения о вызовах функций

#   main НА АССЕМБЛЕРЕ ВЫЗЫВАЕТ Read_Sym НА C

#   Байты из массива Symbols читаются по одному.
#   Если прочтен код цифры, то он преобразуется
#   в 4-байтовое целое, иначе в значение -1.
#   Результат записывается в элементы массив Numbers.

    .include "my-macro" # подключение файла с макроопределениями
```

```
.data # секция данных

Symbols:
    .asciz  "91A23B456C789" # массив символьных кодов
#                цифр и "не цифр"

#   для показа исходного состяния  стека
#   и мест РОН в нем после pusha в основной программ

Ini:    .ascii "INIT"   # стек
EAXF:   .ascii "axF "   # %eax
EDIF:   .ascii "diF "   # %edi
EBPm4:  .ascii "bp-4"   # для показа %ebp через %esi

.bss    # секция общей памяти

 .lcomm Numbers, 40 # массив 4-х байтовых значений цифр

# ---- добавлено для раздельной трансляции ---

# Описание внешних символьных имен

.globl Read_Sym # функция на языке C

.globl Numbers  # определен здесь, Read_Sym пишет в него
результат

# ---- конец добавления ----------------- ---

.global main # точка входа - глобальная метка

.type   main, @function # т.к. собираем gcc а не ld
```

```
.text # секция команд процессора

main:           # вход,  т.к. собираем gcc а не ld
```

Лекционный комментарий.

Т.к. нам нужно подключать библиотеки языка C вызывающая функция на ассемблере должна иметь стандартное имя точки входа main. Выше цветом выделены небольшие изменения в тексте функции `main.S` по сравнению с ее версией на языке ассемблера, которые внесены из-за того, что функции `Read_Sym` реализована на языка C.

```
    nop

#   Индикаторы исходных состояний

#   Стека

    movl Ini, %eax
    movl %eax, 0(%esp)

#   Регистров общего назначения перед pusha

    movl EAXF,  %eax    # первый
    movl EDIF,  %edi    # последний
    movl EBPm4, %esi    # следующий после %ebp
                # !!! %ebp НЕ ТРОГАТЬ !!!

    pusha   # РОН в стек

    pushl $Symbols  # Параметр-2 - адрес массива в стек
```

```
    pushl $8          # Параметр-1 в стек, цикл 0-7

    call Read_Sym     # вызов функции

    addl $8,%esp    # очистить стек от параметров Read_Sym

    popa            # восстановить РОН

    Finish # конец работы, возврат в ОС

    .end   # последняя строка исходного текста
```

Файл Read_Sym.c

```
#include <stdio.h>

    extern long Trans_Sym (char);
    extern long Numbers[];

void Read_Sym(int N, char* Symbols)
{
    int i;

    for (i = 0; i < 8; i++)
    Numbers[i] = Trans_Sym(Symbols[i]);

//    printf("Hello!\n");

    return;
}
```

Файл Trans_Sym.S **не изменился.**

Получение выполняемого файла опишем в файле Makefile**:**

```
main:   main.o Read_Sym.o Trans_Sym.o
        gcc -m32 -gstabs+ -o main main.o Read_Sym.o Trans_Sym.o

main.o: main.S my-macro
```

```
        as -ahlsm=main.lst --32 -gstabs+ -o main.o main.S

# ключ -c не выполнять редактирование связей

Read_Sym.o: Read_Sym.c
        gcc -m32 -c Read_Sym.c

Trans_Sym.o: Trans_Sym.S
        as -ahlsm=Trans_Sym.lst --32 -gstabs+ -o Trans_Sym.o \
Trans_Sym.S
```

Рассмотрим работу этой программы в отладчике.

Вызов функций printf **и** scanf**, входящих в библиотеку** glibc**, из функции на языке ассемблера**

Ранее мы для диалога с терминалом использовали системные вызовы и программы преобразования строки символов в двоичное число и обратно. В общем виде это не простая задача и для ее решения в библиотеке glibc **реализованы несколько десятков функций. Рассмотрим как соглашения о вызовах** ABI **позволяют использовать для этих целей библиотечные функции** scanf **и** printf**.**

Файл call-libc.S

```
#   ABI соглашения о вызовах функций

#    Применение соглашений для вызова функций printf и scanf,
#    входящих в библиотеку glibc

#    В примере покзан ввод и вывод 16 и 32 битовых ЗНАКОВЫХ
#    целых чисел

#    Прототипы из man:

#  Параметр формата для обоих функций передается как адрес
(указатель)

#    int scanf(const char *format, *p1, *p2, ...);
#    параметры после формата - адреса (указатели)
```

```
#    int printf(const char *format, p1, p2, ...);
#    параметры после формата - сами значения для печати

.include "my-macro" # подключение файла с макроопределениями

#   Указание имен функций как внешних для редактирования
#   связей с помощью компилятора gcc, в который встроен
#   механизм подключения функция из библиотеки glibc.

.extern scanf
.extern printf

.data

m:  .short  0
n:  .short  0
l:  .long   0
k:  .long   0
p:  .long   0

#   Форматные строки для передачи их адресов как параметров

#   d - знаковое целое

fmth:
        .string "%hd"   # для scanf: h - 16 бит,

fmthE:
        .string "%hd\n" # для printf

fmtl:
        .string "%ld"   # для scanf: l - 32 бита, d - знаковое
целое

fmtp:
    .string "%ld\n"  # для printf


.text        # секция команд процесора

.global main # точка входа - глобальная метка

.type   main, @function # т.к. собираем gcc а не ld

main:
    nop

#   Вызовы scanf для ввода значений 16 битовых целых
```

```
#   m и n, форматная строка fmth

    Puts "Введите m:"
    pushl $m    # 2 параметр - адрес переменной
    pushl $fmth # 1 параметр - адрес форматной строки
    call scanf
    addl $8, %esp   # очистка стека

    Puts "Введите n:"
    pushl $n    # 2 параметр - адрес переменной
    pushl $fmth # 1 параметр - адрес форматной строки
    call scanf
    addl $8, %esp   # очистка стека

#   получение суммы m и n

    subl %edx, %edx
    addw m, %dx
    addw n, %dx
    movl %edx,p

#   печать суммы

    Puts "m + n ="
    pushl p             # 2 параметр - значение
    pushl $fmthE        # 1 параметр - адрес форматной строки
    call printf         #
    addl $8, %esp       # очистка стека


#   Вызовы scanf  для ввода значений 32
#   битовых целых l и k, форматная строка fmtl

    Puts "Введите l:"
    pushl $l    # 2 параметр - адрес переменной
    pushl $fmtl # 1 параметр - адрес форматной строки
    call scanf
    addl $8, %esp   # очистка стека

    Puts "Введите k:"
    pushl $k    # 2 параметр - адрес переменной
    pushl $fmtl # 1 параметр - адрес форматной строки
    call scanf
    addl $8, %esp   # очистка стека

#   получение суммы k и l

    subl %edx, %edx
    addl k, %edx
```

```
    addl l, %edx
    movl %edx, p

#   печать суммы

    Puts "l + k ="
    pushl p             # 2 параметр - значение
    pushl $fmtp         # 1 параметр - адрес форматной строки
    call printf         #
    addl $8, %esp       # очистка стека


    Finish
.end
```

Вариант предыдущей программы с использованием макроопределений

Повторяющиеся конструкции удобно описывать и задавать с помощью макро определений, которые зададим в файле `libc-macro`**:**

```
/*
 * Макроопределение для ввода с клавиатуры stdin, функция scanf
 * Аргументы:
 *  - Variable имя переменной, принимающей значение
 *  - Format имя форматной строки
*/
.macro  Input  Variable Format
    pushl $\Variable
    pushl $\Format
    call scanf
    addl $8, %esp
.endm

/*
 * Макроопределение для вывода в stdout, функция printf
 * Аргументы:
 *  - Variable имя переменной — значение выводится
 *  - Format имя форматной строки
*/

.macro  Output  Variable Format
    pushl \Variable
    pushl $\Format
    call printf
```

```
    addl $8, %esp
.endm
```

Использование этих макроопределений позволяет существенно упростить программу:

```
#   ABI соглашения о вызовах функций

#    Применение соглашений для вызов функций printf и scanf,
#    входящих в библиотеку glibc.

#    Текст сокращен за счет макрокоманд Input и Output

#    В примере показан ввод и вывод 16 и 32 битовых ЗНАКОВЫХ
#    целых чисел

#    Прототипы из man:

#  Параметр формата для обоих функций  передается как адрес
(указатель)

#    int scanf(const char *format, *p1, *p2, ...);
#    параметры после формата - адреса (указатели)

#    int printf(const char *format, p1, p2, ...);
#    параметры после формата - сами значения для печати

#   Подключение макроопределения Exit, Getchar, Puts, Finish
.include "my-macro"

#   Подключение макроопределений Input (для scanf)
#   и Output (для printf)

.include "libc-macro"

#   Указание имен функций как внешних для редактирования
#   связей с помощью компилятора gcc, в который встроены
#   имя файла библиотеки glibc и путь к ней.

.extern scanf
.extern printf

.data

m:  .short  0
n:  .short  0
```

```
l:  .long   0
k:  .long   0
p:  .long   0

#   Форматные строки для передачи их адресов как параметров

fmth:
        .string "%hd"   # scanf: h - 16 бит, d - знаковое целое

fmthE:
        .string "%hd\n" # printf

fmtl:
        .string "%ld"   # scanf: l - 32 бита, d - знаковое целое

fmtp:
    .string "%ld\n"  # printf


.text       # секция команд процесора

.global main  # точка входа - глобальная метка

.type   main, @function # т.к. собираем gcc а не ld

main:
    nop

#   Ввод значений 16 битовых целых m и n

    Puts "Введите m:"
    Input m fmth

    Puts "Введите n:"
    Input n fmth

#   получение суммы m и n

    subl %edx, %edx
    addw m, %dx
    addw n, %dx
    movl %edx,p

#   печать суммы

      Puts "m + n ="
      Output p fmthE

#   Ввод значений 32 битовых целых l и k
```

```
    Puts "Введите l:"
    Input l fmtl

    Puts "Введите k:"
    Input k fmtl

#   получение суммы k и l

      subl %edx, %edx
    addl k, %edx
    addl l, %edx
    movl %edx, p

      Puts "l + k ="
      Output p fmtp

    Finish
.end
```