Программные уязвимости – на стыке аппаратуры и программного обеспечения

Лекция 1. Исполнение процессов в Linux, binutils, gdb, трассировка программ.

IA 32

```
char * buf =
  "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" /*  1 - 10 */
  "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" /*  11 - 20 */
  "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" /*  21 - 30 */
  "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" /*  31 - 40 */
  "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" /*  41 - 50 */
  "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" /*  51 - 60 */
  "\x90\x90\x90\x90"; /* 61 - 64 */
```

Программа

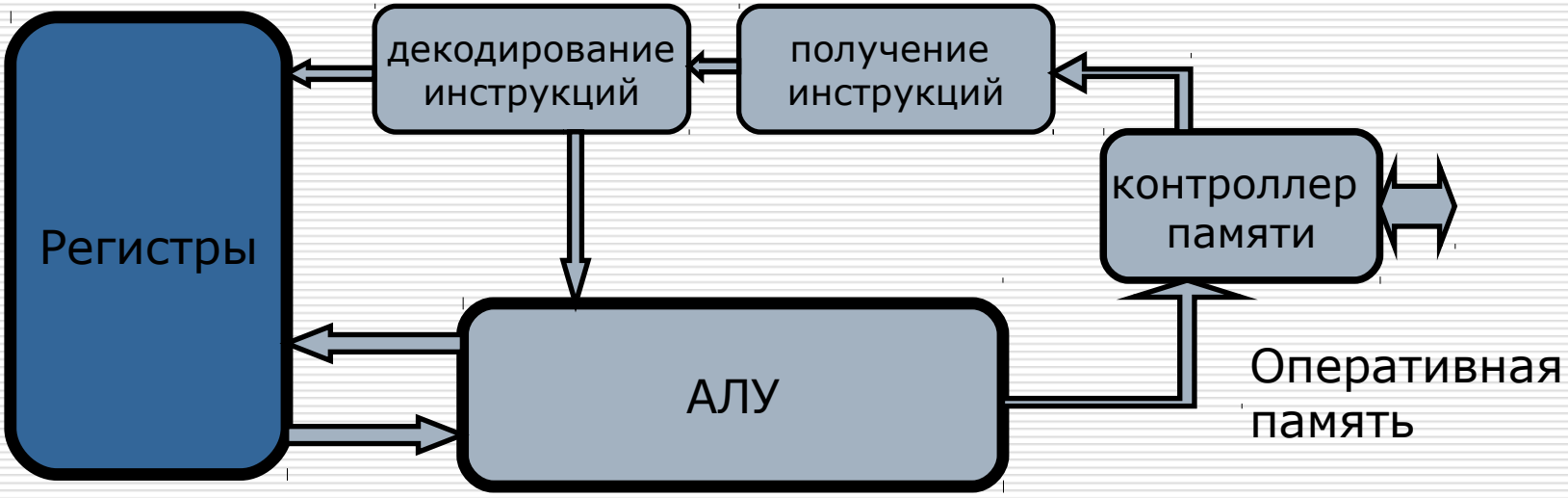
- Язык ассемблера. Синтаксисы Intel и AT&T. Ассемблер as. Дизассемблирование.
- Стек программы. Подпрограммы и функции. Передача аргументов, возврат значения. Системные вызовы.
- Размещение объектов в памяти: статическое, динамическое, автоматическое.
- Структура процесса в Linux.
- GNU binutils: работа с исполимыми файлами.
- Трассировка и отладка: gdb, strace.

Тестовая программа

```
#include "stdio.h"
void print_scrambled(char *message) {
        int i = 3;
        do {
                printf("%c", (*message)+i);
        } while (*++message);
        printf("\n");
}

int main() {
        char * bad_message = NULL;
        char * good_message = "Hello, world.";
        print_scrambled(good_message);
        print_scrambled(bad_message);
}
```

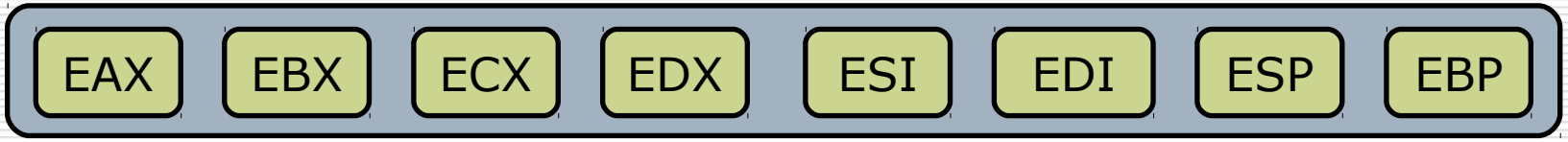
Исполнение программы на процессоре



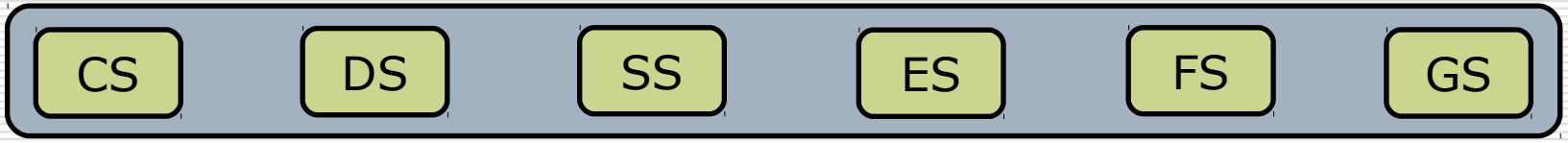
Регистры – хранение данных для использования в инструкциях
Флаги (регистр EFLAGS) – результаты выполнения инструкций
Например: ZF — устанавливается, если результат равен нулю

Регистры

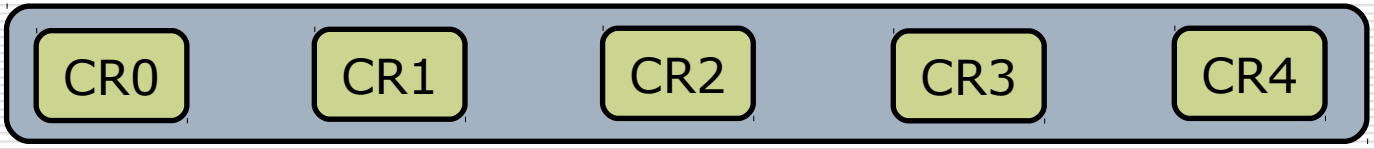
- Общего назначения



- Сегментные



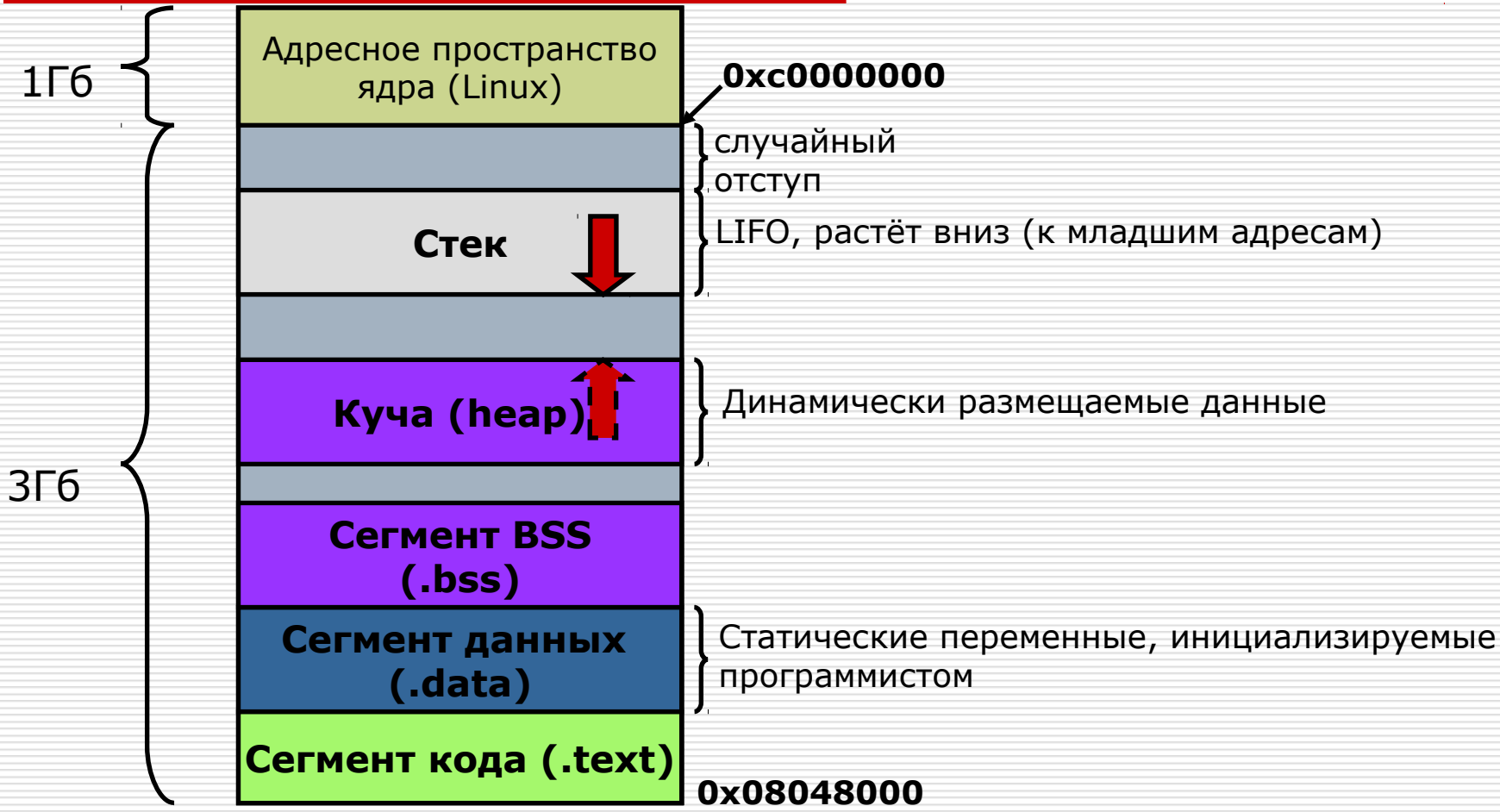
- Указатель инструкций (EIP)
- Управляющие регистры



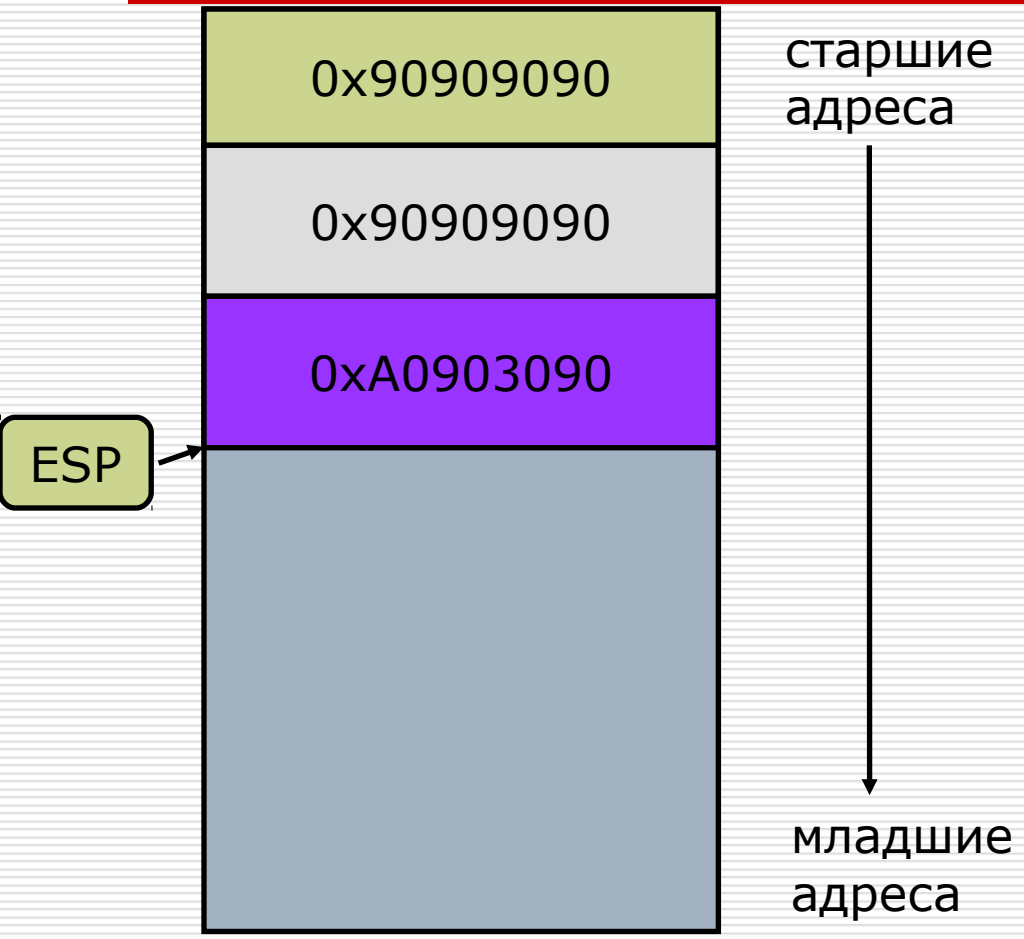
Регистры общего назначения (при автоматической кодогенерации)

- EAX - аккумулятор, для хранения операндов и возвращаемого значения функций
- EBX - указатель на данные
- ECX - регистр счётчика, в циклах
- EDX - регистр данных, указатель ввода-вывода
- ESI EDI - регистры для операций с памятью
- ESP - указатель стека
- EBP - указатель на данные стека (фрейма)

Виртуальная память процесса



Стек - LIFO



- **push** – кладёт значение на стек
- **pop** – забирает значение со стека
- **ESP** – указывает на вершину стека

Подпрограммы и функции

- Подпрограммы
  - Процедуры vs Функции
- Инструкции для реализации подпрограмм:
  - call
  - ret
  - push/pop
  - jmp

Размещение объектов в памяти

- Статическое – сегменты .bss, .data
- Динамическое - heap
- Автоматическое – stack (локальные переменные функций)

```
.globl main
main:
        call foo
        ret
foo:
        pushl %ebp
        movl %esp, %ebp
        subl $4, %esp
        movl $10, -4(%ebp)
        movl %ebp, %esp
        popl %ebp
        ret
```

Вопрос

- foo – некоторый класс C++
- Что не так с этим кодом?

```
{
int i;
foo f;
bar(&f);
}
```

Ассемблер: синтаксис Intel

- Приёмник находится слева от источника
- Числовые константы: 20h
- gdb: **set disassembly-flavor intel**

```
0x080483a4 <print_scrambled+0>: push   ebp
0x080483a5 <print_scrambled+1>: mov    ebp,esp
0x080483a7 <print_scrambled+3>: sub    esp,0x18
0x080483aa <print_scrambled+6>: mov    DWORD PTR [ebp-0x4],0x3
0x080483b1 <print_scrambled+13>:       mov    eax,DWORD PTR [ebp+0x8]
0x080483b4 <print_scrambled+16>:       movzx  eax,BYTE PTR [eax]
0x080483b7 <print_scrambled+19>:       movsx  eax,al
0x080483ba <print_scrambled+22>:       add    eax,DWORD PTR [ebp-0x4]
0x080483bd <print_scrambled+25>:       mov    DWORD PTR [esp],eax
0x080483c0 <print_scrambled+28>:       call   0x80482c8 <putchar@plt>
0x080483c5 <print_scrambled+33>:       add    DWORD PTR [ebp+0x8],0x1
0x080483c9 <print_scrambled+37>:       mov    eax,DWORD PTR [ebp+0x8]
0x080483cc <print_scrambled+40>:       movzx  eax,BYTE PTR [eax]
0x080483cf <print_scrambled+43>:       test   al,al
0x080483d1 <print_scrambled+45>:       jne    0x80483b1 <print_scrambled+13>
0x080483d3 <print_scrambled+47>:       mov    DWORD PTR [esp],0xa
0x080483da <print_scrambled+54>:       call   0x80482c8 <putchar@plt>
0x080483df <print_scrambled+59>:       leave
0x080483e0 <print_scrambled+60>:       ret
```

Ассемблер: синтаксис AT&T

- Приемник находится справа от источника
- Константы: $0x20
- gdb: **set disassembly-flavor att** (default)

```
0x080483a4 <print_scrambled+0>: push   %ebp
0x080483a5 <print_scrambled+1>: mov    %esp,%ebp
0x080483a7 <print_scrambled+3>: sub    $0x18,%esp
0x080483aa <print_scrambled+6>: movl   $0x3,-0x4(%ebp)
0x080483b1 <print_scrambled+13>:       mov    0x8(%ebp),%eax
0x080483b4 <print_scrambled+16>:       movzbl (%eax),%eax
0x080483b7 <print_scrambled+19>:       movsbl %al,%eax
0x080483ba <print_scrambled+22>:       add    -0x4(%ebp),%eax
0x080483bd <print_scrambled+25>:       mov    %eax,(%esp)
0x080483c0 <print_scrambled+28>:       call   0x80482c8 <putchar@plt>
0x080483c5 <print_scrambled+33>:       addl   $0x1,0x8(%ebp)
0x080483c9 <print_scrambled+37>:       mov    0x8(%ebp),%eax
0x080483cc <print_scrambled+40>:       movzbl (%eax),%eax
0x080483cf <print_scrambled+43>:       test   %al,%al
0x080483d1 <print_scrambled+45>:       jne    0x80483b1 <print_scrambled+13>
0x080483d3 <print_scrambled+47>:       movl   $0xa,(%esp)
0x080483da <print_scrambled+54>:       call   0x80482c8 <putchar@plt>
0x080483df <print_scrambled+59>:       leave
0x080483e0 <print_scrambled+60>:       ret
```

Трансляция в объектный ELF через ассемблер

- gcc –S test.c
- *as -a --gstabs -o* test.o test.s
- *ld -m elf_i386 -static /usr/lib/crt1.o /usr/lib/crti.o -lc test.o /usr/lib/crtn.o*

Стандартные обёртки исполнимых ELF

- *Обёртки исполнимых файлов в Linux: crt1.o, crti.o и crtn.o.*
- *crt1.o и crti.o обеспечивают инициализацию программы*
- *crtn.o занимается завершением и очисткой памяти*

Виды исполнимых файлов в UNIX

- Скрипты (текстовые исполнимые файлы)
  - #!/bin/zsh, #!/usr/local/bin/my_brainfuck_interp
- a.out (**a**ssembler **out**put)
  - старый формат исполнимых файлов (PDP-7, PDP-11)
  - не имеет ничего общего с выводом GCC по-умолчанию, кроме названия
- COFF (**C**ommon **O**bject **F**ile **F**ormat)
  - сменил a.out в AT&T Unix System V
- ELF (**E**xecutable and **L**inkable **F**ormat)
  - начиная с AT&T Unix System V release 4

Hint: утилита *file* умеет определять тип файла по его содержимому

Executable Linkable Format

- Может содержать следующие виды бинарного кода:
  - Исполнимые файлы программ
  - Динамически загружаемые библиотеки (*.so)
  - Объектные файлы (*.o)
    - Статические библиотеки (*.a) — архивы объектных файлов, получаемые с помощью утилиты *ar*

Разница между исполнимым файлом и библиотекой в том, что у библиотеки нет единственной точки входа (_start или main), точек входа множество — все экспортируемые функции

Executable Linkable Format

- Может содержать множество секций, включая пользовательские.
- Наиболее важные:
  - .text — содержит код программы
  - .bss — неинициализированные переменные
  - .data — инициализированные переменные
  - .interp — путь к бинарному интерпретатору
  - .init — выполняется до вызова точки входа

Подробное описание формата можно найти в man pages: man ELF

Декорирование (манглинг) имён

- ELF создавался для программ на языках Си и Фортран
- Рассмотрим программу на Си++:

```
c l a s s bar {
    v o i d main ( i n t ) ;
    v o i d main ( i n t , char ) ;
};
i n t main ( i n t a r g c , char ** a r g v ) {
    return 0;
}
```

- Как разместить все эти функции main в одном исполнимом файле?

Декорирование (манглинг) имён

- Компилятор языка Си++ выполняет декорирование имён (name mangling):
  - void main(int,char)->void _Z1mainic(int,char)
  - функции декорируются также одним или двумя символами '_' в качестве префикса
- Способы декорирования могут отличаться от компилятора к компилятору

Compiler	void func(int)	void func(int, char)	void func(void)
Tru64 C++ ANSI	__7func__Fi	__7func__Fic	__7func__Fv
HP aC++ PA-RISC	func__Fi	func__Fic	func__Fv
HP aC++ IA-64	_Z1funci	_Z1funcic	_Z1funcv
GNU GCC 3 and 4			

Бинарный интерпретатор

- Для исполнимых файлов — это линковщик и загрузчик ld
- Исторически расположен в /lib/ld.so
- В GNU/Linux используется /lib/ld.so.* для файлов формата a.out и /lib/ld-linux.so.* для ELF

В системе может быть более одного бинарного интерпретатора. Кроме того, программист может использовать свой собственный

Интерфейсы dlopen() и dlsym()

- Иногда имя внешней функции или библиотеки неизвестно во время компиляции.
- Линковщик предоставляет механизм dlopen() и dlsym():

```
handle = dlopen("/usr/local/lib/libbar.so", RTLD_LOCAL);
*(void **)(&fptr) = dlsym(handle, "my_function");
(*fptr)(32);
```

Документация и примеры: man dlopen

Настройка ld

- /lib/ld*so* использует переменные окружения и файлы конфигурации для поиска динамических библиотек при загрузке приложения
- По-умолчанию поиск производится:
  - в /lib и /usr/lib
  - пути, указанные в /etc/ld.so.conf, /etc/ld.so.cache и /etc/ld.so.preload
- Часто используемые переменные:
  - LD_LIBRARY_PATH
  - LD_PRELOAD
  - LD_TRACE_LOADED_OBJECTS

В чём разница между указанием пути в файле конфигурации и в переменной окружения?

GNU binutils

- Коллекция инструментов для работы с бинарными исполнимыми файлами
- Наиболее важные из них ld и as.
- Кроме того, для задач анализа программ полезными являются:
  - c++filt — позволяет убрать декорирование символов
  - gprof — средство профилировки (для однопоточных программ)
  - nm — выдаёт листинг экспортируемых символов для заданного файла
  - objdump — позволяет дизассемблировать исполнимые файлы
  - size — показывает размер всех секций в ELF
  - strings — показывает все строковые данные в бинарном файле

Objdump

- Дизассемблирование объектных файлов
  - --disassemble-all
- Деманглинг
  - --demangle
- Просмотр содержимого секций ELF, анализ core dumps:

Terminal 1

```
~$ ulimit -c unlimited
~$ cat
Ошибка сегментирования (core dumped)
~$ objdump -f core
```

Terminal 2

```
$ ps -ef | grep cat
gamajun   1418 27316  0 11:42
pts/3    00:00:00 c
$ kill -SIGSEGV 1418
```

Отладка и трассировка

- gdb — GNU debugger

```
~$ gdb /bin/cat core
Core was generated by `cat'.
Program terminated with signal 11, Segmentation fault.
#0  0x0074a416 in __kernel_vsyscall ()
(gdb) bt
#0  0x0074a416 in __kernel_vsyscall ()
#1  0x0085be73 in read () from /lib/i386-linux-gnu/libc.so.6
#2  0x0804cf63 in ?? ()
#3  0x08049f4d in ?? ()
#4  0x007b2e37 in __libc_start_main () from /lib/i386-linux-gnu/libc.so.6
#5  0x08049081 in ?? ()
(gdb) disas
Dump of assembler code for function __kernel_vsyscall:
   0x0074a414 <+0>:int    $0x80
=> 0x0074a416 <+2>:   ret
End of assembler dump.
```

- strace — трассировка системных вызовов и сигналов
  - основан как раз на подмене вызовов через LD_PRELOAD

Задание

- Дана программа:
- Требуется:
  - Скомпилировать её в ассемблер с помощью gcc;
  - Поменять текст функции print_scrambled в ассемблерном представлении таким образом, чтобы выполнялась проверка параметра на NULL и вывод error message, если параметр NULL (error message сделать локальной строковой переменной функции);
  - Транслировать в машинный код с помощью GNU as, слинковать с помощью ld;
  - При помощи gdb вывести листинг ассемблерного кода функции print_scrambled в нотации Intel и в нотации AT&T;
  - Вывести адрес начала сегмента стека в трёх последовательных запусках программы, отключить рандомизацию адресов (ASLR), повторить три запуска.
  - Сравнить вывод objdump тела программы и файла core после SIGSEGV.
  - Выполнить трассировку программы с помощью strace.

```
#include "stdio.h"
void print_scrambled(char
*message) {
      int i = 3;
      do { printf("%c", (*message)
      +i); }
      while (*++message);
      printf("\n");
      }

int main() {
char * bad_message = NULL; char
* good_message = "Hello, world.";
print_scrambled(good_message);
print_scrambled(bad_message);
}
```