Программные уязвимости – на стыке аппаратуры и программного обеспечения

Лекция 1. Архитектура вычислителя, процессы в Linux, gdb.

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

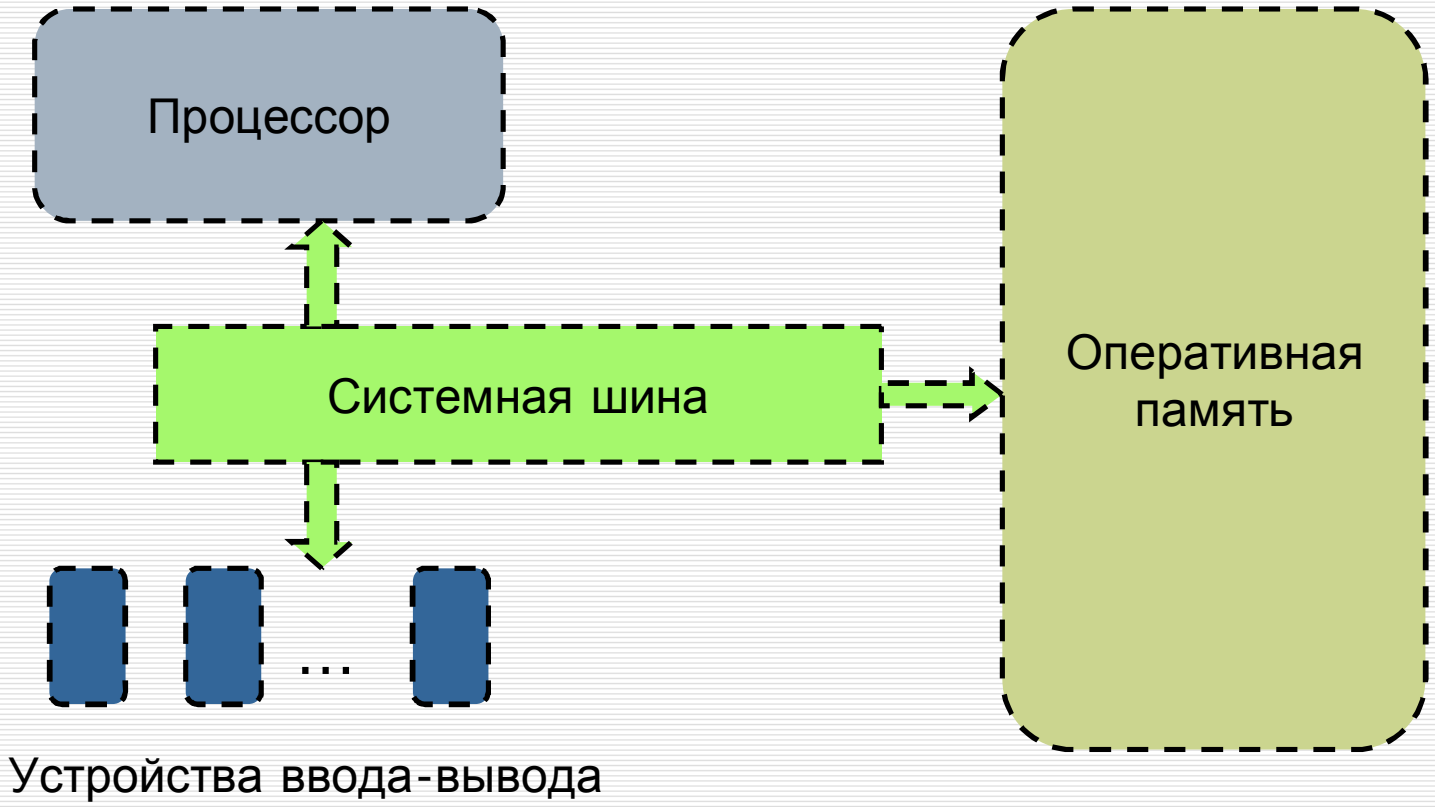
- Язык ассемблера. Синтаксисы Intel и AT&T. Ассемблер as. Дизассемблирование.
- Устройство процессора. Регистры, оперативная память, порты ввода/вывода, прерывания. Стек.
- Подпрограммы и функции. Передача аргументов, возврат значения. Системные вызовы.
- Размещение объектов в памяти: статическое, динамическое, автоматическое.

Тестовая программа

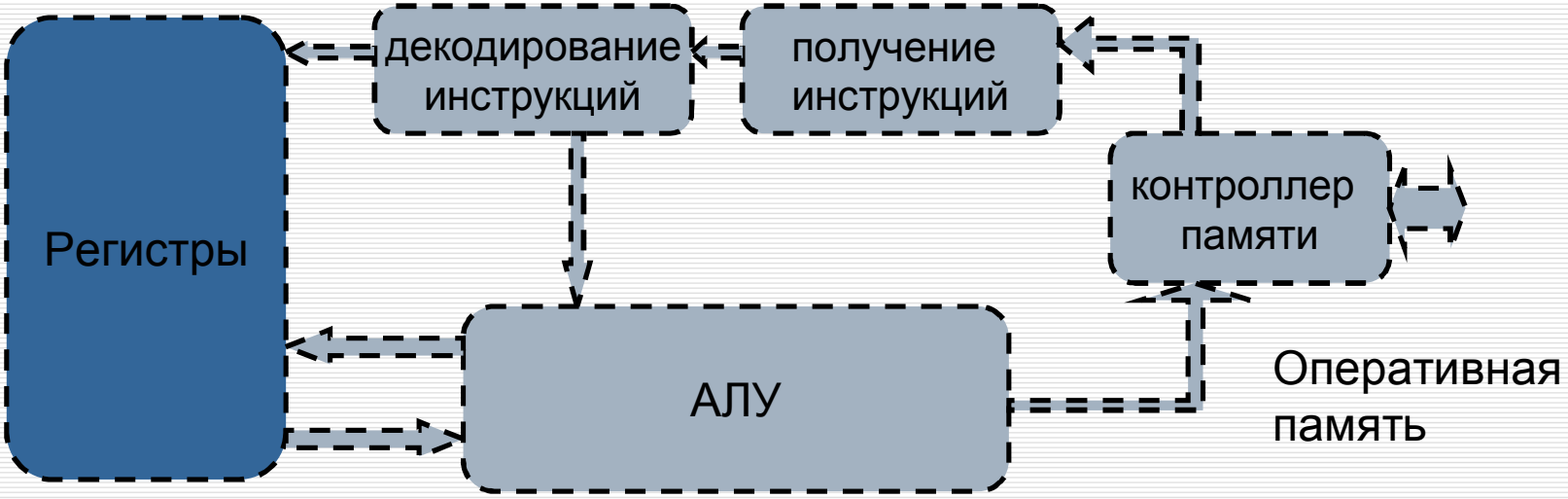
```
#include "stdio.h"
void print_scrambled(char *message) {
        int i = 3;
        do {
              printf("%c", (*message)+i);
        } while (*++message);
        printf("\n");
}

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

Архитектура условного вычислителя



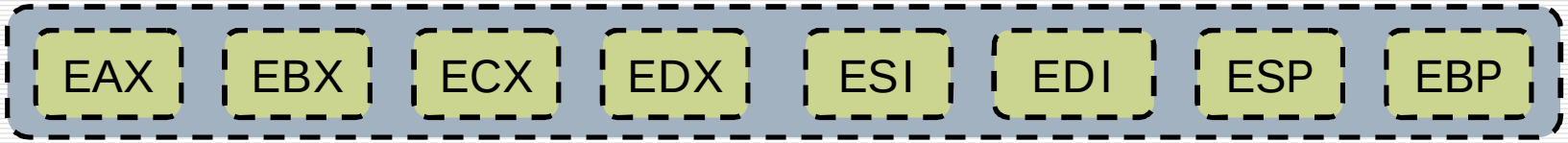
Процессор



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

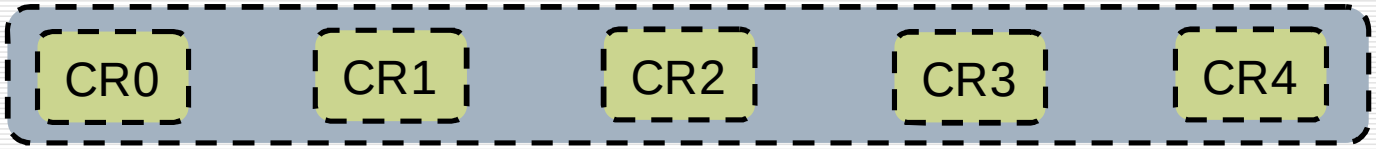
Регистры

- Общего назначения



- Сегментные

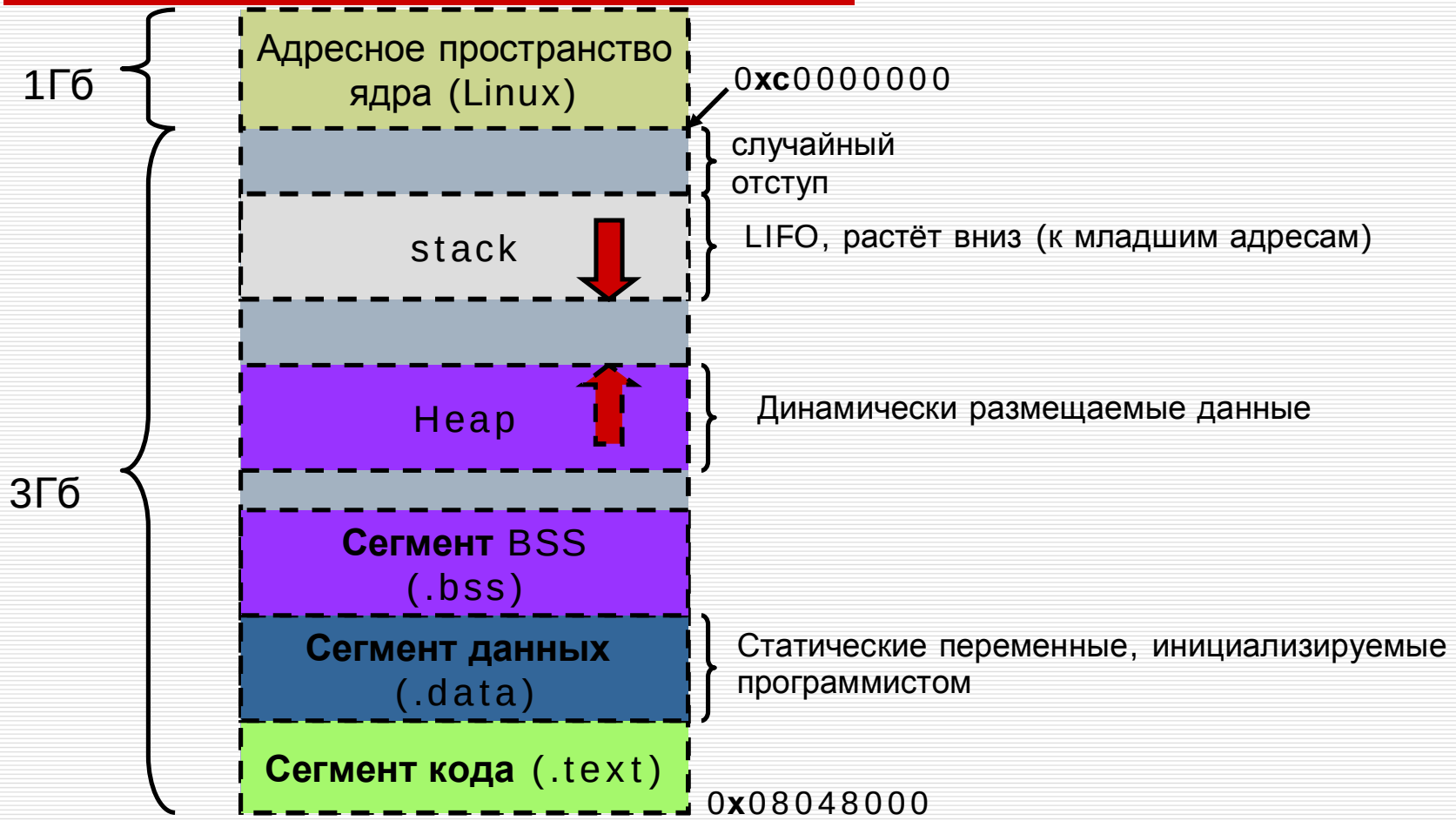
- Указатель инструкций (EIP)
- Управляющие регистры



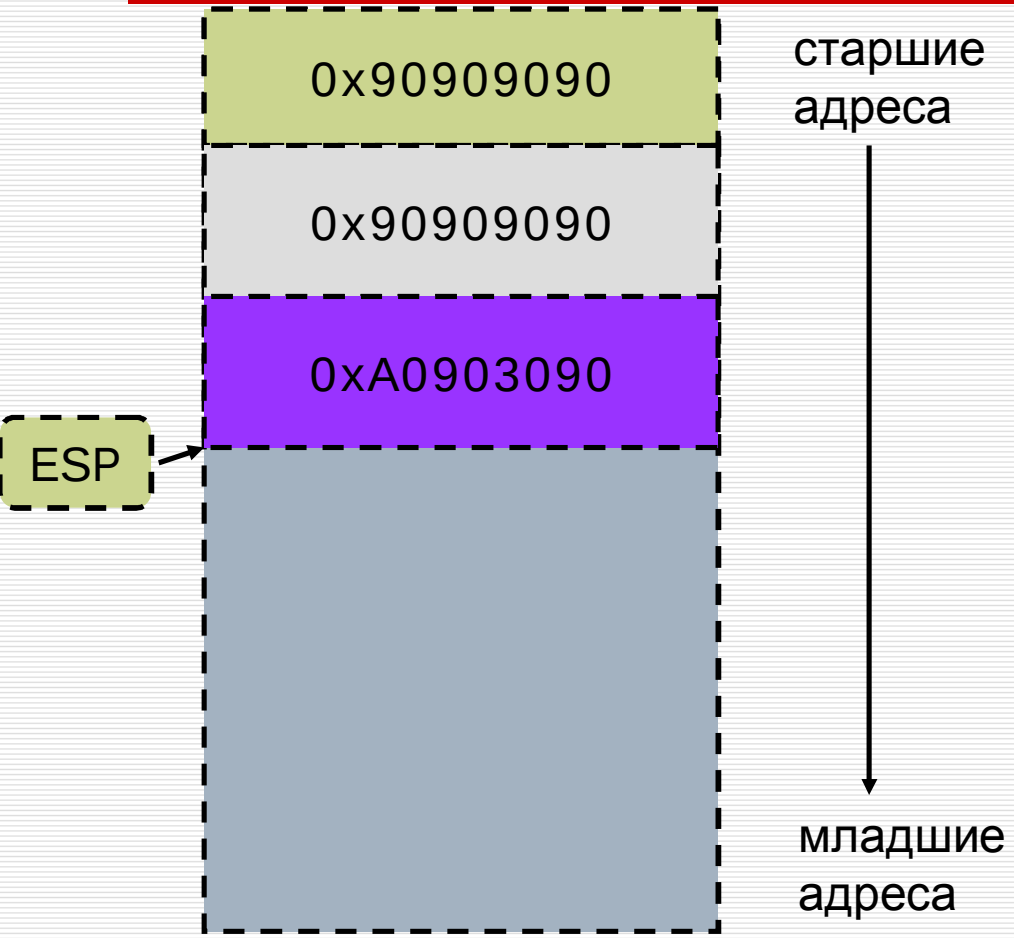
Регистры общего назначения (при автоматической кодогенерации)

- 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



Stack - 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*

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.

```
#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);
}
```