

Internet Security 2

(aka Advanced InetSec)

Reverse Engineering and Binary Analysis

Christian Platzer

Gilbert Wondracek

Markus Kammerstetter

inetsec@seclab.tuwien.ac.at

Edgar Weippl

Overview

*Int. Secure Systems Lab
Technical University Vienna*

- Introduction
- Reverse engineering
 - Intel x86 Assembler Primer
 - static vs. dynamic analysis techniques
 - anti-reverse engineering
- Malicious code analysis
- ICTF Challenge 29

Introduction

*Int. Secure Systems Lab
Technical University Vienna*

- Reverse engineering
 - process of analyzing a system
 - understand its structure and functionality
 - used in different domains (e.g., consumer electronics)
- Software reverse engineering
 - understand architecture (from source code)
 - extract source code (from binary representation)
 - change code functionality (of proprietary program)
 - understand message exchange (of proprietary protocol)

Reverse Engineering

*Int. Secure Systems Lab
Technical University Vienna*

- Application areas
 - copy (steal) technology
 - allow for interoperability
 - Samba (SMB protocol), WINE (Windows API), OpenOffice (MS Office), NTFS (file system structure), ...
 - circumvent copy protection or access restrictions
 - program cracking, creation of license key-generators (keygens)
- Techniques
 - static approaches
 - dynamic approaches

Reverse Engineering

*Int. Secure Systems Lab
Technical University Vienna*

- Static techniques
 - read documentation
 - read source code
 - analyze binary for strings, symbols, and library functions
 - disassemble binary image
- Dynamic techniques
 - observe interaction with environment
 - file system, network, registry
 - observe interaction with operating system
 - system calls
 - debug process

Reverse Engineering

Static Techniques

Static Techniques

- Gathering program information

```
$ cat test.c
```

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

```
int main (int argc, char **argv)
```

```
{
```

```
    if (argc == 2 && strcmp(argv[1], "correctSerial") == 0)
```

```
    {
```

```
        printf("do something useful\n");
```

```
    }
```

```
    else
```

```
    {
```

```
        printf("usage: %s <correct-serial>\n", argv[0]);
```

```
    }
```

```
    return 0;
```

```
}
```

Static Techniques

- Gathering program information
 - strings that the binary contains
 - `strings` command

```
$ strings test
```

```
/lib64/ld-linux-x86-64.so.2  GLIBC_2.2.5
libm.so.6                    fff.
__gmon_start__              fffff.
_Jv_RegisterClasses         l$ L
libc.so.6                   t$(L
puts                         |$0H
printf                       correctSerial
strcmp                       do something useful
libc_start_main              usage: %s <correct-serial>
```

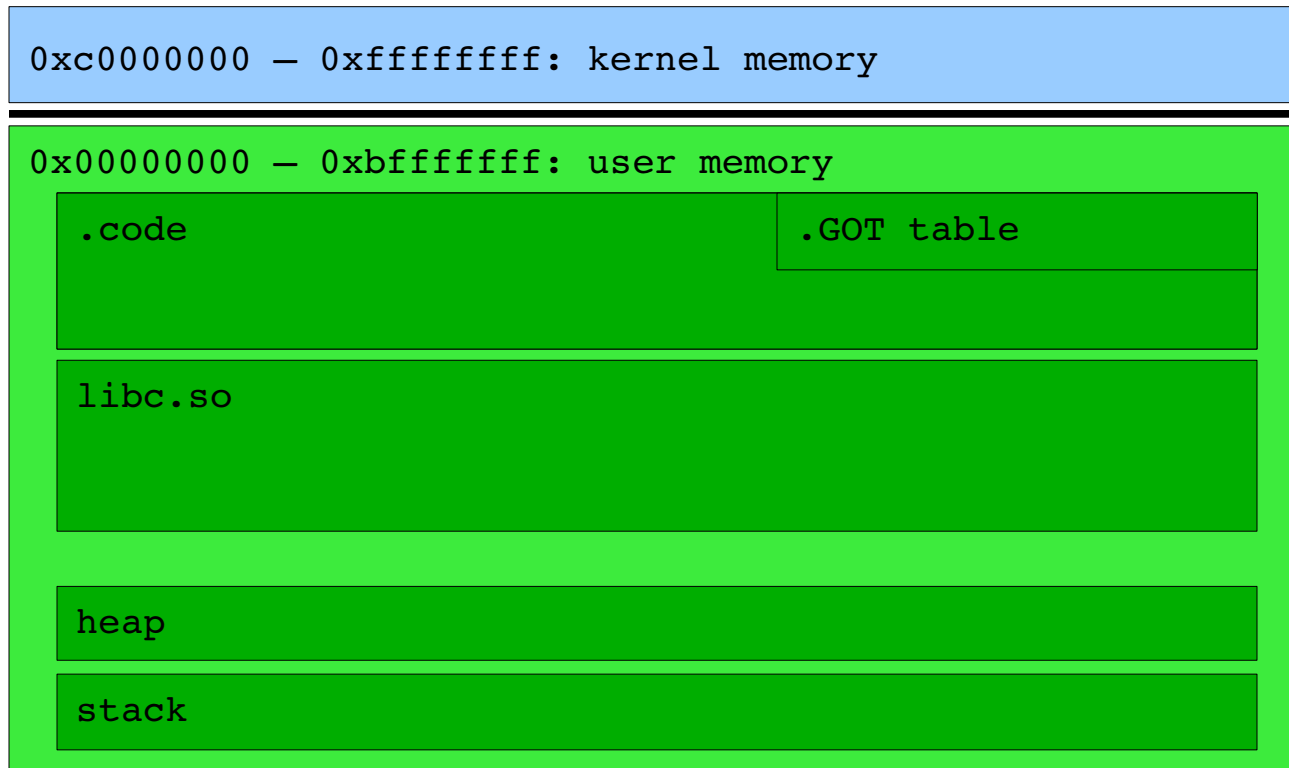
Static Techniques

*Int. Secure Systems Lab
Technical University Vienna*

- Gathering program information
 - library functions that were used
 - easy when program is dynamically linked

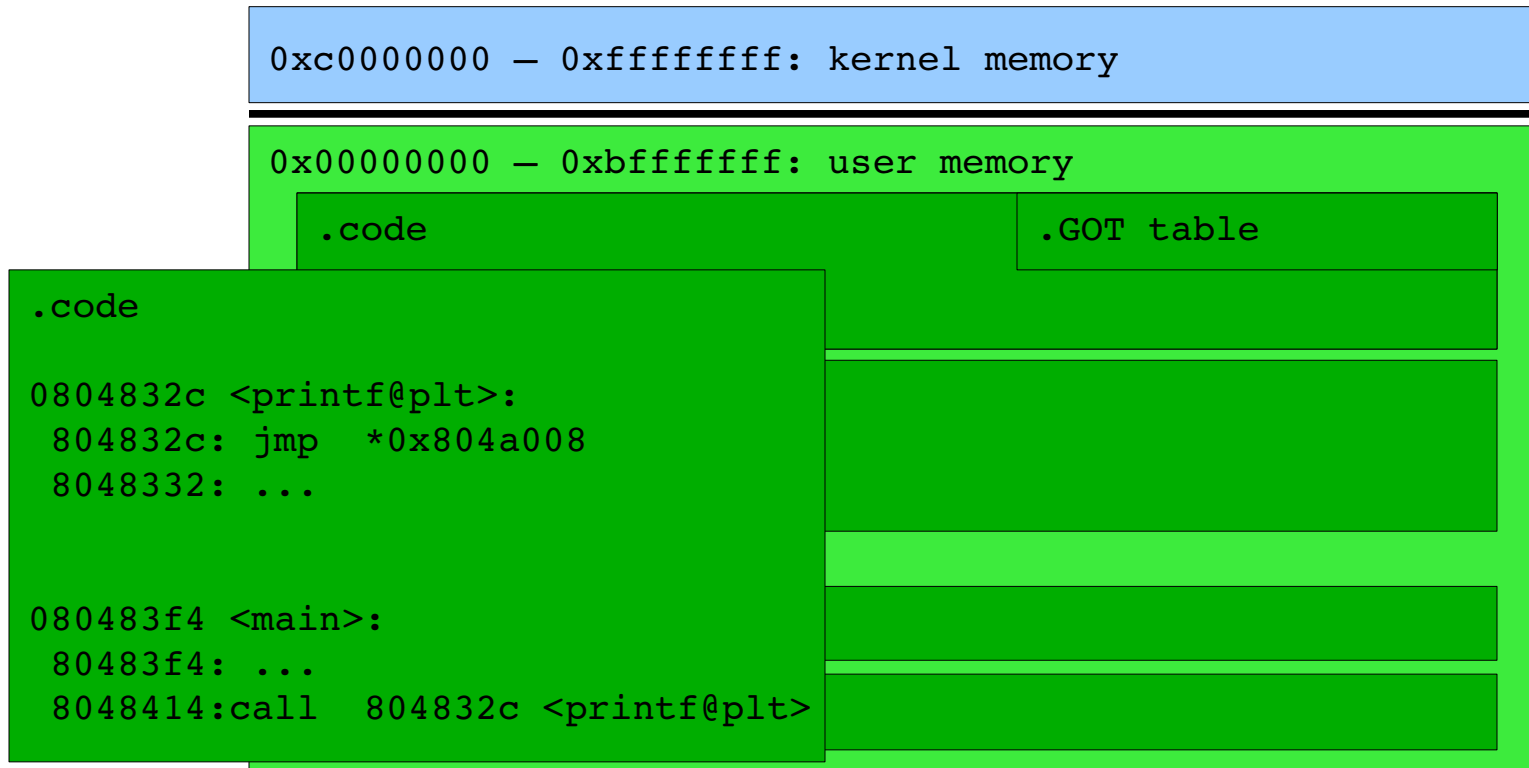
Shared Libraries

- Process layout (32 bit systems)



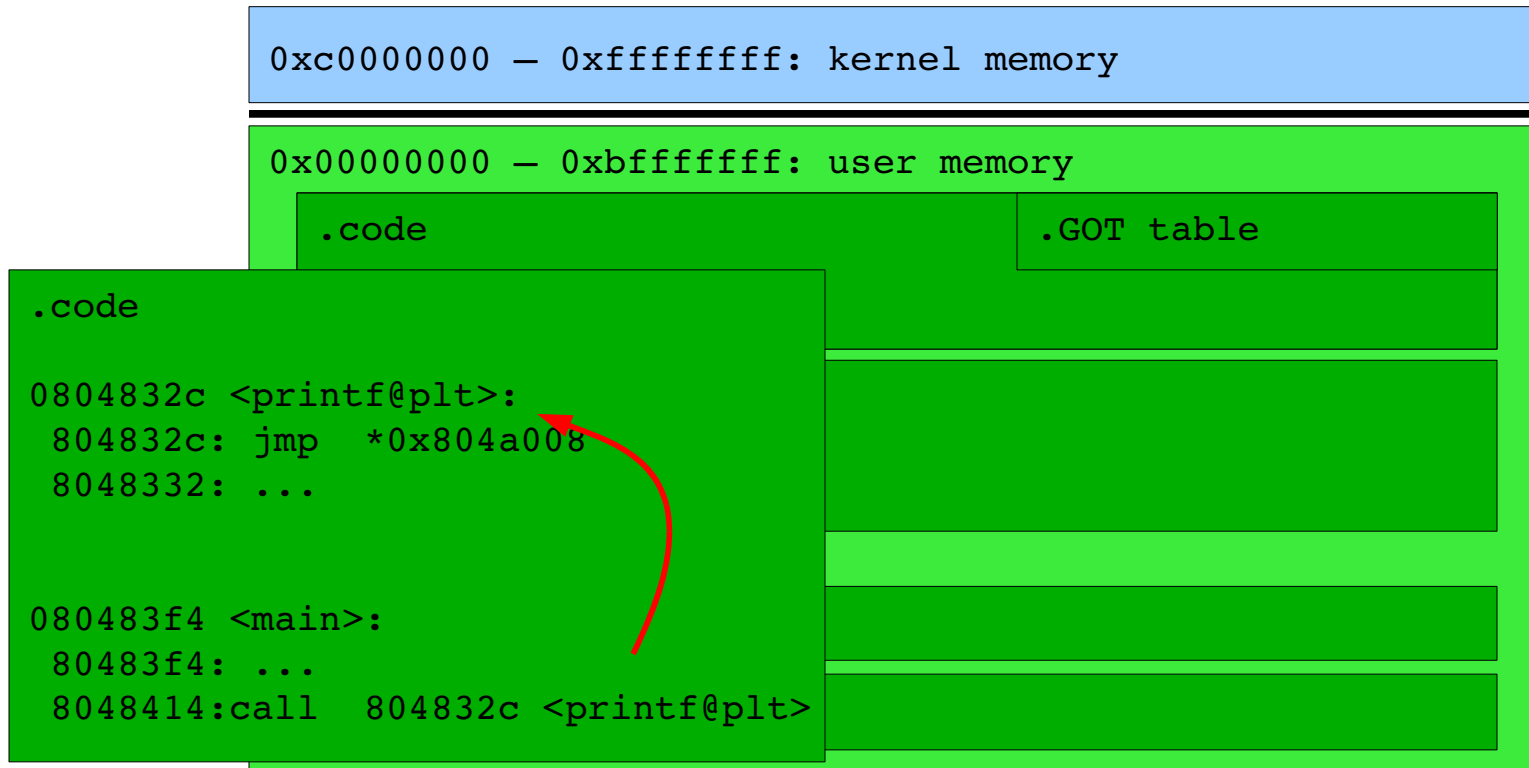
Shared Libraries

- Process layout (32 bit systems)



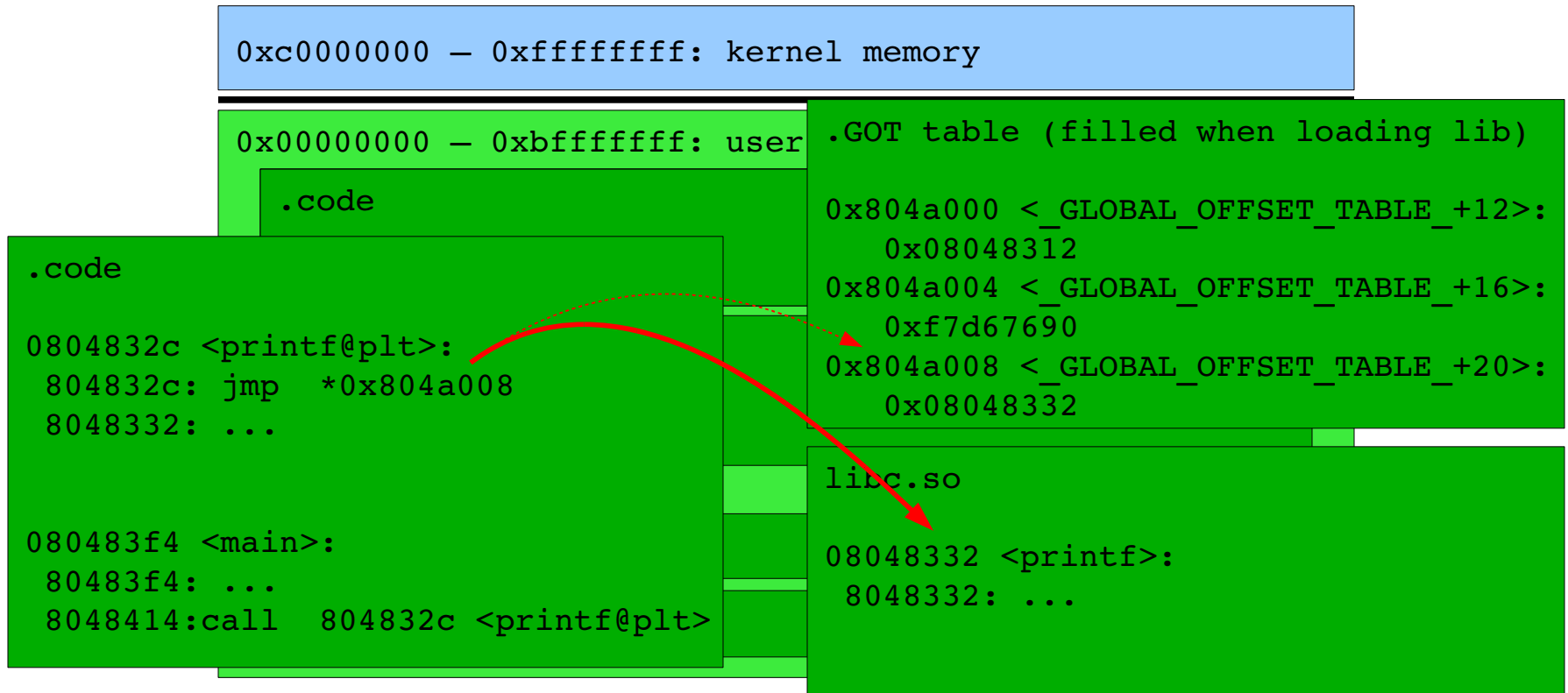
Shared Libraries

- Process layout (32 bit systems)



Shared Libraries

- Process layout (32 bit systems)



Static Techniques

- Gathering program information
 - library functions that were used
 - easy when program is dynamically linked
 - use `ldd` to find imported libraries

```
$ ldd test
linux-vdso.so.1 => (0x00007fff701ff000)
libm.so.6 => /lib/libm.so.6 (0x00007f3f2dd94000)
libc.so.6 => /lib/libc.so.6 (0x00007f3f2da25000)
/lib64/ld-linux-x86-64.so.2 (0x00007f3f2e018000)
```

Static Techniques

- Gathering program information
 - library functions that were used
 - easy when program is dynamically linked
 - use `objdump` to find linked functions

```
$ objdump -R test
```

```
...
```

```
DYNAMIC RELOCATION RECORDS
```

OFFSET	TYPE	VALUE
0000000000601000	R_X86_64_JUMP_SLOT	printf
0000000000601008	R_X86_64_JUMP_SLOT	puts
0000000000601018	R_X86_64_JUMP_SLOT	strcmp

- more difficult when program is statically linked
- use function fingerprints
 - support through tools: IDA or dress

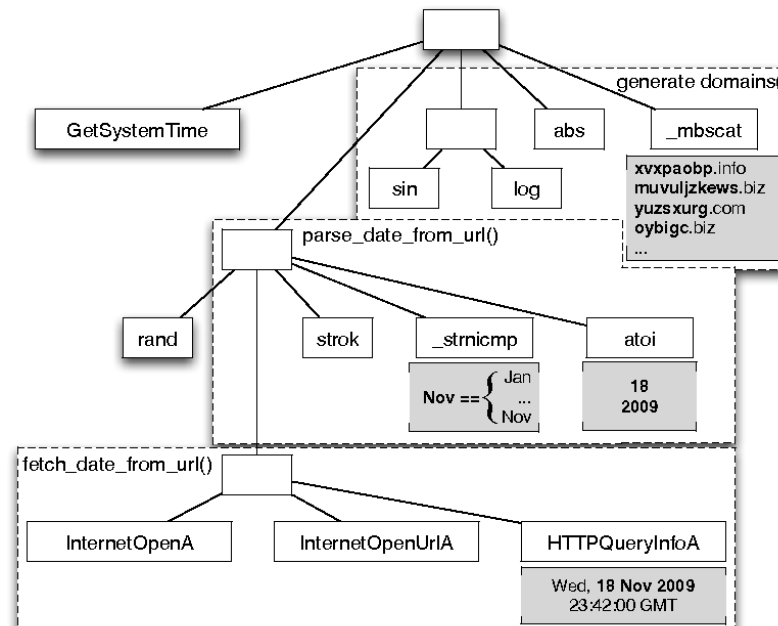
Static Techniques

*Int. Secure Systems Lab
Technical University Vienna*

- Gathering program information
 - program symbols
 - used for debugging (and linking)
 - function names (with start addresses)
 - global variables
 - can be removed with `strip`
 - use `nm` to display symbol information
 - function call trees
 - draw a graph that shows which function calls which other function
 - get an idea of program structure

Static Techniques

- Gathering program information
 - function call trees
 - Conficker.A domain name generation algorithm (DGA)



Static Techniques

*Int. Secure Systems Lab
Technical University Vienna*

- Disassembly
 - process of translating binary stream into machine instructions
- Different levels of difficulty
 - depending on ISA (instruction set architecture)
- Instructions can have
 - fixed length
 - more efficient to decode for processor
 - RISC processors (SPARC, MIPS)
 - variable length
 - use less space for common instructions
 - CISC processors (Intel x86)

Static Techniques

*Int. Secure Systems Lab
Technical University Vienna*

- Fixed length instructions
 - easy to disassemble
 - each address is a multiple of the instruction length
 - even if code contains data (or junk), all program instructions are found
- Variable length instructions
 - difficult to disassemble
 - start addresses of instructions not known in advance
 - disassembler can be desynchronized with respect to actual code
 - force disassembler to output incorrect instructions
 - [obfuscation attack](#)
 - different strategies
 - linear sweep disassembler (i.e. objdump)
 - recursive traversal disassembler (i.e. IDA Pro)

Intel x86 Assembler Primer

*Int. Secure Systems Lab
Technical University Vienna*

- Assembler Language
 - human-readable form of machine instructions
 - must understand the hardware architecture, memory model, and stack
- AT&T syntax vs. Intel syntax

AT&T vs. Intel Syntax

*Int. Secure Systems Lab
Technical University Vienna*

AT&T

mnemonic source(s), destination

Constants: prefixed with \$

Hexadecimal numbers: start with 0x

Registers: prefixed with %

Memory access is of form

`displacement(%base, %index, scale)`

where the result address is

`displacement + %base + %index*scale`

Intel

mnemonic destination, source(s)

No prefix

hexadecimal numbers: start with 0x

Registers: No prefix

Memory access is of form

`<size> [disp + index*4 + base]`

where the result address is

`disp + index*4 + base`

Example:

`dword [ebx + ecx*4 + mem_location]`

AT&T vs. Intel: Example

*Int. Secure Systems Lab
Technical University Vienna*

```
$ objdump -M att -d /bin/l
```

```
...
push %ebp
xor %ecx,%ecx
mov %esp,%ebp
sub $0x8,%esp
mov %ebx,(%esp)
mov 0x8(%ebp),%ebx
mov %esi,0x4(%esp)
mov 0xc(%ebp),%esi
mov (%ebx),%edx
mov 0x4(%ebx),%eax
xor 0x4(%esi),%eax
xor (%esi),%edx
or %edx,%eax
je 8049c60 <exit@plt+0x13c>
...
```

```
$ objdump -M intel -d /bin/l
```

```
...
push ebp
xor ecx,ecx
mov ebp,esp
sub esp,0x8
mov DWORD PTR [esp],ebx
mov ebx,DWORD PTR [ebp+0x8]
mov DWORD PTR [esp+0x4],esi
mov esi,DWORD PTR [ebp+0xc]
mov edx,DWORD PTR [ebx]
mov eax,DWORD PTR [ebx+0x4]
xor eax,DWORD PTR [esi+0x4]
xor edx,DWORD PTR [esi]
or eax,edx
je 8049c60 <exit@plt+0x13c>
...
```

Intel x86 Assembler Primer

*Int. Secure Systems Lab
Technical University Vienna*

- Important mnemonics (instructions)

<code>mov</code>	data transfer
<code>add / sub</code>	arithmetic
<code>cmp / test</code>	compare two values and set control flags
<code>je / jne</code>	conditional jump depending on control flags (branch)
<code>jmp</code>	unconditional jump

- Registers

- local variables of processor
- six 32-bit general purpose registers
 - can be used for calculations, temporary storage of values, ...
`%eax, %ebx, %ecx, %edx, %esi, %edi`
- several 32-bit special purpose registers
 - `%esp` - stack pointer
 - `%ebp` - frame pointer
 - `%eip` - instruction pointer

Intel x86 Assembler Primer

Int. Secure Systems Lab
Technical University Vienna

- Stack
 - managed by stack pointer (%esp) and frame pointer (%ebp)
 - used for
 - function arguments
 - function return address
 - local arguments
- Byte ordering
 - important for multi-byte values (e.g., four byte long value)
 - Intel uses *little endian* ordering
 - how to represent 0x11223344 in memory (at addr)?

```
0x010004 (addr)      : 0x44
0x010005 (addr+1)    : 0x33
0x010006 (addr+2)    : 0x22
0x010007 (addr+3)    : 0x11
```

Intel x86 Assembler Primer

*Int. Secure Systems Lab
Technical University Vienna*

- **if statement**

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int a;

    if(a < 0) {
        printf("A < 0\n");
    }
    else {
        printf("A >= 0\n");
    }
}
```

```
.LC0:
    .string "A < 0\n"

.LC1:
    .string "A >= 0\n"

.globl main
.type    main, @function

main:
    [ function prologue ]
    cmpl  $0, -4(%ebp) /* s = a - 0*/
    jns   .L2          /* if sign bit is not
                        set */

    movl  $.LC0, (%esp)
    call  printf
    jmp   .L3

.L2:
    movl  $.LC1, (%esp)
    call  printf

.L3:
    leave
    ret
```

Intel x86 Assembler Primer

*Int. Secure Systems Lab
Technical University Vienna*

- `while` statement

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int i;

    i = 0;
    while(i < 10)
    {
        printf("%d\n", i);
        i++;
    }
}
```

```
.LC0:
    .string "%d\n"

main:
    [ function prologue ]
    movl    $0, -4(%ebp)

.L2:
    cmpl   $9, -4(%ebp)
    jle    .L4
    jmp    .L3

.L4:
    movl   -4(%ebp), %eax
    movl   %eax, 4(%esp)
    movl   $.LC0, (%esp)
    call  printf
    leal  -4(%ebp), %eax
    incl  (%eax)
    jmp   .L2

.L3:
    leave
    ret
```

Intel x86 Assembler Primer

*Int. Secure Systems Lab
Technical University Vienna*

- `while` statement

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int i;

    i = 0;
    while(i < 10)
    {
        printf("%d\n", i);
        i++;
    }
}
```

```
.LC0:
    .string "%d\n"

main:
    [ function prologue ]
    movl    $0, 0xffffffff(%ebp)

.L2:
    cmpl    $9, 0xffffffff(%ebp)
    jle     .L4
    jmp     .L3

.L4:
    movl    0xffffffff(%ebp), %eax
    movl    %eax, 4(%esp)
    movl    $.LC0, (%esp)
    call   printf
    leal   0xffffffff(%ebp), %eax
    incl   (%eax)
    jmp    .L2

.L3:
    leave
    ret
```

Static Techniques

*Int. Secure Systems Lab
Technical University Vienna*

- Linear sweep disassembler
 - start at beginning of code (.text) section
 - disassemble one instruction after the other
 - assume that well-behaved compiler tightly packs instructions
 - `objdump -d` uses this approach
- Obfuscation Attack
 - insert data (or junk) between instructions and let control flow jump over this garbage
 - disassembler gets confused

```
jmp L1          | 4004cf:  eb 02          | jmp      4004d3
.short 0x4711   | 4004d1:  11 47          | <junk>
L1:
xor %eax, %eax | 4004d3:  31 c0          | xor     %eax,%eax
...            | 4004d5:  b8 00 00 00 00 | mov     $0x0,%eax
               | 4004da:  c9            | leave
ret            | 4004db:  c3            | ret
```

CORRECT

Static Techniques

*Int. Secure Systems Lab
Technical University Vienna*

- Linear sweep disassembler
 - start at beginning of code (.text) section
 - disassemble one instruction after the other
 - assume that well-behaved compiler tightly packs instructions
 - `objdump -d` uses this approach
- Obfuscation Attack
 - insert data (or junk) between instructions and let control flow jump over this garbage
 - disassembler gets confused

```
jmp L1          | 4004cf:  eb 02          | jmp      4004d3
.short 0x4711   | 4004d1:  11 47 31      | adc     %eax,0x31(%edi)
L1:            |
xor %eax, %eax | 4004d4:  c0 b8 00 00 00 00 c9 | sarb   $0xc9,0x0(%eax)
...           |
ret            | 4004db:  c3          | ret
```

Static Techniques

*Int. Secure Systems Lab
Technical University Vienna*

- Recursive traversal disassembler
 - aware of control flow
 - start at program entry point (e.g., determined by ELF header)
 - disassemble one instruction after the other, until branch or jump is found
 - recursively follow both (or single) branch (or jump) targets
 - not all code regions can be reached
 - indirect calls and indirect jumps
 - use a register to calculate target during run-time
 - for these regions, linear sweep is used
 - IDA Pro uses this approach

Static Techniques

*Int. Secure Systems Lab
Technical University Vienna*

- Recursive traversal disassembler
- Obfuscation Attack
 - plain previous attack fails
 - replace direct jumps (calls) by indirect ones
 - force disassembler to revert to linear sweep, and then use previous attack

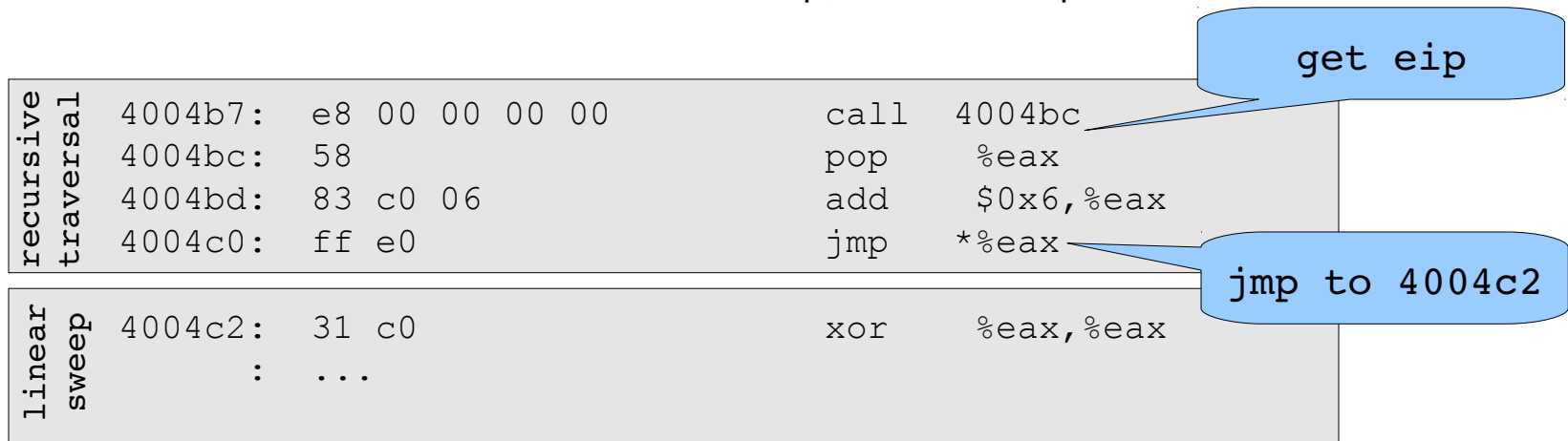
```
4004b7: e8 00 00 00 00    call 4004bc
4004bc: 58                pop    %eax
4004bd: 83 c0 06         add    $0x6,%eax
4004c0: ff e0            jmp    *%eax

4004c2: 31 c0            xor    %eax,%eax
: ...
```

Static Techniques

Int. Secure Systems Lab
Technical University Vienna

- Recursive traversal disassembler
- Obfuscation Attack
 - plain previous attack fails
 - replace direct jumps (calls) by indirect ones
 - force disassembler to revert to linear sweep, and then use previous attack



Reverse Engineering

Dynamic Techniques

Dynamic Techniques

Int. Secure Systems Lab
Technical University Vienna

- General information about process
 - /proc file system
 - /proc/<pid>/ for a process with pid <pid>
 - interesting entries
 - `cmdline` (show command line)
 - `environ` (show environment)
 - `maps` (show memory map, *remember this for the next challenge!!*)
 - `fd` (file descriptors held by program)
 - `exe` (program image)
- Interaction with the environment
 - file system
 - network

Dynamic Techniques

*Int. Secure Systems Lab
Technical University Vienna*

- File system interaction
 - `lsdf`
 - lists all open files associated with processes
- Registry (Windows)
 - `regmon` (Sysinternals)
- Network interaction
 - check for open ports
 - processes that listen for requests or that have active connections
 - `netstat`
 - also shows UNIX domain sockets used for IPC
 - check for actual network traffic
 - `tcpdump`
 - `wireshark`

Dynamic Techniques

*Int. Secure Systems Lab
Technical University Vienna*

- System calls
 - are at the boundary gates between user space and kernel
 - reveal much about a process' operation
 - `strace`
 - powerful tool that can also
 - follow child processes
 - decode more complex system call arguments
 - show signals
 - works via the `ptrace` interface
- Library functions
 - similar to system calls, but dynamically linked libraries
 - `ltrace`

Dynamic Techniques

- strace

```
$ strace echo "hi"
```

```
execve("/bin/echo", ["echo", "hi"], [/* 41 vars */) = 0
brk(0) = 0xddb000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE...) = 0x7f54eac10000
...
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or...)
open("/lib/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1490312, ...}) = 0
mmap(NULL, 3598344, PROT_READ|PROT_EXEC, ...) = 0x7f54ea684000
mprotect(0x7f54ea7ea000, 2093056, PROT_NONE) = 0
...
write(1, "hi\n", 3hi) = 3
close(1) = 0
munmap(0x7f54eaac1000, 4096) = 0
close(2) = 0
exit_group(0) = ?
```

Dynamic Techniques

*Int. Secure Systems Lab
Technical University Vienna*

- ltrace

```
$ ltrace echo "hi"
```

```
__libc_start_main(0x4013e0, 2, 0x7fffb3cfbe78, ...)
getenv("POSIXLY_CORRECT")           = NULL
strrchr("echo", '/')                 = NULL
setlocale(6, "")                     = "en_US.UTF-8"
bindtextdomain("coreutils", "/usr/share/locale") = "/usr/share/locale"
textdomain("coreutils")              = "coreutils"
...
fputs_unlocked(0x7fffb3cfc61e, 0x7f19cdc6a780, 0, 1, 0) = 1
...
fclose(0x7f19cdc6a860)               = 0
...
+++ exited (status 0) +++
```

Dynamic Techniques

*Int. Secure Systems Lab
Technical University Vienna*

- Execute program in a controlled environment
 - sandbox (virtual machine or emulator)
 - debugger
- Advantages
 - can inspect actual program behavior and data values
 - target of indirect jumps (or calls) can be observed
- Disadvantages
 - may accidentally launch attacks
 - anti-debugging mechanisms
 - not all possible traces (paths) can be seen

Dynamic Techniques

*Int. Secure Systems Lab
Technical University Vienna*

- Debugger
 - breakpoints to pause execution
 - when execution reaches a certain point (address)
 - when specified memory is access or modified
 - examine memory and CPU registers
 - modify memory and execution path
- Advanced features
 - attach comments to code
 - data structure and template naming
 - track high level logic
 - file descriptor tracking
 - function fingerprinting

Dynamic Techniques

*Int. Secure Systems Lab
Technical University Vienna*

- Debugger on x86 / Linux
 - use the `ptrace` interface
- `ptrace`
 - allows a process (parent) to monitor another process (child)
 - whenever the child process receives a signal, the parent is notified
 - parent can then
 - access and modify memory image (peek and poke commands)
 - access and modify registers
 - deliver signals
 - `ptrace` can also be used for system call monitoring

Dynamic Techniques

*Int. Secure Systems Lab
Technical University Vienna*

- Breakpoints
 - hardware breakpoints
 - software breakpoints
- Hardware breakpoints
 - special debug registers (e.g., Intel x86)
 - debug registers compared with PC at every instruction
- Software breakpoints
 - debugger inserts (overwrites) target address with an `int 0x03` instruction
 - interrupt causes signal SIGTRAP to be sent to process
 - debugger
 - gets control and restores original instruction
 - single steps to next instruction
 - re-inserts breakpoint

Dynamic Techniques

*Int. Secure Systems Lab
Technical University Vienna*

- Anti-debugging techniques

- detect tracing

- a process can be traced only once

```
if (ptrace(PTRACE_TRACEME, 0, 1, 0) < 0)
    exit(1);
```

- detect breakpoints

- look for int 0x03 instructions

```
if ((* (unsigned *) ((unsigned)<addr>+3) & 0xff) == 0xcc)
    exit(1);
```

Dynamic Techniques

- Anti-debugging techniques (cont.)

- checksum the code

```
if (checksum(text_segment) != valid_checksum)
    exit(1);
```

- register signal handler for debug interrupt

- force interrupt: parent will receive the signal

```
int dbg=1;
void my_handler(int signal) { dbg=0; };
int main(...) {
    signal(SIG_INT, my_handler);
    asm("int 0x03");
    if (dbg)
        exit(1);
}
```

Malicious Code Analysis

Malicious Code Analysis

*Int. Secure Systems Lab
Technical University Vienna*

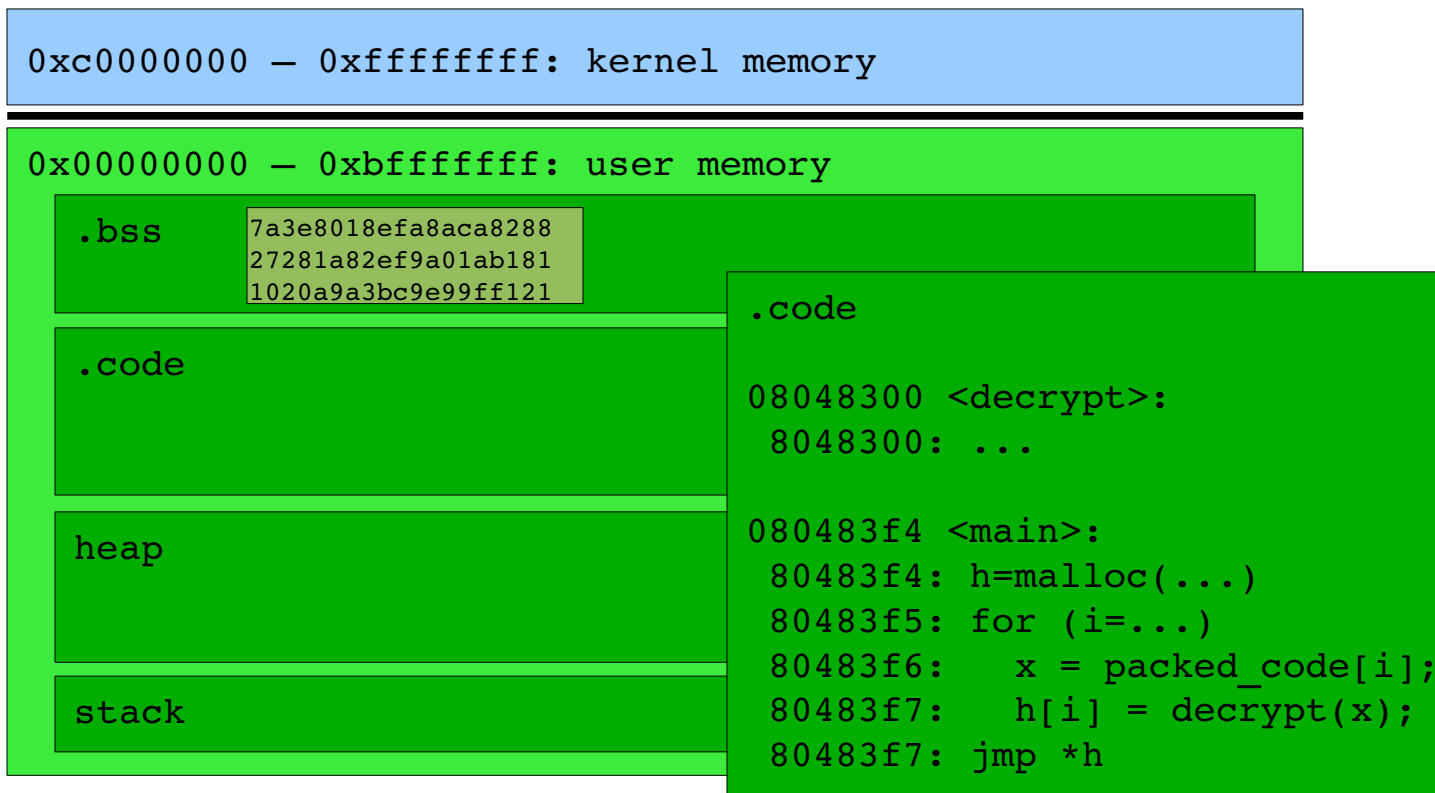
Static analysis vs. dynamic analysis

- Static analysis
 - code is not executed
 - all possible branches can be examined (in theory)
 - quite fast
- Problems of static analysis
 - binary code typically contains very little information
 - functions, variables, type information, ...
 - disassembly difficult (particularly for Intel x86 architecture)
 - obfuscated code
 - packed code, self-modifying code

Malicious Code Analysis

Int. Secure Systems Lab
Technical University Vienna

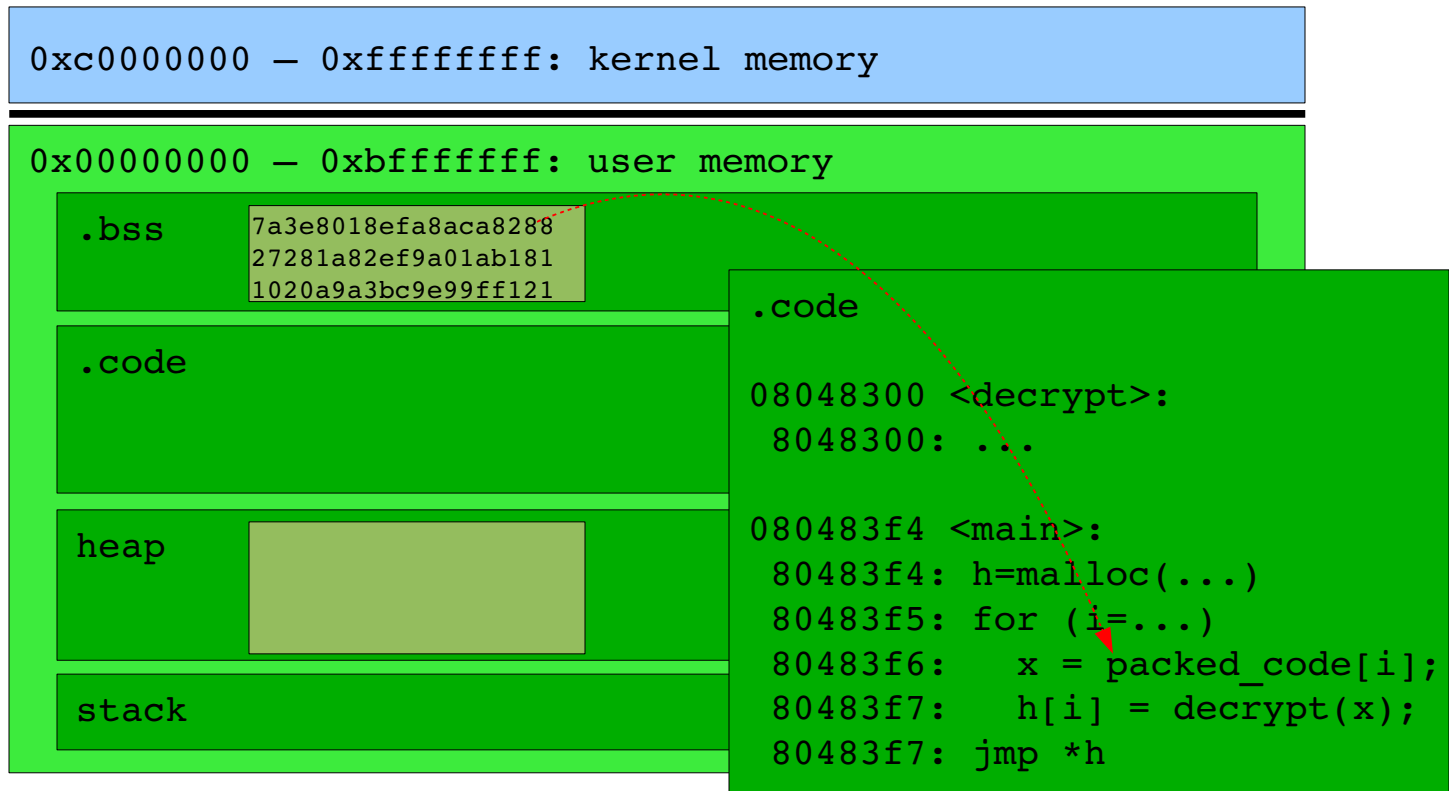
- Packed code (*dynamic unpacking*)



Malicious Code Analysis

Int. Secure Systems Lab
Technical University Vienna

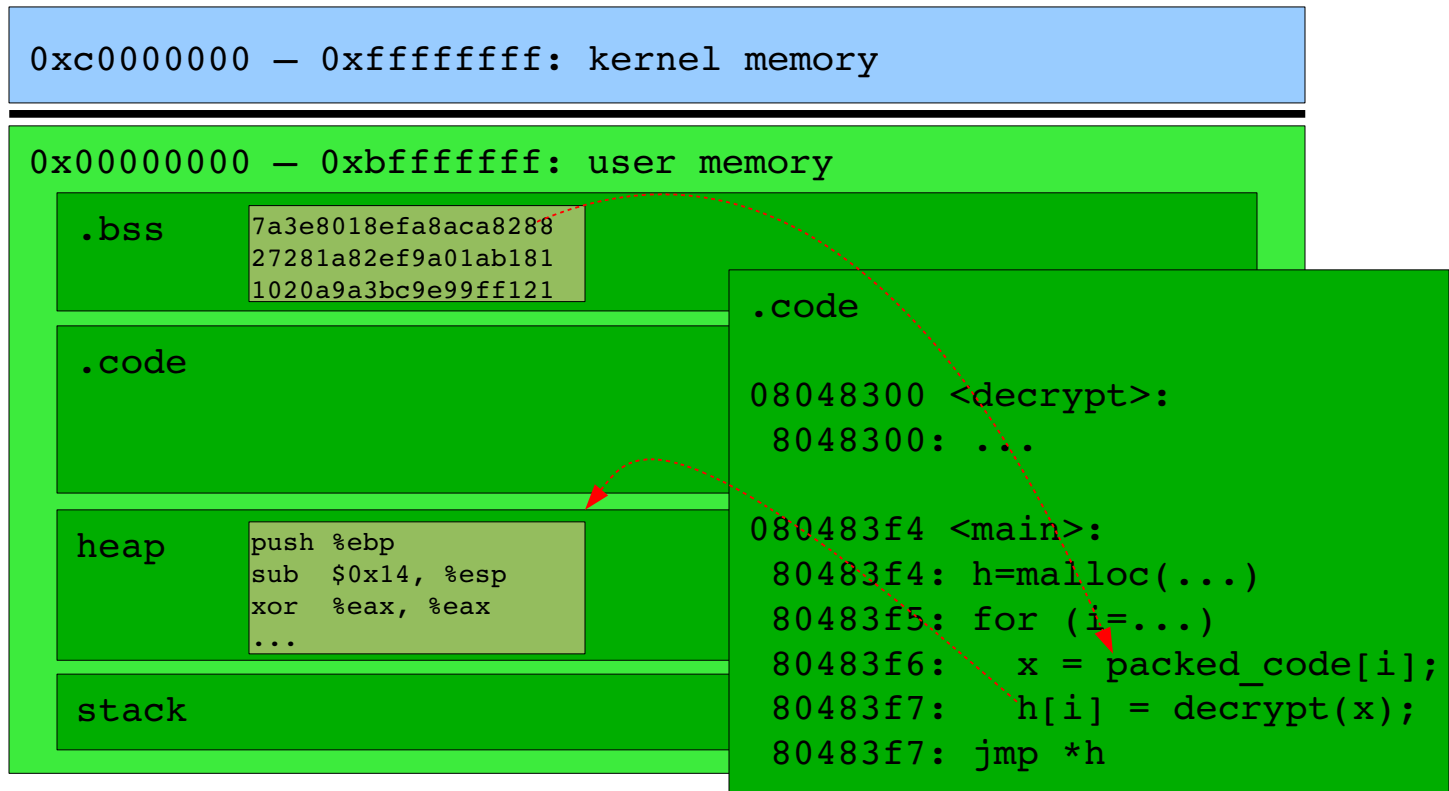
- Packed code (*dynamic unpacking*)



Malicious Code Analysis

Int. Secure Systems Lab
Technical University Vienna

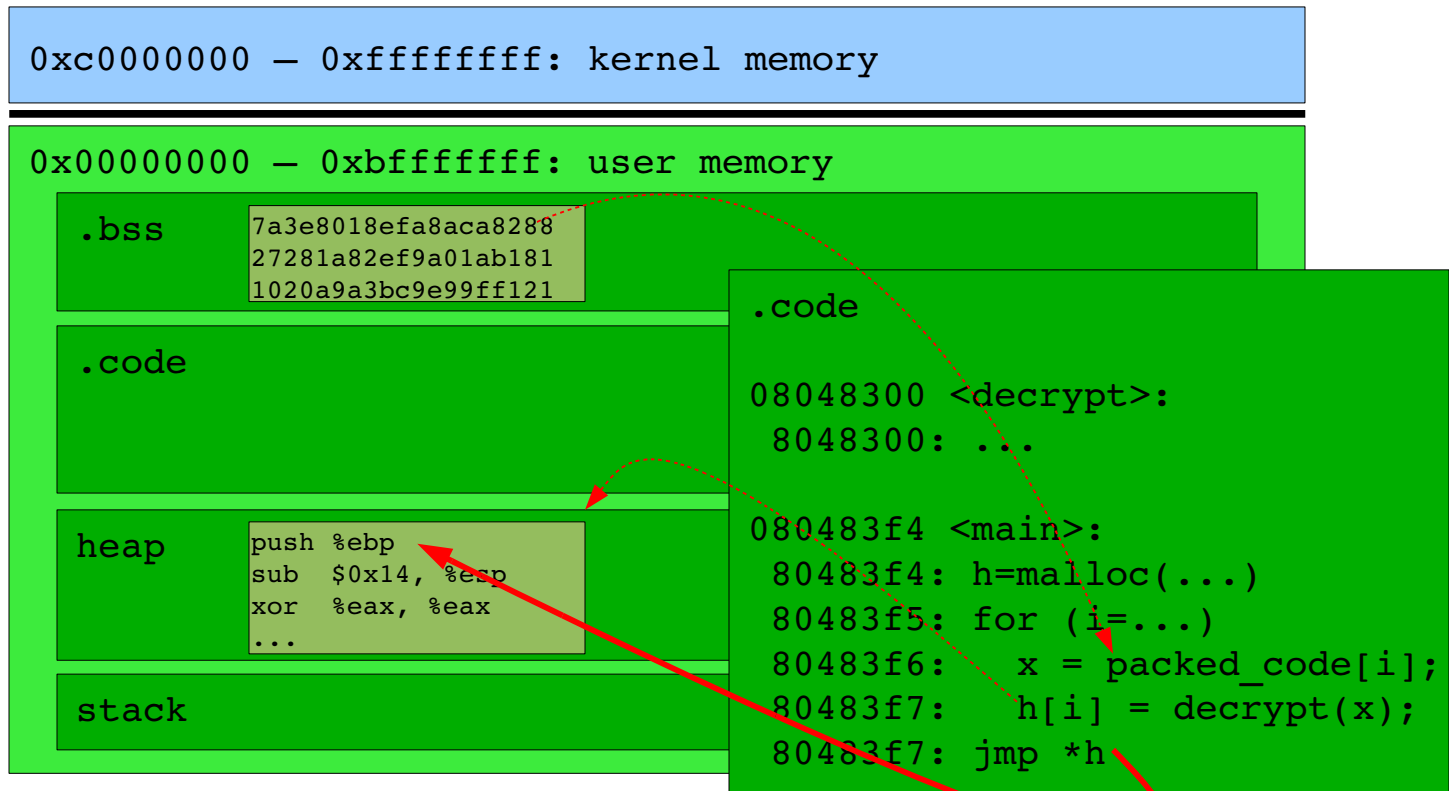
- Packed code (*dynamic unpacking*)



Malicious Code Analysis

Int. Secure Systems Lab
Technical University Vienna

- Packed code (*dynamic unpacking*)



Malicious Code Analysis

Int. Secure Systems Lab
Technical University Vienna

- Dynamic analysis
 - code is executed
 - sees instructions that are actually executed
- Problems of dynamic analysis
 - single path (execution trace) is examined
 - analysis environment possibly not *invisible*
 - analysis environment possibly not *comprehensive*
- Possible analysis environments
 - instrument program
 - instrument operating system
 - instrument hardware

Malicious Code Analysis

*Int. Secure Systems Lab
Technical University Vienna*

- Instrument program
 - analysis operates in same address space as sample
 - manual analysis with debugger
 - Detours (Windows API hooking mechanism)

 - binary under analysis is modified
 - breakpoints are inserted
 - functions are rewritten
 - debug registers are used
 - not invisible, malware can detect analysis
 - can cause significant manual effort

Malicious Code Analysis

*Int. Secure Systems Lab
Technical University Vienna*

- Instrument operating system
 - analysis operates in OS where sample is run
 - Windows system call hooks
 - invisible to (user-mode) malware
 - can cause problems when malware runs in OS kernel
 - limited visibility of activity inside program
 - cannot set function breakpoints

Malicious Code Analysis

*Int. Secure Systems Lab
Technical University Vienna*

- Instrument hardware
 - provide virtual hardware (processor) where sample can execute (sometimes including OS)
 - software emulation of executed instructions
 - analysis observes activity “from the outside”
 - completely transparent to sample (and guest OS)
 - operating system environment needs to be provided
 - Anubis uses this approach: <http://anubis.iseclab.org>

Analysis Report

Int. Secure Systems Lab
Technical University Vienna

- File activity
 - read, write, create, open, ...
- Registry activity
- Service activity
 - start or stop of Windows services (via Service Manager)
- Process activity
 - start, terminate process, inter-process communication
- Network activity
 - API calls and packet (network) logs

Stealth

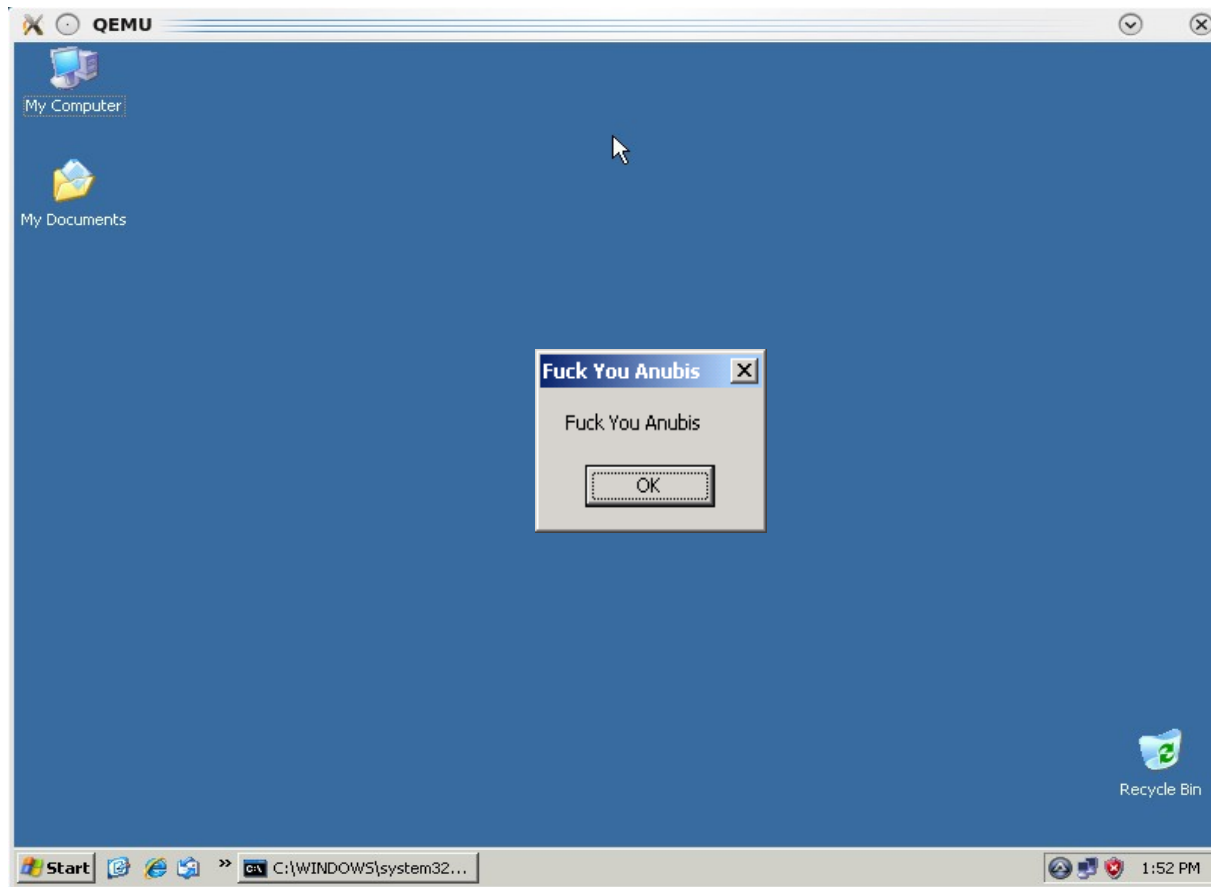
*Int. Secure Systems Lab
Technical University Vienna*

- Virtual machines
 - allow to quickly restore analysis environment
 - identical, clean environment for every analysis run
 - introduces detectable artifacts
- Some detection mechanisms (we have seen)
 - x86 virtualization problems
 - speed of execution
 - check system/installation specific settings
 - computer name, drive label, external IP address, etc.

Stealth

*Int. Secure Systems Lab
Technical University Vienna*

```
$ ./analyze.py --show-window ~/anti_anubis.exe
```



Summary

*Int. Secure Systems Lab
Technical University Vienna*

- Software reverse engineering
 - static & dynamic techniques
- Static techniques
 - check for strings, symbols, and library functions
 - disassembler
- Dynamic techniques
 - system/API call monitoring (`ptrace/ltrace` interface)
 - monitor network and file system activity
 - debugger
- Malicious code analysis

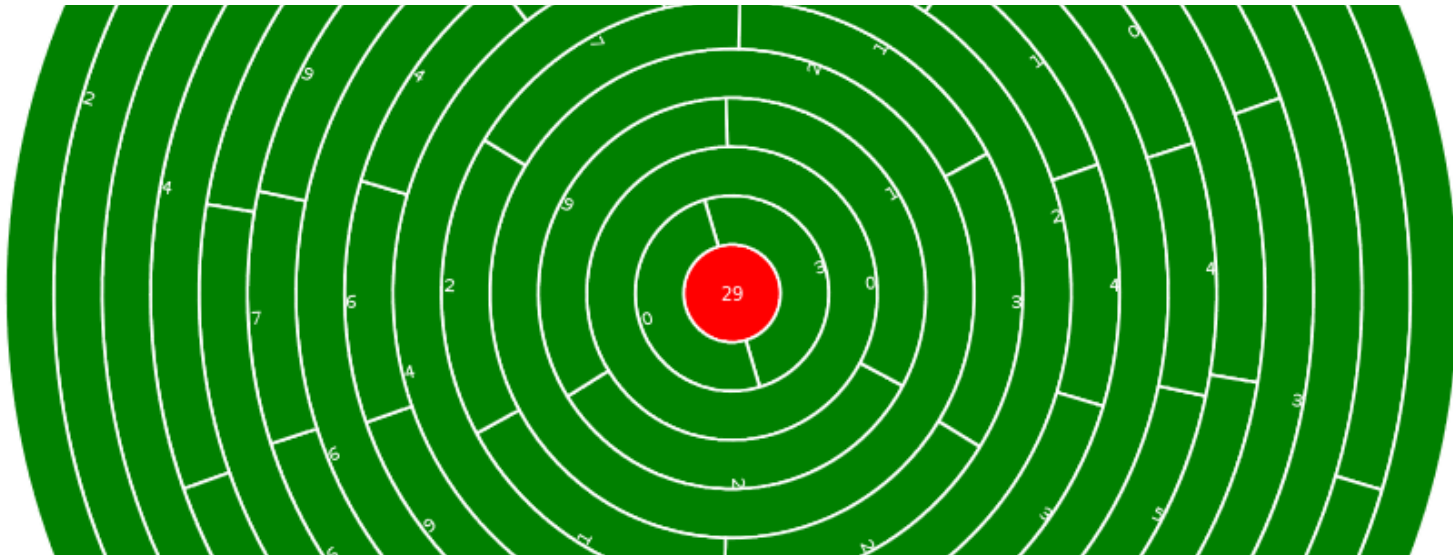
ICTF

Challenge 29

ICTF Challenge 29

*Int. Secure Systems Lab
Technical University Vienna*

- Optional, not relevant for the exam
- Reverse-engineering challenge, 800 points



Reverse1.exe

*Int. Secure Systems Lab
Technical University Vienna*

- User needs to enter 24-byte secret hex key so that the application reveals bank account number
- Bank account number is encrypted (XOR) with secret key
- Win32 PE binary
- Anti-Debugging
- Code obfuscation
- We used IDA Pro to reverse engineer it

Anti-Debug Code (1)

Int. Secure Systems Lab
Technical University Vienna

```
.text:004010AA  
.text:004010AA loc_4010AA: ; CODE XREF: sub_401000+65↑j  
.text:004010AA ; sub_401000:loc_40109C↑j  
.text:004010AA lea    eax, [ebp+timestamp_var_40]  
.text:004010AD push   eax  
.text:004010AE lea    eax, [ebp+timestamp_var_64]  
.text:004010B1 push   eax  
.text:004010B2 call   RTSC_anti_debug_sub_40134A ; get pesky realtime timestamp  
.text:004010B7 pop    ecx  
.text:004010B8 pop    ecx  
.text:004010B9 push   ds:GetProcAddress  
.text:004010BF push   ds:LoadLibraryW  
.text:004010C5 push   [ebp+kernel32_dll_var_34]  
.text:004010C8 push   0A36DC676h  
.text:004010CD call   get_kernel32_IsDebuggerPresent_sub_4012AB  
.text:004010D2 add    esp, 10h  
.text:004010D5 mov    [ebp+IsDebuggerPresent_var_5C], eax  
.text:004010D8 cmp    [ebp+var_3C], 30h  
.text:004010DC jnz   short loc_4010E5  
.text:004010DE call   [ebp+IsDebuggerPresent_var_5C]  
.text:004010E1 test   eax, eax  
.text:004010E3 jle   short loc_4010F9
```

Anti-Debug Code (2)

*Int. Secure Systems Lab
Technical University Vienna*

- Program stops if `isDebuggerPresent()` returns true
- Program measures execution time in CPU ticks
 - If execution is slow (e.g. single-stepping in debugger), it will loop for ages
 - Number of iterations depends on tick-count

Get Around Anti-Debug Tricks

*Int. Secure Systems Lab
Technical University Vienna*

- We could patch the code
 - Means code-modification
 - Program could detect it or behave differently
- Easy and more stealthy way:
 - Use IDAStealth plugin
 - Circumvents common anti-debug tricks
 - Hooks `isDebuggerPresent()`
 - Overwrites tick-count register value (i.e. RDTSC return value)

Code Obfuscation (1)

*Int. Secure Systems Lab
Technical University Vienna*

- Code jumps into custom code-segment (.ck)
- Obfuscation code-blocks:
 - Each block contains the key for the next block
 - Custom encryption/decryption function for each block (XOR)
 - Once decrypted, jumps into next code block

Code Obfuscation (2)

Int. Secure Systems Lab
Technical University Vienna

```
.ck:00404305
.ck:00404305 ; Attributes: noreturn
.ck:00404305
.ck:00404305 sub_404305 proc near ; CODE XREF: .ck:004042F0↑p
.ck:00404305 pop esi ; --> pop off address of key (pushed by call instruction)
.ck:00404306 mov ebx, esi ; ebx --> start of key
.ck:00404308 add esi, 62h ; add 0x62, esi --> start of xored block
.ck:00404308 ; that we want to decrypt
.ck:00404308 call xor_decrypt_sub_40433E ; call de-xor function (ebx=key addr, esi=xored block)
.ck:00404310 push esi ; esi=start of now dexored block
.ck:00404311 push edi
.ck:00404312 mov esi, [ebp+8] ; esi = ptr to given key
.ck:00404315 lea edi, [ebp-18h] ; dest addr
.ck:00404318 mov ecx, 24 ; copy length
.ck:0040431D rep movsb ; copy given key
.ck:0040431F pop edi
.ck:00404320 pop esi
.ck:00404321 mov eax, [ebp+0Ch]
.ck:00404324 push eax
.ck:00404325 lea eax, [ebp-18h] ; ptr to given key
.ck:00404328 push eax
.ck:00404329 call sub_404357 ; jump to dexored code
.ck:00404329 sub_404305 endp
.ck:00404329
```

Code Obfuscation Ideas

*Int. Secure Systems Lab
Technical University Vienna*

- Set breakpoint at point where all decryption is finished ?
 - Each code block encrypts itself at return
- Single-step ?
 - Numerous Code-Blocks
 - Will take ages
- Set SW Breakpoints ?
 - Will lead to wrong code as INT3 instruction will be XORed

Solution (1)

- Each block has the same structure and length
- We can use hardware breakpoints (exec)
- Let's have a look at the calls to the subsequent blocks:
 - First block: `.ck:00404329 call sub_404357`
 - Second Block: `.ck:0040439A call sub_4043C8`
 - Third Block: `.ck:0040440B call sub_404439`
- The distance between two encrypted blocks is always `0x71` bytes !

Solution (2)

Int. Secure Systems Lab
Technical University Vienna

- We can set a HW breakpoint for the call to the N-th block
- Start binary search
 - i.e. 200th block is reached, but 300th block isn't
 - Solution needs to be between 200th and 299th block
 - Continue approach
- Result:
 - there are 255 encryption layers
 - Almost like peeling an onion

Solution (3)

- The 256th block contains the actual function

```
.ck:0040B3E3 ; FUNCTION CHUNK AT .ck:0040B000 SIZE 00000002 BYTES
.ck:0040B3E3 ; FUNCTION CHUNK AT .ck:0040B606 SIZE 00000221 BYTES
.ck:0040B3E3
.ck:0040B3E3 push    ebp
.ck:0040B3E4 mov     ebp, esp
.ck:0040B3E6 mov     eax, [ebp+8] ; ptr to given key
.ck:0040B3E9 cmp     dword ptr [eax], 27149A52h ; check if the first dword is 0x27149a52
.ck:0040B3EF jnz     short loc_40B427 ; return if wrong
.ck:0040B3F1 cmp     dword ptr [eax+4], 8124A07h ; next dword
.ck:0040B3F8 jnz     short loc_40B427 ; and so on ...
.ck:0040B3FA cmp     dword ptr [eax+8], 451CCAB9h
.ck:0040B401 jnz     short loc_40B427
.ck:0040B403 cmp     dword ptr [eax+0Ch], 91827364h
.ck:0040B40A jnz     short loc_40B427
.ck:0040B40C cmp     dword ptr [eax+10h], 45913FDCh
.ck:0040B413 jnz     short loc_40B427
.ck:0040B413 ; -----
.ck:0040B415 db     81h ; ü
.ck:0040B416 db     78h ; x
```

- The key is

529a1427074a1208b9ca1c4564738291dc3f91455763ccb1

We Got It !

Int. Secure Systems Lab
Technical University Vienna

```
C:\WINDOWS\system32\cmd.exe

Directory of C:\Documents and Settings\mne\Desktop\ictf2011 challenge29

12/21/2011  06:41 PM    <DIR>          .
12/21/2011  06:41 PM    <DIR>          ..
12/03/2011  02:34 PM           1,677,947 ida-20111203-143405-1244.dmp
12/03/2011  02:34 PM           1,676,559 ida-20111203-143435-1244.dmp
12/03/2011  02:34 PM           1,676,305 ida-20111203-143444-1244.dmp
12/03/2011  02:35 PM           1,675,325 ida-20111203-143514-348.dmp
12/03/2011  02:35 PM           1,672,105 ida-20111203-143524-348.dmp
12/02/2011  05:01 PM              50,176 reverse1.exe
12/21/2011  06:41 PM           5,013,504 reverse1.id0
12/21/2011  06:41 PM          76,242,944 reverse1.id1
12/04/2011  02:08 PM          81,150,120 reverse1.idb
12/21/2011  06:41 PM              49,152 reverse1.nam
12/21/2011  06:41 PM              84 reverse1.til
12/03/2011  02:32 PM              2,000 test.py
12/04/2011  02:02 PM              293 teststring.txt.txt
                13 File(s)          170,886,514 bytes
                2 Dir(s)    11,809,775,616 bytes free

C:\Documents and Settings\mne\Desktop\ictf2011 challenge29>reverse1.exe
Enter key: 529a1427074a1208b9ca1c4564738291dc3f91455763ccb1
Bank account: 2526390575284-60846167886
C:\Documents and Settings\mne\Desktop\ictf2011 challenge29>
```

That's all folks

*Int. Secure Systems Lab
Technical University Vienna*

- Thank you for your attention
- Merry Christmas and a happy new year !