
Secure Programming II (SecProg 2)

Engin Kirda

kirda@eurecom.fr

Administrative Issues

- Challenge 6 has been announced
 - Not the easiest thing
 - After the lecture today, things should become clearer
- Challenge 7 and 8 to be announced after the holidays
 - A worm or a virus
 - Forensics
- We have an invited talk on the 8th of January
 - Gilbert Wondracek, Underground Economy

Malicious Code (Part 1)

Overview

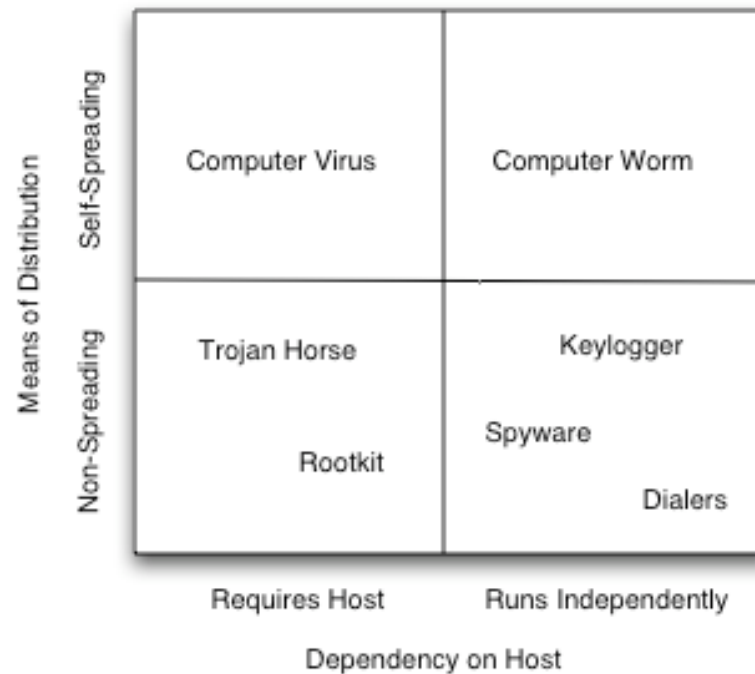
- Introduction to malicious code
 - taxonomy, history, life cycle
- Virus
 - infection strategies, armored viruses, detection
- Worms (in some other lecture)
 - email- and exploit-based worms, spreading strategies
- Trojan horses
 - key logger, rootkits, botnet, spyware

Introduction

- Malicious Code (Malware)
 - software that fulfills malicious intent of author
 - term often used equivalent with virus (due to media coverage)
 - however, many different types exist
 - classic viruses account for only 3% of malware in the wild
- Virus - Definition

A virus is a program that reproduces its own code by attaching itself to other executable files in such a way that the virus code is executed when the infected executable file is executed

Taxonomy



Taxonomy

- Virus
 - self-replicating, infects files (thus requires host)
- Worm
 - self-replicating, spreads over network
- Interaction-based worms (B[e]agle, Netsky, Sobig)
 - spread requires human interaction
 - double-click and execute extension
 - follow link to download executable
- Process-based worms (Code Red, Blaster, Slammer)
 - requires no human interaction
 - exploits vulnerability in network service

A Short History

- 1983 – The first documented experimental virus
 - Fred Cohen's pioneering work; name *coined* by Len Adleman
- 1987 - File infectors
 - Christmas worm hit IBM Mainframes (500,000 replications / hour)
- 1988 - Internet worm
 - Internet worm created by Robert Morris (CERT is created)
- 1991 - First polymorphic virus
- 1995 - First macro virus
- 1999 - Melissa worm (first large scale email virus)

A Short History

- 1999 – Distributed denial of service (DOS)
- 1999 - Kernel Rootkits
 - Knark (modification of system call table)
- 2000 - Spyware
- 2001 - Code Red
 - first large-scale, exploit-based worm
- 2003 - SQL Slammer worm
 - extremely fast propagation
- 2004> - Botnets (e.g., Storm Worm)

Reasons for Malware Prevalence

- Mixing data and code
 - violates important design property of secure systems
 - unfortunately very frequent
- Homogeneous computing base
 - Windows is just a very tempting target
- Unprecedented connectivity
 - easy to attack from safety of home
- Clueless user base
 - many targets available
- Malicious code has become profitable
 - compromised computers can be sold (e.g., spam relay, DoS)

Virus Lifecycle

- Lifecycle
 - reproduce, infect, run payload
- Reproduction phase
 - viruses balance infection versus detection possibility
 - variety of techniques may be used to hide viruses
- Infection phase
 - difficult to predict when infection will take place
 - many viruses stay resident in memory (TSR or process)
- Attack phase
 - e.g., deleting files, changing random data on disk
 - viruses often have bugs (poor coding) so damage can be done
 - Stoned virus expected 360K, floppy, corrupted sectors

Infection Strategies

- Boot viruses
 - master boot record (MBR) of hard disk (first sector on disk)
 - boot sector of partitions
 - e.g., Pakistani Brain virus
 - rather old, but interest is growing again
 - diskless work stations, virtual machine virus (SubVirt)
- File infectors
 - simple overwrite virus (damages original program)
 - parasitic virus
 - append virus code and modify program entry point
 - cavity virus
 - inject code into unused regions of program code

Infection Strategies

- Entry Point Obfuscation
 - virus scanners quickly discovered to search around entry point
 - virus hijacks control later (after program is launched)
 - overwrite import table addresses
 - overwrite function call instructions
- Code Integration
 - merge virus code with program
 - requires disassembly of target
 - difficult task on x86 machines
 - W95/Zmist is a classic example for this technique

Macro Viruses

- Many modern applications support macro languages
 - Microsoft Word, Excel, Outlook
 - macro language is powerful
 - embedded macros automatically executed on load
 - mail app. with Word as an editor
 - mail app. with Internet Explorer to render HTML

The screenshot shows a configuration window for a macro virus. At the top, there are menu items: **•About•**, **•Greets•**, **•Contact•**, **•Min•**, and **•Exit•**. The main content area contains several fields and options:

- Name of Author (Your name):** [Text input field]
- Name of Virus:** [Text input field]
- Special comments, shout outs:** [Text input field]
- Origin (The country you are in):** [Text input field]
- When would you like the infection to take place?**
 - On Open
 - On New
 - On Close
- When would you like the Message Box to be displayed?**
 - On Open
 - On New
 - On Close
- Where would you like the Virus to be created?**
 - On Desktop
 - In Current Directory
- Click here to create your Virus**
- Create•**

Companion Virus

- Companion virus
 - installs a COM file (the virus) for every EXE file found
 - idea is simple: DOS runs COM files before EXE
 - virus will stay memory resident and execute the original file
 - easy to find and eliminate

NTFS ADS Viruses

- NTFS contains a system called Alternate Data Streams (ADS)
 - sometimes used by viruses
 - original intention of ADS is to store meta information with file e.g., has it been downloaded from the Internet?

```
echo 'Hello World' > test.txt  
echo 'This is Hidden' > test.txt:hidden.txt  
notepad test.txt:hidden.txt
```

- Stream we have created is completely invisible
 - most commands do not work on ADSs (e.g., deleting).
 - Explorer and dir will not show the file
 - viruses can make use of ADS to hide code, data, temporary files
 - tool called *streams.exe* from Sysinternals.com is useful for finding such streams

Fast and Slow Infectors

- A fast infector infects any file accessed
 - purpose of fast infection is to ride on the back of anti-virus software
 - infect files as they are being checked
 - can be defeated if the scanner is started from a floppy
- A slow infector only infects files as they are created or modified
 - purpose of slow infection is to attempt to defeat integrity checking
 - piggyback on top of the process which legitimately changes a file
 - if integrity checker has a scanning component, virus can be caught

Virus Defense

- Antivirus Software
 - working horse is signature based detection
 - database of byte-level or instruction-level signatures that match virus
 - wildcards can be used, regular expressions
 - heuristics (check for signs of infection)
 - code execution starts in last section
 - incorrect header size in PE header
 - suspicious code section name
 - patched import address table
- Sandboxing
 - run untrusted applications in restricted environment
 - simplest variation, do not run as Administrator

Tunneling and Camouflage Viruses

- To minimize the probability of its being discovered, a virus could use a number of different techniques
- A tunneling virus attempts to bypass antivirus programs
 - idea is to follow the interrupt chain back down to basic operating system or BIOS interrupt handlers
 - install virus there
 - virus is “underneath” everything – including the checking program
- In the past, possible for a virus to spoof a scanner by camouflaging itself to look like something the scanner was programmed to ignore
 - false alarms of scanners make “ignore” rules necessary

Sparse Infectors and Armored Viruses

- Sparse infector
 - infect every n^{th} time a file is executed
 - infect files only with a certain name
- Armored virus
 - aims to make disassembly difficult
 - exploits fact that x86 code is hard to disassemble
 - Whale (early virus), made extensive use of such techniques
 - manual disassembly is almost always possible but takes more time and is not automated

Polymorphism and Metamorphism

- Polymorphic viruses
 - change layout (shape) with each infection
 - payload is encrypted
 - using different key for each infection
 - makes static string analysis practically impossible
 - of course, encryption routine must be changed as well
 - otherwise, detection is trivial
- Metamorphic techniques
 - create different “versions” of code that look different but have the same semantics (i.e., do the same)

Chernobyl (CIH) Virus

5B 00 00 00 00	pop ebx
8D 4B 42	lea ecx, [ebx + 42h]
51	push ecx
50	push eax
50	push eax
0F 01 4C 24 FE	sidt [esp - 02h]
5B	pop ebx
83 C3 1C	add ebx, 1Ch
FA	cli
8B 2B	mov ebp, [ebx]

```
5B 00 00 00 00 8D 4B 42 51 50 50 0F 01 4C 24 FE 5B
83 C3 1C FA 8B 2B
```

Dead Code Insertion

```
5B 00 00 00 00    pop ebx
8D 4B 42          lea ecx, [ebx + 42h]
51               push ecx
50               push eax
90               nop
50               push eax
40               inc eax
0F 01 4C 24 FE    sidt [esp - 02h]
48               dec eax
5B               pop ebx
83 C3 1C          add ebx, 1Ch
FA               cli
8B 2B            mov ebp, [ebx]
```

```
5B 00 00 00 00 8D 4B 42 51 50 90 50 40 0F 01 4C 24
FE 48 5B 83 C3 1C FA 8B 2B
```

Instruction Reordering

5B 00 00 00 00	pop ebx
EB 09	jmp <S1>
S2:	
50	push eax
0F 01 4C 24 FE	sidt [esp - 02h]
5B	pop ebx
EB 07	jmp <S3>
S1:	
8D 4B 42	lea ecx, [ebx + 42h]
51	push ecx
50	push eax
EB F0	jmp <S2>
S3:	
83 C3 1C	add ebx, 1Ch
FA	cli
8B 2B	mov ebp, [ebx]

```
5B 00 00 00 00 EB 09 50 0F 01 4C 24 FE 5B EB 07 8D
4B 42 51 50 EB F0 83 C3 1C FA 8B 2B
```

Instruction Substitution

```
5B 00 00 00 00    pop ebx
8D 4B 42          lea ecx, [ebx + 42h]
51               push ecx
89 04 24          mov eax, [esp]
83 C4 04          add 04h, esp
50               push eax
0F 01 4C 24 FE    sidt [esp - 02h]
83 04 24 0C       add 1Ch, [esp]
5B               pop ebx
8B 2B            mov ebp, [ebx]
```

```
5B 00 00 00 00 8D 4B 42 51 89 04 24 83 C4 04 50 0F
01 4C 24 FE 83 04 24 0C 5B 8B 2B
```

Advanced Virus Defense

- Most virus techniques very effective against static analysis
- Thus, dynamic analysis techniques introduced
 - virus scanner equipped with emulation engine
 - executes actual instructions (no disassembly problems)
 - runs until polymorphic part unpacks actual virus
 - then, signature matching can be applied
 - emulation must be fast
 - ANUBIS
- Difficulties
 - virus can attempt to detect emulation engine
 - time execution, use exotic (unsupported) instructions, ...
 - insert useless instructions in the beginning of code to deceive scanner

Virus Naming

- Virus writers would like to name themselves
 - this is of course not possible 😊
 - Netsky --> Skynet discussion
- The first identifiers of a virus also get to name it
 - typically, this is a research in an antivirus company
 - however, people work in parallel...
- Each anti-virus company has its own notation
 - causes confusion and often multiple names
 - attempts to create a unique naming and identification notation

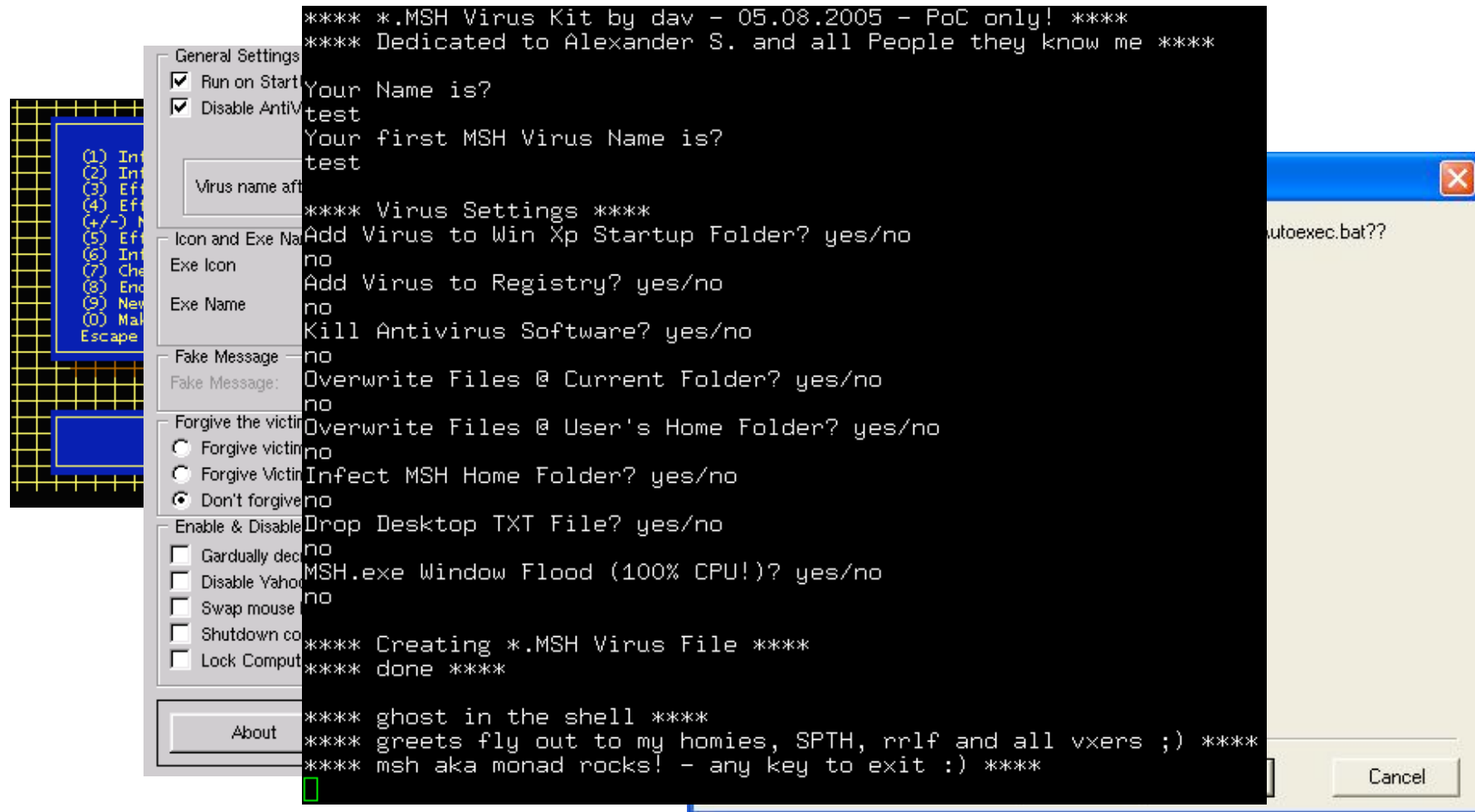
Number of Viruses

- More MS-DOS/Windows viruses than all other types
- Numbers are growing
 - 1991 - 600 to 1,000 viruses
 - 1996 - more than 10,000
 - 2000 - more than 50,000
 - 2005 - more than 125,000
- Only a small percentage is active in the wild
- Particular strong growth in recent years
 - problems with size of signature database
 - incremental updates are necessary
- No consensus on what a “new” virus is

How Serious a Threat are Viruses?

- Important to keep virus threat in perspective
 - antivirus industry is also a business
 - only a small fraction found in the wild
 - according to some, the chance that a cup of coffee will do more damage to your computer is higher
 - probably not true, however ...
 - ... there other threats besides classic file infecting viruses that are more problematic nowadays (worms, Trojan horses)
 - one should also consider the existence of toolkits to generate viruses
 - makes entrance easy for script kiddies

Virus Generation Toolkits



The image shows a screenshot of a virus generation toolkit interface. On the left, there is a vertical menu with numbered options: (1) Int, (2) Int, (3) Ef, (4) Ef, (5) Ef, (6) Int, (7) Che, (8) Enc, (9) Nev, (10) Mal, and an Escape key. The main window is titled "General Settings" and contains several sections:

- General Settings:** Run on Start, Disable AntiV
- Virus name aff:** A text input field containing "test".
- Icon and Exe Na:** Exe Icon, Exe Name
- Fake Message:** Fake Message: (empty)
- Forgive the victi:** Forgive victim, Forgive Victim, Don't forgive
- Enable & Disable:** G ardually dec, Disable Yahoo, Swap mouse, Shutdown co, Lock Comput
- About:** (empty)

The central command prompt window displays the following text:

```
**** *.MSH Virus Kit by dav - 05.08.2005 - PoC only! ****
**** Dedicated to Alexander S. and all People they know me ****
Your Name is?
test
Your first MSH Virus Name is?
test
**** Virus Settings ****
Add Virus to Win Xp Startup Folder? yes/no
no
Add Virus to Registry? yes/no
no
Kill Antivirus Software? yes/no
no
Overwrite Files @ Current Folder? yes/no
no
Overwrite Files @ User's Home Folder? yes/no
no
Forgive victim? yes/no
no
Forgive Victim? yes/no
no
Infect MSH Home Folder? yes/no
no
Drop Desktop TXT File? yes/no
no
MSH.exe Window Flood (100% CPU!)? yes/no
no
**** Creating *.MSH Virus File ****
**** done ****
**** ghost in the shell ****
**** greets fly out to my homies, SPTH, rrlf and all vxers ;) ****
**** msh aka monad rocks! - any key to exit :) ****
```

On the right, a smaller dialog box titled "autoexec.bat??" is visible with a "Cancel" button.

Computer Worms

A self-replicating program able to propagate itself across networks, typically having a detrimental effect.

(Oxford English Dictionary)

- Worms either
 - exploit vulnerabilities that affect large number of hosts
 - send copies of worm body via email
- Difference to classic virus is *autonomous* spread over network
- Speed of spreading is constantly increasing
- Make use of techniques known by virus writers for long time

Reverse Engineering

Introduction

- 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

- Application areas
 - copy (steal) technology
 - allow for interoperability
 - Samba (SMB protocol)
 - WINE (Windows API)
 - OpenOffice (MS Office)
 - circumvent copy protection or access restrictions
 - program cracking
- Techniques
 - static approaches
 - dynamic approaches

Reverse Engineering

- 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

Static Techniques

- Gathering program information
 - strings that the binary contains
 - `strings` command
 - library functions that are used
 - easy when program is dynamically linked
 - use `ldd`

```
[ek@adana]$ ldd /bin/bash
linux-gate.so.1 => (0xffffe000)
libdl.so.2 => /lib/libdl.so.2 (0x40021000)
libc.so.6 => /lib/libc.so.6 (0x40024000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```
 - more difficult when program is statically linked
 - use function fingerprints

Static Techniques

- 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

- Disassembly
 - process of translating binary stream into machine instructions
- Different level 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

- Fixed length instructions
 - easy to disassemble
 - take each address that is multiple of instruction length as instruction start
 - 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
 - recursive traversal disassembler

Intel x86 Assembler Primer

- Assembler Language
 - human-readable form of machine instructions
 - must understand the hardware architecture, memory model, and stack
- AT&T syntax
 - `mnemonic source(s), destination`
 - standalone numerical constants are prefixed with a `$`
 - hexadecimal numbers start with `0x`
 - registers are specified with `%`
 - memory access is of form `displacement(%base, %index, scale)`
where the result address is `displacement + %base + %index*scale`

Intel x86 Assembler Primer

- Important mnemonics (instructions)

 - `mov` data transfer

 - `add / sub` arithmetic

 - `cmp / test` compare two values and set control flags

 - `je / jne` conditional jump depending on control flags (branch)

 - `jmp` 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

- 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 0x03020100 in memory?

0x040	0
0x041	1
0x042	2
0x043	3

Intel x86 Assembler Primer

- 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

- 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
```

Static Techniques

- 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 Label1          | 8048000: 74 02          | je 8048004
.short 0x4711       | 8048002: 47             | inc    %edi
                    | 8048003: 11 90 90 90 90 90 | adc %edx,0x90909090(%eax)
Label1:             | 8048004:          | <Label1>
```

Static Techniques

- 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
- 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

Dynamic Techniques

- 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)
 - `fd` (file descriptor to program image)
- Interaction with the environment
 - file system
 - network

Dynamic Techniques

- File system interaction
 - `lsdf`
 - lists all open files associated with processes
- Windows Registry
 - `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

- System calls
 - are at the boundary 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

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

Dynamic Techniques

- 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

- 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

- 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

- 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);
```

- checksum the code

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

Reverse Engineering

- Goals
 - focused exploration
 - deep understanding
- Case study
 - copy protection mechanism
 - program expects name and serial number
 - when serial number is incorrect, program exits
 - otherwise, we are fine
- Changes in the binary
 - can be done with `hexedit`

Reverse Engineering

- Focused exploration
 - *bypass check routines*
 - locate the point where the failed check is reported
 - find the routine that checks the password
 - find the location where the results of this routine are used
 - slightly modify the jump instruction
- Deep understanding
 - *key generation*
 - locate the checking routine
 - analyze the disassembly
 - run through a few different cases with the debugger
 - understand what check code does and develop code that creates appropriate keys

Summary

- Reverse Engineering
 - process of understanding the structure and functionality of system
- Software reverse engineering
 - static techniques
 - dynamic techniques
- Static techniques
 - check for strings, symbols, and library functions
 - disassembler
- Dynamic techniques
 - monitor network and file system activity
 - system call monitoring (`ptrace` interface)
 - debugger