CSC 405
Introduction to Computer Security

Reverse Engineering

Alexandros Kapravelos
kapravelos@ncsu.edu

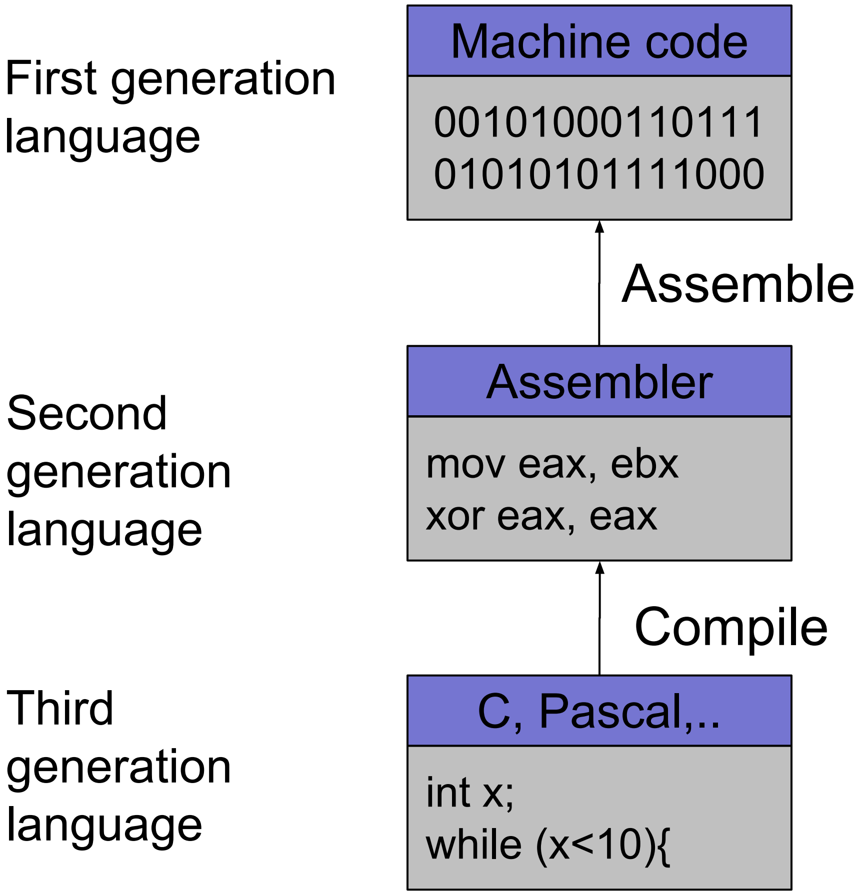
(Derived from slides by Chris Kruegel)

Introduction

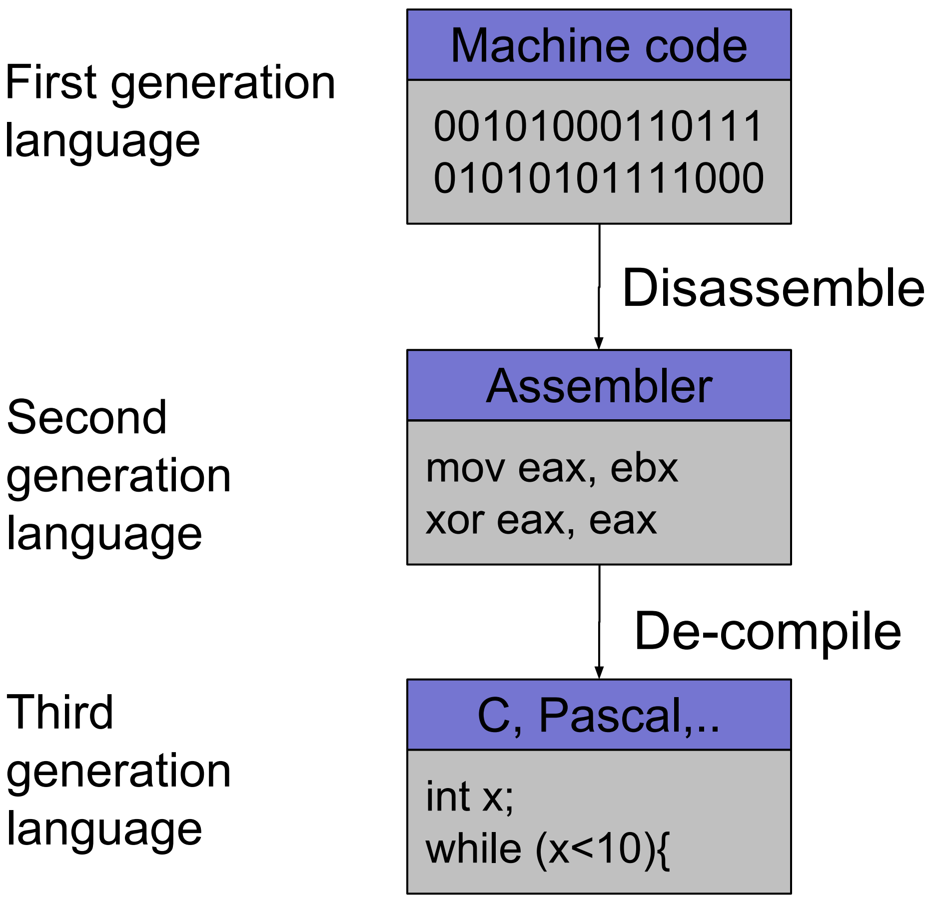
- Reverse engineering
  - process of analyzing a system
  - understand its structure and functionality
  - used in different domains (e.g., consumer electronics)

- 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)

Software Engineering



Software Reverse Engineering



Going Back is Hard!

- Fully-automated disassemble/de-compilation of arbitrary machine-code is theoretically an undecidable problem

- Disassembling problems
  - hard to distinguish code (instructions) from data

- De-compilation problems
  - structure is lost
    - data types are lost, names and labels are lost
  - no one-to-one mapping
    - same code can be compiled into different (equivalent) assembler blocks
    - assembler block can be the result of different pieces of code

Why Reverse Engineering

- Software interoperability
  - Samba (SMB Protocol)
  - OpenOffice (MS Office document formats)
- Emulation
  - Wine (Windows API)
  - React-OS (Windows OS)
- Malware analysis
- Program cracking
- Compiler validation

Analyzing a Binary

Static Analysis

- Identify the file type and its characteristics
  - architecture, OS, executable format...
- Extract strings
  - commands, password, protocol keywords...
- Identify libraries and imported symbols
  - network calls, file system, crypto libraries
- Disassemble
  - program overview
  - finding and understanding important functions
    - by locating interesting imports, calls, strings...

Analyzing a Binary

Dynamic Analysis

- Memory dump
  - extract code after decryption, find passwords...
- Library/system call/instruction trace
  - determine the flow of execution
  - interaction with OS
- Debugging running process
  - inspect variables, data received by the network, complex algorithms..
- Network sniffer
  - find network activities
  - understand the protocol

Static Techniques

- Gathering program information
  - get some rough idea about binary (`file`)

```
linux util # file sil
sil: ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), for GNU/Linux 2.6.9, dynamically linked (uses s
hared libs), not stripped
```

  - strings that the binary contains (`strings`)

```
linux util # strings sil | head -n 5
/lib/ld-linux.so.2
_Jv_RegisterClasses
__gmon_start__
libc.so.6
puts
```

Static Techniques

- Examining the program (ELF) header (`elfsh`)

```
[ELF HEADER]
[Object sil, MAGIC 0x464C457F]

Architecture          :        Intel 80386   ELF Version          :             1
Object type           :  Executable object   SHT strtab index     :            25
Data encoding         :      Little endian   SHT foffset          :          4061
PHT foffset           :                 52   SHT entries number   :            28
PHT entries number    :                  8   SHT entry size       :            40
PHT entry size        :                 32   ELF header size      :            52
Entry point           :          0x8048500   [_start]
{PAX FLAGS = 0x0}
PAX_PAGEEXEC          :           Disabled   PAX_EMULTRAMP        :  Not emulated
PAX_MPROTECT          :         Restricted   PAX_RANDMMAP         :    Randomized
PAX_RANDEXEC          :     Not randomized   PAX_SEGMEXEC         :       Enabled
```

Program entry point

Static Techniques

Interesting "shared" library –
used for (fast) system calls

- Used libraries
  - easier when program is dynamically linked (ldd)

```
linux util # ldd sil
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/libc.so.6 (0xb7e99000)
        /lib/ld-linux.so.2 (0xb7fcf000)
```

  - more difficult when program is statically linked

```
linux util # gcc -static -o sil-static simple.c
linux util # ldd sil-static
        not a dynamic executable
linux util # file sil-static
sil-static: ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), for GNU/Linux 2.6.9, statically linked, not stripped
```

Static Techniques

Looking at `linux-gate.so.1`

```
linux util # cat /proc/self/maps | tail -n 1
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
linux util # dd if=/proc/self/mem of=linux-gate.dso bs=4096 skip=1048574
 count=1 2> /dev/null
linux util # objdump -d linux-gate.dso | head -n 11

linux-gate.dso:     file format elf32-i386

Disassembly of section .text:

ffffe400 <__kernel_vsyscall>:
ffffe400:       51                      push   %ecx
ffffe401:       52                      push   %edx
ffffe402:       55                      push   %ebp
ffffe403:       89 e5                   mov    %esp,%ebp
ffffe405:       0f 34                   sysenter
```

Static Techniques

- Used library functions
  - again, easier when program is dynamically linked (`nm -D`)

```
linux util # nm -D sil | tail -n8
         U fprintf
         U fwrite
         U getopt
         U opendir
08049bb4 B optind
         U puts
         U readdir
08049bb0 B stderr
```

  - more difficult when program is statically linked

```
linux util # nm -D sil-static
nm: sil-static: No symbols
linux util # ls -la sil*
-rwxr-xr-x 1 root chris   8017 Jan 21 20:37 sil
-rwxr-xr-x 1 root chris 544850 Jan 21 20:58 sil-static
```

Static Techniques

Recognizing libraries in statically-linked programs

- Basic idea
  - create a checksum (hash) for bytes in a library function
- Problems
  - many library functions (some of which are very short)
  - variable bytes – due to dynamic linking, load-time patching, linker optimizations
- Solution
  - more complex pattern file
  - uses checksums that take into account variable parts
  - implemented in `IDA Pro` as: Fast Library Identification and Recognition Technology (FLIRT)

Static Techniques

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

Static Techniques

Displaying program symbols

```
linux util # nm sil | grep " T"
080488c7 T __i686.get_pc_thunk.bx
08048850 T __libc_csu_fini
08048860 T __libc_csu_init
08048904 T _fini
08048420 T _init
08048500 T _start
080485cd T display_directory
080486bd T main
080485a4 T usage
linux util # strip sil
linux util # nm sil | grep " T"
nm: sil: no symbols
```

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
  - more difficult to disassemble
  - start addresses of instructions not known in advance
  - different strategies
    - linear sweep disassembler
    - recursive traversal disassembler
  - disassembler can be desynchronized with respect to actual code

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 %

Intel x86 Assembler Primer

- 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`

- Important mnemonics (instructions)

  `mov` data transfer

  `add` / `sub` arithmetic

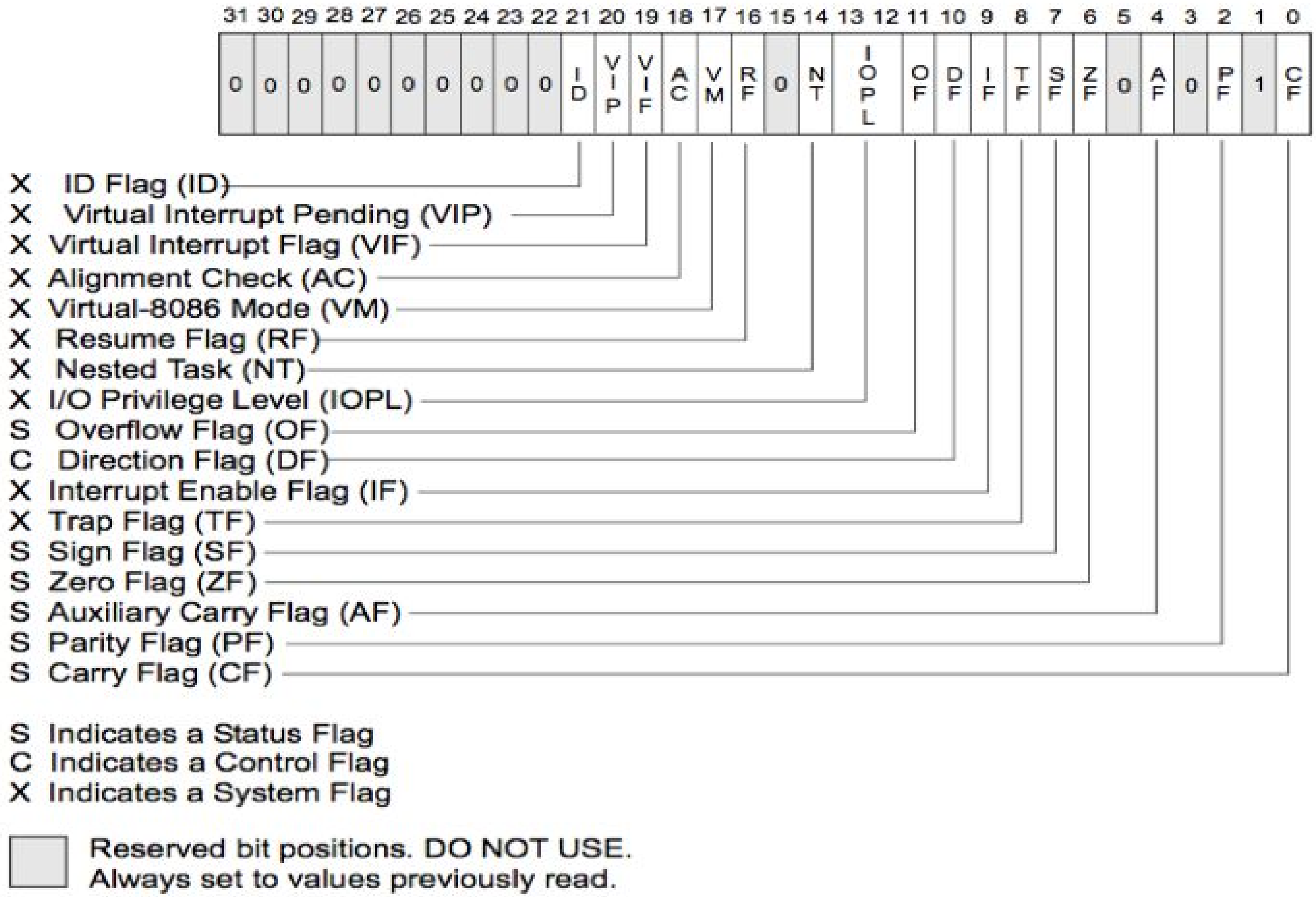
  `cmp` / `test` compare two values and set control flags

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

  `jmp` unconditional jump

Intel x86 Assembler Primer

Status (EFLAGS) Register



Intel x86 Assembler Primer

- Status (EFLAGS) Register
  - used for control flow decision
  - set implicit by many operations (arithmetic, logic)

- Flags typically used for control flow
  - CF (carry flag)
    - set when operation “carries out” most significant bit
  - ZF (zero flag)
    - set when operation yields zero
  - SF (signed flag)
    - set when operation yields negative result
  - OF (overflow flag)
    - set when operation causes 2’s complement overflow
  - PF (parity flag)
    - set when the number of ones in result of operation is even

Intel x86 Assembler Primer

Instruction	Synonym	Jump condition	Description
jmp label		1	direct jump
jmp *operand		1	indirect jump
je label	jz	ZF	equal/zero
jne label	jnz	~ZF	not equal/zero
js label		SF	negative
jns label		~SF	non-negative
jg label	jnle	~(SF ^ OF) & ~ZF	greater than (signed)
jge label	jnl	(~SF ^ OF)	greater or equal (signed)
jl label	jnge	SF ^ OF	less than (signed)
jle label	jng	(SF ^ OF) | ZF	less or equal (signed)
ja label	jnbe	~CF & ~ZF	above (unsigned)
jae label	jnb	~CF	above or equal (unsigned)
jb label	jnae	CF	below (unsigned)
jbe label	jna	CF | ZF	below or equal (unsigned)

Intel x86 Assembler Primer

- When are flags set?
  - implicit, as a side effect of many operations
  - can use explicit compare / test operations
- Compare

  `cmp b, a` [note the order of operands]
  - computes (a – b) but does not overwrite destination
  - sets ZF (if a == b), SF (if a < b) [and also OF and CF]
- How is a branch operation implemented
  - typically, two step process

    first, a compare/test instruction

    followed by the appropriate jump instruction

Intel x86 Assembler Primer

- Program can access data stored in memory
  - memory is just a linear (flat) array of memory cells (bytes)
  - accessed in different ways (called addressing modes)

- Most general fashion
  - `address: displacement(%base, %index, scale)`

    where the result address is `displacement + %base + %index*scale`

- Simplified variants are also possible
  - use only displacement → direct addressing
  - use only single register → register addressing

Intel x86 Assembler Primer

- Stack
  - managed by stack pointer (%esp) and frame pointer (%ebp)
  - special commands (push, pop)
  - 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

```
# no input
# returns a status code, you can view it by typing echo $?
# %ebx holds the return code
.section .text
.globl _start

_start:
movl   $1, %eax    # This is the system call for exiting program
movl   $0, %ebx    # This value is returned as status
int    $0x80  # This interrupt calls the kernel, to execute sys call
```

Intel x86 Assembler Primer

- So how do we create the application?
  - we need to assemble and link the code
  - this can be done by using the assembler *as (or gcc)*

- Assemble

```
as exit.s -o exit.o |
  gcc -c -o exit.o exit.s
```

- Link

```
ld -o exit exit.o |
  gcc -nostartfiles -o exit exit.o
```

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) /* compute: a - 0   */
        jns     .L2          /* jump, if sign bit
                          not set: a >= 0  */
        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
```

Intel x86 Assembler Primer

Task: Find the maximum of a list of numbers

- Questions to ask:
  - Where will the numbers be stored?
  - How do we find the maximum number?
  - How much storage do we need?
  - Will registers be enough or is memory needed?

- Let us designate registers for the task at hand:
  - %edi holds position in list
  - %ebx will hold current highest
  - %eax will hold current element examined