CSC 591
Systems Attacks and Defenses

Reverse Engineering
Part 1

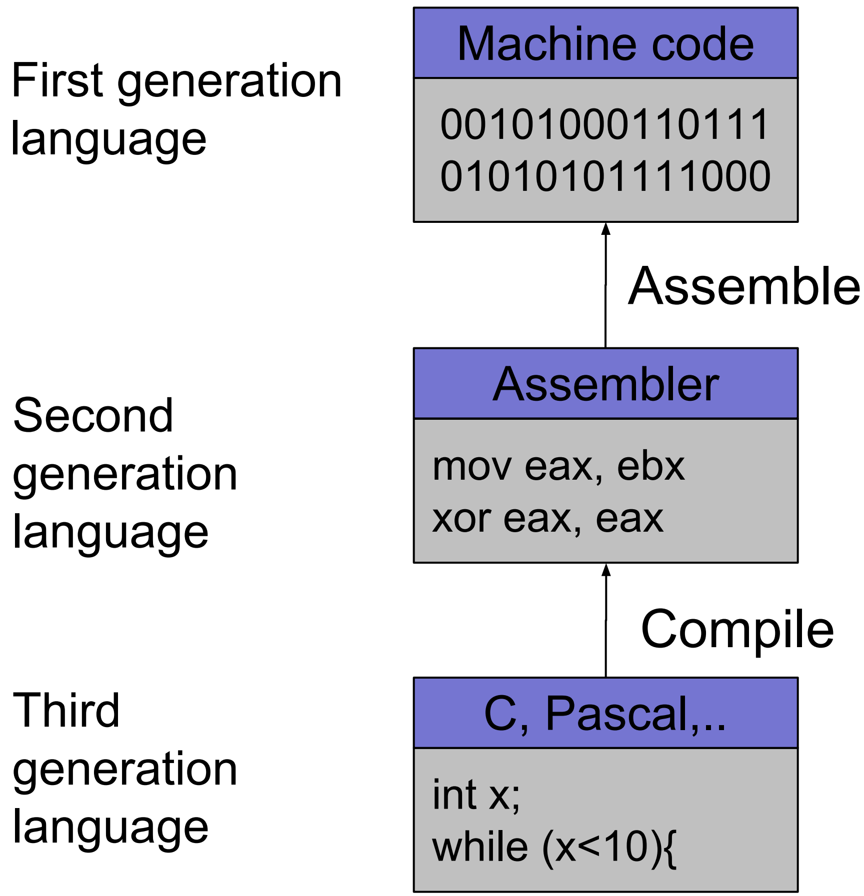
Alexandros Kapravelos
akaprav@ncsu.edu

Introduction

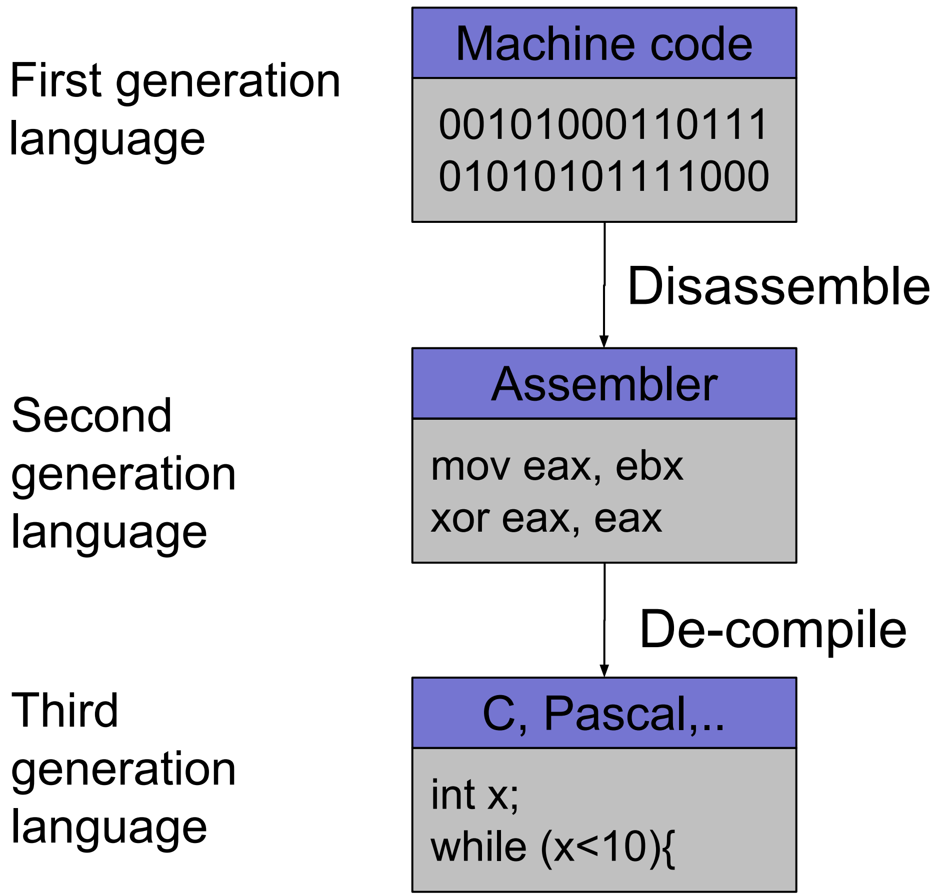
- Reverse engineering
  - process of analyzing a system
  - understand its structure and functionality
  - used in different domains (e.g., consumer electronics)

- 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)
- Legacy software
  - Onlive
- 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

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

Interesting “shared” library used for (fast) system calls

```
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
```

  - n

```
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

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

Let's break LSD

```
#include <stdio.h>

int main() {
   printf("Hello, world!\n");
   return 0;
}
$ gcc hello.c -o hello
$ ./hello
Hello, world!
```

Objdump disassembly

```
0804840b <main>:
   804840b:       8d 4c 24 04             lea    0x4(%esp),%ecx
   804840f:       83 e4 f0                and    $0xfffffff0,%esp
   8048412:       ff 71 fc                pushl  -0x4(%ecx)
   8048415:       55                      push   %ebp
   8048416:       89 e5                   mov    %esp,%ebp
   8048418:       51                      push   %ecx
   8048419:       83 ec 04                sub    $0x4,%esp
   804841c:       83 ec 0c                sub    $0xc,%esp
   804841f:       68 c0 84 04 08          push   $0x80484c0
   8048424:       e8 b7 fe ff ff          call   80482e0 <puts@plt>
   8048429:       83 c4 10                add    $0x10,%esp
   804842c:       b8 00 00 00 00          mov    $0x0,%eax
   8048431:       8b 4d fc                mov    -0x4(%ebp),%ecx
   8048434:       c9                      leave
   8048435:       8d 61 fc                lea    -0x4(%ecx),%esp
   8048438:       c3                      ret
```

```
$ objdump -D hello
```

radare2 disassembly

```
[0x08048310]> pdf@main
/ (fcn) sym.main 46
|           0x0804840b    8d4c2404     lea ecx, [esp+0x4]
|           0x0804840f    83e4f0       and esp, 0xfffffff0
|           0x08048412    ff71fc       push dword [ecx-0x4]
|           0x08048415    55           push ebp
|           0x08048416    89e5         mov ebp, esp
|           0x08048418    51           push ecx
|           0x08048419    83ec04       sub esp, 0x4
|           0x0804841c    83ec0c       sub esp, 0xc
|           ; DATA XREF from 0x080484c0 (fcn.080484b8)
|           0x0804841f    68c0840408   push str.Helloworld ; 0x080484c0
|           ; CODE (CALL) XREF from 0x080482e6 (fcn.080482e6)
|           ; CODE (CALL) XREF from 0x080482f6 (fcn.080482f6)
|           ; CODE (CALL) XREF from 0x08048306 (fcn.08048306)
|           0x08048424    e8b7feffff   call 0x1080482e0 ; (sym.imp.puts)
|              sym.imp.puts(unk, unk, unk, unk)
|           0x08048429    83c410       add esp, 0x10
|           0x0804842c    b800000000   mov eax, 0x0
|           0x08048431    8b4dfc       mov ecx, [ebp-0x4]
|           0x08048434    c9           leave
|           0x08048435    8d61fc       lea esp, [ecx-0x4]
\           0x08048438    c3           ret
```

Let’s patch the program

```
$ radare2 -Aw hello
[0x08048310]> 0x08048419
[0x08048419]> wx eb01   #(jmp 0x804841c)
```

We patched a 3-byte instruction with a 2-byte instruction. What is going to happen now with disassembly?!

Disassembly fails!

```
[0x08048310]> pdf@main
/ (fcn) sym.main 46
|           0x0804840b    8d4c2404     lea ecx, [esp+0x4]
|           0x0804840f    83e4f0       and esp, 0xfffffff0
|           0x08048412    ff71fc       push dword [ecx-0x4]
|           0x08048415    55           push ebp
|           0x08048416    89e5         mov ebp, esp
|           0x08048418    51           push ecx
|       ,=< 0x08048419    eb01         jmp loc.0804841c
|       |   0x0804841b    0483         add al, 0x83
|           0x0804841d    ec           in al, dx
|           0x0804841e    0c68         or al, 0x68
|           0x08048420    c0840408e8b. rol byte [esp+eax-0x14817f8], 0xff
|           0x08048428    ff83c410b800 inc dword [ebx+0xb810c4]
|           0x0804842e    0000         add [eax], al
|           0x08048430    008b4dfcc98d add [ebx-0x723603b3], cl
|           0x08048436    61           popad
|           0x08048437    fc           cld
\           0x08048438    c3           ret
```

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

```
.text:0804840B ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:0804840B                 public main
.text:0804840B main            proc near               ; DATA XREF: _start+17o
.text:0804840B var_4           = dword ptr -4
.text:0804840B argc            = dword ptr  0Ch
.text:0804840B argv            = dword ptr  10h
.text:0804840B envp            = dword ptr  14h
.text:0804840B                 lea     ecx, [esp+4]
.text:0804840F                 and     esp, 0FFFFFFF0h
.text:08048412                 push    dword ptr [ecx-4]
.text:08048415                 push    ebp
.text:08048416                 mov     ebp, esp
.text:08048418                 push    ecx
.text:08048419                 jmp     short loc_804841C
.text:08048419 ; ---------------------------------------------------------------------------
.text:0804841B                 db 4
.text:0804841C ; ---------------------------------------------------------------------------
.text:0804841C loc_804841C:                            ; CODE XREF: main+Ej
.text:0804841C                 sub     esp, 0Ch
.text:0804841F                 push    offset s        ; "Hello, world!"
.text:08048424                 call    _puts
.text:08048429                 add     esp, 10h
.text:0804842C                 mov     eax, 0
.text:08048431                 mov     ecx, [ebp+var_4]
.text:08048434                 leave
.text:08048435                 lea     esp, [ecx-4]
.text:08048438                 retn
.text:08048438 main            endp%
```

Dynamic Techniques

- General information about a 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
  - lsof
  - 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
    - ethereal/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 attack/malware
  - 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

Challenges

- Reverse engineering is difficult by itself
  - a lot of data to handle
  - low level information
  - creative process, experience very valuable
  - tools can only help so much
- Additional challenges
  - compiler code optimization
  - code obfuscation
  - anti-disassemble techniques
  - anti-debugging techniques

Your Security Zen

Introducing managed SSL for Google App Engine

Google is a Certificate Authority

SSL is on by default

source: https://cloudplatform.googleblog.com/2017/09/introducing-managed-SSL-for-Google-App-Engine.html

Your Security Zen

Chrome's Plan to Distrust Symantec Certificates

The Chrome team and the PKI community converged upon a plan to reduce, and ultimately remove, trust in Symantec's infrastructure in order to uphold users' security and privacy when browsing the web

Starting with Chrome 66, Chrome will remove trust in Symantec-issued certificates

source: https://security.googleblog.com/2017/09/chromes-plan-to-distrust-symantec.html