Special Topics in Security
ECE 5968

Engin Kirda
ek@ccs.neu.edu

Northeastern University

Recap: Your practical abilities so far ;)

- You have experience with *suid* programs and exploitation of local vulnerabilities on UNIX systems
- You are now knowledgeable about web security
  – You've learned about and practiced the most popular exploitation techniques
- You have seen the most popular classes of memory corruption problems
  – You have experience with local buffer overflow exploits
- What remains:
  – Reverse engineering, testing…

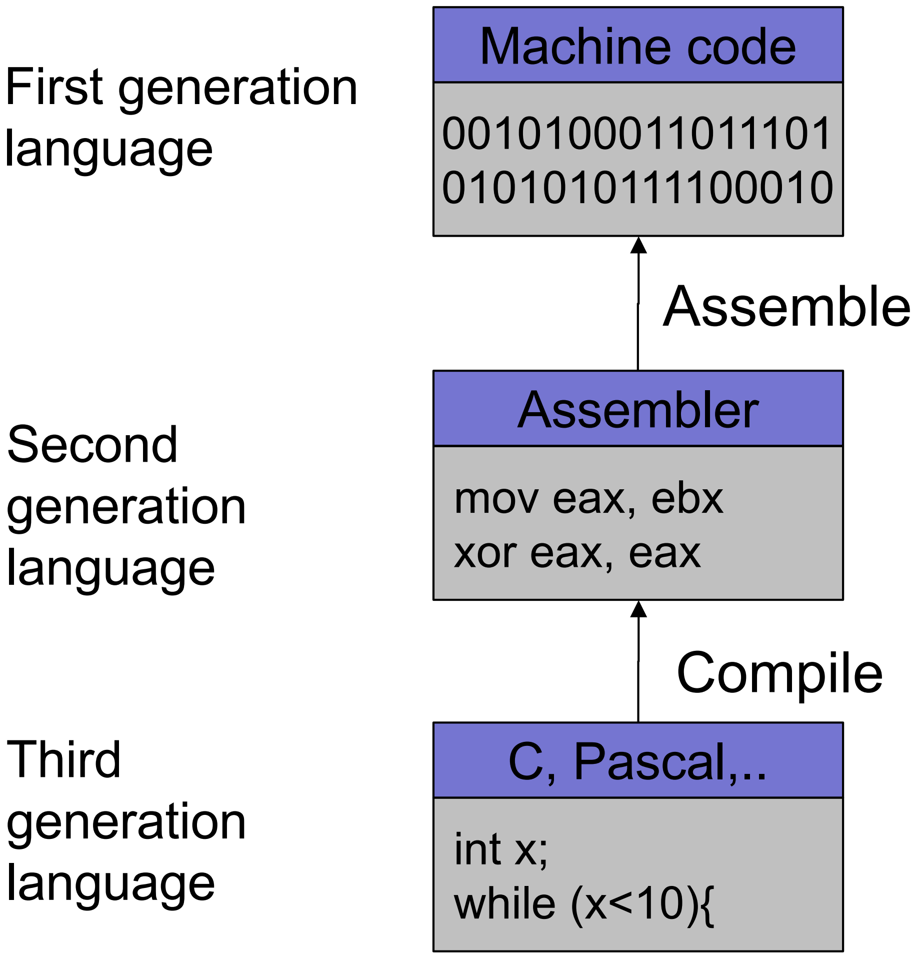
Reverse Engineering

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)

Software Engineering

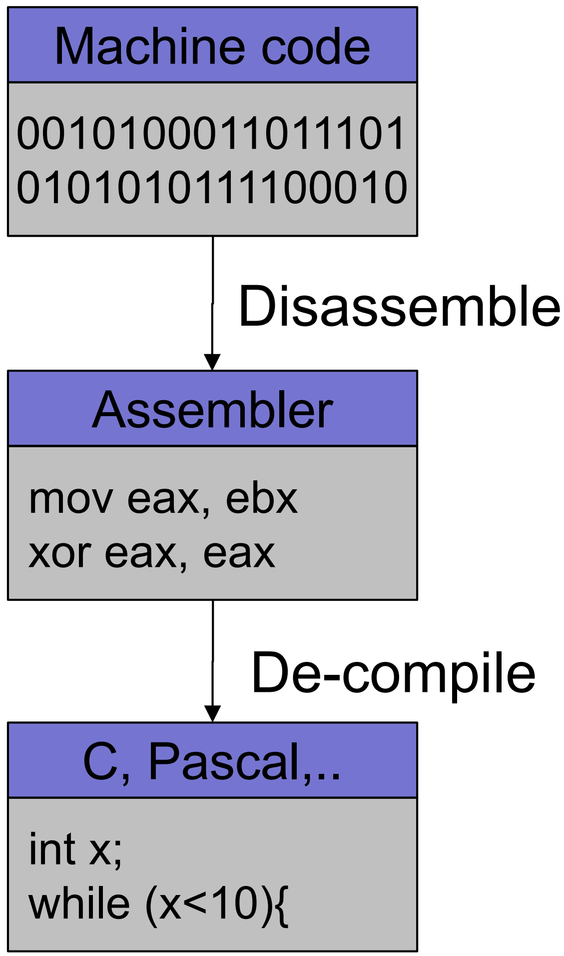


Software Reverse Engineering

First generation language

Second generation language

Third generation language



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 and vulnerability detection
- Compiler validation

Analyzing Byte Code

- Languages such as Java / C# have become popular
  - When these languages are compiled, byte code is generated
  - Byte code is “interpreted” and run in a virtual machine (e.g., JVM)
- Byte code may look “cryptic” and impossible to read, but unlike native applications, it is easy to “de-compile”
  - It is not as ambiguous as binary code (i.e., machine code) generated when compiling a native application (e.g., C)
  - Tools exist that make it easy to analyze and reverse engineer byte code
    - E.g., JAD for decompiling Java class files

Demo: JAD, Reverse Engineering Java Byte Code

Obfuscating Source Code

- Byte code belonging to Java/C# is easy to analyze
  - Hence, to hide functionality, code obfuscation has to be used
    - Security by obscurity
  - e.g., ProGuard – comprehensive tool, transforms source code to make it more difficult to understand
  - Problem:
    - Obfuscation makes it time consuming to read the code, but not impossible

```
char*M,A,Z,E=40,J[40],T[40];main(C){for(*J=A=scanf(M="%d",&C); -- E; J[ E]
    =T [E ]= E) printf("._"); for(;(A-=Z=!Z) || (printf("\n|" ) , A = 39 ,C -- ) ; Z ||
    printf (M ))M[Z]=Z[A-(E =A[J-Z])&&!C & A == T[ A]
    |6<<27<rand()||!C&!Z?J[T[E]=T[A]]=E,J[T[A]=A-Z]=A,"_.":" |"];}
```

Classic Tricks for Obfuscation

- Unobfuscated Java code:

```
public class HelloWorld
{
public static main(String argv[])
{
      System.out.println(“Hello World);
}
}
```

Classic Tricks for Obfuscation

- Let’s obfuscate the code a little...:

```
class y {
public void z() {
      System.out.println(“Hello World);
}
}
public class q {
public static main(String x[])  {
        new y().z();
}
}
```

Classic Tricks for Obfuscation

- Let's obfuscate the code a little bit more…:

```
class p() {
public String x() {
        return("Hello World");
}
}
class y {
public void z() {
      System.out.println(new p().x());
}
}
```

Classic Tricks for Obfuscation

```
class p {
public String x() {
      return("Hello World");
}}
class y {
public void z() {
     System.out.println(new p().x());
}}
public class q {
public static main(String x[])  {
      new y().z();
}}
```

A Real Story…

- A bank uses an online digital signature system for secure online banking
  - The solution is based on Java applets running in the browser
  - The code is highly obfuscated (manual analysis takes time)
  - The CEO of company claims that the code is secure against malware…
    - … because the applet is running in the browser
  - After all the things you've learned in class… my question to you:
  - How can you compromise the security of this applet solution? Ideas?

A Real Story…

- You can always compromise…
  - … the Java Virtual Machine (JVM)!
  - The byte code being executed is interpreted by the JVM and it relies completely on the libraries provided by the JVM
  - For example, if the applet uses FileStreamReader() to read the contents of the file
    - You can intercept and feed it anything you want
- The key lesson to remember in any security situation
  - One CANNOT guarantee secure execution on an UNTRUSTED platform
  - This lesson is often forgotten in practice

DEMO: Manipulating JDK Libraries…

Analyzing a Binary

<u>Static Analysis</u>

- 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

<u>Dynamic Analysis</u>

- 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

DEMO: Let’s first create a binary to analyze… PasswordChecker

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
```

DEMO: Using command-line tools (file, strings)

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`)

```
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
```

DEMO: Checking linked libraries (ldd)

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
```

DEMO: Checking out symbols (nm)

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 Compare

- 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

If Statement Mapping

- `If` statement

```c
#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
```

While Statement Mapping

- `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
```

Global vs. Local Assignments

- Local variables
  - stored in the current stack frame
  - referenced relative to frame pointer (or stack pointer)

```
DIR *d;
d = opendir(s);
```

```
call 80484a8 <opendir@plt>
mov  %eax,0xfffffff0(%ebp)
```

- Global variables
  - referenced by absolute address (or offset to segment)

```
char *progname;
void usage() {
  char *s;
  s = progname;
```

```
mov  0x8049bbc,%eax
mov  %eax,0xfffffffc(%ebp)
```

Function Calls Recap

- Function arguments
  - can be passed in different fashions, depending on the calling convention

- Calling conventions
  - cdecl
    use the stack to pass arguments, caller cleans stack
  - stdcall
    use the stack to pass arguments, callee cleans stack
  - fastcall
    pass first two arguments in registers, rest on stack

- Argument access
  - with cdecl, use relative offset of base pointer
  - similar to local variables, but positive offset

Static Techniques

… after this x86 assembler recap, back to disassembling…

- 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

DEMO: Disassembling code with objdump –d, and gdb, with and without symbols

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

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
  - `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`

DEMO: Checking binary with strace

Dynamic Techniques

- Execute program in a controlled environment
  - sandbox / debugger
  - gdb is your friend, remember? ;)
- 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 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