CS 290
Host-based Security and Malware

Christopher Kruegel
chris@cs.ucsb.edu
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

First generation language

Machine code

0010100011011101
0101010111100010

Assemble

Second generation language

Assembler

mov eax, ebx
xor eax, eax

Compile

Third generation language

C, Pascal,..

int x;
while (x<10){
Software Reverse Engineering

First generation language

Machine code

0010100011011101
0101010111100010

Disassemble

Second generation language

Assembler

mov eax, ebx
xor eax, eax

De-compile

Third generation language

C, Pascal,..

int x;
while (x<10){
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 shared 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 size, MAGIC 0x464C457F]

Architecture : Intel 80386
Object type    : Executable object
Data encoding  : Little endian
PHT offset     : 52
PHT entries number : 8
PHT entry size : 32
Entry point    : 0x8048500
{PAX_FLAGS = 0x0}
PAX_PAGEEXEC   : Disabled
PAX_MPREAD    : Restricted
PAX_RANDEXEC  : Not randomized

ELF Version : 1
SHT strtab index : 25
SHT foffset : 4061
SHT entries number : 28
SHT entry size : 40
ELF header size : 52
[_start]

Program entry point

CS 290: Host-based security and malware
Static Techniques

- Used libraries
  - easier when program is dynamically linked (`ldd`)
    
    ```
    linux util # ldd sil
    linux-gate.so.1 => (0xffffffff000)
    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-ffffff000 r-xp 00000000 00:00 0 [vds]
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 <__kern_vsyscalls>`:

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

- Used library functions
  - again, easier when program is dynamically linked (\texttt{nm -D})

\begin{verbatim}
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
\end{verbatim}

- more difficult when program is statically linked

\begin{verbatim}
linux util # nm -D sil-static
nm: sil-static: No symbols
linux util # ls -la sil*
-rw-r-xr-x 1 root chris 8017 Jan 21 20:37 sil
-rw-r-xr-x 1 root chris 544850 Jan 21 20:58 sil-static
\end{verbatim}
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

```bash
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
Status (EFLAGS) Register

- ID Flag (ID)
- Virtual Interrupt Pending (VIP)
- Virtual Interrupt Flag (VIF)
- Alignment Check (AC)
- Virtual-8086 Mode (VM)
- Resume Flag (RF)
- Nested Task (NT)
- I/O Privilege Level (IOPL)
- Overflow Flag (OF)
- Direction Flag (DF)
- Interrupt Enable Flag (IF)
- Trap Flag (TF)
- Sign Flag (SF)
- Zero Flag (ZF)
- Auxiliary Carry Flag (AF)
- Parity Flag (PF)
- Carry Flag (CF)

- S Indicates a Status Flag
- C Indicates a Control Flag
- X Indicates a System Flag

Reserved bit positions. DO NOT USE. Always set to values previously read.
• 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
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Jump condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp label</td>
<td></td>
<td>1</td>
<td>direct jump</td>
</tr>
<tr>
<td>jmp *operand</td>
<td></td>
<td>1</td>
<td>indirect jump</td>
</tr>
<tr>
<td>je label</td>
<td>jz</td>
<td>ZF</td>
<td>equal/zero</td>
</tr>
<tr>
<td>jne label</td>
<td>jnz</td>
<td>~ZF</td>
<td>not equal/zero</td>
</tr>
<tr>
<td>js label</td>
<td></td>
<td>SF</td>
<td>negative</td>
</tr>
<tr>
<td>jns label</td>
<td></td>
<td>~SF</td>
<td>non-negative</td>
</tr>
<tr>
<td>jg label</td>
<td>jnle</td>
<td>~(SF ^ OF) &amp; ~ZF</td>
<td>greater than (signed)</td>
</tr>
<tr>
<td>jge label</td>
<td>jnle</td>
<td>(~SF ^ OF)</td>
<td>greater or equal (signed)</td>
</tr>
<tr>
<td>jl label</td>
<td>jnle</td>
<td>SF ^ OF</td>
<td>less than (signed)</td>
</tr>
<tr>
<td>jle label</td>
<td>jnle</td>
<td>(SF ^ OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>ja label</td>
<td>jnbe</td>
<td>~CF &amp; ~ZF</td>
<td>above (unsigned)</td>
</tr>
<tr>
<td>jae label</td>
<td>jnbe</td>
<td>~CF</td>
<td>above or equal (unsigned)</td>
</tr>
<tr>
<td>jb label</td>
<td>jnbe</td>
<td>CF</td>
<td>below (unsigned)</td>
</tr>
<tr>
<td>jbe label</td>
<td>jnbe</td>
<td>CF</td>
<td>ZF</td>
</tr>
</tbody>
</table>
• When are flags set?
  – implicit, as a side effect of many operations
  – can use explicit compare / test operations

• Compare
  \texttt{cmp b, a} \quad [\text{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
• 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?
    
    | Address | Value |
    |---------|-------|
    | 0x040   | 0     |
    | 0x041   | 1     |
    | 0x042   | 2     |
    | 0x043   | 3     |
# no input
# returns a status code, you can view it by typing echo $?
# %ebx holds the return code

.section .data
.section .text
.globl _start

_start:

mov $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
• 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
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
Intel x86 Assembler - Algorithm

- Check if %eax is zero (i.e., termination sign)
  - if yes, exit
  - if not, increase current position %edi

- Load next value in the list to %eax
  - we need to think about what addressing mode to use here

- Compare %eax (current value) with %ebx (highest value so far)
  - if the current value is higher, replace %ebx

- Repeat
.section .data
    data_items:
       .long 3, 67, 34, 222, 45, 75, 54, 34, 44, 33, 22, 11, 66, 0

.section .text
.globl _start

_start:
    movl $0, %edi       # Reset index
    movl data_items(%edi,4), %eax
    movl %eax, %ebx     # First item is the biggest so far
start_loop:

```
cmpl  $0, %eax
je    loop_exit
incl  %edi        # Increment edi
movl  data_items(,%edi,4), %eax  # Load the next value
cmpl  %ebx, %eax   # Compare ebx with eax
jle   start_loop  # If it is less, just jump to the beginning
movl  %eax, %ebx  # Otherwise, store the new largest number
jmp   start_loop
```

loop_exit:

```
movl  $1, %eax    # Remember the exit sys call? It is 1
int   $0x80
```
• **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");
    }
}
```

```assembly
.globl main
.type main, @function
main:
    [ function prologue ]
    cmp1 $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**

```c
#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)
    call printf
    leal -4(%ebp), %eax
    incl (%eax)
    jmp .L2
    .L3:
    leave
    ret
```
Static Techniques

... after this x86 assembler digression, 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
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`
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
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
Anti-Disassembly

- Against static analysis (disassembler)
- Confusion attack
  - targets linear sweep disassembler
  - insert data (or junk) between instructions and let control flow jump over this garbage
  - disassembler gets desynchronized with true instructions

```
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>
```
Anti-Disassembly

- Advanced confusion attack
  - targets recursive traversal disassembler
  - replace direct jumps (calls) by indirect ones (branch functions)
  - force disassembler to revert to linear sweep, then use previous attack
Anti-Debugging

- **Against dynamic analysis (debugger)**
  - detect tracing
    - a process can be traced only once
      
      ```c
      if (ptrace(PTRACE_TRACEME, 0, 1, 0) < 0)
      exit(1);
      ```
  
  - detect breakpoints
    - look for int 0x03 instructions
      
      ```c
      if (((unsigned *)(unsigned)<addr>+3) & 0xff)==0xcc)
      exit(1);
      ```
  
  - checksum the code
    
    ```c
    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
Malicious Code Analysis

*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**
  - undecidable in general case, approximations necessary
  - 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

- **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

- 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

• 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

• Virtual machines
  – allow to quickly restore analysis environment
  – might be detectable (x86 virtualization problems)
Malicious Code Analysis

• 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
  – limited environment could be detected, but faster
  – complete environment is comprehensive, but slower

  – Anubis uses this approach
Semantic Gap

• Analysis observes activity “from the outside”
  – how to relate instructions executed on processor with activity inside the operating system and sample?
  – must encode some knowledge about guest OS and processes
    • we chose to target MS Windows on Intel x86 machines
  – Qemu used as emulation environment

• Questions
  – how to identify which instructions belong to malware sample?
  – what to analyze?
  – how to implement analysis?
Tracking Malware Process

- Executed instructions must be assigned to correct process
  - make use of CR3 register
    - control register used for virtual memory management
    - holds physical address of the base of the page directory
    - unique for each process
    - ensures that virtual address spaces are disjoint
  - reloaded at every context switch

- Obtaining value of CR3 register
  - use small component inside operating system
  - extracts value from kernel task structure (EPROCESS)
  - process started in suspended state
Analysis Information

- Process interacts with operating system via system calls
  - needs OS for every interaction with environment
    - file system, network, registry, …
  - monitor system calls
  - unfortunately, on Windows, system calls largely undocumented and can change without notice
  - developers are supposed to use Windows API, which denotes a collection of stable, user-mode, shared libraries
  - of course, Windows API can be bypassed

→ we monitor Windows API calls and NT kernel calls
Analysis Report

• 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 logs
Analysis Implementation

• Windows API and kernel calls must be hooked *without* interfering with the program (and its memory)

• Can be implemented in straightforward fashion
  – compare address of instruction executed with function start address
  – Qemu allows to do this efficiently
  – start addresses extracted from library export tables

• Arbitrary addresses are possible
  – analysis has fine-grain control
Analysis Implementation

• Information about function invocation alone insufficient
  – we also want to know about arguments

• Callback invoked when “breakpoint” reached
  – allows us to extract argument values
  – semantic gap again, Anubis only knows about physical address space
  – appropriate read functions require argument type information (structure and size of arguments), similar to advanced debuggers
  – would be tedious to write by hand, automatic support desirable

• Callbacks are invoked on function call and function return
Argument Extraction

- Generate routines to extract argument values automatically
  - provide values directly to callback routine
  - requires knowledge about argument types

- Code generation component
  - processes Windows header files (read type definitions)
  - parameters may be in and out parameters (or both)
  - parameters may be of unknown length
    - zero-terminated strings
    - arrays where second parameters specifies length
  - headers files were manually augmented
  - complex structures (with pointers) are traversed automatically
Argument Extraction

• Problem
  – not all argument values need to present in physical memory
  – Anubis has only access to (emulated) physical memory
  – argument values could be swapped out
  – other problem are shared libraries that are not paged in
    • e.g., due to lazy evaluation

• Solution
  – make process access virtual address(es)
  – forces Windows to bring in memory (page fault handler)
  – Anubis has to (transparently) inject such access into process
Code Injection

• **Modify control flow of target process**
  – necessary to bring in paged out memory content
  – but also useful for calling arbitrary functions in context of target

• **Calling functions**
  – distinguish between open and create file (or registry key)
  – distinguish between files and directories
  – resolve unknown handles

• **Can be easily done with Qemu**
  – without modifying process’ code
  – only problem is to push argument values onto process stack
  – dual problem to reading arguments, generator can be reused
Stealthiness

- One obvious difference between machine and emulator
  \( \rightarrow \) time of execution

- Time could be used to detect our system
  - emulation allows to address these issues
  - certain instructions can be dynamically modified to return innocently looking results
  - for example, RTC (real-time clock) - RTDSC instruction