Virt-ICE: next generation debugger for malware analysis

Black Hat USA 2010, Las Vegas
July 29th

NGUYEN Anh Quynh, Kuniyasu SUZAKI

AIST, Japan

Who are we?

- From the National Institute of Advanced Industrial Science and Technology (AIST), Japan
- NGUYEN Anh Quynh, Post doctor researcher
    - VNSecurity member (http://vnsecurity.net)
- Kuniyasu SUZAKI, senior researcher (PhD)
- Multiple interests: Operating System, Virtualization, Trusted computing, malware analysis, forensic, rootkits, IDS, ...

VM-related research areas

- Practical security problems regarding Virtual Machine (VM)
  - Protect VM
    - Live memory forensic for VM
    - Malware scanner for VM
  - Leverage VM for various security-related areas
    - Dynamic binary analysis
    - Vulnerability research
    - etc ...

Virt-ICE preview

- A new debugger, specially built to analyze malware
- Have new approach to fix most problems of current debuggers
- Provide rich functionalities targeting malware analyst
  - To ease the job of malware analyst

Presentation overview

- Problems of debugger in malaware analysis
- Virt-ICE solution
    - Architecture, Design & Implementation
    - Main features
- Live demo
- Discussions
- Conclusions
- Q & A

Part I

- Problems of debugger in malaware analysis
- Virt-ICE solution
  - Architecture, Design & Implementation
  - Main features
- Live demo
- Discussions
- Conclusions
- Q & A

Malware analysis

- Static analysis
  - Disassemble/decompile malware binary code
  - Analyze dead-list to understand its activities
    - Most malware are packed and obfuscated
- Dynamic analysis
  - Run malware and monitor its activities at run-time
  - Analyze malware when it is running, lively

Debugger against malware

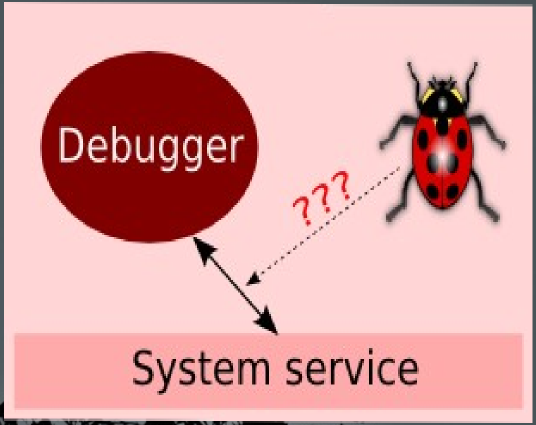
- Run malware under the monitor of a debugger
  - Disassemble/Decompile malware binary
  - Monitor execution flow
    - Using software/hardware breakpoints
  - Monitor data flow
    - Using memory watchpoints
  - Single-step for fine-granularity tracing
  - etc ...

Problems of debugger

- Malware can detect debugger and change behavior
  - Knowing that it is being debugged/monitored, malware can behave differently
  - Xu et al [NDSS08] reported the popularity of anti-debugging malware
    - 93.9% malware have anti-debugger techniques!
- Malware can tamper with debugger
  - Fool debugger, to make it function incorrectly
  - Attack debugger

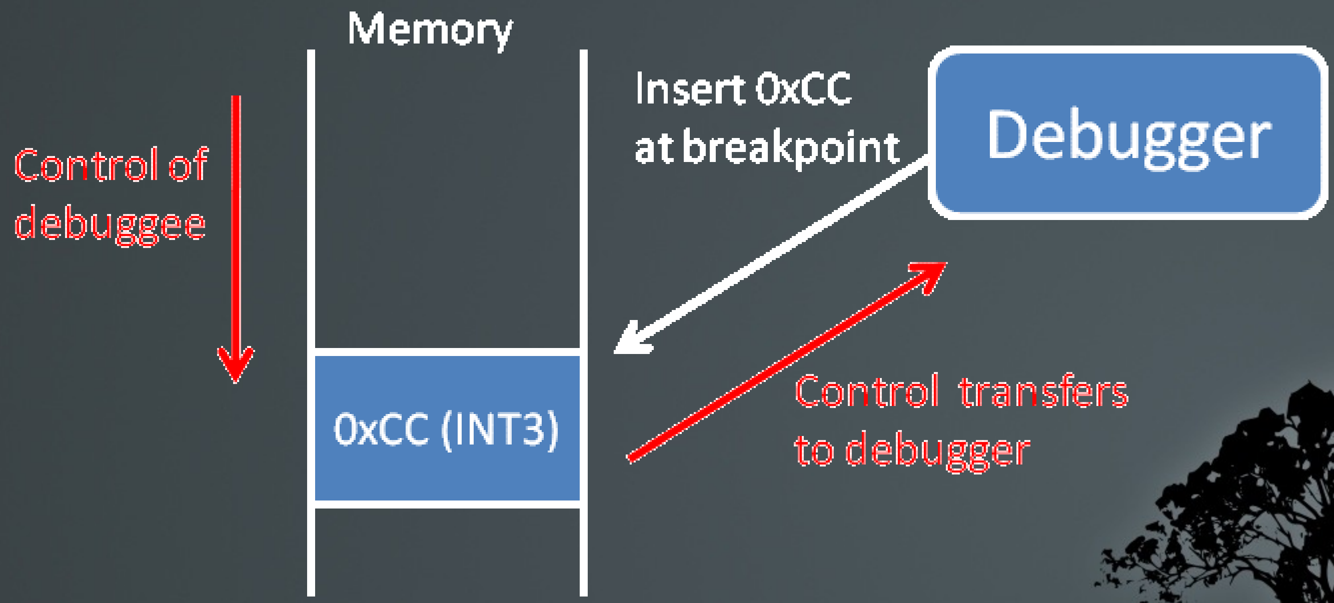
Detecting debugger (1)

- Debugger uses system service to handle debug events
  - Windows OS leaves traces in various places about the existence of debugger
    - PEB::NtGlobalFlag
    - PEB::BeingDebugged
  - Windows OS even provide some APIs for applications (and for malware, too) to check if a debugger is running
    - IsDebuggerPresent()
    - CheckRemoteDebuggerPresent()
    - NtQueryInformationProcess()
    - NtQuerySystemInformation()
    - NtQueryObject()



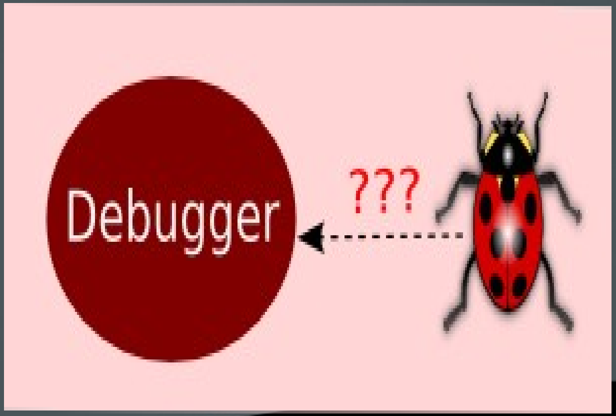
Detecting debugger (2)

- Debugger modify malware
  - Write software breakpoints (0xCC insn) into process memory
  - Malware can perform self-checking its code to detect the integrity violation



Detecting debugger (3)

- Debugger is visible in the same system
  - Detect that a debugger is installed in system
  - Detect that a debugger is running
    - Look for special processes, windows of particular debuggers
    - Look for special registries of particular debuggers
    - Look for special kernel devices using by particular debuggers
    - Etc ...



Tamper with debugger

- Tamper with debugger operation to make it work incorrectly
  - Modify hardware breakpoint value if debugger uses hardware breakpoints
  - Reset software breakpoint (0xCC byte) to original value, so debugger is not triggered any more
- Directly attack debugger
  - For ex: terminate debugger with TerminateProcess() function

Demo

- Detecting debugger is easy and unfortunately, increasingly complicated techniques are introduced
  - Peter Ferrie, Anti-unpacker tricks - series 1~9
    - and more is still coming :-(
- Attack and tamper with debugger is trivial
- Unfortuanately, unfixable!!

Why these problems?

Unfixable debugger

- Because debugger is never designed to analyze malware in the first place
  - Only for legitimate software, built and debugged by developers to find software bugs
  - Developers never write software to defeat his debugger :-)
    - Unfortunately, malware does that with lots of sophisticated tricks

Part II

- Problems of debugger in malaware analysis
- Virt-ICE solution
    - Architecture, Design & Implementation
    - Main features
- Live demo
- Discussions
- Conclusions
- Q & A

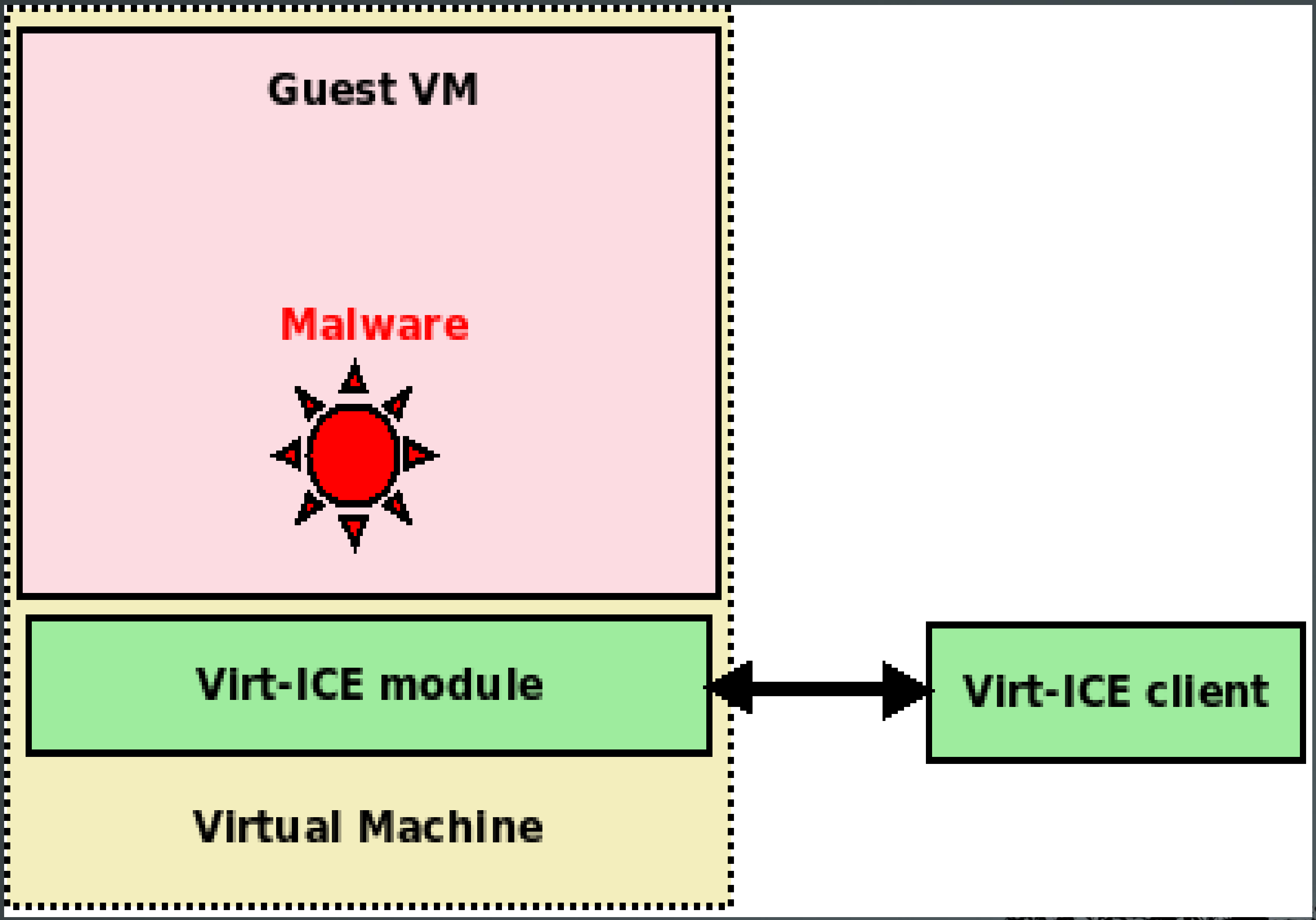
Ideas to solve problems

- Make the debugger invisible to malware
  - Malware cannot see the debugger
- Put the debugger out of the reach of malware
  - Having debugger in another protection domain, so malware cannot attack it

Virt-ICE approach

- Run malware inside Virtual Machine (VM)
  - Not introduce any problem, because analyst already used VM for malware analysis that for a long time
  - Fine-grain instrument guest VM to intercept guest anytime/anywhere we want to
- Put the debugger in hypervisor/emulator layer
  - Out of the reach of malware running inside guest VM

Virt-ICE architecture



Other benefits

- Whole system view, so whole system analysis is possbile
- Ring 0 code (rootkits included) debugging is better than anything else available out there!
  - Debug anywhere is possible

Fix the unfixable problems

- Virt-ICE is invisible to malware
  - Debugger uses system service for debugging?
    - Not more, because instrumentation from bottom can do even provide better mechnism for debugging anywhere
  - Debugger modify malware process?
    - Instrumentation never modifies malware process
  - Debugger is present in the same domain with malware?
    - Stay in emulator layer, and never uses any agent inside guest
- Virt-ICE cannot be attacked by malware
  - Guaranteed by VM design

Virt-ICE requirement

- Understand guest context from outside
- Instrument guest VM execution
  - So it is possible to set breakpoint, watchpoint, ... anywhere
- Access to VM context
  - Read/write to VM memory
  - Read/write to CPU context
- Manage VM
  - Pause, resume VM

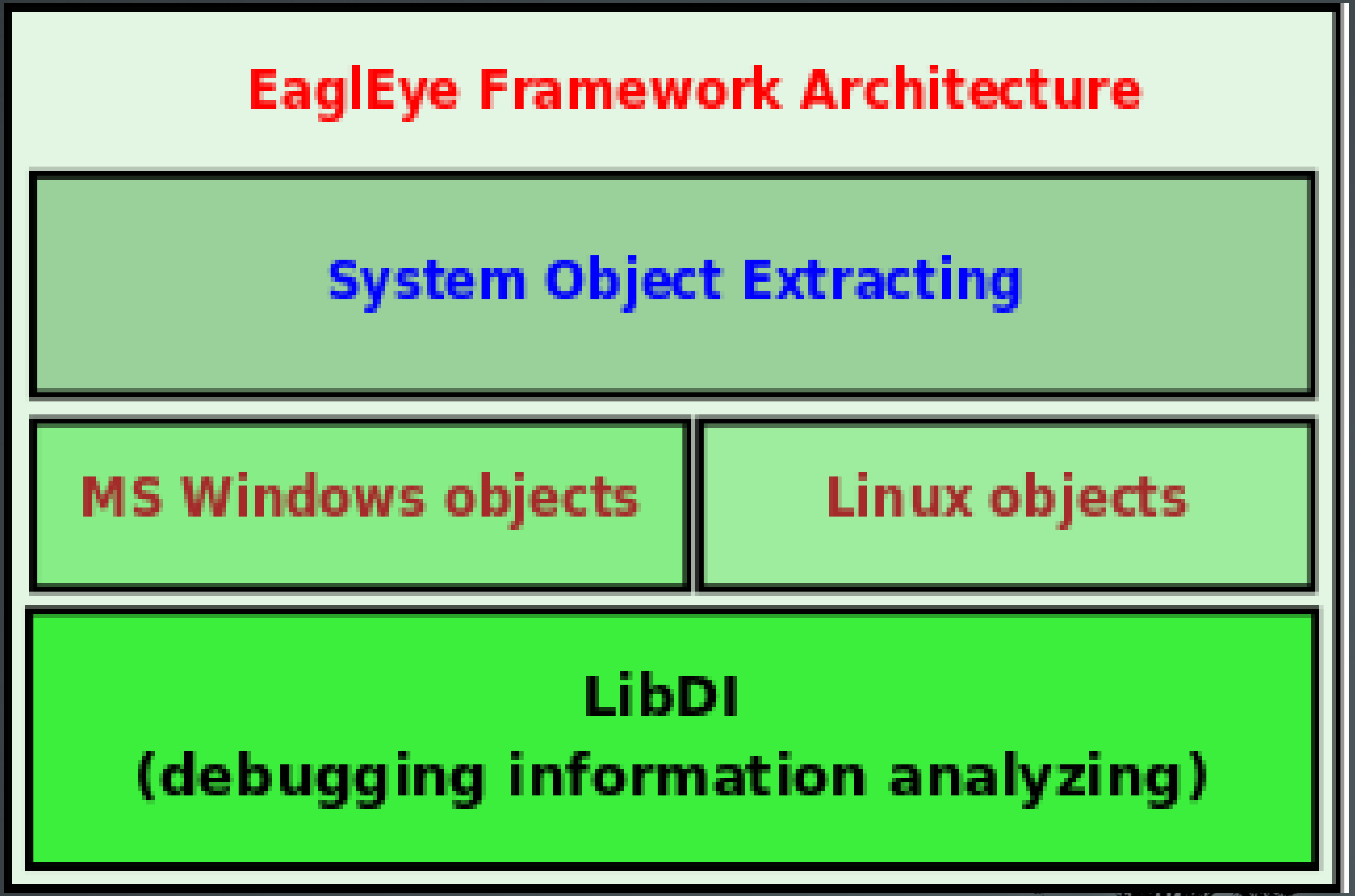
Understand guest context

- Must be done from outside, without any support of guest VM
  - VM instrospection problem
- Leverage works from last year
  - See Syscan '09, FrHack '09, HITB '09, DeepSec '09
  - EaglEye framework
    - Extract OS semantic objects from VM's memory
    - Support Windows OS

EaglEye Framework

- Get access to guest memory and CPU context from host
  - Provided by Kobuta framework (see later)
- Retrieve OS-objects from virtual/physical memory of guest VM
  - Focus on important objects, especially which usually exploited by malware
    - Network ports, connections
    - Processes, DLL, registries, ...
    - Kernel modules
    - etc...

EaglEye architecture



Challenges

- Retrieve semantic objects requires excellent understanding on OS internals
  - Locate the objects
  - Actually retrieve objects and its internals
    - How the objects are structured?
      - Structure size?
      - Structure members?
      - Member offset?
      - Member size?
      - ...

Locate OS's objects

- Kernel modules
- Processes/threads
- System handles
- Open files
- Registries
- DLLs
- Network connections/ports
- Drivers, symbolic links, ...

Retrieve objects' intenals

- Must understand object structure
  - Might change between Windows versions, or even Service Pack

```
struct _EPROCESS {
    KPROCESS Pcb;                          → offset 0, size 0x6c
    EX_PUSH_LOCK ProcessLock; → offset 0x6c, size 4
    LARGE_INTEGER CreateTime;              → offset 0x70, size 8
    LARGE_INTEGER ExitTime;                → offset 0x78, size 8
    EX_RUNDOWN_REF RundownProtect;         → offset 0x80, size 4
    ....
```

Current solutions?

- Hardcode all the popular objects, with offsets & size of popular fields?
  - Does by everybody else
  - But this is far from good enough!
    - Limited to objects you specify
    - Limited to only the offsets you specify

A dream ...

- To be able to query structure of all the objects, with their fields
    - Support all kind of OS, with different versions
    - On demand, at run-time, with all kind of objects
    - Various questions are possible
        - What is the size of this object?
        - What is the offset of this member field in this object?
        - ...

... Comes true: LibDI

- Satisfy all the above requests, and make your deam come true
    - Come in a shape of another framework
    - Rely on public information on OS objects
        - OS independence
            - Windows and Linux are well supported so far
        - Have information in debugging formats DWARF , and extract their structure out at run-time

Windows internals information

- ReactOS file header prototypes
  - Free & open to public (http://www.reactos.org)
  - Support Win2k3 and up.
    - Windows XP and prior are not supported

Sample ReactOS code

```
typedef struct _EPROCESS { … // removed some fields for brevity
#if (NTDDI_VERSION < NTDDI_WS03)
        FAST_MUTEX WorkingSetLock;
#endif
    ULONG WorkingSetPage;
#if (NTDDI_VERSION >= NTDDI_LONGHORN)
    EX_PUSH_LOCK AddressCreationLock;
    PETHREAD RotateInProgress;
#else
    KGUARDED_MUTEX AddressCreationLock;
    KSPIN_LOCK HyperSpaceLock;
#endif
… } EPROCESS, *PEPROCESS;
```

Windows objects

- Compile ReactOS file header prototypes with debugging information
- Dynamically extract out information from object files

```
g++  -g windows.c  -DNTDDI_XPSP3 -c -o windows_XPSP3.o
```

Windows objects - Problems

- ReactOS only supports Win2k3 and up
- Need to patch ReactOS headers to support WinXP and prior versions
    - From Windows debugging symbols data
    - Patch size is small
- Fix incorrect and not updated data structures
    - Windows Vista, Windows 2008
- Patch to support recent Windows OS, like Windows 7

Sample LibDI API

```c
/* <libdi/di.h> */

/* Get the struct size, given its struct name */

int di_struct_size(di_t h, char *struct_name);

/* Get the size of a field of a struct, given names of struct and member. */

int di_member_size(di_t h, char *struct_name, char *struct_member);

/* Get the offset of a field member of a struct */

int di_member_offset(di_t h, char *struct_name, char *struct_member);
```

Sample code using LibDI

```
#include <libdi/di.h>
...
di_t h;
/* Initialize LibDI to get a LibDI handle */
di_open("windows_XPSP3.o", &h);
/* retrieve the size of _EPROCESS */
int s1 = di_struct_size(h, "_EPROCESS");
/* retrieve the size of _EPROCESS::CreateTime */
int m1 = di_member_size(h, "_EPROCESS", "CreateTime");
/* retrieve the offset of _EPROCESS::CreateTime */
int o1 = di_member_offset(h, "_EPROCESS", "CreateTime");
/* close when you are done with LibDI */
di_close(h);
```

EaglEye: retrieve objects

- Separate API for each kind of objects
- Designed so it is hard to be abused or tampered by guest VM
  - Get first object in the list of objects
    - Usually the head of object list must be located
    - Or by scanning the pool memory, or scanning in physical memory
      - Using pattern-matching technique
  - Get next objects
  - One by one, until reach the last object

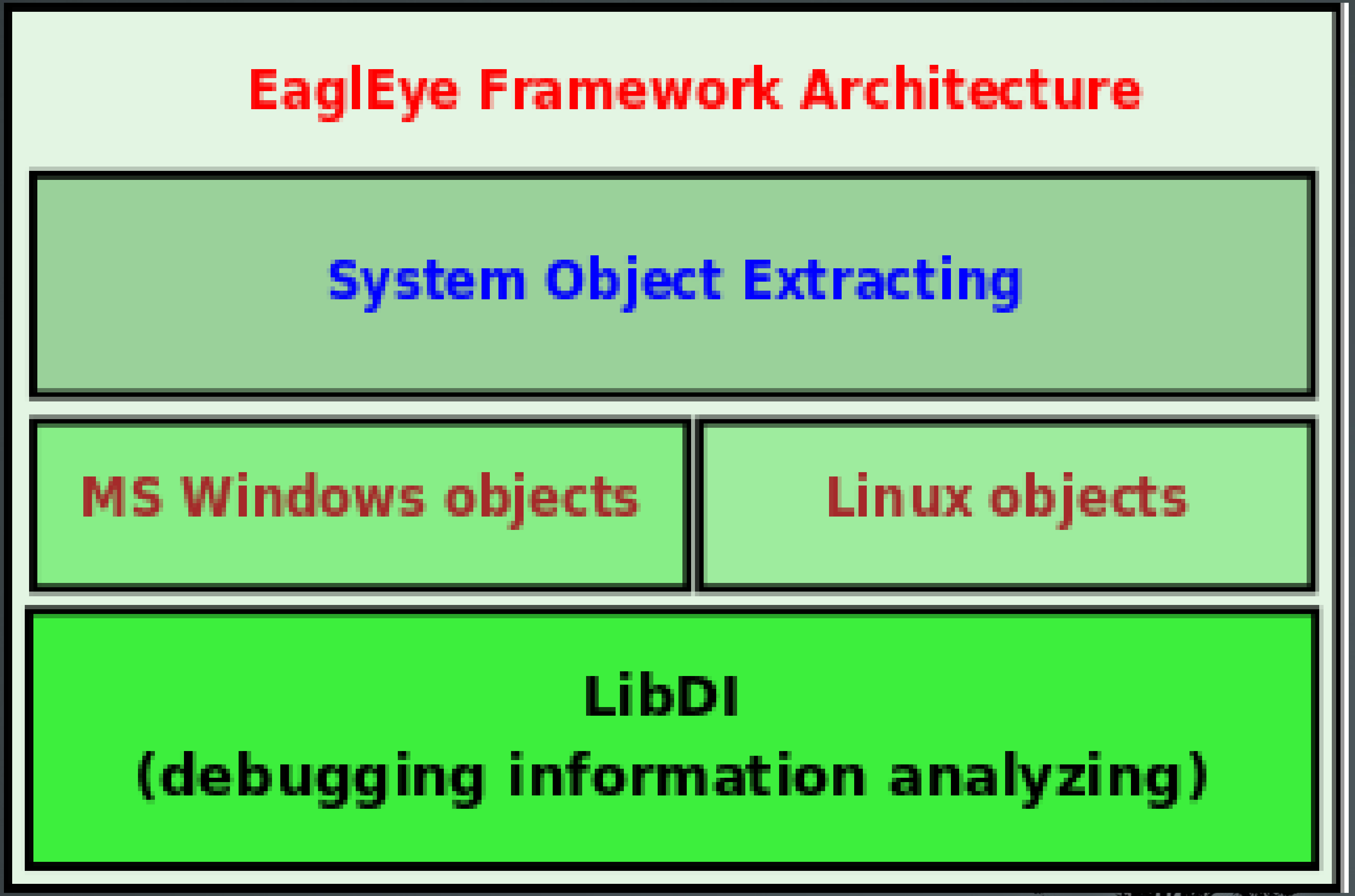
Sample EaglEye API (1)

```
/* <eagleye/eagleye.h> */

/* @task: output value, pointed the the kernel memory keep task info */

int ee_get_task_first(ee_t h, unsigned long *task);

/* @task: output value, pointed the the kernel memory keep task info */

int ee_get_task_next(ee_t h, unsigned long *task);

/* get the pointer to the process struct, given the process's pid.

int ee_get_task_pid(ee_t h, unsigned long pid, unsigned long *task);

/* get the first open dll file of a task with a given process id.

 * on return, dll points to the userspace memory that keeps dll info */

int ee_get_task_dll_first(ee_t h, unsigned long pid, unsigned long *dll);

/* get the next open dll file of a task with a given process id.

int ee_get_task_dll_next(ee_t h, unsigned long *dll);
```

Sample EaglEye API (2)

```
/* <eagleye/windows.h> */

/* get process image filename, given its EPROCESS address */
int windows_task_imagename(ee_t h, unsigned long eprocess, char *name,
    unsigned int count);

/* get process id, given its EPROCESS address */
int windows_task_pid(ee_t h, unsigned long eprocess, unsigned long *pid);

/* get parent process id, given its EPROCESS address */
int windows_task_ppid(ee_t h, unsigned long eprocess, unsigned long
    *ppid);

/* get process cmdline, given its EPROCESS address */
int windows_task_cmdline(ee_t h, unsigned long eprocess, char *cmdline,
    unsigned int count);
```

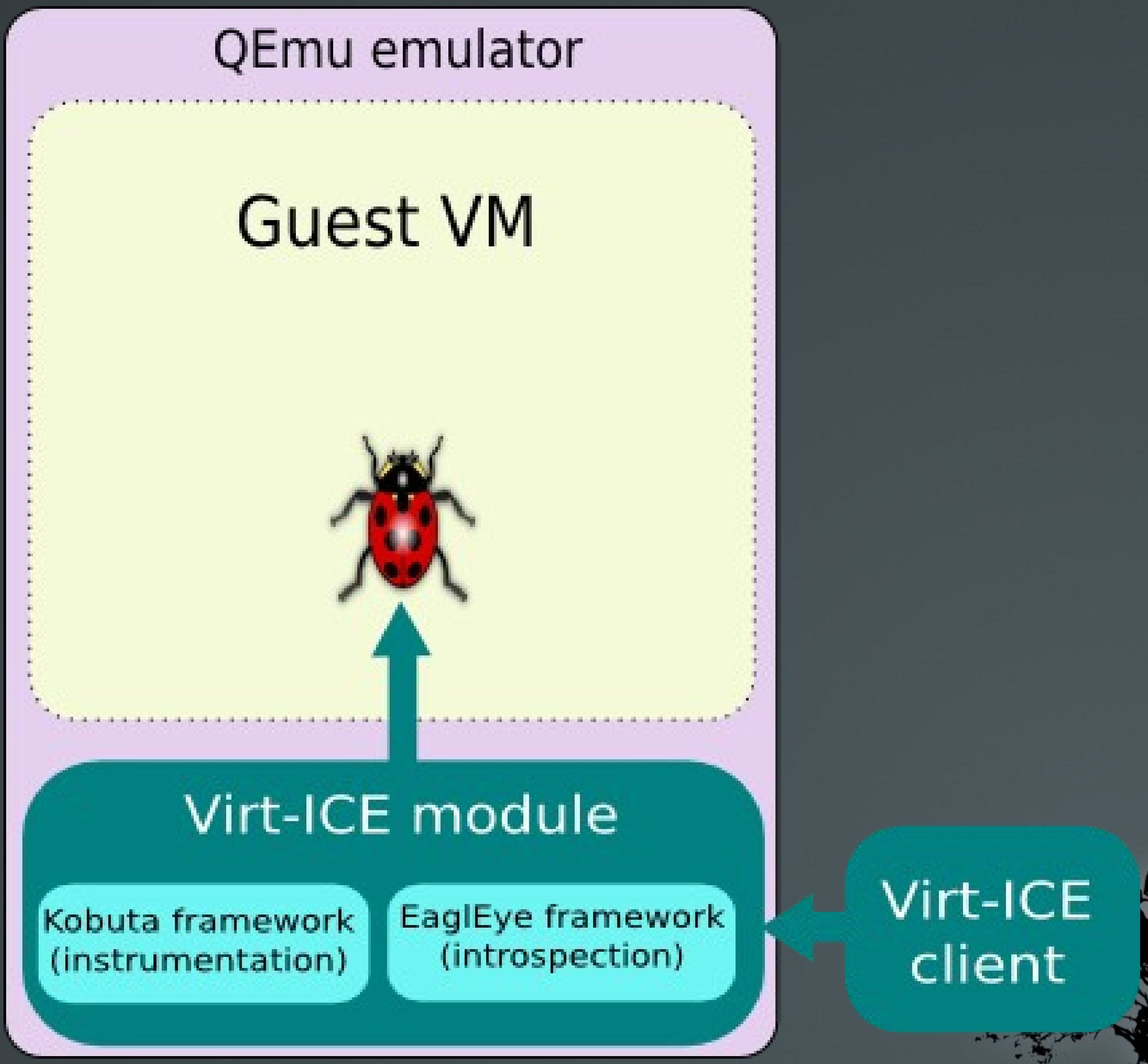
EaglEye architecture



Virt-ICE design

- Choose VM for Virt-ICE
  - Open source, so customizable (therefore VMWare is not suitable)
    - Xen? KVM?
    - VirtualBox?
    - Bochs?
    - Qemu?
      - 0.12.4 version

Virt-ICE architecture



Instrument guest VM

- Kobuta framework
  - Generic instrumentation framework
    - Not only for Virt-ICE, but other internal projects
  - Instrument binary translation process
    - Put hooks at right places to call out to external instrumentation handlers
  - Support dynamic loaded module built on top of Kobuta
    - Module provides external instrumentation handlers to be executed when called from Kobuta hooks

Instrument guest VM – Challenges

- Originally, QEMU provides no support for instrumentation
  - We are on our own, and have to build Kobuta instrumentation framework from scratch
- QEMU uses Just-in-time (JIT) compiler to perform binary translation
  - Translated code is saved, and is not translated again if available in cache
    - We have to dig deeply into the translation process of QEMU to provide instrumental hooks

QEMU JIT compiler

- Translate guest code to TCG Intermediate Representative (IR), then translate TCG IR to native (host) code to execute on host
- The translated code is cached to be reused (to improve performance)
- Translation is done on code block basis
- To improve performance, full CPU context (registers, segments, CR*, ...) is only saved at the end of each translated block
  - So CPU context is only guaranteed to be synchronized at begining of each block
  - At middle of a block, CPU context is out-of-synch
    - We have to synchronize CPU context ourselves when needed
    - On x86, only EFLAGS value is out-of-sync

Instrumentation hooks

- Instrumentation is at TCG IR level (after target code is translated to TCG IR)
  - This is required due to translated code is cached for future reference
- At all cost, avoid putting static hooks into architecture related code, so supporting all architecture can be done universally
  - Instruction level instrumentation is exception
  - Architecture specific instrumentation is also exception
    - Update CR0/2/3/4, RDMSR, WRMSR, …
    - SYSENTER/SYSEXIT
- Make sure performance overhead is minimized when no instrumentation hook is registered

Sample Kobuta instrumentation

```
/* target-i386/op_helper.c */
void helper_sysenter(void)
{  ....
  if (kobuta_ins_sysenter) {      /* SYSENTER hook has been registered? */
    /* Then is it necessary to synchronize CPU context? */
    if (kobuta_ins_sysenter_cpusync == KOBUTA_CPUSYNC_ENABLE)
      kobuta_syn_cpucontext();    /* Synchronize CPU context on demand*/
    kobuta_sysenter();   /* Finally, execute all registered handlers for SYSENTER */
  } ...
}
```

Kobuta framework

- Hooking various places useful for generic purposes
- Fine grain instrumentation
    - Begin/end of instruction/block
    - Jump/call insn
    - Interrupt begin/end
    - Sysenter/Sysexit/Syscall/Sysret
    - Input/Output insn
    - Update control registers (CR0, CR2, CR3, CR4)
    - RDMSR, WRMSR (read/write to Model-Specific-Register)
    - Memory access (read/write)

Performance challenge

- Vanilla QEMU is quite slow
- Accelerate QEMU with KQEMU
    - Software based solution to run most instructions directly on CPU
    - Dynamically enable and disable with Kobuta layer
        - Turn on KQEMU when there nobody registeres for Kobuta
        - Turn off KQEMU when instrumentation is required
    - Support dropped from QEMU 0.12.0 version
        - Had to forward-port to 0.12.4

Kobuta module

- Need to register with Kobuta framework for interested instrumentation events
  - Then provide instrumentation handlers for those events
  - Handlers be executed when events happen in guest VM
- Leverage exported functions (from Kobuta framework) to manage guest VM
  - Pause and Resume VM on demand
  - Read and write to VM's memory (physical & virtual memory) and CPU context
  - Dynamically enable/disable instrumentation hooks

Kobuta module

- Design Kobuta module to be just a Dynamic Linked module
  - .so file in Linux, .DLL file in Windows
  - Loadable into Qemu process, and supported by OS services
  - Easy to implement your Kobuta module (just a normal DL module running in host OS)

Manage Kobuta module

- Manage Kobuta modules
    - Extend QEMU with new command kmodule
    - Allow unlimited number of Kobuta modules to be loaded at the same time
    - Reloading module with different parameter is supported
- Load module into Qemu process
    - Simply using DLL service provided by host OS
        - dlopen() in Linux, LoadLibrary() in Windows
    - Load module with a string parameter
- Unload module from Qemu process
    - Also use DLL service of host OS
        - dlclose() in Linux
    - But how about code (instrumentation handlers) still running?

Unloading Kobuta module

- Use reference counter for Kobuta instrumentation handlers
  - Associate each handler with a ref counter
  - Increase counter before running a handler, and decrease it when done
  - Only run a hanlder when its module is in enable state
  - Have a manage thread to unload Kobuta module
    - Firstly, put the module in disable state
    - Signal the module to interrupt itself
    - Periodically checking for ref counter, and unload module when refcount = 0

Export functions (1)

- Kobuta module needs to manage guest VM
  - Pause & resume the guest
  - Access to guest memory and CPU context
  - Register instrumentation hooks and instrumentation handlers with Kobuta framework
  - But all these functions stay deeply inside QEMU and Kobuta layer
    - Need to export them out for external Kobuta module to use

Export functions (2)

- Two ways to export these functions from QEMU/Kobuta to external modules
  - Refactor QEMU code to export required functions out to an external DLL library
    - The same DLL lib can be linked to both QEMU and Kobuta module
    - Complicated due to too much code needed to be refactored
  - Selectively exports needed function pointers to Kobuta module
    - Transfer these pointers to Kobuta module when external module when loading it
    - Extremely easy to implement, and require minimum modifycation to QEMU

Exported functions (3)

```
/* kobuta.h */

struct kobuta_ins {
   kobuta_cpu_t cpu_read;         /* read CPU context */
   kobuta_cpu_t cpu_write;        /* modify CPU context */
   kobuta_pmem_rw_t mem_rw;       /* physical memory read/write */
   uint64_t ram_size;             /* memory size of guest VM */
   kobuta_virt2phys_t v2p;        /* find physical address of a virtual address */
   kobuta_vm_t vm_pause;          /* request to pause guest VM */
   kobuta_vm_t vm_resume;         /* request to resume guest VM */
   kobuta_manager_t event_manager; /* manage instrumentation hooks */
   int unload();                  /* request (from Kobuta layer) to unload this module */
};
```

Exported functions (4)

```
enum kobuta_handler_reg_t {
        KOBUTA_HANDLER_INSTALL,  KOBUTA_HANDLER_DELETE, ...
};
enum kobuta_cpusync_t {
        KOBUTA_CPUSYNC_DISABLE = 0, KOBUTA_CPUSYNC_ENABLE, ...
};
enum kobuta_event_t {
    KOBUTA_EVENT_JMPCALL, KOBUTA_EVENT_INSN_BEGIN,
    KOBUTA_EVENT_INSN_END,  KOBUTA_EVENT_SYSENTER,
    KOBUTA_EVENT_MEM_READ, KOBUTA_EVENT_MEM_WRITE, ....
};
typedef void (*kobuta_manager_t)(enum kobuta_handler_reg_t reg,
    enum kobuta_event_t event, enum kobuta_cpusync_t sync,
    void *func);
```

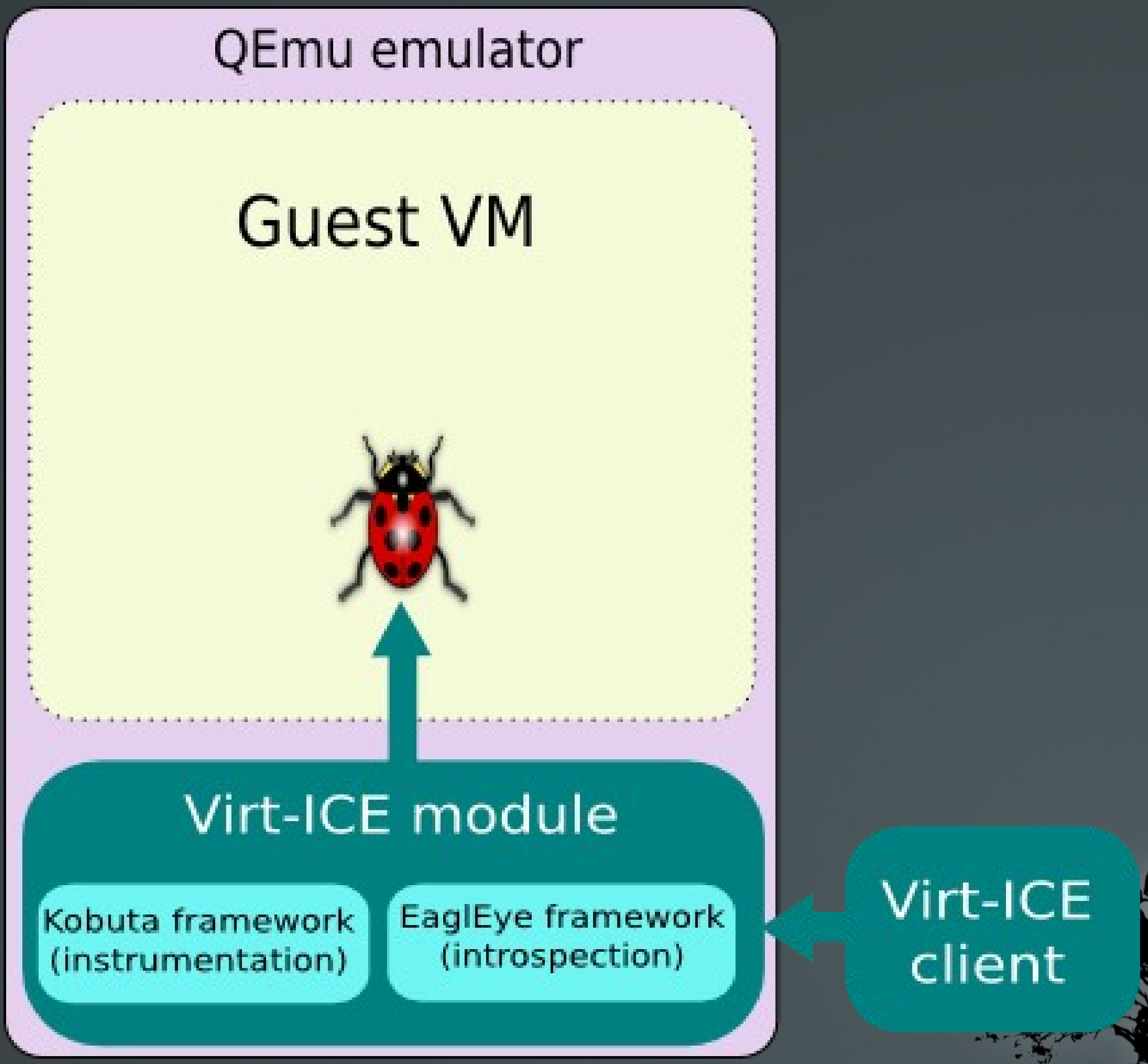
Sample of Kobuta module

```
static void sysenter(void)

{

}

int k_module_init(struct kobuta_ins *ins, const char *args)

{   ...

    ins->event_manager(KOBUTA_HANDLER_INSTALL,
     KOBUTA_EVENT_SYSENTER, KOBUTA_CPUSYNC_DISABLE, sysenter);

    ...

}

int k_module_exit(void)

{

    return 0;

}
```

Virt-ICE debugger design

- A Virt-ICE server: a Kobuta module
  - Register related instrumentation hooks (on demand)
    - JmpCall (to intercept function call)
    - Begin/end insn (for single-step purpose)
    - Begin/end interrupts (to intercept syscalls thru Int 2E)
    - Sysenter/sysexit (to intercept syscalls)
    - Memory access events (to intercept memory read and write)
  - Leverage EaglEye framework to access to objects in guest memory
- A Virt-ICE client
  - Simple front-end to send request and receive results from Virt-ICE module

Virt-ICE architecture



Handling request for Virt-ICE

- Have a separate thread to handle external commands from Virt-ICE client
  - TCP protocol
  - Receive commands from client
    - Built-in protocol for exchanging data between module ↔ client
    - Debugging commands (disasm, breakpoints, watchpoints, singlestep, etc)
    - Monitoring VM status
  - Using exported functions from Kobuta to manage VM
    - Read/write CPU context and memory
    - Run VM into single-step mode
    - Enable instrumentations on demand

Virt-ICE generic commands

- Inspect malware process running inside VM
  - pe: PE file analyzing
  - view: View memory in hex/string format
  - dump: Dump memory out (physical or process or kernel)
  - write: Write to memory
  - search: Searching (pattern matching, regex, ...)
  - ps/pstree: Processes
  - dlls: DLLs, registry: Registries, files: Open files, vad: VADs
  - kmod: Kernel modules
  - address: Attributes of a memory address
  - connection: Open network connections, socket: open sockets
  - disasm: Disassemble memory range
  - register: View all the registers

Virt-ICE debug commands

- Set execution breakpoint: db -s <address>
    - Set syscall breakpoint
- Set memory watchpoint: db -m <address> -c <count> -t <R|W|A>
- Single-step: db -s
- Step over: db -O
- Run until RET: db -R
- Disassemble
- Pause guest VM: db -C | Ctrl+C
- Resume guest VM: db -r

Virt-ICE advanced features

- Malware behavior monitoring
    - API monitoring: db -M <filename>
        - Popular Windows APIs (with semantic arguments)
            - Kernel32, User32, GDI32, AdvApi32, WS2_32, Shell32, OLE32, ...
        - Malware related API monitoring
            - File, Registry, Http, Keylogger, Process, Service, Code injection, ...
    - Syscall monitoring (with semantic arguments)
        - db -Y [filename|ALL|NULL]
- Report anti-debugging techniques used by malware
    - db -A
    - Focus on most popular tricks so far

Demo

Part IV

- Problems of debugger in malaware analysis
- Virt-ICE solution
    - Architecture, Design & Implementation
    - Main features
- Live demo
- Discussions
- Conclusions
- Q & A

Anti Virt-ICE

- Detecting Virt-ICE?
  - Timing attack based on delay execution introduced by the Kobuta instrumentation framework
  - Timing debugger delay using external clock
    - Everybody suffers, not only us!
    - We fix the problem with internal clock, however
- Attack Virt-ICE?
  - Not possible by design due to strong isolation between guest and emulator
- Anti-virtualization malware?
  - Out-of-scope of this research
  - Everybody suffers, too :-)

Future plan - Development

- Improve binary analysis
  - More semantic information
  - GUI ?
- Unpacking tool (in progress)
- Taint analysis tool (in progress)
- Improve performance
    - Using KVM to speed up even further
      - Even currently, KQEMU is not too bad, either
- Re-playable debugger
  - So replay debug process is possible
  - Take snapshot of memory and HDD and rollback

Conclusions

- Virt-ICE is a new debugger that can fix most problems of current debuggers against malware
  - Leverage VM technology
  - Invisible (mostly) against malware
  - Tamper-resistant against malware
  - Provide rich functionality for malware analysis

References

- Peter Ferrie [VIRUS BULLETIN]
  - Anti-Unpacker tricks (series)
- Xu Chen [NDSS08]
  - Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware
- BitBlaze project
  - Presented in BH US' 10 (yesterday)
  - TEMU framework targets tainting analysis
    - Not a generic instrumentation framework like Kobuta
    - Based on old version of QEMU (0.9) with very different JIT engine

Virt-ICE: next generation debugger for malware analysis

Q & A

NGUYEN Anh Quynh <aquynh @ gmail.com>

Kuniyasu SUZAKI <k.suzaki @ aist.go.jp>