Precursors of Security and Performance

Instrumentation and Tracing of Systems

Suchakra Sharma

17th August 2017
Security and DevOps Meetup, Santa Clara
Le Plan

Observability Layers

Instrumentation of Systems
  - Challenges
  - Techniques

Foundations
  - Performance
  - Security

Security Tooling
  - Preventive (Isolation)
  - Passive (Monitoring)
  - Active (Protection)
Suchakra

- Staff Scientist, ShiftLeft Inc.
- PhD, DORSAL Lab, Polytechnique Montréal - University of Montréal
- Loves tracing, security, performance analysis, hardware dev, poutine and samosas
- @tuxology
Observability Layers

- Host OS
- Hardware

Kernel functions, hardware perf counters
Observability Layers

- Guest OS
- Virtualization
- Containerization
- Host OS
- Hardware

Kernel functions, Hypervisor, Custom APIs
Kernel functions, hardware perf counters
Observability Layers

- Hardware
- Virtualization
- Containerization
- Applications
  - Host OS
  - Guest OS
  - Virtualization
  - Applications
    - Libs
    - Kernel functions, hardware
    - Hypervisor, Custom APIs
    - Library functions, Syscalls
    - Kernel functions, perf counters
Observability Layers

- **Functions**
  - Application functions
  - Library functions, Syscalls
  - Kernel functions, Hypervisor, Custom APIs
  - Kernel functions, hardware perf counters

- **Applications**
  - Guest OS
  - Virtualization
  - Containerization

- **Host OS**
- **Hardware**

©2017 ShiftLeft Inc.
Observability Layers

- Hardware
- Virtualization
- Containerization
- Applications
- Host OS
- Guest OS

- Instructions
  - Branches, calls

- Functions
  - Application functions
  - Library functions, Syscalls

- Applications
  - Kernel functions, Hypervisor, Custom APIs
  - Kernel functions, hardware perf counters

- Libs
- Virtualization
- Containerization
Challenges

- Horizontal spread of services has increased

- Apps are distributed across machines and geographies
- Visibility across horizontal and vertical layers
- Preventive, Monitoring and Enforcing security for cloud-native applications is non-trivial now
- Developers need awareness of production setups in the modern world
Challenges

- Horizontal spread of services has increased

- Apps are distributed across machines and geographies

- Visibility across horizontal and vertical layers

- Preventive, Monitoring and Enforcing security for cloud-native applications is non-trivial now

- Developers need awareness of production setups in the modern world
Instrumenting Systems

Why

- Applications can now assist you in performance and security analysis
  - Understand program and data flow
  - Analyze timings and compare executions
- Powerful debugging using traces where debugging is prohibitively expensive

How

- Simple. Insert extra code at desired locations in any layer of the system (app, library, host OS)
- Add a `printf()` function. Congrats 🎉
void set_tire_dim() {
    tire_dia = 26;
    tire_width 2;
}
Instrumentation
Instrumentation

Tracepoints
Detailed trace data
Program Flow

Detailed trace data
Instrumenting Systems

Static Instrumentation
- Development time, eg. insert code that takes a timestamp at function entry and saves it
- Compile time. Compiler inserts hooks that you can latch onto at runtime

Dynamic Instrumentation
- Patch a live application, insert your own observation code, let it run
  - Reliability
  - Security
- Translate code to another form, instrument it, run it synthetically
Instrumenting Systems

- Compiled Observation Function
- Program Flow, Arguments, Latency
Kernel Tracing with Kprobes

- Dynamic instrumentation based on trap or jump-pad based instrumentation

```c
do_sys_open()
```
Kernel Tracing with Kprobes

- Dynamic instrumentation based on trap or jump-pad based instrumentation

Prepare Kprobe

```
jmp
```
```
do_sys_open()
```
Kernel Tracing with Kprobes

- Dynamic instrumentation based on trap or jump-pad based instrumentation

Prepare Kprobe

```
jmp
```

do_sys_open()

```
Save registers
CALL pre_handler
Restore registers
JMP back
```
Kernel Tracing with Kprobes

- Dynamic instrumentation based on trap or jump-pad based instrumentation

Prepare Kprobe

```
do_sys_open()
```

```
jmp
```

Save registers
CALL pre_handler
Restore registers
JMP back

User handler
eBPF program
Case Study - I

Kernel Tracing with Kprobes

- Dynamic instrumentation based on trap or jump-pad based instrumentation
Kernel Tracing with Kprobes

- Dynamic instrumentation based on trap or jump-pad based instrumentation
Kernel Tracing with Kprobes
- Dynamic instrumentation based on trap or jump-pad based instrumentation

**Diagram**
- **Run Kprobe**
  - `do_sys_open()`
  - `jmp`
- **Save registers**
  - `CALL pre_handler`
  - `Restore registers`
- **JMP back**
- **User handler**
  - `eBPF program`
Kernel Tracing with Kprobes

- Dynamic instrumentation based on trap or jump-pad based instrumentation

```
do_sys_open()
```
Kernel Tracing with Kprobes

- Dynamic instrumentation based on trap or jump-pad based instrumentation

```
do_sys_open()
jmp
Save registers
CALL pre_handler
Restore registers
JMP back
User handler
```

©2017 ShiftLeft Inc.
Other Techniques

Ftrace - Dynamic (Kernel)
- Build kernel functions with `mcount (-pg in GCC)`
- Patch it to NOPs at boot. Add `jmp` to handler for activated functions
- Add hooks, save function arguments, timestamp
- Generate function graph

Static Tracepoints (Kernel)
- `trace_*` in most kernel functions
- Uses `TRACE_EVENT` based static tracepoints
- Well defined kernel trace events, can be attached to `perf`, Ftrace (Static)
Case Study - I

eBPF + Kprobes

BPF Program

trace.bpf

LLVM/Clang

bpf()

Verifier + JIT

BPF Code

Kprobe

Kernel Function

BPF Map

Trace Pipe

Perf Buffer

Userspace

Monitor/Store

Read Events

Kernel
Case Study

eBPF + Kprobe

- IOVisor BCC – Python, C++, Lua, Go (gobpf) APIs
- Compile BPF programs directly via LLVM interface
- Helper functions to manage maps, buffers, probes

Example

```python
from bcc import BPF

prog = ""
int hello(void *ctx) {
    bpf_trace_printk("Hello, World!\n");
    return 0;
}
""

b = BPF(text=prog)
b.attach_kprobe(event="sys_clone", fn_name="hello")
print "PID MESSAGE"
b.trace_print(fmt="{} {}")
```

Complete Program
trace_fields.py

Prog compiled to BPF bytecode

Attach to Kprobe event

Print trace pipe
Case Study

eBPF + Uprobes Example

```
#include <uapi/linux/ptrace.h>
#include <uapi/linux/limits.h>

int get_fname(struct pt_regs *ctx) {
    if (!ctx->si)
        return 0;
    char str[NAME_MAX] = {};
    bpf_probe_read(&str, sizeof(str), (void *)ctx->si);
    bpf_trace_printk("%s\n", &str);
    return 0;
}
```

```
b = BPF(text=bpf_text)
b.attach_uprobe(name="/usr/bin/vim", sym="readfile", fn_name="get_fname")
```

Output

```
# ./vim-test.py
TASK   PID    FILENAME
vim    23707  /tmp/wololo
```
Linux Security Modules (LSM)

- Static Instrumentation in the kernel code
- Hooks to attach LSM implementations for defining and inserting MAC policies
  - SELinux
  - AppArmor, LandLock LSM (eBPF)

**syscall from userspace**

```c
open()
do_sys_open()
do_filp_open()
path_openat()
...
vfs_open()
do_dentry_open()
security_file_open()
```

**LSM call**
Case Study - II

SELinux

Policy Management Server


©2017 ShiftLeft Inc.
SELinux

Policy Management Server

 SELinux Policy

Compile and Build Binary

Policy Load

Process open("foo")

Query policy

selinuxfs

Security Server

Access Vector Cache

LSM HOOKS

Kernel Resource Access

Policy Store
Case Study - II

SELinux

- SELinux Policy
- Compile and Build Binary
- Policy Management Server
  - Policy Load
  - selinuxfs
  - Security Server
  - Policy Store
- Access Vector Cache
  - LSM HOOKS
  - Query policy
  - Deny access
- Audit Log
  - type=AVC msg=audit(XXXXX):
    avc: denied { getattr } for pid=18923
    comm="httpd"
    path="/var/www/html/file42"
    dev=dm-0
    ino=284133
Securing Stuff
Securing Strategy

Preventive (Isolation)

- Virtualization (Hypervisor/VMs)
  - Inherent isolation, by virtue of hardware/software design
  - Robust, smaller attack surface
- Linux Namespaces/Cgroups (Containers)
  - Isolation by host kernel/userspace support
  - Isolate resources and groups of processes
  - Used to define containers
- Linux Capabilities: Not just root/non-root now
- SECCOMP* (Application)
  - Can be used to sandbox process and allow/deny syscalls
Passive (Monitoring)

- System level logs and audit messages (Auditd)
  - Get summary of AVC denials/syscalls to keep track of interesting events
- Hook to system events such as capability, syscalls custom userspace events
- Monitor network events across layers
Securing Strategy

Active (Protection)

- LSM Modules
  - Protect infrastructure and implement policies
  - Policies can now be programmable (eBPF)
  - Support with modern containers
    (policies for Docker, rkt, K8s)
- SECCOMP
  - Policies for Docker, rkt, runc
- Custom application/library instrumentation
Securing Strategy

Insertion Spectrum

- Two variables for defining and inserting security

Diagram: Graph with axes labeled "Computing Layer" and "Software State"
Securing Strategy

Insertion Spectrum

- Two variables for defining and inserting security
Securing Strategy

Insertion Spectrum

- Two variables for defining and inserting security
References

Links
- **Kprobes Kernel Docs**

- **Secrets of Ftrace Function Tracer (Steven Rostedt)**
  - https://lwn.net/Articles/370423/

- **Linux Performance/Tracing (Brendan Gregg & Julia Evans & Honeycomb)**
  - https://jvns.ca/blog/2017/07/05/linux-tracing-systems/
  - https://honeycomb.io/blog/categories/instrumentation/

- **Linux Security/Containers (Jessie Frazelle & Jérôme Petazzoni)**
  - https://blog.jessfraz.com/post/a-rant-on-usable-security/
  - https://www.slideshare.net/jpetazzo/anatomy-of-a-container-namespaces-cgroups-some-filesystem-magic-linuxcon

- **eBPF/Landlock LSM (IOVisor devs, Mickaël Salaün et al.)**
Fin

suchakra@shiftleft.io
@tuxology