Summary for Winter Holiday

Haonan Lu, Wenxuan Shi, Xueying Zhang

COMPASS

March 2, 2021

1 Outline

The ETM Debugging Project

- Sync.
- Read the data flow derivation python code.

Group Project (Reverse Debugging)

- Put together a document, introduce to “reverse debugging”.
- For example, GDB reverse debugging doesn’t need any data flow. It use the GDB interface to directly execute instructions on the machine.
- Keep focusing on the replay mechanism.

The ETM debugging project

Original Design:

a replay machanism -> **reproduce** the bug -> help to find buggy instructions.

Currently Design:

restore correct control and data flow -> **static** analysis -> find buggy instructions.

Challange

Since I was about to research on the replay machanism, there is nothing I can do for now.

The Group Project (Reverse Debugging)

Record and **Replay**.
Depend on the effect of control flow and data flow restoration.
Need more feedback from Record part. Syncing...

- What can I do for the ETM debugging project?
- Explore different ways of parallel programs replay machanism. (modify priority, hook and add delay, ...)

- In replay stage, we cannot re-construct the return value of **syscall**
- Find a way to log all **syscalls** in the record stage

```
static void el0_svc_common(struct pt_regs *regs, int scno, int sc_nr,
                           const syscall_fn_t syscall_table[])
{
  // ... some pre-check ...

    invoke_syscall(regs, scno, sc_nr, syscall_table);

  // ... tracing status check

trace_exit:
    syscall_trace_exit(regs);
}
```

```
static void invoke_syscall(struct pt_regs *regs, unsigned int scno,
                           unsigned int sc_nr,
                           const syscall_fn_t syscall_table[])
{
    long ret;

    // ... some checks and find in syscall_table

    regs->regs[0] = ret;
}
```

Linux has provided hook positions.

```
void syscall_trace_exit(struct pt_regs *regs)
{
    audit_syscall_exit(regs);

    if (test_thread_flag(TIF_SYSCALL_TRACEPOINT))
        trace_sys_exit(regs, regs_return_value(regs));

    if (test_thread_flag(TIF_SYSCALL_TRACE))
        tracehook_report_syscall(regs, PTRACE_SYSCALL_EXIT);

    rseq_syscall(regs);
}
```

Besides, we also need hook in the enter of syscall, which also provided by Linux.

```
int syscall_trace_enter(struct pt_regs *regs)
{
    // ... some pre-check ...

    if (test_thread_flag(TIF_SYSCALL_TRACEPOINT))
        trace_sys_enter(regs, regs->syscallno);

    // ... audit syscall entry

    return regs->syscallno;
}
```

3 Outline

- **sysdig** is a universal system visibility tool.
- sysdig leverages `tracepoints` and load drivers to capture kernel events.

A receipt for using sysdig to capture syscalls for a process named zsh and with pid 3981

```
root@ubuntu:/home# sysdig proc.name=zsh and proc.pid=3981
3344 ... < read res=1 data=z
3345 ... > rt_sigprocmask
3346 ... < rt_sigprocmask
3349 ... > fcntl fd=0(<f>/dev/pts/2) cmd=1(F_DUPFD)
3350 ... < fcntl res=11(<f>/dev/pts/2)
3351 ... > close fd=0(<f>/dev/pts/2)
3352 ... < close res=0
3353 ... > openat
3357 ... < openat fd=0(<f>/dev/null) dirfd=-100(AT_FDCWD) ...
3362 ... > mmap addr=0 length=16384 prot=3(PROT_READ|PROT_WRITE) ...
3363 ... < mmap res=7F3CB2E39000 vm_size=53820 vm_rss=6456 vm_swap=0
3364 ... > rt_sigprocmask
3365 ... < rt_sigprocmask
```

I have made following attempts last week:

- Compile and Install Sysdig on Juno: *\textcolor{red}{Failed to compile finally}*
- Install Debian on Juno and then install Sysdig: *\textcolor{red}{Critical isssue: cannot use ETM}*
- Write syscall hook manually: *\textcolor{blue}{In progress}*

But what do these bemused numbers represent for?

```
[  207.532130] [my_sysdig:] call syscall 0x122bbec0, 2: 0x3f, 3:0x8d66090
[  207.532135] [my_sysdig:] call syscall 0x11d4bec0, 2: 0x3f, 3:0x8d66090
[  207.532137] [my_sysdig:] call syscall 0x11cf3ec0, 2: 0x42, 3:0x8d66090
[  207.532141] [my_sysdig:] call syscall 0x122bbec0, 2: 0x3f, 3:0x8d66090
[  207.532143] [my_sysdig:] call syscall 0x11cf3ec0, 2: 0x3f, 3:0x8d66090
[  207.532146] [my_sysdig:] call syscall 0x122bbec0, 2: 0x3f, 3:0x8d66090
[  207.532150] [my_sysdig:] call syscall 0x11cf3ec0, 2: 0x16, 3:0x8d66090
[  207.532151] [my_sysdig:] call syscall 0x122bbec0, 2: 0x3f, 3:0x8d66090
[  207.532155] [my_sysdig:] call syscall 0x11d4bec0, 2: 0x3f, 3:0x8d66090
[  207.532159] [my_sysdig:] call syscall 0x122bbec0, 2: 0x3f, 3:0x8d66090
[  207.532162] [my_sysdig:] call syscall 0x122bbec0, 2: 0x3f, 3:0x8d66090
[  207.532166] [my_sysdig:] call syscall 0x11d4bec0, 2: 0x39, 3:0x8d66090
[  207.532169] [my_sysdig:] call syscall 0x11cf3ec0, 2: 0x42, 3:0x8d66090
[  207.532173] [my_sysdig:] call syscall 0x122bbec0, 2: 0x3f, 3:0x8d66090
```

- We need install sysdig from source code and configure all dependencies manually.
- Or we could also hook syscalls as sysdig does.

We need do some experiments to:

- figure out the overhead.
- judge whether the information captured is enough.
- make a demo to finish original schedule.

4 Acknowledgement

- Chang Zhu: Introduce Sysdig
- HongYi Lu: Help to locates the syscall number(?) in a mysterious way