Crashing the Kernel for Fun and Profit

GUUG FFG 2013

Stefan Seyfried <seyfried@b1-systems.de>
Ralf Lang <lang@b1-systems.de>
B1 Systems GmbH

http://www.b1-systems.de

March 1, 2013

Crashing the Kernel for Fun and Profit

A way to test systems monitoring software.

Actually, the correct tagline for this talk is:

Hanging the Kernel for Fun and Profit

(but it's not as catchy)

Hanging the Kernel: Why?

- to test systems monitoring software
- it's fun and you learn quite some things about kernel internals

Interjection: Systems Monitoring

Or "Why would you want to crash the Kernel?"

- Systems monitoring is necessary for ensuring availability
- Automated restart of nonresponding machines
- With lots of servers, automated response is useful
- Lower support load with automatic recovery

Systems Monitoring: The Situation

- Large Xen Setup (about 1000 Hypervisors)
- Custom built infrastructure to automatically start, stop, migrate and deploy VMs
- 2nd Level support team can fix almost everything, in the worst case a new VM is deployed and the application is migrated onto the new VM

Systems Monitoring: Automatic Recovery

- Detect if a VM is "dead". Reliably. Really.
- Also detect undead VMs - looks alive, but actually is not.
- If the VM is dead, restart it
- Some additional conditions apply

System Recovery: Additional Constraints

- A VM problem might actually be a problem of the Hypervisor
- Possible hint: all VMs on a Hypervisor die at the same time
- Basic rule: if a VM does not sart, use a different Hypervisor
- Heuristics to not even try to use the original Hypervisor

System Recovery: Testing

- Worst case: errors in the monitoring / recovery tool cause downtime
- To avoid false positives, realistic tests are needed
- For good tests, realistic hangs / crashes have to be injected

Why not just SysRQ-C?

- hard crashes are actually relatively easy to detect
- crashdump setups or hypervisors notice the crash and just restart the machine
- "ping" stops working
- hangs are not that obvious

Different kinds of system hangs

all encountered on real systems:

- ping still works, but no `accept()` from sshd
- `sshd` connects, but cannot start the shell
- some NFS subtrees are not accessible anymore (every process that tries to access them hangs)
- `/proc` related problems (`ps` and friends just hang)
- block device access hangs
- ...

How to simulate such hangs from a kernel module?

- almost all those hangs are probably locking problems
- write a module that grabs one of those locks
- done

Unfortunately, it is not *that* easy . . .

- many of those locks are in kernel core and not exported (a module *should not* mess with them)
- some of the interesting locks are hard to find (they belong to the `task_struct` or similar dynamic data structures, which are not easily available)
- apparently, kernel is designed to not be broken

The dirty hacks . . .

- find symbol address in `System.map` . . .
- . . . and put that into the module
- unfortunately, those addresses change for every kernel version / flavour
- fortunately, there is `kallsyms_lookup_name()`

What is already there

- some different methods to hang the kernel
- most of them hang it hard
- one method to do a soft hang (ping still works)

This was actually easy to do ...

What would be nice to have

- more "soft" hang methods
- breaking only parts of the system
  - block devices blocking, but stuff already in FS cache will keep working
  - disturbing only some processes
- This will be harder to implement.
- We might need to walk the task list or the list of block devices and somehow disturb them.

Why not just use systemtap?

- (depending on the viewpoint) writing a kernel module is easier
- systemtap needs debuginfo matching the running kernel, which is huge

Where is the code?

Source code for the module and a short README is at:

`https://github.com/seife/hangman-ko`

Thanks

Questions?

Comments and suggestions: info@b1-systems.de