A design proposal for Xen hotpatching

Martin Pohlack

2014-10-17
Hotpatching building blocks (Linux / Xen)

1. **Preparing:**
   Linux: create special kernel module
   Xen:  ?

2. **Loading:**
   Linux: kernel module
   Xen:  ?

3. **Splicing:**
   Linux: relocation, ftrace, kprobes, ...
   Xen:  ?
1. Preparing hotpatches

- No stable API or ABI
  - Target-specific hotpatches → Build ID for Xen
  - Freeze build environment (gcc, gas)
- Source + patch + compiler output
- Apply patch, build → patched objects
- Binary comparison of trees → changed objects and fns.
- Rebuild with -ffunction-sections → extract changed fns.
- + some glue → hotpatch
- Link against target-specific xen-syms,
- Tag with build-ID
2. Loading hotpatches

- Module system for Xen (similar to Linux' but simpler)
- Activation / deactivation callback into glue
- Linking and relocation in userland (Linux 2.4 insmod)
3. Splicing: how

- Function granularity
- JMP instruction in old function start
  - Redirect to new code
  - x86, ± 2GB → 5B
- Atomically for all target functions
- Anesthesia required for Xen
3. Splicing: when

- **Linux: Kpatch / kGraft**
  - Machine halt vs. incremental patching
  - Permanent kernel threads
  - Some functions never left (e.g., `schedule()`)
  - Inspect kernel stacks

- **Xen: Simpler design possible**
  - No permanent threads, stacks not preserved
  - Global barrier with *timeout* at HV exit, abort and retry
Implementation challenges: Reproducible builds

- Capturing original build environments
  - gcc & gas version stability
  - Koji integration, build tag stability over time

- Xen build system
  - Time: incremental builds hard with Xen
  - compile.h, auto-generated for each build

- Build paths and line numbers
  - In normative parts via __LINE__ / __FILE__ → larger patches
  - Normalize patches + environment, and / or
  - Deep binary comparison logic
Implementation challenges: Detecting modified objects

• Compare at what level?
  – "Normalized" disassembler view vs. memcmp
  – Symbol stability: static (fn.14077), local (.LC27)
  – Deep inspection of .rodata.str*, strings, local jmp tables (switch etc.)
  – Exception tables

• -ffunction-sections
  – No support in Xen, but can be compiled
  – __init etc. → multiple functions in single section, .init.text usually not target of hotpatch
Implementation challenges: Inter-hotpatch dependencies

- Single function multiple times
- Ordered hotpatch building & loading
Implementation challenges: Hotpatch unloading

• Auto-generated modules
  – Arbitrary code, hard to reason about
  – ! module coding conventions (register pointers to itself)
→ Unloading may be unsafe
Discussion

• Shared user-space tooling Linux / Xen?
  – Generalize kpatch / kgraft, also Xen?
• Full-blown module system for Xen?
  – Where should relocation happen, user-space or in Xen?
  – Non-unique local symbols?
  – Hotpatch signing?
• ftrace for Xen?
• gcc-specific assumptions build into tools: icc, clang?
• LTO?
• Hotpatch inter-dependencies?
• Hotpatch unloading safety?