SUSE® Linux Enterprise Live Patching Roadmap: Zero Downtime

Hannes Kühnemund
SUSE Product Management

Session 1, Nov 4, 9:00 AM @ 9-Ontvangkamer
Session 2, Nov 6, 9:00 AM @ 9-Ontvangkamer
Agenda

Quiz

Vulnerabilities

Patching

Downtime

Compliance & Costs

Live Patching
Agenda

Quiz

Vulnerabilities

Patching

Downtime

Compliance & Costs

Live Patching
Quiz
Quiz
Quiz
Quiz
Agenda

Quiz

Vulnerabilities

Patching

Downtime

Compliance & Costs

Live Patching
How safe is your data?

- **80M** (May-2015)
  - Social Security Numbers
  - email addresses
  - physical addresses

- **83M** (Aug-2014)
  - email addresses
  - physical addresses

- **47,000** (Nov-2014)
  - Salary records
  - employment information

- **109M** (Sept-2014)
  - Credit card numbers
  - email addresses

- **145M** (May-2014)
  - email addresses
  - physical addresses
  - login credentials

- **110M** (Dec-2013)
  - Credit card numbers

Since 2005, more than 75 data breaches in which 1,000,000 or more records were compromised have been publicly disclosed.

But what about the non-disclosed ones?

Vulnerabilities – Patching – Downtime

**Year** | **# of vulnerabilities**
---|---
2010 | 4.258
2011 | 3.532
2012 | 4.347
2013 | 4.794
2014 | 7.038

**Vulnerability distribution by product type - 2014**

- **Application**
- **OS**
- **Hardware**

**Kernel / OS** | **# of vulnerabilities (H/M/L)**
---|---
Apple Max OS X | 147 (64/67/16)
Apple iOS | 127 (32/72/23)
Linux | 119 (24/74/21)
Microsoft Windows Server 2008 | 38 (26/12/0)

Agenda

Quiz

Vulnerabilities

Patching

Downtime

Compliance & Costs

Live Patching
Vulnerabilities – **Patching** – Downtime

- Bug Fixing
- New Features
- Compliance
- Legal Requirements
Agenda

Quiz

Vulnerabilities

Patching

Downtime

Compliance & Costs

Live Patching
As a result of your most significant disruption, which of the following turned out to be the greatest impacts to your organization? (Rank 1-3)

- Customer compensation: 13%
- Loss of partner trust/confidence: 16%
- Damaged corporate reputation: 26%
- Loss of customer confidence: 33%
- Lost revenue: 35%
- Lost business opportunities: 37%
- Loss of employee morale: 37%
- Loss of employee productivity: 72%

Base: 66 global disaster a decision-makers and influencers who have declared a disaster or had a major business disruption (multiple responses accepted)

Source: Forrester/Disaster Recovery Journal November 2013 Global Disaster Recovery Preparedness Online Survey
... and it is always about the money
Agenda

Quiz

Vulnerabilities

Patching

Downtime

Compliance & Costs

Live Patching
A Hypothesis on Compliance & Costs
Where are your costs relative to the amount of compliance?

Are they here? Or here?

And what if I need to have a high compliance level?

Are the costs somewhere along this line, independent of the compliance level?
There is always good and bad ...

<table>
<thead>
<tr>
<th>good</th>
<th>bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bug fixing for stability</td>
<td>Risk of Reboot</td>
</tr>
<tr>
<td>Adding new functions</td>
<td>Cost of downtime</td>
</tr>
<tr>
<td>Compliance</td>
<td>Hard to schedule</td>
</tr>
<tr>
<td>Legal requirements</td>
<td>SLA</td>
</tr>
</tbody>
</table>
What if we can get rid of the bad ...?

**good**
- Bug fixing for stability
- Adding new functions
- Compliance
- Legal requirements

**bad**
- Risk of Reboot
- Cost of downtime
- Hard to schedule
- SLA

---

FUT20715- SUSE Linux Enterprise Live Patching Roadmap: Zero Downtime
Agenda

Quiz

Vulnerabilities

Patching

Downtime

Compliance & Costs

Live Patching
The Live Patching Equation

Security & Stability + Less Downtime = Improved Service Availability

Patching without reboot = Live Patching
Key Highlights

- Available for SLES 12 (x86-64)
- Provides fixes for Kernel bugs which affect
  - Security
  - Stability
  - Data Integrity
- No runtime performance impact
- No interruption of applications while patching
- Allows full review of patch source code
- PTFs can be delivered as Live Patches
- Patches available for kernels up to 1y old
Change Management

Common tiers of change management

1. Incident response
   “We are down, actively exploited …”

2. Emergency change
   “We could go down, are vulnerable …”

3. Scheduled change
   “Time is not critical, we keep safe”
NASA JPL
Hale telescope PALM-3000 Adaptive optics

- 5m telescope with adaptive optics on Mount Palomar
- Avoid atmospheric blurring in Real Time
- Control 3888 segments of a deformable mirror with a latency <250 μs
- Reboot?
Google NC data center

- 139,200 servers in the Lenoir location
- Close to 15+ Exabytes storage (by 2013 ...)
- Light weight Applications built on full redundancy, probably stateless
- Reboot?
SAP HANA
In-memory database and analytics engine

- 4-24 TB of RAM
- All operations done in memory
- Disk used for journalling
- Active-Passive HA
- Fail-over measured in seconds
- Reboot?

HP DL980 w/ 12 TB RAM
AWS Demo & 60-Day Eval

https://www.youtube.com/watch?v=8pOha28_69o

https://www.suse.com/products/live-patching/
SLE Live Patching

"What comes after SUSECon 2015"

<table>
<thead>
<tr>
<th>Lineup:</th>
<th>Arrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Live Patches for LTSS Kernels</td>
<td>SOON</td>
</tr>
<tr>
<td>&amp; Live Patches for PTF Kernels</td>
<td></td>
</tr>
</tbody>
</table>

Your ticket into the world of reboot-less Linux patching
Downtime isn’t an option.

Towards Zero Downtime with SUSE.

www.suse.com/zerodowntime
Thank you!

Hannes Kühnemund
SUSE Product Management
hkuehnemund@suse.com
www.twitter.com/hakuehnemund
www.linkedin.com/in/hanneskuehnemund

Appendix
SLE Live Patching Workflow

1. Trigger to create a kernel patch
2. Identify severity
   - uncritical: Create regular kernel patch → Make kernel in Repo available → Install Kernel → Reboot → Up to date and secure kernel
   - critical: Create Live Patching update → Make patch in Repo available → Install Live Kernel Patch → Up to date and secure kernel
3. Optional: Reboot
History of Dynamic Software Updates
1943: Manhattan project – punchcards

- IBM punchcard automatic calculators were used to crunch the numbers
- A month before the Trinity nuclear device test, the question was: “What will the yield be, how much energy will be released?”
- The calculation would normally take three months to complete – recalculating any batches with errors
- Multiple colored punch cards introduced to fix errors in calculations while the calculator was running
- Trinity test site, 16ms after initiation
Modern history of DSU: C language

• DSU: Dynamic Software Updates
  - the goal is to be able to fix bugs and add features
  - either by changing some functions
  - or replacing the whole program

• Let's focus only on C
  - the Linux kernel is (mostly) in C
  - all the major techniques were developed for C
  - C most closely matches the system ABI
1991-1993: PoDUS (University of Michigan)

- The first DSU to work on C in Berkeley Unix
- Uses binary *overwriting* of code *segments*
- The first to include *Activeness Safety*
  - functions are only changed when not running or on stack
- No state format changes allowed

Segment overwriting

Activeness Safety
1994: Deepak Gupta's DSU (IIT)

- Proved that the safety of applying an update is undecidable in general.
  - by reduction to the halting problem
- Replaces whole program with a new version
- Introduces State Transfer
  - no state transformation yet, no state format changes allowed

Whole program replacement

State Transfer
1998: Erlang (Ericsson)

- Not C, an own language, with DSU built-in
- Replacing functions on the fly
- Relies on the programmer for safety
- The first *commercially deployed* DSU
  - widely deployed in telecommunications systems

*Commercially deployed*
2006: Ginseng (U of Maryland, U of Cambridge, ETH Zurich)

- Introducing automated *patch generation*
- Uses *function indirection* and *lazy migration*
- Introducing *type safety*
  - Decides which functions to call based on matching data types

*Patch generation*

*Type safety*

*Function indirection*

*Lazy migration*
2008: Ksplice (MIT, Oracle)

- First to patch *Linux kernel*
- Stops kernel execution for activeness check
  - restarts and tries again later when active
- Uses jumps patched into functions for redirection
  - solves the call by pointer problem

*Commercially deployed*

*Kernel patching*

*Activeness safety*

*Binary patching*
2009: UpStare (Arizona State University)

- Introduces Stack reconstruction
  - rebuilds stacks to match the new software
- Immediate patching
  - no Activeness safety required

Stack reconstruction
Immediate patching
2011: Kitsune and Ekiden (University of Maryland)

- Introduces *State transformation*
  - transforms state to match the new software
- Uses controlled updating with *safe points*

**State transformation**

**Safe points**
2014: kGraft (SUSE)

- Linux kernel patching
- Immediate patching with lazy migration
  - Function type safety
- Commercially deployed

Commercially deployed
Linux kernel
Immediate
Lazy migration
kGraft
kGraft goals

- Applying limited scope fixes to the *Linux kernel*
  - security, stability and corruption fixes

- Require *minimal changes* to the source code
  - no changes outside the kGraft engine itself

- Have no runtime *performance* impact
  - full speed of execution

- *No interruption* of applications while patching
  - full speed of execution

- Allow *full review* of patch source code
  - for accountability and security purposes
Patch Generation
Manual Approach

- kGraft offers a way to create patches entirely by hand
- The source of the patch is then a single C file
  - easy to review, easy to maintain in a VCS like git
- Add fixed functions
- Create a list of functions to be replaced
- Call kGraft: `kgr_patch_kernel();`
- Compile
- Insert as a .ko module
- Done
```c
#include <linux/module.h>
#include <linux/kgraft.h>

static bool kgr_new_capable(int cap)
{
    printk(KERN_DEBUG "we added a printk to capable()\n");
    return ns_capable(&init_user_ns, cap);
}

static struct kgr_patch patch = {
    .name = "sample_kgraft_patch",
    .owner = THIS_MODULE,
    .patches = { KGR_PATCH(capable, kgr_new_capable, true),
                KGR_PATCH_END }
};

static int __init kgr_patcher_init(void)
{
    return kgr_patch_kernel(&patch);
}
static void __exit kgr_patcher_cleanup(void)
{
    kgr_patch_remove(&patch);
}

module_init(kgr_patcher_init);
module_exit(kgr_patcher_cleanup);

MODULE_LICENSE("GPL");
```
Patch Generation Pitfalls

Inlining

- Inlining is when a compiler decides that it's worth copying a function as a whole instead of calling it.

- kGraft offers a tool that given a list of functions, based on DWARF debug information, expands it with all functions where any function from the list was included.
Patch Generation Pitfalls

Static symbols

- Static or unexported symbols are common
- But they may be used from the patched function
- The kernel keeps a list of all symbols: `kallsyms`

```c
int patched_fn(void)
{
    kgr_orig_static_fn();
}

static int __init kgr_patcher_init(void)
{
    kgr_orig_static_fn = (static_fn_proto)kallsyms_lookup_name("static_fn");
    if (!kgr_orig_static_fn) {
        pr_err("kgr: function %s not resolved\n", "static_fn");
        return -ENOENT;
    }

    ...}
```
Patch Generation Pitfalls
IPA-SRA

- Compiler optimization, enabled with -O2 and higher
  - Inter-procedural scalar replacement of aggregates
- Gives a significant performance boost
- But also a disaster for patching
  - Can modify CALL at the end of a function into JMP if it's the last statement of a function
  - Can transform arguments passed by reference to arguments passed by value if the value is never changed
  - Can create variants of a function with fewer arguments if the removed argument doesn't have any impact on what the function does in a specific case
- Again, DWARF to the rescue (and more work creating the patch)
Patch Generation Tooling
Automated Approach

• Now we have a full list of functions to replace
• How about automating the rest?
• Recompile the kernel such that each function has its own section
  - ffunction-sections
  - fdata-sections
• And extract the functions into a single object file
  - Patched objcopy
• Generate a .c stub file linking with the object file
  - Shell scripts
Patch Lifecycle
More Details

• Build
  - Identify changed function set
  - Expand set based on inlining and IPA/SRA compiler decisions
  - Extract functions from built image (or source code)
  - Create/adapt framework kernel module source code
  - Build kernel module

• Load
  - insmod

• Run
  - Address redirection using ftrace
  - Lazy per-thread migration
Call Redirection

How Does It Work

• Use of ftrace framework
  - gcc -pg is used to generate calls to _fentry_() at the beginning of every function
  - ftrace replaces each of these calls with NOP during boot, removing runtime overhead
  - When a tracer registers with ftrace, the NOP is runtime patched to a CALL again
  - kGraft uses a tracer, too, but then asks ftrace to change the return address to the new function
  - And that's it, call is redirected
Call redirection
ftrace: SMP-safe code modification
Call Redirection
ftrace: return address modification mechanism
Call Redirection
The Final Hurdle

• Changing a single function is easy
  – since ftrace patches at runtime, you just flip the switch

• What if a patch contains multiple functions that depend on each other?
  – Number of arguments changes
  – Types of arguments change
  – Return type change
  – Or semantics change

• We need a consistency model
  – Lazy migration enforcing function type safety
kGraft Consistency Model
Keeping Threads Intact

- We want to avoid calling a new function from old and vice versa: Function type safety
- Execution threads in kernel are of four types
  - interrupts (initiated by hardware, non-nesting)
  - user threads (enter kernel through SYSCALL)
  - kernel threads (infinite sleeping loops in kernel)
  - idle tasks (active when there is nothing else to do)
- We want to make sure a thread calls either all old functions or all new
- And we can migrate them one by one to 'new' as they enter/exit execution
- No stopping for anybody
kGraft Consistency Model

![Diagram of kGraft Consistency Model]

- Kernel
  - kernel_func
    - heavy work
    - buggy_func()
  - reality_check
    - which universe are you coming from?
- Userspace
  - buggy_func
  - fixed_func
kGraft Consistency Model

Complications

• How about eternal sleepers?
  – like getty on a console
  – They'll never exit the kernel
  – They'll never be migrated to 'new'
  – They'll block completion of the patching process forever

• #1 Wake them up
  – sending a fake signal (SIGKGRAFT)
  – the signal exits the syscall and transparently restarts it

• #2 Just ignore them
  – once they wake up to do anything, they'll be migrated to 'new'
  – so they're not a security risk
Ksplice Consistency Model
Making a Clean Cut

• Ksplice uses *Activeness safety*

• First `stop_kernel();`
  
    - that stops all CPUs completely, including all applications

• Then, check all stacks, whether any thread is stopped within a patched function

• If yes, resume kernel and try again later
  
    - and hope it'll be better next time

• If not, flip the switch on all functions and resume the kernel

• The system may be stopped for 10-40ms typical