Reboot Reloaded

Patching the Linux Kernel Online

Vojtěch Pavlík
Director SUSE® Labs
SUSE vojech@suse.com

Dr. Udo Seidel
Head of Linux Strategy and Server Automation
Amadeus useidel@amadeus.com

Agenda

What to Expect

- Who and why?
- History
- Helicopter View
- Deep Dive
- Demo
- Summary

SUSE

Who and Why

Vojtech Pavlik

The First One

- Education in Biophysics and EE
- Using Linux since 1994
  - Slackware 1.1.2, kernel 0.97
- Did a bit of kernel development
  - input driver layer, USB, IDE, I^2C, network drivers, x86-64 port, HPET timers ...
- Joined SUSE in 1999
- Director SUSE Labs (kernel, toolchain, Samba) @SUSE

SUSE

Udo Seidel

The Other

- Teacher of mathematics and physics
- PhD in experimental physics
- Started with Linux in 1996
- Linux/UNIX trainer
- Solution engineer in HPC and CAx environment
- Head of the Linux Strategy and Server Automation @Amadeus

SUSE

Kernel Updates

Why at All?

- Business critical workload on Linux
  - Bug fixing
  - New functions
  - Improvements
  - External requirements
- Importance of Security
  - PCI-DSS
  - SAESS-16
  - ISO 27001

SUSE

Reboot

What Is Wrong With That?

- Missing High Availability
  - Penny-wise
  - Change of business/platform goals
- Procedures, Operations
  - Organizational lock
  - Too many share holders
- External requirements
  - Hosting
  - Maintenance windows

SUSE

Reboot

Questioning ...

- Really?
- Always?

==> **Don't think so!**

SUSE

History

History

Long Long Time Ago

- Mainframes
  - Still done at Amadeus
  - Kernel = Control Programm
- Dynamic Software Update
  - Erlang: Hot Code Loading
  - Modula: DYMOS
- Very early Linux days
  - Mainly user space
  - Manual

SUSE

History

More Recently

- Rootkits
  - First around 2000
  - Not only Linux
- Ksplice
  - Initial release in April 2008
  - Oracle acquisition in 2011
- CRIU
  - *C*heckpoint/*R*estore of application *I*n *U*ser space
  - Part of OpenVZ project

SUSE

Helicopter View

kGraft Patches

General Remarks

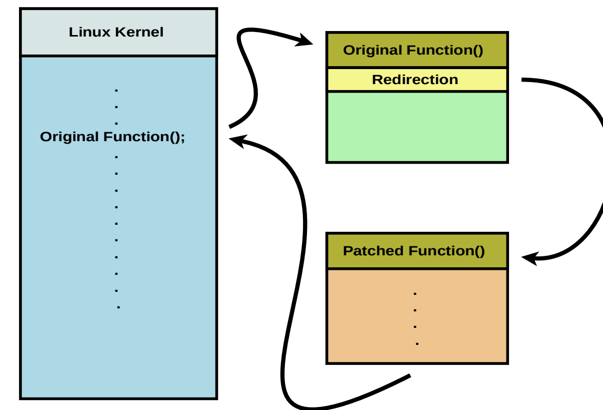
- Cover small changes
  - Severe security bugs
  - Data layout changes => extra work/caution
  - Small update steps
- To be built as Kernel module
- Open questions ... requirements
  - Stack-able
  - Revert-able
  - Supercede-able

SUSE

Live Patching

High Level Procedure

- Find differences
- Create replacement functions
- Load and link patched functions
- Redirect code execution to patched functions
  - Only callee is changed
  - Jump inserted as first instruction
  - Callers unchanged (calling via pointer possible)



Finding Differences

Approaches

- Source code comparison
  - Obvious
  - Requires programming language skills
  - Complex analysis
  - Manual, prone to error, but easy to review
- Object code comparison
  - Can be fully automated
  - Tends to generate false positives
  - But semantic analysis still needs to be done by a human
  - Reviewing object code for correctness is hard to impossible

SUSE

Deep Dive

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 new functions
- Create a list of functions to be replaced
- Call kGraft: **`kgr_patch_kernel();`**
- Compile
- Insert as a .ko module
- Done

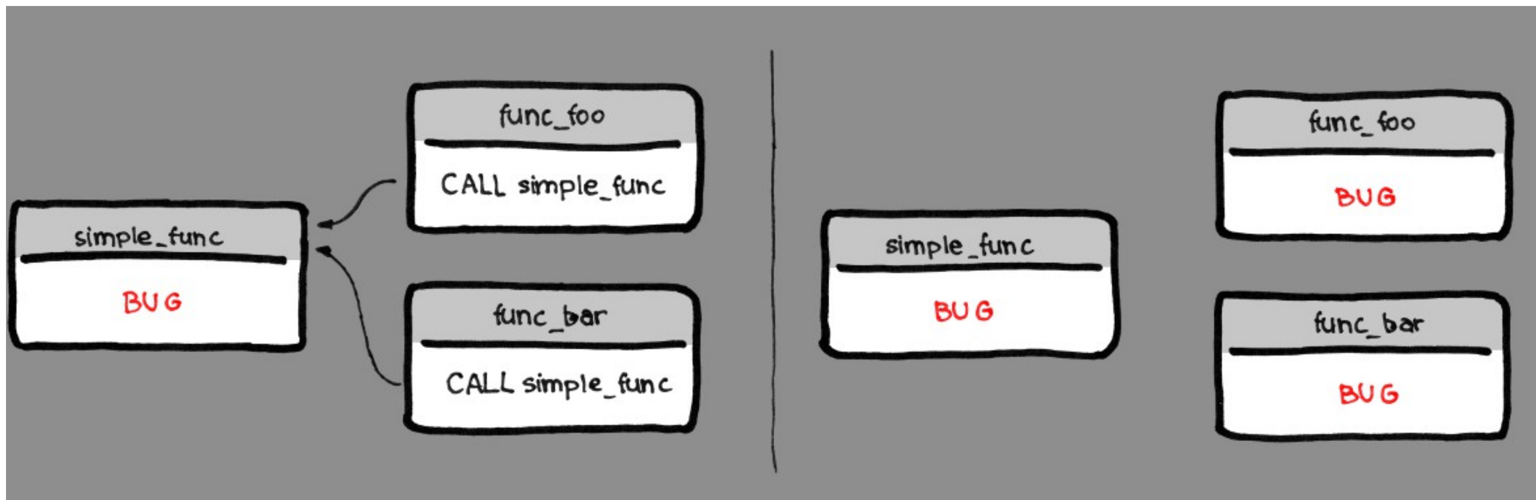
SUSE

```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`**

```
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 starement 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

SUSE

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

SUSE

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

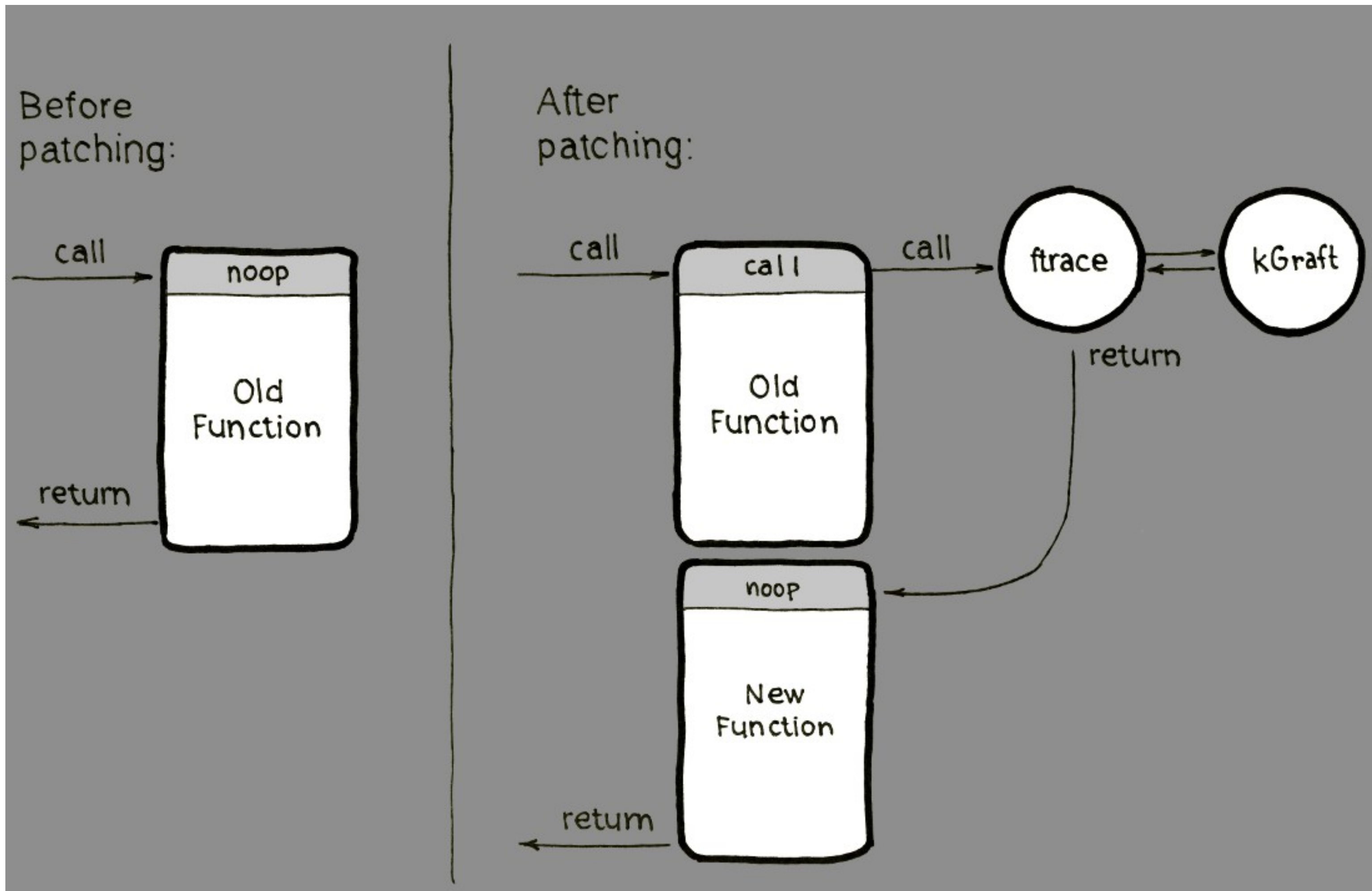
Call Redirection

How Does It Work

- Use of **ftrace** framework
  - gcc -pg is used to generate calls to `_mcount()`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

SUSE

Call Redirection



Call Redirection

More Pitfalls

- `_mcount()` is a regular call
  - the stack is already set up, registers saved
  - returning to `_mcount` location in a different function is not possible
- We could tear down the stack first
  - possible, but rather inefficient and ugly
- `-mfentry` saves us.
  - modifies `-pg` to call `_fentry()` instead
  - `_fentry()` is called before stack setup or anything else
  - redirection works well

SUSE

Call Redirection

Other Architectures

- Only x86-64 implements **`-mfentry`**
- And support for register tracers and modifying the return address is also laching outside x86-64
- So porting kGraft to other architectures is mostly improving ftrace there
- s390x was easy, _**`mcount()`**is reasonably sane there
- ppc64le is tougher, gcc implements **`-mprofile-kernel`**, which is similar to **`-mfentry`**, but other architectural details complicate things (work in progress)
- arm64 would be next

SUSE

Call Redirection

Modules

- A buggy module may not even be loaded while patch is applied
  - kGraft will remember there is an outstanding patch
  - and patch the module if/when it's loaded

SUSE

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**

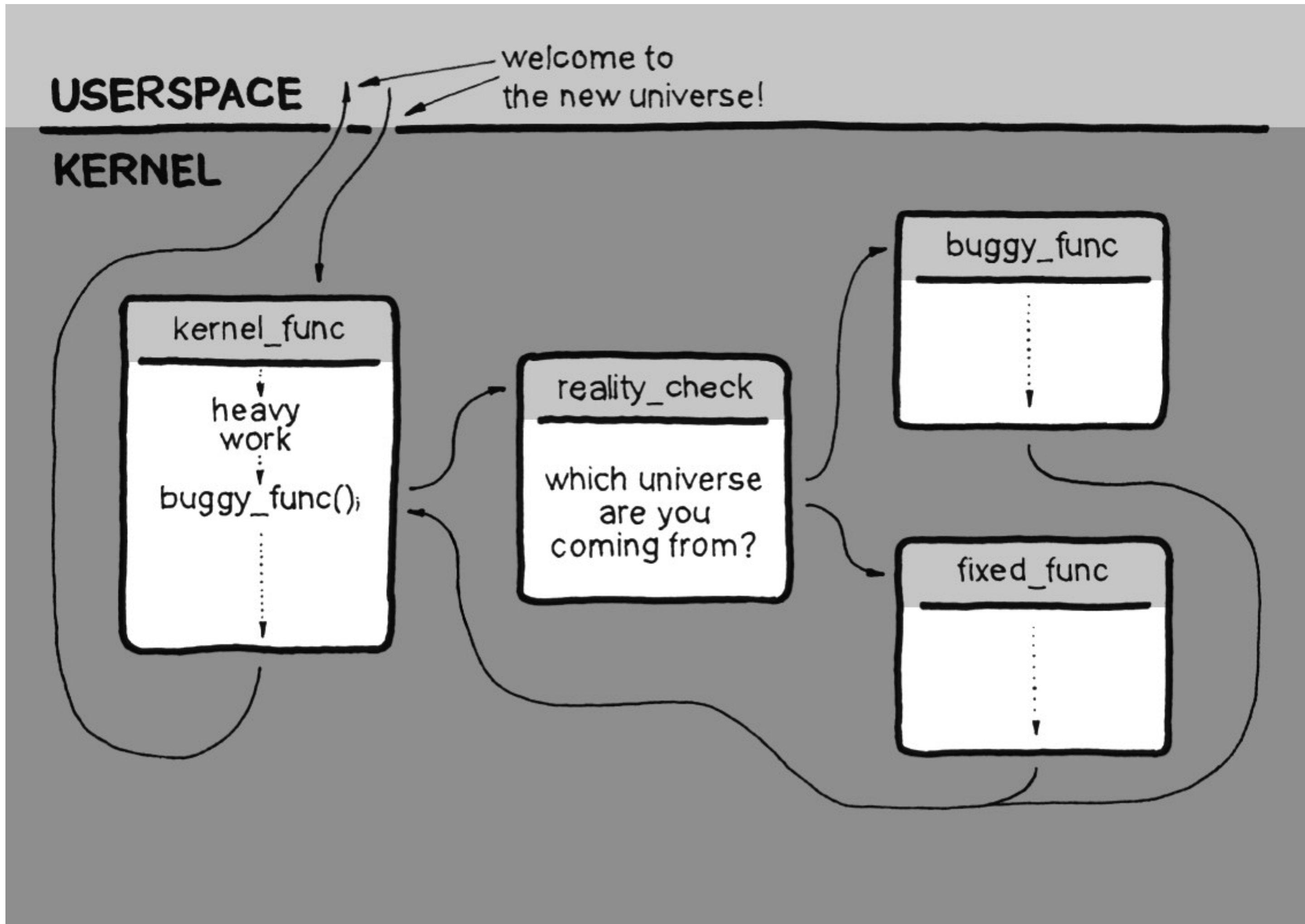
SUSE

kGraft Consistency Model

Keeping Threads Intact

- We want to avoid calling a new function from old and vice versa
- Execution threads in kernel are of three types
  - interrupts (initiated by hardware, non-nesting)
  - user threads (enter kernel through SYSCALL)
  - kernel threads (infinite sleeping loops in kernel)
- 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



SUSE

kGraft Consistency Model

Complications

- How about eternal sleepers?
  - like **`getty`** on a console 10
  - 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 signal (SIGSTOP/SIGCONT) is enough
- #2 Just ignore them
  - once they wake up to do *anything,* they'll be migrated to 'new'
  - so they're not a security risk
  - kGraft can apply the next patch even while current is in progress so no worry there either (parallel patching)

kpatch (Ksplice) Consistency Model

Making a Clean Cut

- kpatch wants the same: to avoid calling a new function from old and vice versa
- 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

kpatch (Ksplice) Consistency Model

Limitations

- Eternal sleepers
  - may sleep in a patched function and never leave it
  - kpatch will simply fail to patch after several tries
  - waking doesn't help, it'd be needed to wake an patch in the same instant, which is not feasible
- Latency
  - The system will be stopped for 10-40ms typical
- New hope
  - Masami Hiramatsu (Hitachi) has implemented a non-stopping kpatch using a refcounting scheme. Latency is gone, wakeups work for sleepers
  - Combined model (Masami+kGraft) possible

Multiple Patches

When One Fix Isn't Enough

- kGraft allows patches to be stacked
- Two types of patches
  - incremental
  - all-in-one
- Incremental patches
  - allow independent fixes, targeted for upstream devs
- All-in-one
  - contain all previous fixes, replace any previous patches
  - this is what SUSE uses for Live Patching
- Patch removal
  - possible, first use **`sysfs`** to trigger unpatch, then **`rmmod`**

Upstream Status

- Agreement since LPC Düsseldorf
  - Without userspace and consistency models, kGraft and kpatch are fairly similar
  - And that's what'll go upstream - a trivial implementation without a consistency model whatsoever that both kGraft and kpatch can build on - as a joint effort of SUSE and Red Hat
- Selectable consistency models
  - Based on what kind of consistency a specific patch needs, it can ask the patching engine for more than 'no consistency'
- Joint consistency model
  - May be possible based on Masami's work

SUSE

Packaging

- kGraft patch is packaged as a KMP RPM
  - with a single module (kgraft-patcher.ko) inside
  - hard dependency on kernel module version
  - updating the kGraft RPM patches the kernel immediately
  - kGraft thus works with any patch management mechanism
    - rpm, zypper, SUSE Manager, ...
  - and is trivial to deploy

SUSE

Demo

Live Demo

Where Do We Start

- kGraft enabled Linux kernel
  - Jiri's GIT repo
  - Object re-direction
  - Udo's GITHub repo for kGraft-tools
- Patching uptime_proc_show()

SUSE

Live Demo

Do Something :-)

Summary

Summary

What To Take Away Today

- Online Linux kernel patching is back!
  – Opensource, GPL
  – Ftrace and kernel modules
  – Different implementations
- Open-heart surgery
- Safety first
  – Safety = Security
  – Safety = Consistency
- Still quite a way to go

SUSE

Summary

What Is Coming

- Merge with upstream
- Alignment/collaboration with similar projects
- Enterprise readiness

SUSE

SUSE®
We adapt. You succeed.

Unpublished Work of SUSE LLC. All Rights Reserved.
This work is an unpublished work and contains confidential, proprietary and trade secret information of SUSE LLC. Access to this work is restricted to SUSE employees who have a need to know to perform tasks within the scope of their assignments. No part of this work may be practiced, performed, copied, distributed, revised, modified, translated, abridged, condensed, expanded, collected, or adapted without the prior written consent of SUSE.
Any use or exploitation of this work without authorization could subject the perpetrator to criminal and civil liability.

General Disclaimer
This document is not to be construed as a promise by any participating company to develop, deliver, or market a product. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. SUSE makes no representations or warranties with respect to the contents of this document, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. The development, release, and timing of features or functionality described for SUSE products remains at the sole discretion of SUSE. Further, SUSE reserves the right to revise this document and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes. All SUSE marks referenced in this presentation are trademarks or registered trademarks of Novell, Inc. in the United States and other countries. All third-party trademarks are the property of their respective owners.