



# Paradigm shift

---

Live patching as a core security concern



# Agenda

---



- What is live patching?
- What problems does it solve?
- What can be live patched?
- How are live patches created and how do they differ from regular patches?
- Integrating a live patching solution into an automated workflow

# Live Patching - How it started

---



- In 2006, Jeff Arnold, a student at MIT, started working on what would eventually become ksplice
- A way to apply changes, specifically security patches, to running Linux kernels *without interrupting processes or users*
- The motivation was a hacked system while a patch existed but couldn't be applied in a timely fashion



# Live Patching - 16 years gone by, the same problem exists

- Deploying security patches on time is still heavily constrained by the availability of maintenance windows
- For 56% of organizations, patches for Critical or High-Priority vulnerabilities take, on average, 5 to 6 weeks to be deployed\*
- That is a very large risk window for your organization and a very large window of opportunity for malicious actors

- “State of Enterprise Linux Security Management” - A study conducted by the Ponemon Institute, targeting IT leaders across different industries. Can be download here: <https://tuxcare.com/introducing-the-state-of-enterprise-linux-security-report/>



# The underlying issue

---



- The way patch operations are done has not changed, even when the technology available has
- The rate at which new vulnerabilities appear has been steadily growing year-on-year (5% per year for Kernel vulnerabilities alone)
- System downtime is one of the big friction points between IT teams and other business units

# Context



- According to Cloudflare\*, Log4j was being actively exploited **before** public disclosure
- Patching **30 days** after disclosure means that your systems are vulnerable for 30 days in addition to whatever happened before disclosure
- Hackers don't need your help
- Log4j first exploit attempts after public disclosure – **9 minutes**

• <https://blog.cloudflare.com/exploitation-of-cve-2021-44228-before-public-disclosure-and-evolution-of-waf-evasion-patterns/>



# Compliance rules are changing

- 30 days patch window is a **"best practice"** today
  - Shrunk from 6 months, to 3 months, to 1 month within a decade
- Ransomware attacks are causing people to start questioning if that window is too long
- Patching within 30 days will meet compliance... but not security needs





# Live patching to the rescue

- Live patching is a better way to deploy patches
- This was recognized by different teams when kSplice was acquired and became focused solely on Oracle's own distribution
- Three new projects, kPatch, kGraft and KernelCare, were started as alternatives





# The patches

---

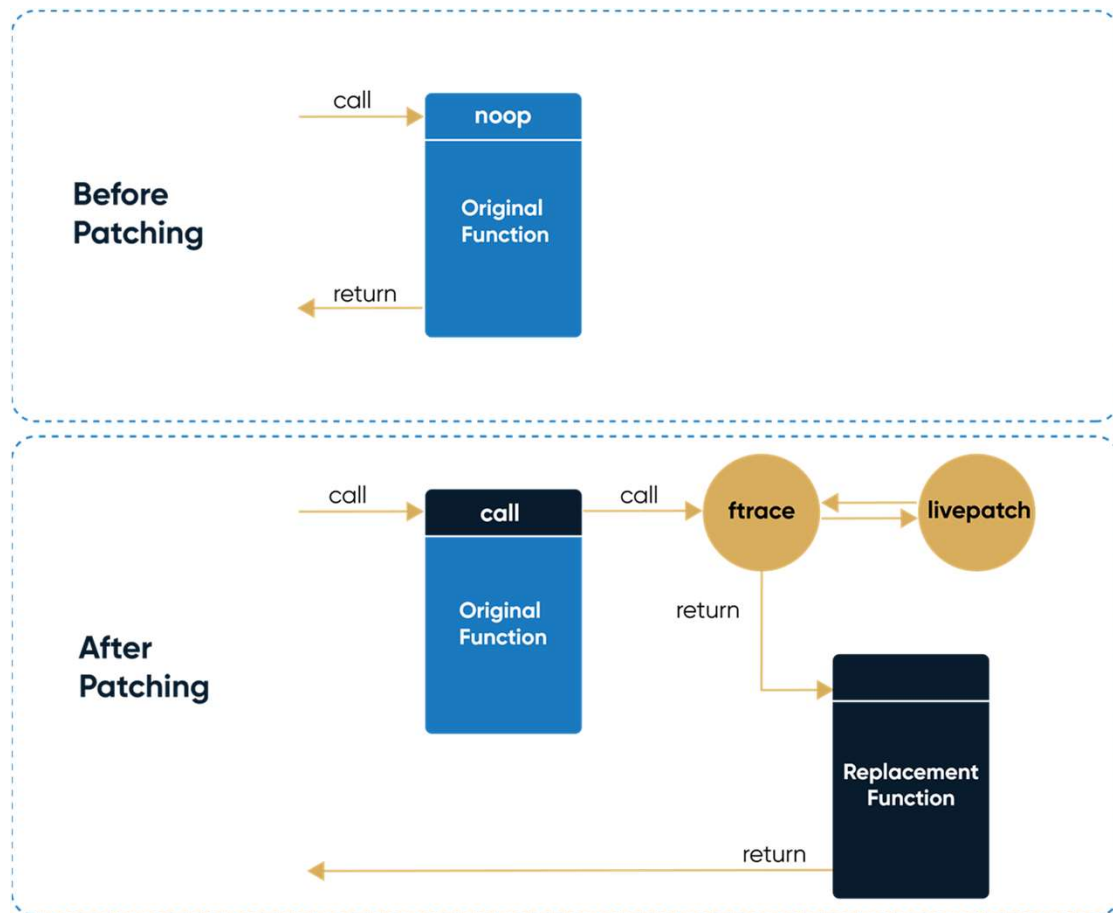


- Replacing running code is not trivial, but simple function code changes are easier than bigger changes touching multiple functions
- Most vulnerabilities are fixed with one-liners: bounds-check, off-by-one, input validation
- As long as function signatures or data structures don't change, live patching is easier

# The different Kernel facilities for Live Patching



- There are usually multiple ways to do something in the Linux Kernel
- Live patching follows this trend:
  - Livepatch - patch creation/loading
  - Ftrace - tracing/debugging tool repurposed to aid live patching by adding custom code at specific instructions
  - Kprobes - debugging tool repurposed to aid live patching with breakpoints in specific functions
  - eBPF - A mechanism to attach logic at certain hook points in the kernel



# Ftrace



- Tracing mechanism originally built to help developers writing code for the Kernel
- With it, you can add custom code handlers that run whenever a specific instruction is reached
- It can be used to redirect execution away from the “bad” function to the “corrected” version



# Kprobes



- Generic way of adding breakpoints at any instruction
- By setting the breakpoint at strategic locations before applying a patch, you can check if the function you want to change is in use (for example, by checking the stack)
- It's desirable that the function being patched is not in use. Some tools will wait for the right opportunity to apply a live patch

For more information: <https://www.kernel.org/doc/html/latest/livepatch/livepatch.html>



# How they all work together (or not)

---

- The most common way to use them is adding kprobe's breakpoints inside of ftrace handler code at the start of the functions you want to change, then using livepatch to load the fixed code into the kernel memory space
- Some of their functionality conflicts with each other (some ftrace instructions can be overwritten by some livepatch calls, for example)
- Kprobes and ftrace are repurposed kernel facilities, not designed with live patching in mind
- Third-party live patching solutions will often have their own custom ways for achieving live patching

# Livepatch



- Eventually, the kPatch and kGraft project's common features were merged into “livepatch” and included in the Linux Kernel
- Livepatch contains basic functionality for live patch creation and loading
- Some third-party tools for live patching use livepatch functionality, while others have opted for their own implementation to achieve the same goal

For more information: <https://www.kernel.org/doc/html/latest/livepatch/livepatch.html>



# Consistency model

---

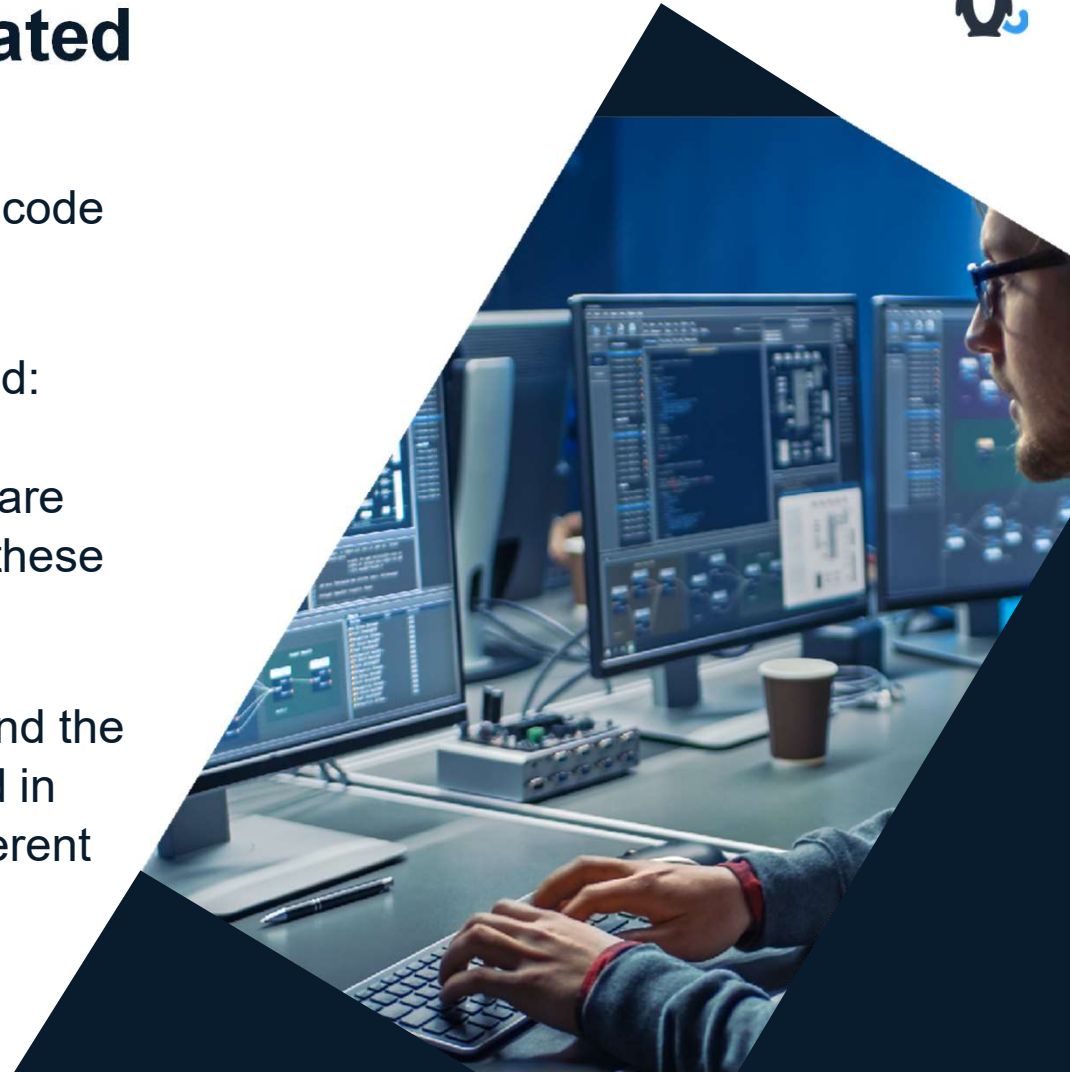
- The code execution should be in a “safe” situation before applying a patch
- You don’t want to replace a function when it’s in use, or active locks are originating from it, or memory assignments are not yet released
- Live patching tools will have mechanisms that “wait” for this safe situation to appear, and may even refuse to apply a patch if they can’t find one
- Maintaining consistency is one of the trickier parts of the process

For more information: <https://www.kernel.org/doc/html/latest/livepatch/livepatch.html#consistency-model>



# Patches - How they are created

- Like “regular” patches, it all starts with the code that fixes the problem
- Live patching specific issues are addressed: function signature changes, data structure changes, non maskable interrupts. These are some of the new concerns when creating these patches
- A binary “diff” between the running code and the new code has to be created and packaged in such a way that it can be deployed on different systems





# Build environments

---



- Tiny differences in build environments can create big changes in binary outputs
- This means that binary “diffs” will be unusable
- Compiler, linker, and binutils changes like different versions or flags will all likely invalidate patch creation
- Every different kernel version or distribution supported by a live patching tool requires a different build environment

# Testing



- Live patches, like traditional patches, need to be tested
- Not just for actually fixing what they are supposed to fix, but also for unexpected side-effects of being deployed live
- It's desirable to use extensive test automation when creating live patches and to continually expand the tests with validation for every new fix introduced
- Traditional Kernel tests\* often not suitable for live patching integration

\*For more information: <https://www.kernel.org/doc/html/latest/dev-tools/testing-overview.html>

# Lifetime of a life patch



Patch creation



Loading into kernel memory space with a live patching provider-specific tool



Enabling/activating the patch

**Maintaining the consistency of the kernel, the live patching tool will probably:**



Pause the process briefly



Check if the function is in use or not



If it's safe, apply the code change to redirect the function to the new code. If not, retry later.



Unpause the process



Disabling/removing the patch



## Other ways to 'update' kernel without reboot

- Kexec lets you replace the running kernel with another
  - Requires a whole new kernel to be loaded and restarts the existing user space processes





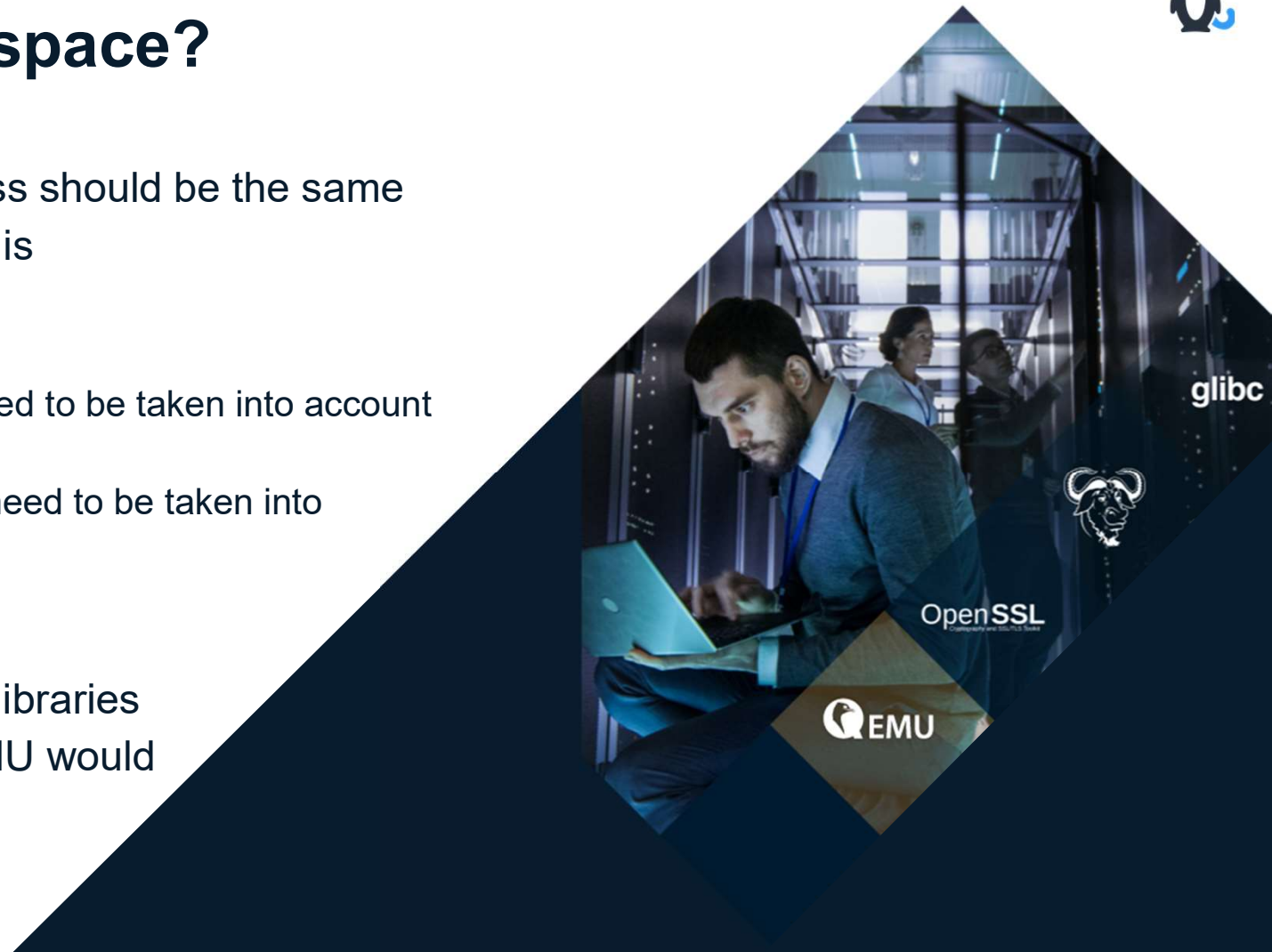
# What about userspace?

Patching one running process should be the same no matter what that process is

## Except:

- coroutines / schedulers need to be taken into account
- stack usage (exceptions) need to be taken into account

Services like databases or libraries like OpenSSL, glibc & QEMU would benefit from live patching.



# Creating a patch - Example (1/5)



## Original code

```
// foo.c
#include <stdio.h>
#include <time.h>

void i_m_being_patched(void)
{
    printf("i'm unpatched!\n");
}

int main(void)
{
    while (1) {
        i_m_being_patched();
        sleep(1);
    }
}
```

## Patched code

```
// bar.c
#include <stdio.h>
#include <time.h>

void i_m_being_patched(void)
{
    printf("you patched my %s\n", "tralala");
}

int main(void)
{
    while (1) {
        i_m_being_patched();
        sleep(1);
    }
}
```

For more information and the complete test example: <https://github.com/cloudlinux/libcare/blob/master/docs/internals.rst> (under "Manual patch Creation")



## Creating a patch - Example (2/5)

We want to compare the assembly code obtained through:

```
$ gcc -S foo.c
$ gcc -S bar.c
```

Looking at the changes with diff, we can see the added text and the code changes.



```
$ diff -u foo.s bar.s
--- foo.s      2016-07-16 16:09:16.635239145 +0300
+++ bar.s      2016-07-16 16:10:43.035575542 +0300
@@ -1,7 +1,9 @@
- .file "foo.c"
+ .file "bar.c"
+ .section .rodata
+ .LC0:
- .string "i'm unpatched!"
+ .string "tralala"
+ .LC1:
+ .string "you patched my %s\n"
+ .text
+ .globl i_m_being_patched
+ .type i_m_being_patched, @function
@@ -13,8 +15,10 @@
+ .cfi_offset 6, -16
+ movq %rsp, %rbp
+ .cfi_def_cfa_register 6
- movl $.LC0, %edi
- call puts
+ movl $.LC0, %esi
+ movl $.LC1, %edi
+ movl $0, %eax
+ call printf
+ popq %rbp
+ .cfi_def_cfa 7, 8
+ ret
```

For more information and the complete test example: <https://github.com/cloudlinux/libcare/blob/master/docs/internals.rst> (under "Manual patch Creation")



## Creating a patch - Example (3/5)

Every live patching solution has its own scripts to aid in patch creation, and so does TuxCare's. We use "kpatch\_gensrc" to create a patch file from the binary diff:

```
kpatch_gensrc --os=rhel6 -i foo.s -i bar.s -o foobar.s
```

This will create a very long file with assembly code and special sections for the patch deployment tool. You can find the complete output at the github repository linked below.

```
.file "foo.c"
#----- var -----
.section .rodata
.LC0:
.string "i'm unpatched!"
#----- func -----
.text
.globl i_m_being_patched
.type i_m_being_patched, @function
i_m_being_patched:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl $.LC0, %edi
call puts
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size i_m_being_patched, .-i_m_being_patched
```

(...)

For more information and the complete test example: <https://github.com/cloudlinux/libcare/blob/master/docs/internals.rst> (under "Manual patch Creation")



## Creating a patch - Example (4/5)



Now moving from assembly to the compiled code, using the special linker flag “-q” to store relocation information:

```
$ gcc -o foo foo.s
$ gcc -o foobar foobar.s -Wl,-q
```

Some clean up is still needed:

```
$ kpatch_strip --strip foobar foobar.stripped
$ stat -c '%n: %s' foobar foobar.stripped
foobar: 10900
foobar.stripped: 6584
```

There is still a process for fixing relocations that is detailed in the github repository and not shown here. After that is done, the binary patch is finally obtained with:

```
$ readelf -n foo | grep 'Build ID'
Build ID: 9e898b990912e176275b1da24c30803288095cd1
```

```
$ kpatch_make -b "9e898b990912e176275b1da24c30803288095cd1" \
  foobar.stripped -o foo.kpatch
```

For more information and the complete test example: <https://github.com/cloudlinux/libcare/blob/master/docs/internals.rst> (under “Manual patch Creation”)

## Creating a patch - Example (5/5)



Applying the patch:

```
(terminal1) $ ./foo
i'm unpatched!
i'm unpatched!
...
(terminal2) $ kpatch_ctl -v patch -p $(pidof foo) ./foo.kpatch
...
(terminal1)
you patched my tralala
you patched my tralala
```



*Note that the actual file on disk for the original process has not been touched, so that a restart would bring back the original behavior.*

For more information and the complete test example: <https://github.com/cloudlinux/libcare/blob/master/docs/internals.rst> (under “Manual patch Creation”)

# Automating TuxCare's live patching



Add the KernelCare installation one-liner to your Linux system deployment scripts: `curl -s -L https://kernelcare.com/installer | bash`



Centrally manage the patch deployment process through ePortal



Patch deployment:

**Either fully automatic or manual deployment:**



If fully automatic, systems will check for new patches every 4 hours (configurable)



If manual, patches can be centrally approved for all systems or for groups of systems



All deployed patches can be reverted if so desired (a performance regression, for example)




No files on-disk are touched



Never schedule another maintenance window just for patching

# Integrations



- As live patching only changes code in-memory, vulnerability and other management tools may incorrectly flag a system as vulnerable even after it is patched
- KernelCare integrates out-of-the-box with the most common tools and management solutions to provide accurate reporting and auditing information (Nessus, Qualys, Rapid7, Puppet, Ansible, Chef, Datadog, Crowdstrike)
-  orcharhino



# Paradigm shift

- IT is usually very fast at adopting new technologies but rather slow at changing processes
- Live patching is **reliable** and **proven** — and a better way to deploy patches
- It doesn't just shorten maintenance windows — it completely eliminates their use for patching
- You no longer need to choose which CVEs to patch – simply **patch all** of them – there is no downside
- In addition to being secure faster, it gives you back control over your operations – you're no longer simply **reacting** to new threats, you're actually **planning** your response





# Thank you!

Get in touch:

 [jcorreia@tuxcare.com](mailto:jcorreia@tuxcare.com)

 [@jcorreiacl](https://twitter.com/jcorreiacl)

