Patching with Xen LivePatch

Non disruptive patching of hypervisor

Konrad Rzeszutek Wilk
Oracle
Software Development Director

Ross Lagerwall
Citrix
Software Engineer

Agenda:

- Non disruptive patching.
- Why would you want this?
- Other known patching techniques.
- Patching!
- Tiny details.
- Roadmap.

What is this?

- Replacing compiled functions with new code.

```
const char *xen_extra_version(void)
{
      return XEN_EXTRAVERSION;
}
```

=>

```
const char *xen_extra_version(void)
{
      return "Hello World";
}
```

```
push   %rbp
mov    %rsp,%rbp
lea    0x16698b(%rip),%rax
leaveq
retq
```

=>

```
push   %rbp
mov    %rsp,%rbp
lea    0x29333b(%rip),%rax
leaveq
retq
```

- While hypervisor is running with guests.

ORACLE

Why binary patching? Why not migrate to another host?

- Local storage (SATA?),
- PCI pass-through (SR-IOV),
- NUMA locality,
- Giant guests (memory or CPU) and cannot fit on other hosts,
- Or system administrator simply does not want to reboot host:
  - Can or want to **only** during scheduled maintaince windows.
- Patching is almost instantenous

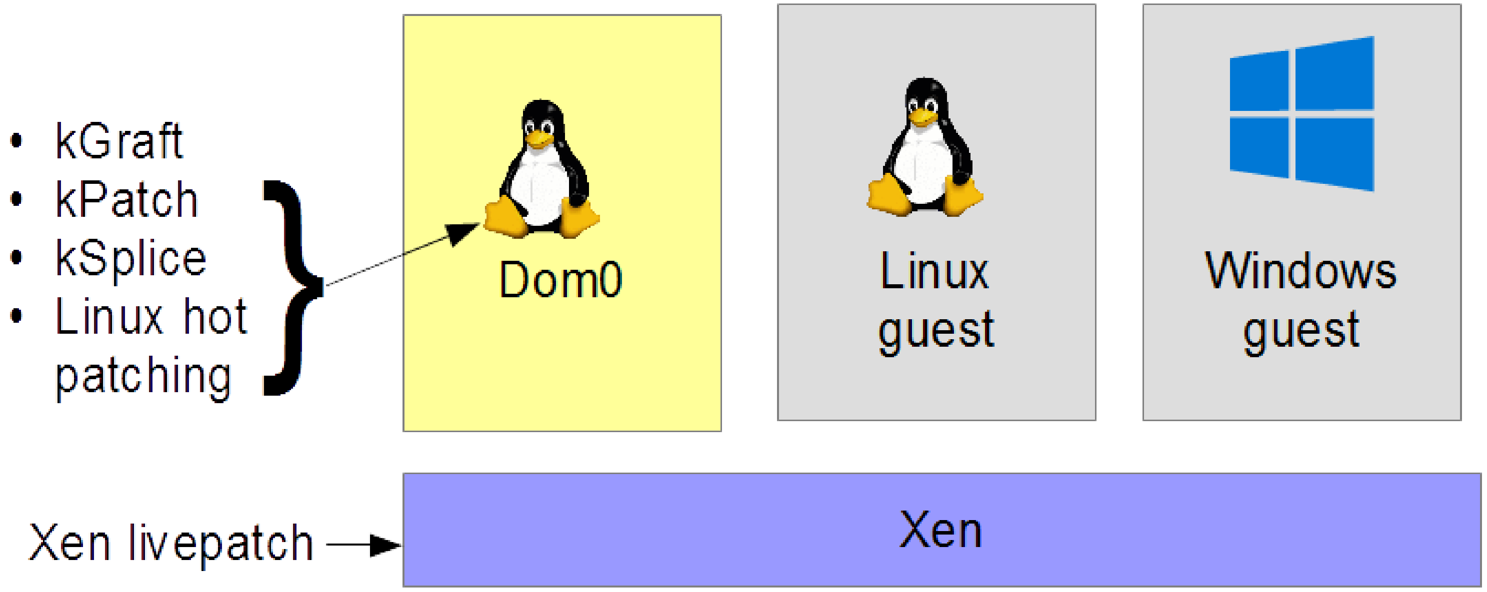
ORACLE

Known patching techniques.

- On Linux:
  - kGraft (SuSE).
  - kPatch (Red Hat).
  - kSplice (Oracle).
  - Linux live-patching (upstream) – common paths of kGraft + kPatch.
- On Xen:
  - Xen Livepatch (Oracle, Citrix), with Amazon contributing to design.
    - http://wiki.xenproject.org/wiki/LivePatch
    - http://xenbits.xen.org/docs/unstable/misc/livepatch.html
  - Amazon's internal hotpatching design:
    - http://www.linuxplumbersconf.net/2014/ocw//system/presentations/2421/original/xen_hotpatching-2014-10-16.pdf

ORACLE

Non disruptive patching options.



And their functionality:

Level	Name	Function + Data	Patching of data structures	Inline patching
Userspace	kSplice userpace (glibc,openssl)	✓	✓	✓
Kernel	Linux hot patching	✓		
	kGraft (SuSE)	✓		
	kPatch (Red Hat)	✓	✓[via hooks]	
	kSplice	✓	✓	✓
Hypervisor	Xen livepatch	✓	✓[via hooks, hopefully in Xen 4.8]	

ORACLE

Patching!

- At first blush this sounds like binary translation – we convert old code to new code:

```
mov %rsp,%rax
and $0xffffffffffff8000,%rax
```

```
movq $0x0,-0x48(%rbp)
movq $0x0,-0x40(%rbp)
movq $0x0,-0x38(%rbp)
mov %rsp,%rax
and $0xffffffffffff8000,%rax
```

Extra 18 bytes of code.

- XSA-132 “domctl/sysctl: don’t leak hypervisor stack to toolstack” – change inside arch_do_domctl.
- But nobody can translate the code for us. We **NEED** to change the code in memory while the hypervisor is executing.

Patching: inserting new code.

- But adding in code means moving other code as well:

```
arch_do_domctl:
        55 48 89 E5 48 89 FB 90
        89 05 A4 9C 1E 00 8B 13
        48 8D 05 83 71 12 00 8B
        14 90 48 B8 00 00 00 80
        D0 82 FF FF 48 8D 04 02
        49 89 06 8B 03 83 C0 01
        89 03 89 C0 48 89 05 7F
        9C 1E 00 48 8D 3D D0 12
        17 00 E8 E3 EC FF FF B8
        48 89 E0 48 25 00 80 FF
        FF 00 00 00 48 8B 1C 24
        4C 8B 64 24 08 4C 8B 6C
        24 10 4C 8B 74 24 18 C9
do_domctl:
        55 48 89 E5 48 81 EC 70
        01 00 00 48 89 5D D8 4C
                  …
```

```
55 48 89 E5 48 89 FB 90
89 05 A4 9C 1E 00 8B 13
48 8D 05 83 71 12 00 8B
14 90 48 B8 00 00 00 80
D0 82 FF FF 48 8D 04 02
49 89 06 8B 03 83 C0 01
89 03 89 C0 48 89 05 7F
9C 1E 00 48 8D 3D D0 12
17 00 E8 E3 EC FF FF B8
48 C7 45 B8 00 00 00 00
48 C7 45 C0 00 00 00 00
48 C7 45 C8 00 00 00 00
48 89 E0 48 25 00 80 FF
FF 00 00 00 48 8B 1C 24
4C 8B 64 24 08 4C 8B 6C
24 10 4C 8B 74 24 18 C9
C3 90 90 90 90 90 90 90
90 90 55 48 89 E5 48 81
```

- Otherwise we end up executing nonsense code at old location!

Patching: Jumping

- We could add padding in all the functions to deal with this. But what if the amount of changes is **greater** than the padding?
- Jump!
  - Allocate new memory.
  - Copy new code in memory.
  - Check that nobody is running old code.
  - Compute offset from old code to new code.
  - Add trampoline jump to new code.

ORACLE

Patching: 1) Allocate + copy new code in

- New **arch_do_domctl** code at newly allocated memory space:

```
<arch_do_domctl>:

    55                          push   %rbp
    48 89 e5                    mov    %rsp,%rbp
    41 57                       push   %r15
…
48 c7 45 b8 00 00 00 00 movq $0x0,-0x48(%rbp)
48 c7 45 c0 00 00 00 00 movq $0x0,-0x40(%rbp)
48 c7 45 c8 00 00 00 00 movq $0x0,-0x38(%rbp)
48 89 e0                mov %rsp,%rax
48 25 00 80 ff ff       and $0xffffffffffff8000,%rax
```

ORACLE

Patching: 2) Check code 3) Compute offset

- Check that arch_do_domctl is not being executed.
- Figure out offset from new to old code.

```
<arch_do_domctl>:

    55                      push   %rbp
    48 89 e5                mov    %rsp,%rbp
    41 57                   push   %r15
…
48 89 e0                    mov %rsp,%rax
48 25 00 80 ff ff           and $0xffffffffffff8000,%rax
```

```
<arch_do_domctl>:

    55                      push   %rbp
    48 89 e5                mov    %rsp,%rbp
    41 57                   push   %r15
…
48 c7 45 b8 00 00 00 00 movq $0x0,-0x48(%rbp)
48 c7 45 c0 00 00 00 00 movq $0x0,-0x40(%rbp)
48 c7 45 c8 00 00 00 00 movq $0x0,-0x38(%rbp)
48 89 e0                mov %rsp,%rax
48 25 00 80 ff ff       and $0xffffffffffff8000,%rax
```

ORACLE

Patching: 4) Add trampoline

- Add trampoline:

```
<arch_do_domctl>:

    E9 1A 97 EA FF       jmpq    <arch_do_domctl>[NEW]
…
48 89 e0                 mov %rsp,%rax
48 25 00 80 ff ff        and $0xffffffffffff8000,%rax
```

```
<arch_do_domctl>:

    55                          push   %rbp
    48 89 e5                    mov    %rsp,%rbp
    41 57                       push   %r15
…
48 c7 45 b8 00 00 00 00 movq $0x0,-0x48(%rbp)
48 c7 45 c0 00 00 00 00 movq $0x0,-0x40(%rbp)
48 c7 45 c8 00 00 00 00 movq $0x0,-0x38(%rbp)
48 89 e0                mov %rsp,%rax
48 25 00 80 ff ff       and $0xffffffffffff8000,%rax
```

ORACLE

Patching: Conclusion

- For code just need to over-write start of function with:

```
…
E9 1A 97 EA FF      jmpq   <arch_do_domctl>[NEW] …
```

- For data it can be inline replacement (changing in .data values):

```
<opt_noreboot>:
   00 00
    ...
```

→

```
<opt_noreboot>:
   00 01
    ...
```

ORACLE

That was easy, what is the fuss about?

- Relocation of symbols – data or functions:

```
…
8b 0d 53 80 fb ff       mov    -0x47fad(%rip),%ecx        # ffff82d0802848c0 <pfn_pdx_hole_shift>
…
```

  Need to compute new code/data the offsets to other functions, data structures, etc.

  – Xen hyprvisor now has an ELF final dynamic linker to resolve this.

- Correctness: Is the old code the same as what the hot-patch had been based on? Using an **build-id** (unique value) generated by compiler.
  – The tools to generate payloads need to embed the correct **build-id**
  – Allows also to stack payloads on top of each other (with each having an unique **build-id** and depending on prior payload's **build-id**):

ORACLE

Payloads dependencies – and how **build-id** are used for that.

- Hypervisor build-id (0x17ac1..)
  - Payload test1 (build-id: 0x8ef93.., depends on 0x17ac1..)
    - Payload test2 (build-id: b409fb.., depends on 0x8ef93..)
      - And so on.
  - Can apply payloads on top of each other.
  - Can also replace the chain of them with a new one:
  - Hypervisor build-id (0x17ac1..)
    - Payload test1 (build-id: 0x8ef93.., depends on 0x17ac1..)
      - …
    - Payload replace (build-id: 0x99432.., depends on 0x17ac1..)

ORACLE

How do guarantee we don’t patch code which may be in this (or another) CPU cache/stack?

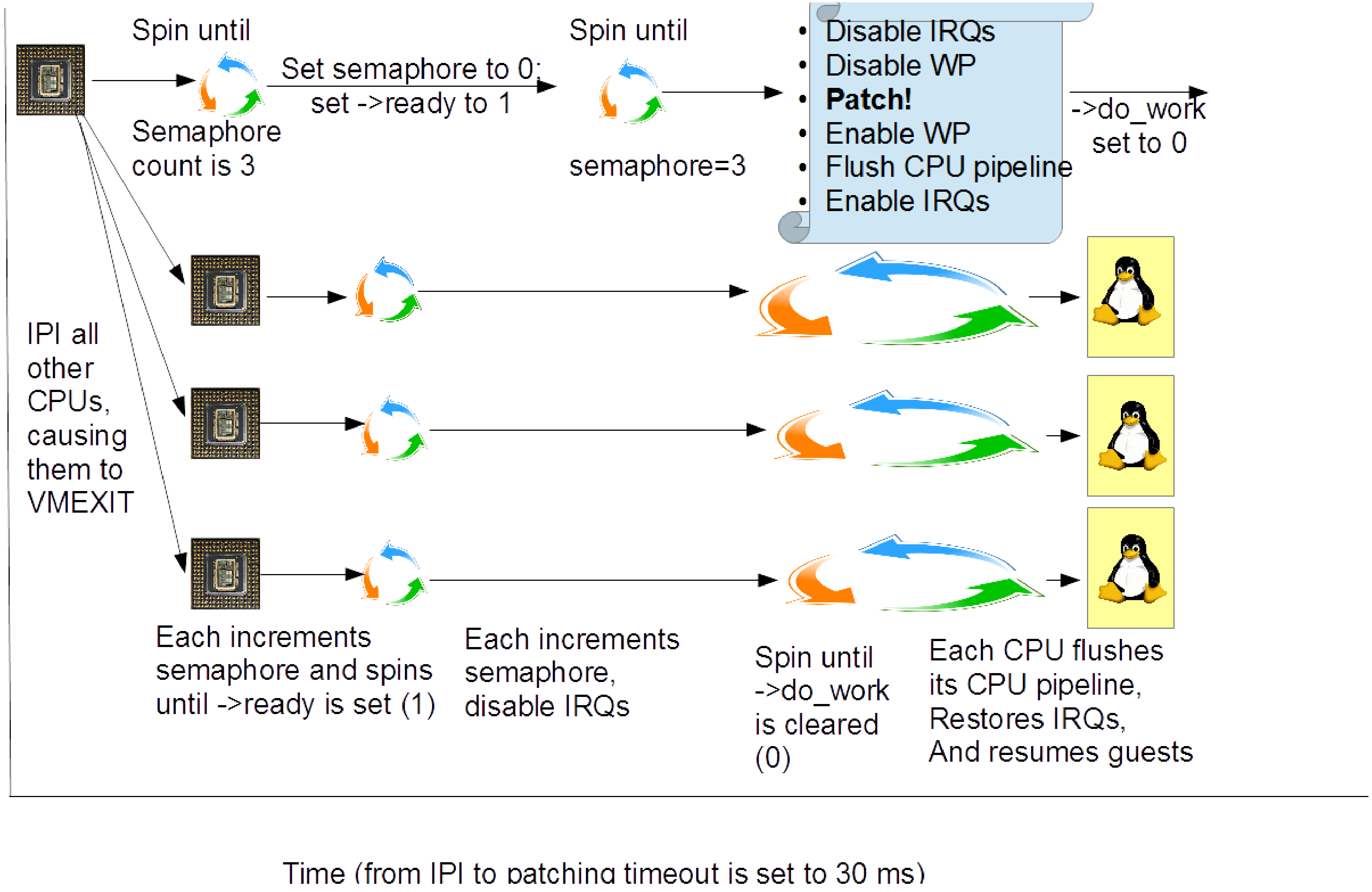
- Stack checking: Cannot patch the function which is in use by another CPU!
  - We patch when the hypervisor has no stack – at deterministic point.
- A two stage rendezvous mechanism:
  - Schedule_work sets per_cpu(work_to_do) and global do_work.
  - Whoever gets first to **check_for_livepatach_work** is master, all others are sub-ordinates. **check_for_livepatch_work** called in VMEXIT handlers and idle_loop loop.
    - Master IPIs all other CPUs to call function which sets per_cpu(work_to_do)
    - Slave CPUs IPI handler is called. It sets per_cpu(work_to_do), and right before entering to the guest calls **check_for_livepatch_work**. Spins waiting until ->ready is set.
    - Master spins until all CPUs have incremented a atomic counter (aka – all subordinates are waiting on ->ready). Sets ->ready=1.

ORACLE

Hypervisor patching code

- Master signals to sub-ordinates to disable IRQs (we don’t want IRQ handlers to run as we may be patching them).
  - Sub-ordinates disable IRQs, and spin waiting on patching (->do_work) to be complete.
- Master disables IRQs, disables Write Protection on read-only memory and patches code, re-enables Write Protection.
- Master enables IRQs, clears ->do_work.
- Sub-ordinates stop spinning, flush their pipeline, and restore IRQs.
- Master prints that it has finished patching.
- Same mechanism for revert and replace - only what’s written into the trampoline differs.

ORACLE



Spin until
Semaphore count is 3
Set semaphore to 0; set ->ready to 1
Spin until
semaphore=3
• Disable IRQs
• Disable WP
• **Patch!**
• Enable WP
• Flush CPU pipeline
• Enable IRQs
->do_work set to 0
IPI all other CPUs, causing them to VMEXIT
Each increments semaphore and spins until ->ready is set (1)
Each increments semaphore, disable IRQs
Spin until ->do_work is cleared (0)
Each CPU flushes its CPU pipeline, Restores IRQs, And resumes guests
Time (from IPI to patching timeout is set to 30 ms)

ORACLE

Tool side functionality:

- Query what payloads have been loaded and their status (checked, applied).
- Upload new payloads.
- Apply, revert or replace payloads.

Roadmap – Further work in hypervisor:

- /proc/xen/xensyms needs symbols introduced by payloads
- Signature verification code.
- NMI and MCE handling when patching
- OSSTest
- ARM64 support

ORACLE

Roadmap – Further work in tools:

- Sensibly patching assembly code (probably requires HV changes too)
- Ensure that .config is unchanged between the original build and the patched build
- General livepatch-build improvements to increase the success rate to patch anything close to 100%.
- Merge xen-livepatch tool into xl.

ORACLE

Questions and Answer

Backup slides

Signature verification:

- The signature is to be appended at the end of the ELF payload prefixed with the string: ~Module signature appended~\n
- Signature header afterwards matches Linux’s one.

ORACLE

Screenshot of xen-livepatch

```
-bash-4.1# xl info | grep xen_version
xen_version            : 4.8-unstable
-bash-4.1# xen-livepatch load /usr/lib/debug/xen_hello_world.livepatch
Uploading /usr/lib/debug/xen_hello_world.livepatch (16897 bytes)
Performing apply:. completed
-bash-4.1# xen-livepatch list
 ID                                     | status
----------------------------------------+------------
xen_hello_world                         | APPLIED
-bash-4.1# xl info | grep xen_version
xen_version            : 4.8Hello World
-bash-4.1#
```

Building live patches

Live patches are binary files containing code to be loaded by the hypervisor — like kernel modules.

How are these created?

Enter livepatch-build-tools!

http://xenbits.xen.org/gitweb/?p=livepatch-build-tools.git

livepatch-build-tools is based on kpatch-build

Building live patches: Inputs

```
$ livepatch-build -s xen -c orig.config \
    --depends 55776af8c7377e7191d733797543b87a59631c50 \
    -p xsa182.patch -o outdir
```

Takes as input:

- The exact source tree from the running Xen.
- The .config from the original build of Xen.
- A build-id onto which the livepatch will be applied.
- A source patch.

Building live patches: Process

livepatch-build does:

1. Build Xen
2. Apply Patch
3. Build Xen with "-ffunction-sections -fdata-sections"
4. Unapply patch
5. Build Xen again with "-ffunction-sections -fdata-sections"
6. Create a livepatch from the changed object files.

Building live patches: Diff

For each pair of changed objects, ‘original’ and ‘patched’, run `create-diff-tool`:

- Load objects and check that the headers match.
- Adjust the ELFs to make them easier to process:
  - Replace section symbols with function/object symbols
  - Rename mangled symbols: `.isra.` `.part.` `.constprop.`
    - `map_domain_page.isra.9` ➞ `map_domain_page.isra.2`

Building live patches: Diff

- Correlate sections: for each section in ‘original’, find its twin in ‘patched’.
- Correlate symbols: for each symbol in ‘original’, find its twin in ‘patched’.
- Correlate static locals: match randomly named static local variables from ‘original’ to ‘patched’.
  - Static locals are correlated if they have the same base name and are referenced by a pair of correlated sections.
  - `avail_static.16247` → `avail_static.24561`

Building live patches: Diff

- Compare and mark as SAME, CHANGED or NEW.
- For each CHANGED function or NEW global, include it and its references recursively.
- Handle special sections (bug frames, altinstructions, exception tables).

Building live patches: Diff

- Rename local symbols to match the format used by Xen (filename#symbolname).
- For each CHANGED function, create an entry in a special livepatch section (`.livepatch.funcs`).
- Write out the new object file.

Building live patches: Link

Link all the diff object files into a single ELF file, adding:

- A dependency section containing the target build id,
- and a new build id for the object file.

This object file gets uploaded to the hypervisor.

Pitfalls when building live patches: Assembly

There are some XSAs which patch assembly, for example XSA-183. It is not currently possible to generate a livepatch using livepatch-build.

- Have less assembly (yay!).
- Rewrite assembly into self-contained functional units (aka assembler functions) with entries in the symbol table.
- Inline patching of assembly (when possible).

Pitfalls when building live patches: Data

- New data and read-only data is handled correctly.
- Changing initialized data or existing data structures is hard so such changes are prevented.
- Use hook functions to allow code to be executed at various stages during the patch apply (or revert) process.
  - Allows data to be transformed during patch apply, even if the data is dynamically allocated
  - Allows once-off initializations.
- Use shadow variables to attach new members to existing data structures.
- Hopefully in Xen 4.8.

Pitfalls when building live patches: Visibility

- Changing the type or visibility of a symbol is not allowed.
- Issue when building a patch for XSA-58.
- `put_old_guest_table` goes from local symbol to a global symbol.
- Rename the function (e.g. `lp_put_old_guest_table`) then replace all references to the old name with the new name.
- This isn't ideal because it means potentially many functions need to be changed unnecessarily, but it is the current solution.

Pitfalls when building live patches: __init

- Tool prevents changes to __init — doesn't make sense anyway.
- Use a hook function to make the equivalent change during patch load.
- Need to verify per-patch that it is actually safe since to do this → otherwise reboot!

Pitfalls when building live patches: __LINE__

- __LINE__ causes many functions to be CHANGED and included in the ouput.
- Not necessarily a problem since the size is small, but it is harder to analyze.
- dprintk uses __LINE__ — not in release build
- Patches coming to reduce uses of __LINE__ to zero for a release build.

Pitfalls when building live patches: leaks

- Even if the patch is trivial to build and apply, it is not necessarily correct — XSA-100
- Freed pages aren't scrubbed after live patch is applied.
  - Schedule an asynchronous scrub of the free heap
  - Scrub before handing pages to the guest.
- Do not blindly trust the tools with the output they generate.

Demo!