# CRIU: Reworking vDSO proxyfication, syscall restart

Dmitry Safonov, Andrey Vagin

Linux Plumbers

2019

# Take a timeout

- There is a bunch of syscalls that take a timeout. Mostly they block while waiting for some events: `poll()`, `select()` and `futex()` with one exception for `nanosleep()`.

# Take a timeout

- There is a bunch of syscalls that take a timeout. Mostly they block while waiting for some events: `poll()`, `select()` and `futex()` with one exception for `nanosleep()`.
- While blocking on some event, userspace expects that signal handlers are still invoked (IOW, interruptible sleep).
- It's also expected that SIGSTOP/SIGCONT and terminating signals could do their job.

# Restarting a syscall

The kernel tries to deal with timeout invisible to the userspace:

- When a task blocked with timeout receives a signal ( `signal_pending()` ) kernel fills the content of `task_struct::restart_block` . Basically, it saves the timeout that's left unwaited and a list of arguments necessary to restart the syscall (i.e.: list of fds for `poll()` ).

# Restarting a syscall

The kernel tries to deal with timeout invisible to the userspace:

- When a task blocked with timeout receives a signal (`signal_pending()`) kernel fills the content of `task_struct::restart_block`. Basically, it saves the timeout that's left unwaited and a list of arguments necessary to restart the syscall (i.e.: list of fds for `poll()`).
- Then the syscall returns `-ERESTART_RESTARTBLOCK`

# Restarting a syscall

The kernel tries to deal with timeout invisible to the userspace:

- When a task blocked with timeout receives a signal (`signal_pending()`) kernel fills the content of `task_struct::restart_block`. Basically, it saves the timeout that's left unwaited and a list of arguments necessary to restart the syscall (i.e.: list of fds for `poll()`).
- Then the syscall returns `-ERESTART_RESTARTBLOCK`
- Now before going back to userspace, the kernel checks pending signals and starts processing them. If the signal should be delivered, the original syscall returns `-EINTR`.

# Restarting a syscall

The kernel tries to deal with timeout invisible to the userspace:

- When a task blocked with timeout receives a signal ( `signal_pending()` ) kernel fills the content of `task_struct::restart_block` . Basically, it saves the timeout that's left unwaited and a list of arguments necessary to restart the syscall (i.e.: list of fds for `poll()` ).
- Then the syscall returns `-ERESTART_RESTARTBLOCK`
- Now before going back to userspace, the kernel checks pending signals and starts processing them. If the signal should be delivered, the original syscall returns `-EINTR` .
- If it shouldn't be delivered (i.e.: `SIG_IGN` , `SIGSTOP` , `SIGCONT` , etc), the kernel patches `ip` in the regset and sets `__NR_restart_syscall` into register with syscall number

# Restarting a syscall (cont.)

- Userspace calls the proper syscall instruction, but with patched by the kernel syscall number `__NR_restart_syscall`
- `restart_syscall()` checks `task_struct::restart_block` and calls back the function that was interrupted by the signal

# Now in the game comes CRIU

- Somebody noticed that if you keep migrating a container fast enough, timeout-based syscalls may never stop blocking...

# Now in the game comes CRIU

- Somebody noticed that if you keep migrating a container fast enough, timeout-based syscalls may never stop blocking...
- Currently, if a syscall was interrupted by a checkpoint, it's restarted with original regset (by patching `ip`).
- Which means that the syscall is always restarted with an initial timeout

# Now in the game comes CRIU

- Somebody noticed that if you keep migrating a container fast enough, timeout-based syscalls may never stop blocking...
- Currently, if a syscall was interrupted by a checkpoint, it's restarted with original regset (by patching `ip`).
- Which means that the syscall is always restarted with an initial timeout
- CRIU can't use `restart_syscall()` as on restore you have a brand new `task_struct` without any `restart_block` filled

# A new PTRACE_GET_RESTART_BLOCK request

- A patches set to refactor `restart_block` and unify *offset* for different syscalls

# A new PTRACE_GET_RESTART_BLOCK request

- A patches set to refactor `restart_block` and unify *offset* for different syscalls
- Introducing a new `ptrace` API to dump the unwaited offset time seemed easy

# A new PTRACE_GET_RESTART_BLOCK request

- A patches set to refactor `restart_block` and unify *offset* for different syscalls
- Introducing a new `ptrace` API to dump the unwaited offset time seemed easy
- But can syscall parameter *actually* be patched in regset on restore?
- Compiler expects the register being unchanged, so probably it can't be updated

# A new PTRACE_SET_RESTART_BLOCK request?

There is no *expected* way to set `restart_block` from userspace, and the argument in syscall shouldn't be changed. But there are standing questions how-to design this ptrace() API:

- How to get/set the callback function?

# A new PTRACE_SET_RESTART_BLOCK request?

There is no *expected* way to set `restart_block` from userspace, and the argument in syscall shouldn't be changed. But there are standing questions how-to design this ptrace() API:

- How to get/set the callback function?
- Verification and the format of `restart_block` members

# A different & radical approach

- Is it possible to remove `restart_block` and `__NR_restart_syscall`?
- Probably, instead of checking `signal_pending()`, is it possible to ignore non-deliverable signals and call `do_signal_stop()` in case of `SIGSTOP`?

# vDSO proxyfication - The problem

- On migration the vDSO (virtual Dynamic Shared Object) can be different between the nodes.
- It can be different even on the same machine if the kernel changes between Checkpoint and Restore.
- The only kernel guarantee is that old function symbols persist

# vDSO proxyfication - The problem

- On migration the vDSO (virtual Dynamic Shared Object) can be different between the nodes.
- It can be different even on the same machine if the kernel changes between Checkpoint and Restore.
- The only kernel guarantee is that old function symbols persist
- The old vDSO can't be used "as is" on Restore as VVAR variables layout can differ
- Loosing fast syscalls (especially timing) after migration to newer kernel isn't very exciting idea

# vDSO proxyfication - Detecting changes

- To detect changes in vDSO/VVAR pair, ELF symbols table is parsed to get entries to vDSO and their offsets
- The parsing done once a boot time and the layout of vDSO is stored in tmpfs
- Original vDSO VMA image is stored with CRIU images (files)

# vDSO proxyfication - Detecting changes

- To detect changes in vDSO/VVAR pair, ELF symbols table is parsed to get entries to vDSO and their offsets
- The parsing done once a boot time and the layout of vDSO is stored in tmpfs
- Original vDSO VMA image is stored with CRIU images (files)
- If the layout stays the same on Restore - nothing to do, just move vDSO where it was

# vDSO proxyfication - Jump trampolines

When the layout is different on Restore:

- Map the original vDSO code blob at the same place
- Move/map vDSO provided by kernel at free space
- Patch all entries on old vDSO with ”jump trampolines” - per-architecture (arm32/arm64/s390/x86) jumps like

  ```
  mov <addr> %eax ; jmp %eax
  ```

# vDSO proxyfication - Downsides

- There is a new mapping appearing after migration to newer kernel
- Not safe: if an application is Checkpointed on the bytes being patched - it likely will crash the program after Restore
- Not safe: if an application is Checkpointed on the code after jump blob - it may dereference VVAR which is not saved by CRIU :)

# vDSO proxyfication - Downsides

- There is a new mapping appearing after migration to newer kernel
- Not safe: if an application is Checkpointed on the bytes being patched - it likely will crash the program after Restore
- Not safe: if an application is Checkpointed on the code after jump blob - it may dereference VVAR which is not saved by CRIU :)
- Andrei also mentioned that it's not safe if the application is processing a signal, it may return from handler on the patched bytes. What's worse - it's not possible to tell for sure if the application is running signal handler.

# vDSO proxyfication - First hacky ways to solve those issues

- Put a breakpoint just after bytes to be patched and don't Checkpoint on that place
- Add an eBPF that saves the stack pointer on signal deliver and if `rt_sigreturn()` was about to return on the old vDSO, patch `%ip` to go into a new vDSO image.

# New idea: no vDSO proxyfication anymore

- Concentrating on the problem that's being solved:

# New idea: no vDSO proxyfication anymore

- Concentrating on the problem that's being solved:
- The original problem is that VVAR can't be used with the old vDSO
- That issue could be addressed by dynamically re-link new VVAR page with the old vDSO code

# New idea: no vDSO proxyfication anymore

- Concentrating on the problem that's being solved:
- The original problem is that VVAR can't be used with the old vDSO
- That issue could be addressed by dynamically re-link new VVAR page with the old vDSO code
- That means exporting VVAR symbol table and vDSO relocation table
- But the dynamic linker probably shouldn't interfere with vDSO/VVAR usually as it may link some variable to vDSO instead the one from VVAR