NAME

pkeys − overview of Memory Protection Keys

DESCRIPTION

Memory Protection Keys (pkeys) are an extension to existing page-based memory permissions. Normal page permissions using page tables require expensive system calls and TLB invalidations when changing permissions. Memory Protection Keys provide a mechanism for changing protections without requiring modification of the page tables on every permission change.

To use pkeys, software must first "tag" a page in the page tables with a pkey. After this tag is in place, an application only has to change the contents of a register in order to remove write access, or all access to a tagged page.

Protection keys work in conjunction with the existing **PROT_READ**, **PROT_WRITE**, and **PROT_EXEC** permissions passed to system calls such as **mprotect**(2) and **mmap**(2), but always act to further restrict these traditional permission mechanisms.

If a process performs an access that violates pkey restrictions, it receives a **SIGSEGV** signal. See **sigaction**(2) for details of the information available with that signal.

To use the pkeys feature, the processor must support it, and the kernel must contain support for the feature on a given processor. As of early 2016 only future Intel x86 processors are supported, and this hardware supports 16 protection keys in each process. However, pkey 0 is used as the default key, so a maximum of 15 are available for actual application use. The default key is assigned to any memory region for which a pkey has not been explicitly assigned via **pkey_mprotect**(2).

Protection keys have the potential to add a layer of security and reliability to applications. But they have not been primarily designed as a security feature. For instance, WRPKRU is a completely unprivileged instruction, so pkeys are useless in any case that an attacker controls the PKRU register or can execute arbitrary instructions.

Applications should be very careful to ensure that they do not "leak" protection keys. For instance, before calling **pkey_free**(2), the application should be sure that no memory has that pkey assigned. If the application left the freed pkey assigned, a future user of that pkey might inadvertently change the permissions of an unrelated data structure, which could impact security or stability. The kernel currently allows in-use pkeys to have **pkey_free**(2) called on them because it would have processor or memory performance implications to perform the additional checks needed to disallow it. Implementation of the necessary checks is left up to applications. Applications may implement these checks by searching the */proc/*pid*/smaps* file for memory regions with the pkey assigned. Further details can be found in **proc**(5).

Any application wanting to use protection keys needs to be able to function without them. They might be unavailable because the hardware that the application runs on does not support them, the kernel code does not contain support, the kernel support has been disabled, or because the keys have all been allocated, perhaps by a library the application is using. It is recommended that applications wanting to use protection keys should simply call **pkey_alloc**(2) and test whether the call succeeds, instead of attempting to detect support for the feature in any other way.

Although unnecessary, hardware support for protection keys may be enumerated with the *cpuid* instruction. Details of how to do this can be found in the Intel Software Developers Manual. The kernel performs this enumeration and exposes the information in */proc/cpuinfo* under the "flags" field. The string "pku" in this field indicates hardware support for protection keys and the string "ospke" indicates that the kernel contains and has enabled protection keys support.

Applications using threads and protection keys should be especially careful. Threads inherit the protection key rights of the parent at the time of the **clone**(2), system call. Applications should either ensure that their own permissions are appropriate for child threads at the time when **clone**(2) is called, or ensure that each child thread can perform its own initialization of protection key rights.

Signal Handler Behavior

Each time a signal handler is invoked (including nested signals), the thread is temporarily given a new, default set of protection key rights that override the rights from the interrupted context. This means that

applications must re-establish their desired protection key rights upon entering a signal handler if the desired rights differ from the defaults. The rights of any interrupted context are restored when the signal handler returns.

This signal behavior is unusual and is due to the fact that the x86 PKRU register (which stores protection key access rights) is managed with the same hardware mechanism (XSAVE) that manages floating-point registers. The signal behavior is the same as that of floating-point registers.

Protection Keys system calls

The Linux kernel implements the following pkey-related system calls: **pkey_mprotect**(2), **pkey_alloc**(2), and **pkey_free**(2).

The Linux pkey system calls are available only if the kernel was configured and built with the **CONFIG_X86_INTEL_MEMORY_PROTECTION_KEYS** option.

EXAMPLES

The program below allocates a page of memory with read and write permissions. It then writes some data to the memory and successfully reads it back. After that, it attempts to allocate a protection key and disallows access to the page by using the WRPKRU instruction. It then tries to access the page, which we now expect to cause a fatal signal to the application.

```
$ ./a.out
buffer contains: 73
about to read buffer again...
Segmentation fault (core dumped)
```

Program source

```
#define _GNU_SOURCE
#include <err.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

int
main(void)
{
    int status;
    int pkey;
    int *buffer;

    /*
     * Allocate one page of memory.
     */
    buffer = mmap(NULL, getpagesize(), PROT_READ | PROT_WRITE,
                  MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
    if (buffer == MAP_FAILED)
        err(EXIT_FAILURE, "mmap");

    /*
     * Put some random data into the page (still OK to touch).
     */
    *buffer = __LINE__;
    printf("buffer contains: %d\n", *buffer);

    /*
     * Allocate a protection key:
```

```
         */
        pkey = pkey_alloc(0, 0);
        if (pkey == -1)
            err(EXIT_FAILURE, "pkey_alloc");

        /*
         * Disable access to any memory with "pkey" set,
         * even though there is none right now.
         */
        status = pkey_set(pkey, PKEY_DISABLE_ACCESS);
        if (status)
            err(EXIT_FAILURE, "pkey_set");

        /*
         * Set the protection key on "buffer".
         * Note that it is still read/write as far as mprotect() is
         * concerned and the previous pkey_set() overrides it.
         */
        status = pkey_mprotect(buffer, getpagesize(),
                               PROT_READ | PROT_WRITE, pkey);
        if (status == -1)
            err(EXIT_FAILURE, "pkey_mprotect");

        printf("about to read buffer again...\n");

        /*
         * This will crash, because we have disallowed access.
         */
        printf("buffer contains: %d\n", *buffer);

        status = pkey_free(pkey);
        if (status == -1)
            err(EXIT_FAILURE, "pkey_free");

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

pkey_alloc(2), **pkey_free**(2), **pkey_mprotect**(2), **sigaction**(2)