Introduction

If you reached this page because of a message like this printed by a Go program:

```
runtime: note: your Linux kernel may be buggy
runtime: note: see https://go.dev/wiki/LinuxKernelSignalVectorBug
runtime: note: mlock workaround for kernel bug failed with errno <number>
```

then you are using a Linux kernel that may have a bug. This kernel bug may have caused memory corruption in your Go program, and may have caused your Go program to crash.

If you understand why your program crashed, then you can ignore this page.

Otherwise, this page will explain what the kernel bug is, and includes a C program that you can use to check whether your kernel has the bug.

Bug description

A bug was introduced in Linux kernel version 5.2: if a signal is delivered to a thread, and delivering the signal requires faulting in pages of the thread signal stack, then AVX YMM registers may be corrupted upon returning from the signal to the program. If the program was executing some function that uses the YMM registers, that function can behave unpredictably.

The bug only happens on systems with an x86 processor. The bug affects programs written in any language. The bug only affects programs that receive signals. Among programs that receive signals, the bug is more likely to affect programs that use an alternate signal stack. The bug only affects programs that use the YMM registers. In Go programs in particular the bug will normally cause memory corruption, as Go programs primarily use the YMM registers to implement copying one memory buffer to another.

The bug was reported to the Linux kernel developers. It was quickly fixed. The bug fix was not ported back to the Linux kernel 5.2 series. The bug was fixed in Linux kernel versions 5.3.15, 5.4.2, and 5.5 and later.

The bug is only present if the kernel was compiled with GCC 9 or later.

The bug is present in vanilla Linux kernel versions 5.2.x for any x, 5.3.0 through 5.3.14, and 5.4.0 and 5.4.1. However, many distros that are shipping those kernel versions have in fact backported the patch (which is very small). And, some distros are still compiling their kernel with GCC 8, in which case the kernel does not have the bug.

In other words, even if your kernel is in the vulnerable range, there is a good chance that it is not vulnerable to the bug.

Bug test

To test whether your kernel has the bug, you can run the following C program (click on "Details" to see the program). On a buggy kernel, it will fail almost immediately. On a kernel without the bug, it will run for 60 seconds and then exit with a 0 status.

```c
// Build with: gcc -pthread test.c
//
// This demonstrates an issue where AVX state becomes corrupted when a
// signal is delivered where the signal stack pages aren't faulted in.
//
// There appear to be three necessary ingredients, which are marked
// with "!!!" below:
//
// 1. A thread doing AVX operations using YMM registers.
//
// 2. A signal where the kernel must fault in stack pages to write the
//    signal context.
//
// 3. Context switches. Having a single task isn't sufficient.

#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/mman.h>
#include <sys/prctl.h>
#include <sys/wait.h>

static int sigs;

static stack_t altstack;
static pthread_t tid;

static void die(const char* msg, int err) {
  if (err != 0) {
    fprintf(stderr, "%s: %s\n", msg, strerror(err));
  } else {
    fprintf(stderr, "%s\n", msg);
  }
  exit(EXIT_FAILURE);
}
```

```
void handler(int sig __attribute__((unused)),
             siginfo_t* info __attribute__((unused)),
             void* context __attribute__((unused))) {
  sigs++;
}

void* sender(void *arg) {
  int err;

  for (;;) {
    usleep(100);
    err = pthread_kill(tid, SIGWINCH);
    if (err != 0)
      die("pthread_kill", err);
  }
  return NULL;
}

void dump(const char *label, unsigned char *data) {
  printf("%s =", label);
  for (int i = 0; i < 32; i++)
    printf(" %02x", data[i]);
  printf("\n");
}

void doAVX(void) {
  unsigned char input[32];
  unsigned char output[32];

  // Set input to a known pattern.
  for (int i = 0; i < sizeof input; i++)
    input[i] = i;
  // Mix our PID in so we detect cross-process leakage, though this
  // doesn't appear to be what's happening.
  pid_t pid = getpid();
  memcpy(input, &pid, sizeof pid);

  while (1) {
    for (int i = 0; i < 1000; i++) {
      // !!! Do some computation we can check using YMM registers.
      asm volatile(
        "vmovdqu %1, %%ymm0;"
        "vmovdqa %%ymm0, %%ymm1;"
        "vmovdqa %%ymm1, %%ymm2;"
        "vmovdqa %%ymm2, %%ymm3;"
```

```
        "vmovdqu %%ymm3, %0;"
        : "=m" (output)
        : "m" (input)
        : "memory", "ymm0", "ymm1", "ymm2", "ymm3");
      // Check that input == output.
      if (memcmp(input, output, sizeof input) != 0) {
        dump("input ", input);
        dump("output", output);
        die("mismatch", 0);
      }
    }

    // !!! Release the pages of the signal stack. This is necessary
    // because the error happens when copy_fpstate_to_sigframe enters
    // the failure path that handles faulting in the stack pages.
    // (mmap with MMAP_FIXED also works.)
    //
    // (We do this here to ensure it doesn't race with the signal
    // itself.)
    if (madvise(altstack.ss_sp, altstack.ss_size, MADV_DONTNEED) != 0)
      die("madvise", errno);
  }
}

void doTest() {
  // Create an alternate signal stack so we can release its pages.
  void *altSigstack = mmap(NULL, SIGSTKSZ, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
  if (altSigstack == MAP_FAILED)
    die("mmap failed", errno);
  altstack.ss_sp = altSigstack;
  altstack.ss_size = SIGSTKSZ;
  if (sigaltstack(&altstack, NULL) < 0)
    die("sigaltstack", errno);

  // Install SIGWINCH handler.
  struct sigaction sa = {
    .sa_sigaction = handler,
    .sa_flags = SA_ONSTACK | SA_RESTART,
  };
  sigfillset(&sa.sa_mask);
  if (sigaction(SIGWINCH, &sa, NULL) < 0)
    die("sigaction", errno);

  // Start thread to send SIGWINCH.
  int err;
  pthread_t ctid;
```

```
  tid = pthread_self();
  if ((err = pthread_create(&ctid, NULL, sender, NULL)) != 0)
    die("pthread_create sender", err);

  // Run test.
  doAVX();
}

void *exiter(void *arg) {
  sleep(60);
  exit(0);
}

int main() {
  int err;
  pthread_t ctid;

  // !!! We need several processes to cause context switches. Threads
  // probably also work. I don't know if the other tasks also need to
  // be doing AVX operations, but here we do.
  int nproc = sysconf(_SC_NPROCESSORS_ONLN);
  for (int i = 0; i < 2 * nproc; i++) {
    pid_t child = fork();
    if (child < 0) {
      die("fork failed", errno);
    } else if (child == 0) {
      // Exit if the parent dies.
      prctl(PR_SET_PDEATHSIG, SIGKILL, 0, 0, 0);
      doTest();
    }
  }

  // Exit after a while.
  if ((err = pthread_create(&ctid, NULL, exiter, NULL)) != 0)
    die("pthread_create exiter", err);

  // Wait for a failure.
  int status;
  if (wait(&status) < 0)
    die("wait", errno);
  if (status == 0)
    die("child unexpectedly exited with success", 0);
  fprintf(stderr, "child process failed\n");
  exit(1);
}
```

What to do

If your kernel version is in the range that may contain the bug, run the C program above to see if it fails. If it fails, your kernel is buggy. You should upgrade to a newer kernel. There is no workaround for this bug.

Go programs built with 1.14 will attempt to mitigate the bug by using the `mlock` system call to lock the signal stack page into memory. This works because the bug only occurs if the signal stack page has to be faulted in. However, this use of `mlock` can fail. If you see the message

```
runtime: note: mlock workaround for kernel bug failed with errno 12
```

the `errno 12` (also known as `ENOMEM`) means that `mlock` failed because the system set a limit on the amount of memory that a program can lock. If you can increase the limit, the program may succeed. This is done using `ulimit -l`. When running a program in a docker container, you can increase the limit by invoking docker with the option `-ulimit memlock=67108864`.

If you cannot increase the `mlock` limit, then you can make the bug less likely to interfere with your program by setting the environment variable `GODEBUG=asyncpreemptoff=1` when running a Go program. However, this just makes your program less likely to suffer memory corruption (because it reduces the number of signals that your program will receive). The bug is still present, and memory corruption may still occur.

Questions?

Ask on the mailing list golang-nuts@googlegroups.com, or on any Go forum as described at Questions.

Details

To see more details on how the bug affects Go programs and how it was detected and understood, see #35777 and #35326.