LINUX | 21.07.2023

Let's Embed a Go Program into the Linux Kernel

Today, we would like to present a lesser-known feature of the Linux kernel. Instead of launching a program from a file system, regardless of whether it's virtual or not, it is also possible to embed a user-space program directly into the kernel image itself and start it from there.

Introduction

At first glance, this might sound outlandish and of little use, but there are cases where the kernel needs to execute helper programs. Prominent examples include the module loader helper, `/sbin/modprobe`, or the Spanning Tree Protocol (STP) helper for Linux's 802.1d Ethernet bridge subsystem, `/sbin/bridge-stp`. On Linux, programs are typically launched from a file system using the `execve()` system call. The kernel reads the initial parts of the specified file and hands over execution to user space. Various helper functions exist on top of this system call, but they all have in common that the program to be executed is started from a file. The same applies to the Linux user mode helper API, which allows executing a program from a driver. Both scenarios require the program to be installed and loadable. In most cases, having the program installed is a trivial problem if you have control over all user space, as is the case with a typical Linux distribution.

However, things become complicated if you are a Board Support Package (BSP) supplier, either external or in-house, and don't always have easy access to the workflow that determines what to install in the root file system. Ensuring that a program is actually loadable from the current root file system is a different story. A common problem arises in setups where the root file system is attached using network storage, such as NFS or iSCSI. If the network connection is unavailable, all access to the root file system fails, and the kernel is unable to run helpers anymore. This is where the mechanism we'd like to present in this blog post can help.

Let's assume we have a device driver called `embedded_prog` that requires a user space helper program, which must be executable at any time. First, we need the program we want to embed. Strictly speaking, any program will do, but we need to ensure that the program in question has no dependencies on the file system. Linking it statically provides benefits. Go programs are statically linked by default, and to illustrate that the following approach works with any kind of program, we have chosen to embed a Go program into the kernel. The driver itself is very simple. All it does is execute the helper program as soon as the driver is loaded, and when the program writes something into the pipe, the contents are directly logged into the kernel log buffer.

The Go Program

To keep things simple, the program itself will be straightforward. It will write 'Hello, world!' to standard output every five seconds.

```
1    package main
2
3    import (
4            "fmt"
5            "time"
6    )
7
8    func main() {
9            for {
10                   fmt.Println("Hello, world!")
11                   time.Sleep(5 * time.Second)
12           }
13   }
```

Since we want to be independent of the root file system, we need to ensure that the program does not access files from the root file system. We can achieve this by using strace to trace its system calls.

```
$ strace -fe trace=file ./eprog_user
execve("./eprog_user", ["./eprog_user"], 0x7fff743fbd50 /* 61 vars */) = 0
openat(AT_FDCWD, "/sys/kernel/mm/transparent_hugepage/hpage_pmd_size", O_RDONLY) = 3
strace: Process 17908 attached
strace: Process 17909 attached
strace: Process 17910 attached
strace: Process 17911 attached
[pid 17905] --- SIGURG {si_signo=SIGURG, si_code=SI_TKILL, si_pid=17905, si_uid=1000
Hello, world!
Hello, world!
^C
strace: Process 17905 detached
strace: Process 17908 detached
strace: Process 17909 detached
strace: Process 17910 detached
strace: Process 17911 detached
```

Looks good! Access to files in `/proc` or `/sys` is fine. These are virtual file systems and can be assumed to be present and working at all times. They are part of the Linux ABI interface.

Linux's User Mode Helper API

If your device driver needs to execute a user space program, the User Mode Helper API is the way to go. The core function to use the API is `call_usermodehelper()`, which has the following signature:

```
int call_usermodehelper(const char *path, char **argv, char **envp, int wait);
```

Similar to the `execve()` system call, `call_usermodehelper()` takes parameters such as the filename of the program to be executed, supplied arguments, environment, and a flag denoting whether the call should be asynchronous or not. However, this is not what we want for running an embedded program within the kernel image itself.

Since Linux v4.18, a more advanced API called user mode driver is available to run a user mode helper. The basic idea behind this API is that instead of specifying a path to a file on the root file system, an arbitrary buffer can be provided. The content of this buffer will be executed in user space just like a regular program.

```
int umd_load_blob(struct umd_info *info, const void *data, size_t len);
int fork_usermode_driver(struct umd_info *info);
```

Using `umd_load_blob()`, a program context is created from a buffer, and later it is executed by `fork_usermode_driver()`. In addition to running the program, the User Mode Driver API also establishes a pipe between the program and the kernel, enabling inexpensive communication between them.

But how do we get our Go program into that buffer? We don't want a new user space interface where the program has to be loaded into the kernel first. We want it to be part of the kernel image.

GNU Assember to the Rescue!

The kernel build system, specifically the GNU assembler (gas), can assist us in embedding the Go program into the resulting kernel image during the build process. By using a gas file, `eprog_user_blob.S`, we can instruct the assembler to generate a C object file with two symbols: `embedded_umh_start` and `embedded_umh_end`. These symbols will contain the contents of a binary file specified by the `.incbin` command, and this file can be arbitrary.

```
1           .section .init.rodata, "a"
2           .global embedded_umh_start
3   embedded_umh_start:
4           .incbin "drivers/misc/embedded_prog/eprog_user"
5           .global embedded_umh_end
6   embedded_umh_end:
```

If we link the resulting object file, `eprog_user_blob.o`, with our Linux device driver, we can utilize the symbols to locate the contents of the Go program within the kernel image. This makes loading and executing the program from a buffer straightforward. In this case, the buffer passed to `umd_load_blob()` is a memory location within

the read-only section of the in-memory kernel image. The linker has kindly prepared everything for us at compile time!

The code snippet provided below demonstrates the essential segments of the device driver, `eprog_kern.c`, responsible for loading and executing the Go program:

```
1    struct umd_info eprog_ctx = {
2            .driver_name = "eprog_user",
3    };
4
5    umd_load_blob(&eprog_ctx, &embedded_umh_start, &embedded_umh_end - &embedded_u
6    fork_usermode_driver(&eprog_ctx);
```

Using the `nm` command, we can observe the symbols in the driver. If the driver is built as a loadable module, you can run `nm` on `eprog.o`. Otherwise, if it is built directly into the kernel, you can run `nm` on `vmlinux`.

```
$ nm eprog.o | grep embedded_umh_
00000000001bc4e4 R embedded_umh_end
0000000000000000 R embedded_umh_start
```

```
$ nm vmlinux | grep embedded_umh_
0000000060688a00 d embedded_umh_end
00000000604cc51c d embedded_umh_start
```

Gluing it all Together

So far we have the following components:

- eprog_kern.c: The kernel driver that runs `umd_load_blob()` and `fork_usermode_driver()`.
- eprog_user_blob.S, The assembly source file we use to embed the Go program into a C object.
- gohello/hello.go, Our Go program, located in a subdirectory of our driver.

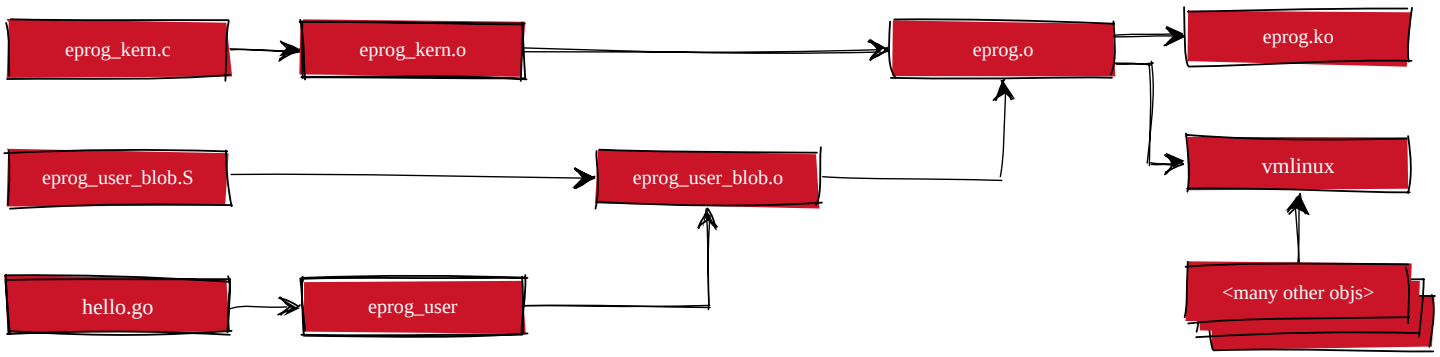
Using the following Makefile, all components are build and connected together:

```
1    $(obj)/eprog_user_blob.o: $(obj)/eprog_user
2    $(obj)/eprog_user: $(srctree)/drivers/misc/embedded_prog/gohello/hello.go
3            # TODO: Add support for cross builds
4            go build -o $(obj)/eprog_user $(srctree)/drivers/misc/embedded_prog/go
5
```

```
6     obj-$(CONFIG_EMBEDDED_PROG) += eprog.o
7     eprog-objs += eprog_kern.o eprog_user_blob.o
```

Linux's GNU make-based build system is rather flexible: it allows us to build our Go program even while building the kernel.

The following graphic outlines how all components are connected:



Demo

```
$ insmod eprog.ko
$ ps fax
  PID TTY      STAT   TIME COMMAND
    2 ?        S      0:00 [kthreadd]
[..]
   25 ?        Sl     0:00  \_ eprog_user
[..]
   30 ?        R      0:00 ps fax
$ dmesg -w
[   53.300000] eprog: From userspace: Hello, world!
[   58.310000] eprog: From userspace: Hello, world!
[...]
```

We observe a new process, `eprog_user`, right after loading the driver module. It is not a kernel thread, the name is not in square brackets and it is not a child of PID 1 but of `kthreadd`. Every five seconds the Go program will write `Hello, world!` via standard output to the kernel driver which prints the string to the kernel log buffer. Once the module is unloaded, the process is killed.

Summary

We have seen that using a small assembly file and the User Mode Driver mechanism, it is possible to embed any kind of executable into the kernel image and run it later from there. It works with any kind of program; we used a

Go binary because it is statically linked and has no dependencies on the root file system. If you're using your own program and want to be completely independent of the root file system, make sure that you know in detail which files it will process. While the number of valid use cases for the presented feature is arguably small, it is still interesting to have and opens the door to new possibilities. The example driver is rather simple, but there are numerous other ways to utilize this feature. Please keep in mind that adding programs to the kernel image comes with a cost: the program as a whole will always stay in memory.

The full example driver is available on git.kernel.org.

PUBLISH DATE	**21.07.2023**
CATEGORY	**Linux**
AUTHORS	**Richard Weinberger**

Get in touch

office@sigma-star.at (PGP Key)
+43 5 9980 400 00 (email preferred)

sigma star gmbh
Eduard-Bodem-Gasse 6, 1st floor
6020 Innsbruck | Austria