LKM loading kernel restrictions

Published: 2018-02-17 · Archived: 2026-04-06 01:53:13 UTC

Loading arbitrary kernel modules dynamically has always been a gray area between usability oriented and security oriented Linux developers & users. In this post I will present what options are available today from the Linux kernel and the most popular kernel hardening patch, the grsecurity. Those will give you some ideas on how those projects deal with the threat of Linux kernel's LKMs (Loadable Kernel Modules).

Threat

This can be split to two main categories, allowing dynamic LKM loading introduces the following two threats:

- Malicious LKMs. That's more or less rootkits or similar malware that an adversary can load for various operations, most commonly to hide specific activities from the user-space.
- Vulnerable LKM loading. Imagine that you have a 0day exploit on a specific network driver but this is not loaded by default. If you can trigger a dynamic loading then you can use your code to exploit it and compromise the system. This is what this vector is about.

Linux kernel and KSPP

The KSPP (Kernel Self-Protection Project) of the Linux kernel tried to fix this issue with the introduction of the kernel modules access restriction. Below you can see the exact description that Linux kernel's documentation has for this restriction.

```
Restricting access to kernel modules

The kernel should never allow an unprivileged user the ability to load specific
kernel modules, since that would provide a facility to unexpectedly extend the
available attack surface. (The on-demand loading of modules via their predefined
subsystems, e.g. MODULE_ALIAS_*, is considered "expected" here, though additional
consideration should be given even to these.) For example, loading a filesystem
module via an unprivileged socket API is nonsense: only the root or physically
local user should trigger filesystem module loading. (And even this can be up
for debate in some scenarios.)
To protect against even privileged users, systems may need to either disable
module loading entirely (e.g. monolithic kernel builds or modules_disabled
sysctl), or provide signed modules (e.g. CONFIG_MODULE_SIG_FORCE, or dm-crypt
```

```
13  with LoadPin), to keep from having root load arbitrary kernel code via the
14  module loader interface.
15
16
```

The most restrictive way is via modules_disabled sysctl variable which is available by default on the Linux kernel. This can either be set dynamically as you see here.

```
1  sysctl -w kernel.modules_disabled=1
```

Or permanently as part of the runtime kernel configuration as you can see here.

```
1  echo 'kernel.modules_disabled=1' >> /etc/sysctl.d/99-custom.conf
```

In both cases, the result is the same. Basically, the above change its default value from “0” to “1”. You can find the exact definition of this variable in kernel/sysctl.c.

```
1
    kernel/sysctl.c
2
        {
3
            .procname    =  "modules_disabled" ,
4
            .data        = &modules_disabled,
5
            .maxlen        =  sizeof ( int ),
6
            .mode        = 0644,
7
            .proc_handler    = proc_dointvec_minmax,
8
            .extra1        = &one,
9
            .extra2        = &one,
10
        },
11
```

If we look into kernel/module.c we will see that if modules_disabled has a non-zero value it is not allowing LKM loading (may_init_module()) or even unloading (delete_module() system call) of any LKM. Below you can see the module initialization code that requires both the SYS_MODULE POSIX capability, and modules_disabled to be zero.

```
1
2
3
4
5
6
7
```

```
static int may_init_module( void )
{
      if (!capable(CAP_SYS_MODULE) || modules_disabled)
            return -EPERM;
      return 0;
}
```

Similarly, the delete_module() system call has the exact same check as shown below. In both cases, failure of those will result in a Permission Denied (EPERM) error.

```
1
2
3
4
5
6
7
8
9
```

```
SYSCALL_DEFINE2(delete_module, const char __user *, name_user,
            unsigned int , flags)
{
      struct module *mod;
      char name[MODULE_NAME_LEN];
      int ret, forced = 0;
      if (!capable(CAP_SYS_MODULE) || modules_disabled)
            return -EPERM;
```

Looking in kernel/kmod.c we can also see another check, before the kernel module loading request is passed to call_modprobe() to get loaded in the kernel, the __request_module() function verifies that modprobe_path is set, meaning the LKM is not loaded via an API or socket instead of /sbin/modprobe command.

```
1
```

```
int __request_module( bool wait, const char *fmt, ...)
```

```
2   {
3       va_list args;
4       char module_name[MODULE_NAME_LEN];
5       int ret;
6       WARN_ON_ONCE(wait && current_is_async());
7       if (!modprobe_path[0])
8           return 0;
9
10
11
12
13
14
15
16
```

The above were the features that Linux kernel had for years to protect against this threat. The downside though is that completely disabling loading and unloading of LKMs can break some legitimate operations such as system upgrades, reboots on systems that load modules after boot, automation configuring software RAID devices after boot, etc.

To deal with the above, on 22 May 2017 the KSPP team proposed a patch to __request_module() (still to be added to the kernel) which follows a different approach.

```
1   int __request_module( bool wait,  int allow_cap,  const char *fmt, ...)
2   {
3       ...
4       if (!modprobe_path[0])
5           return 0;
```

```
6       ...
7       ret = security_kernel_module_request(module_name, allow_cap);
8       if (ret)
9
10
11
```

What you see here is that in the very early stage of the kernel module loading security_kernel_module_request() is invoked with the module to be loaded as well as allow_cap variable which can be set to either “0” or “1”. If its value is positive, the security subsystem will trust the caller to load modules with specific predifned (hardcoded) aliases. This should allow auto-loading of specific aliases. This was done to close a design flaw of the Linux kernel where although all modules required the CAP_SYS_MODULE capability to load modules (which is already checked as shown earlier), the network modules required the CAP_NET_ADMIN capability which completely bypassed the previously described controls. Using this modified __request_module() it is ensured that only specific modules that are allowed by the security subsystem will be able to auto-load. However, it is also crucial to note that to this date, the only security subsystem that utilizes security_kernel_module_request() hook is the SELinux.

Before we move on with grsecurity, it is important to note that in 07 November 2010 Dan Rosenberg proposed a replacement of modules_disabled, the modules_restrict which was a copy of grsecurity’s logic. It had three values, 0 (disabled), 1 (only root can load/unload LKMs), 2 (no one can load/unload – same as modules_disabled). You can see the check that it was adding to __request_module() below.

```
1
        if (current_uid() && modules_restrict)
2
                return -EPERM;
3
```

However, this was never added to the upstream kernel so there is no need to dive more into the details behind it. Just as an overview, here is the proposed kernel configuration option documentation for modules_restrict.

```
1       modules_restrict:
2       A toggle value indicating if modules are allowed to be loaded
3       in an otherwise modular kernel.  This toggle defaults to off
4       (0), but can be set true (1).  Once true, modules can be
```

```
5   neither loaded nor unloaded, and the toggle cannot be set back
6   to false.
7   A value indicating if module loading is restricted in an
8   otherwise modular kernel.  This value defaults to off (0),
9   but can be set to (1) or (2).  If set to (1), modules cannot
10  be auto-loaded by non-root users, for example by creating a
11  socket using a packet family that is compiled as a module and
12  not already loaded.  If set to (2), modules can neither be
13  loaded nor unloaded, and the value can no longer be changed.
14
```

grsecurity

Unfortunately, grsecurity stable patches are no longer publicly available. For this reason, in this article I will be using the grsecurity patch for kernel releases 3.1 to 4.9.24. For the LKM loading hardening grsecurity offers a kernel configuration option known as MODHARDEN (Harden Module Auto-loading). If we go back to ___request_module() in kernel/kmod.c we will see how this feature works.

```
1   
2       ret = security_kernel_module_request(module_name);
3       if (ret)
4           return ret;
5   #ifdef CONFIG_GRKERNSEC_MODHARDEN
6       if (uid_eq(current_uid(), GLOBAL_ROOT_UID)) {
7           read_lock(&tasklist_lock);
8           if (! strcmp (current->comm, "mount" ) &&
9               current->real_parent && ! strncmp (current->real_parent->comm, "udisk" ,
    5)) {
10              read_unlock(&tasklist_lock);
11              printk(KERN_ALERT "grsec: denied attempt to auto-load fs module %.64s by
    udisks\n" , module_name);
12  
```

```
13              return -EPERM;
14          }
15          read_unlock(&tasklist_lock);
16      }
17  #endif
```

The check in this case is relatively simple, it verifies that the caller's UID is the same as the static global UID of root user. This ensure that only users with UID=0 can load kernel modules which completely eliminates the cases of unprivileged users exploiting flaws that are allowing them to request kernel module loading. To overcome the network kernel modules issue grsecurity followed a different approach which maintains the capability check (which is currently used by a very limited amount of security subsystems) but redirects all loading to the ___request_module() function to ensure that only root can load them.

```
1
2   void dev_load( struct net *net, const char *name)
3   {
4       ...
5       no_module = !dev;
6       if (no_module && capable(CAP_NET_ADMIN))
7           no_module = request_module( "netdev-%s" , name);
8       if (no_module && capable(CAP_SYS_MODULE)) {
9   #ifdef CONFIG_GRKERNSEC_MODHARDEN
10          ___request_module( true , "grsec_modharden_netdev" , "%s" , name);
11  #else
12          request_module( "%s" , name);
13  #endif
14  }
```

Furthermore, grsecurity identified that a similar security design flaw also exists in the filesystem modules loading (still to be identified and fixed in the upstream kernel), which was fixed in a similar manner. Below is the grsecurity version of fs/filesystems.c's get_fs_type() function which is ensuring that filesystem modules are loaded only by root user.

```
1  	fs = __get_fs_type(name, len);
2  	#ifdef CONFIG_GRKERNSEC_MODHARDEN
3  	if (!fs && (___request_module( true , "grsec_modharden_fs" , "fs-%.*s" , len, name) == 0))
4  	#else
5  	if (!fs && (request_module( "fs-%.*s" , len, name) == 0))
6  	#endif
7  	fs = __get_fs_type(name, len);
```

This Linux kernel design flaw allows loading of non-filesystem kernel modules via mount. How grsecurity detects those is quite clever and can be found in simplify_symbols() function of kernel/module.c. What it does is ensuring that the the arguments of the module are copied to the kernel side, and then checks the module's loading information in the symbol table to ensure that the loaded module is trying to register a filesystem instead of any arbitrary kernel module.

```
1  	#ifdef CONFIG_GRKERNSEC_MODHARDEN
2  		int is_fs_load = 0;
3  		int register_filesystem_found = 0;
4  		char *p;
5  		p = strstr (mod->args, "grsec_modharden_fs" );
6  		if (p) {
7  			char *endptr = p + sizeof ( "grsec_modharden_fs" ) - 1;
8  			memmove (p, endptr, strlen (mod->args) - (unsigned int )(endptr - mod->args) + 1);
9  			is_fs_load = 1;
10 		}
11
```

```
#endif
      for (i = 1; i < symsec->sh_size /  sizeof (Elf_Sym); i++) {
          const char *name = info->strtab + sym[i].st_name;
#ifdef CONFIG_GRKERNSEC_MODHARDEN
          if (is_fs_load && ! strcmp (name,  "register_filesystem" ))
              register_filesystem_found = 1;
#endif
switch (sym[i].st_shndx) {
```

To help in detection of malicious users trying to exploit this Linux kernel design flaw, grsecurity has also an alerting mechanism in place which immediately logs any attempts to load Linux kernel modules that are not filesystems using this design flaw. Meaning loading a kernel module via “mount” without that being an actual filesystem module.

```
#ifdef CONFIG_GRKERNSEC_MODHARDEN
      if (is_fs_load && !register_filesystem_found) {
          printk(KERN_ALERT  "grsec: Denied attempt to load non-fs module %.64s through mount\n" , mod->name);
          ret = -EPERM;
      }
#endif
```

```
8       return ret;
9   }
```

Security wise grsecurity's approach is by far the most complete but unfortunately, by default the Linux kernel doesn't have anything even close to this.
