The Linux Kernel's Attack Surface
(and how we can reduce it)

Anil Kurmus
kur@zurich.ibm.com ak@kernel.build
twitter: @kurmus
Systems security researcher at IBM Research – Zurich
“If you don’t have a dog, your neighbor can’t poison it.”

Sergey Nikitin, *If You Don’t Have an Aunt* (Russian song)
The Linux kernel: All you (n)ever wanted
The Linux kernel: All you (n)ever wanted

- RDS protocol
- Perf events
The Linux kernel: All you (n)ever wanted

- RDS protocol
- Perf events
- /proc/pid/mem
The Linux kernel: All you (n)ever wanted

- RDS protocol
- Perf events
- /proc/pid/mem
- Berkley Packet Filter (BPF)
The Linux kernel: All you (n)ever wanted

- RDS protocol CVE-2010-3904
- Perf events CVE-2013-2094
- /proc/pid/mem CVE-2012-0056
- BPF CVE-2010-4158
How popular are those features?
How popular are those features?
How popular are those features?
How popular are those features?

A /proc/PID/mem vulnerability [LWN.net]
It was part of a patch set that was specifically targeted at allowing debuggers to write to the memory of processes easily via the /proc/PID/mem file.

Linux Local Privilege Escalation via SUID /proc/pid/mem Write | Nerdling Sappie
There are no restrictions on opening; anyone can open the /proc/pid/mem file for any process (subject to the ordinary VFS restrictions).

Linux kernel 2.2.x /proc/pid/mem mmap() vulnerability
The /proc/pid/mem interface is designed to enable one application to, under certain conditions, access the memory of another application in a convenient way.

c - mmap on /proc/pid/mem - Stack Overflow
Has anybody succeeded in mmaping a /proc/pid/mem file with Linux kernel 2.6? I am getting an ENODEV (No such device) error. My call looks like this

Advisory: Linux kernel 2.2.x /proc/pid/mem mmap() vulnerability
Details: The /proc/pid/mem interface is designed to enable one application to, under certain conditions, access the memory of another application in a convenient way.

Vulnerability Note VU#470151 - ...privilege escalation via SUID /proc/pid/mem...

Tech Patterns :: patch: prevent Privilege Escalation via SUID /proc/pid/mem...
How popular are those features?

A /proc/PID/mem vulnerability [LWN]
It was part of a patch set that was specifically large processes easily via the /proc/PID/mem file.
LWN.net/Articles/476947  More from lwn.net

Linux Local Privilege Escalation via : Sappy
There are no restrictions on opening; anyone can o ordinary VFS restrictions.
blog.zx2c4.com/749  More from blog.zx2c4.com

Linux kernel 2.2.x /proc/pid/mem m
The /proc/pid/mem interface is designed to enable
memory of another application in a convenient way.
net-security.org/vuln.php?id=2314  More from net

C - mmap on /proc/pid/mem - Stack
Has anybody succeeded in mapping a /proc/pid/f
(No such device) error. My call looks like this
stackoverflow.com/questions/5216328/mmap-on-pr

Advisory: Linux kernel 2.2.x /proc/pi Details: The /proc/pid/mem interface is designed
access the memory of another application in a com
securityfocus.com/advisories/4797  More from sec

Vulnerability Note VU#470151 - ...pr
/proc/pid/mem...
Linux Kernel local privilege escalation via SUID /pr
Last revised: 28 Jan 2012.
kb.cort.org/vuls/id/470151  More from kb.cort.org

Tech Patterns :: patch: prevent Privil
/proc/pid/mem...

Berkeley Packet Filter - Wikipedia, the free encyclopedia
The Berkeley Packet Filter or BPF provides, on some Unix-like systems, a raw interface to data
link layers, permitting raw link-layer packets to be sent and received.

A JIT for packet filters [LWN.net]
The Linux BPF implementation can be found in net/core/filter.c; it provides "standard" BPF along with
a number of Linux-specific ancillary instructions which can test whether a packet is marked, which CPU
the filter is running on, which interface the packet arrived on, and more.
lwn.net/Articles/437361/  More from lwn.net

BPF - What is BPF - About.com Linux
Define BPF - from the Linux / Linux / Computing glossary at About.com.
about.com/cs/linux101/g/bpf.htm  More from about.com

Linux Networking - View topic - BPF for Linux
Does anyone know if there is any kind of Implementation of BSD Packet Filter (BPF) for Linux? Since it's a
kernel level implementation, I'm hoping that there's a loadable module out there.
linuxmisc.com/2-linux-networking/ol213881048b53c.htm  More from linuxmisc.com

Man Page for bpf (freebsd Section 4) - The UNIX and Linux Forums
bpf(4) - Berkeley Packet Filter. Man page for bpf(4) in the man set for freebsd at The UNIX and Linux
Forums.
unix.com/man-page/FreeBSD/4/bpf/  More from unix.com

net/bpf.h not installed - LinuxQuestions.org
Hi there, I tried to install pf(4) (passive operating fingerprinting tool) of camtuf, and looks like it didn't find
net/bpf.h. Code: [root@localhost pf]0
linuxquestions.org/questions/linux-software-2/net-bpf-h-no...  More from linuxquestions.org

Linux' packet mmap(2), BPF, and Netsniff-NG
Linux' packet mmap(2), BPF, and Netsniff-NG (Plumber's guide to find the needle in the network packet
haystack) DanieL Boekmann <boekmann@redhat.com>
pub.netsniff-ng.org/paper/devconf_2013.pdf  More from pub.netsniff-ng.org

Linux Socket Filter - In the Beginning was the Light
Linux Socket Filter - In the Beginning was the Light
Linux BPF EVENTS - not exploit - CVE-2013-0394 ...
Linux BPF EVENTS - not exploit - CVE-2013-0394 (quick way to fix it) Recently a quite critical flaw has
How popular are those features?

Large attack surface for no reason?
Research questions (1/2)

Q1: Is it possible to precisely define the kernel attack surface? How can it be measured?
Q2: Can we develop kernel protection mechanisms whose attack surface reduction is quantifiable? To what extent can these mechanisms be applied to Linux in practice?
This talk

P1: Kernel Attack Surface Quantification (NDSS'13)
This talk

P1: Kernel Attack Surface Quantification (NDSS'13)

P2: Compile-time Kernel Tailoring (HotDep'13, NDSS'13)
This talk

P1: Kernel Attack Surface Quantification (NDSS'13)

P2: Compile-time Kernel Tailoring (HotDep'13, NDSS'13)

P3: Run-time Kernel Trimming (Eurosec'11, DIMVA'14, CCS'14)
Measuring Kernel Attack Surface


Existing approaches and limitations

- Typically in OS research: measure TCB size in source lines of code.
  - Fiasco 15K SLOC; Minix 3 4K SLOC; Flicker 250 SLOC
  - Linux 3.0 10M SLOC;

- However:
  - Source files that are not compiled? Configuration-dependent code?
  - Loadable kernel modules (LKM)s? On-demand loadable kernel modules?
  - Code that is not reachable from the system call interface? Initialization code?
  - Code that is only reachable by privileged processes?
General Idea

- Attack surface \(\simeq\) attacker-reachable code
  - Idea: use reachability over kernel call graph
  - Assumptions on the attacker and kernel? (*security model*)

- Measurements: code quality metrics
  - SLOCs, CVEs, ...
Obtaining the attack surface: an example
Obtaining the attack surface: an example
Obtaining the attack surface: an example
Obtaining the attack surface: an example
Obtaining the attack surface: an example
Obtaining the attack surface: an example
Obtaining the attack surface: an example
Obtaining the attack surface: an example
Attack surface measurement: AS1 with SLOC metric

\[ \Sigma = 370 \text{ SLOC} \]
Attack surface measurements: summary

- Program source and configuration
- Entry and barrier functions
- Call graph: functions and calls
- Attack surface
- Attack surface metric

\[ AS1_\mu (G_{AS}) = \sum_{i \in FAS} \mu(i) \]

\[ \sum = 370 \text{ SLOC} \]
Attack surface measurements: summary

What security model?
IsolSec Linux Kernel Security Model

- Application (privileged)
- Application (unprivileged)
- Attacker controls unprivileged process
- System call interface
- Core Kernel
- LKM
- Hardware interface
- Hardware
- LKM (on-demand loadable)
- LKM (driver)
- LKM (other)
- attacker entry
- partial a.s.
- running kernel
IsolSec Linux Kernel Security Model

- Application (privileged)
- Application (unprivileged)
- System call interface
- Core Kernel
- LKM
- Hardware interface
- Hardware
- LKM (on-demand loadable)
- LKM (driver)
- LKM (other)

Attacker controls unprivileged process

attacker entry
- partial a.s.
- running kernel
IsolSec Linux Kernel Security Model

- Entry functions:
  - system calls
- Barrier functions:
  - Functions calling `capable()`

**Application** (privileged) → **Application** (unprivileged)

**System call interface**

**Core Kernel**

**Hardware interface**

**LKM**
- **LKM (on-demand loadable)**
- **LKM (driver)**
- **LKM (other)**

**Hardware**

- Partial a.s. running kernel

Attacker controls unprivileged process
IsolSec Linux Kernel Security Model

- **Application (privileged)**
- **Application (unprivileged)**
- **System call interface**
- **Core Kernel**
- **LKM**
- **Hardware interface**
- **Hardware**

- **LKM (on-demand loadable)**
- **LKM (driver)**
- **LKM (other)**

- Attacker controls unprivileged process
- Drivers and non-ODL LKMs are not considered

- Entry functions:
  - system calls
- Barrier functions:
  - Functions calling `capable()`

- Attack entry
  - partial a.s.
  - running kernel
IsolSec Linux Kernel Security Model

- **Application (privileged)**
- **Application (unprivileged)**

**System call interface**

- **Core Kernel**
- **LKM**

**Hardware interface**

- **Hardware**

**LKM (on-demand loadable)**

- **LKM (driver)**
- **LKM (other)**

- **Attacker controls unprivileged process**

- **Drivers and non-ODL LKMs are not considered**

- **Entry functions:**
  - system calls

- **Barrier functions:**
  - Functions calling `capable()`
  - Drivers and “other” LKMs

- **attacker entry**

  - partial a.s.
  - running kernel
IsolSec Linux Kernel Security Model

- **Entry functions**:  
  - system calls

- **Barrier functions**:  
  - Functions calling `capable()`  
  - Drivers and “other” LKMs  
  - (procfs, sysfs, debugfs)

- Attacker controls unprivileged process

- Drivers and non-ODL LKMs are not considered

- Attacker entry
  - partial a.s.
  - running kernel
IsolSec Linux Kernel Security Model

- **Application (privileged)**
- **Application (unprivileged)**
- **System call interface**
- **Core Kernel**
- **LKM**
- **Hardware interface**
- **LKM (on-demand loadable)**
- **LKM (driver)**
- **LKM (other)**
- **Hardware**
- **attacker entry**
  - partial a.s.
  - running kernel

**Drivers and non-ODL LKMs are not considered**

- **Entry functions:**
  - system calls
- **Barrier functions:**
  - Functions calling `capable()`
  - Drivers and “other” LKMs
  - (procfs, sysfs, debugfs)
- **Purpose:** estimating the attack surface from an untrusted, unprivileged process
StaticSec Linux Kernel Security Model

Application (privileged)

Application (unprivileged)

System call interface

Core Kernel

LKM

Hardware interface

Hardware

LKM (on-demand loadable)

LKM (driver)

LKM (other)

Attacker controls unprivileged process

LKMs cannot be On-demand loaded

attacker entry

partial a.s.

running kernel
GenSec Linux Kernel Security Model

Application (privileged)

Application (unprivileged)

System call interface

Core Kernel

LKM

LKM (on-demand loadable)

LKM (driver)

LKM (other)

Hardware interface

Hardware

attacker entry

attack surface

running kernel
GenSec Linux Kernel Security Model

- Entry functions: all
- Barrier functions: none
GenSec Linux Kernel Security Model

- Entry functions: all
- Barrier functions: none
- Overestimates attack surface: attacker is privileged?
- Not all LKMs can be loaded
- Purpose: upper bound, TCB point of view
Compile-time Kernel Tailoring


Making the kernel smaller

~ 5000 features
(ubuntu 12.04)

~ 500 features
(realistic use case)
Making the kernel smaller

Remove unnecessary features from the kernel by leveraging built-in configurability

~ 5000 features (ubuntu 12.04)

~ 500 features (realistic use case)
Make (menuconfig) your way to a smaller kernel

Now with ~5K features to choose from! (on x86)
Don't take my word for it

[ RFC ] Simplifying kernel configuration for distro issues
87 messages

Linus Torvalds <torvalds@linuxfoundation.org> Fri, Jul 13, 2012 at 10:37 PM
To: Dave Jones <davej@redhat.com>, Greg Kroah-Hartman <greg@kroah.com>, Ubuntu Kernel Team <kernel-team@lists.ubuntu.com>, Debian Kernel Team <debian-kernel@lists.debian.org>, OpenSUSE Kernel Team <opensuse-kernel@opensuse.org>
Cc: Linux Kernel Mailing List <linux-kernel@vger.kernel.org>

So this has long been one of my pet configuration peeves: as a user I am perfectly happy answering the questions about what kinds of hardware I want the kernel to support (I kind of know that), but many of the "support infrastructure" questions are very opaque, and I have no idea which of the them any particular distribution actually depends on.

And it tends to change over time. For example, F14 (iirc) started using TMPFS and TMPFS_POSIX_ACL/XATTR for /dev. And starting in F16, the initrd setup requires DEVTMPFS and DEVTMPFS_MOUNT. There's been several times when I started with my old minimal config, and the resulting kernel would boot, but something wouldn't quite work right, and it can be very subtle indeed.

Similarly, the distro ends up having very particular requirements for exactly *which* security models it uses and needs, and they tend to change over time. And now with systemd, CGROUPS suddenly aren't just esoteric things that no normal person would want to use, but are used for basic infrastructure. And I remember being surprised by OpenSUSE suddenly needing the RAW table support for netfilter, because it had a NOTRACK rule or something.
Don't take my word for it

[RFC] Simplifying kernel configuration for distro issues
87 messages

Linus Torvalds <torvalds@linuxfoundation.org> Fri, Jul 13, 2012 at 10:37 PM
To: Dave Jones <davej@redhat.com>, Greg Kroah-Hartman <greg@kroah.com>, Ubuntu Kernel Team <kernel-team@lists.ubuntu.com>, Debian Kernel Team <debian-kernel@lists.debian.org>, OpenSUSE Kernel Team <opensuse-kernel@opensuse.org>
Cc: Linux Kernel Mailing List <linux-kernel@vger.kernel.org>

“many of the support infrastructure questions are very opaque, and I have no idea which of them any particular distribution actually depends on.”

And it tends to change over time. For example, F14 (jfc) started using TMPFS and TMPFS_POSIX_ACL/XATTR for /dev. And starting in F16, the initrd setup requires DEVTMPFS and DEVTMPFS_MOUNT. There's been several times when I started with my old minimal config, and the resulting kernel would boot, but something wouldn't quite work right, and it can be very subtle indeed.

Similarly, the distro ends up having very particular requirements for exactly *which* security models it uses and needs, and they tend to change over time. And now with systemd, CGROUPS suddenly aren't just esoteric things that no normal person would want to use, but are used for basic infrastructure. And I remember being surprised by OpenSUSE suddenly needing the RAW table support for netfilter, because it had a NOTRACK rule or something.
Automatic Kernel-Configuration Tailoring
Automatic Kernel-Configuration Tailoring

Distribution kernel and use case
Automatic Kernel-Configuration Tailoring

Distribution kernel and use case

Tailored kernel
Automatic Kernel-Configuration Tailoring

Distribution kernel and use case

run workload and collect **trace**
correlate to **source line** locations and **#ifdefs**
correlate to **features** and take into account **feature dependencies**
solve formula and derive a **kernel configuration**

Tailored kernel
Automatic Kernel-Configuration Tailoring

Distribution kernel and use case

run workload and collect trace

correlate to source line locations and #ifdefs

correlate to features and take into account feature dependencies

solve formula and derive a kernel configuration

Tailored kernel
Resulting kernel

- arch: 33%
- block: 15%
- crypto: 71%
- drivers: 95%
- fs: 86%
- ipc: 38%
- kernel: 34%
- lib: 25%
- mm: 8%
- net: 87%
- sound: 100%
- others: 62%

- removed files from tailored kernel compared to Ubuntu standard
- source files in both kernels
Resulting kernel

![Graph showing the comparison between files removed and source files in both kernels.]

- **arch**: 33%
- **block**: 15%
- **crypto**: 71%
- **drivers**: 95%
- **fs**: 86%
- **ipc**: 38%
- **kernel**: 34%
- **lib**: 25%
- **mm**: 8%
- **net**: 87%
- **sound**: 100%
- **others**: 62%

- **removed files from tailored kernel compared to Ubuntu standard**
- **source files in both kernels**
Resulting kernel

- How much **attack surface reduction**?
Selected results of the evaluation

- Typical server use case: LAMP
Results: tracing

- Httpperf benchmark triggers new features
  - Stabilizes at 495 features
- Skipfish: high coverage of the web application
  - Goes beyond real-world workload

Tracing at “feature-granularity” converges quickly
Results: attack surface reduction
Results: attack surface reduction
Results: attack surface reduction

85% reduction in attack surface.
Results: attack surface reduction

85%
Results: attack surface reduction

- GenSec
- IsolSec

85% attack surface reduction
Results: attack surface reduction

85% reduction in attack surface for GenSec.

82% reduction in attack surface for IsolSec.
Results: attack surface reduction

- GenSec: 85% reduction
- IsolSec: 82% reduction

The graph illustrates the reduction in attack surface in million lines of code, with categories such as drivers, fs, net, sound, kernel, and others.
Run-time Kernel Trimming

  

Same idea, more attack surface reduction!

- The promises of run-time attack surface reduction:
  - **More granular**
    - E.g., function-level instead of configuration-level
  - **Application-specific**
    - Different application may exercise different kernel functionality
- Challenges:
  - **Performance overhead** of run-time instrumentation
  - **False positives**
The false positive challenge

- machine 1: qemu-kvm
- machine 1: sshd
- machine 2: mysqld
- machine 2: sshd

Number of remaining kernel functions vs. time elapsed since first system call of the application (s)
The false positive challenge
Run-time kernel attack surface reduction

1. Pre-learning
   - All kernel functions

2. Learning
   - System set
     - `mwait_idle`
     - `__phys_addr`
   - Security contexts
     - `sshd_t`
     - `mysqld_t`

3. Analysis
   - Workload
   - Kernel sources
     - Dependence graph

4. Enforcement
   - Enforcement sets
Run-time kernel attack surface reduction
Run-time kernel attack surface reduction
Phase 1: Pre-learning

- Heuristic approach to improve performance
- Functions hit with frequency above a (dynamically computed) threshold are ignored
- Example:

```c
ext4_fsbblk_t ext4_mb_new_blocks(...) 
{
    ...
    while (ar->len && ext4_claim_free_blocks(sbi,
            ar->len)) {
        /* let others to free the space */
        yield();
        ar->len = ar->len >> 1;
    }
    ...
}
```

- Pre-learning reduces performance overhead
Phase 3: Analysis

- Group functions together to reduce false positives
- 4 different modes
  - No grouping
  - File grouping
  - Directory grouping
  - Cluster grouping
Phase 4: Enforcement

- Can't terminate process
  - False positives
  - Shared kernel state

- Two choices:
  - Logging (IDS)
  - Hardened mode enforcement via split kernel [CCS'14]
Split Kernel overview

- Build kernel with and without hardening
- Chose at run-time whether to run in hardened mode
- Performance impact of hardening greatly reduced
Selected results of the evaluation

- Real-world workload on RHEL 6 development server
  - Total observation time: 403 days
Attack surface reduction vs. convergence rate
Attack surface reduction vs. convergence rate

Convergence time (% of total observation time)

Attack Surface Reduction in SLOC (%)
Attack surface reduction vs. convergence rate

Convergence time (% of total observation time)

Attack Surface Reduction in SLOC (%)
Conclusion
Conclusion

- The kernel attack surface can be quantified
- This can be used to evaluate the effectiveness of kernel attack surface reduction
- Kernel attack surface reduction is effective in preventing kernel exploits:
  - Compile-time Tailoring
    - Prevents 285 out of 485 CVEs.
    - For well-defined use cases (e.g., embedded systems)
  - Run-time Trimming
    - Prevents up to 184 out of 262 CVEs.
    - More flexible, higher ASR but slower convergence rate
- Both mechanism aim to be practical
  - no significant overhead
  - non-intrusive
References


