Quantifying and Reducing Kernel Attack Surface

Anil Kurmus
kur@zurich.ibm.com - ak@kernel.build
@kurmus
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
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?

- **Perf Wiki**
  perf: Linux profiling with performance counters... More than just counters... Introduction. This is the wiki page for the perf performance counters subsystem in Linux.
  perf.wiki.kernel.org/index.php/Main_Page

- **Unofficial Linux Perf Events Performance Counter Web-Page**
  The Unofficial Linux Perf Events Web-Page because the perf_events developers don't seem that excited about writing documentation. The nearly un-googleable “Perf Events” subsystem was merged into the Linux kernel in version 2.6.31 (originally called “Performance Counters for Linux” (PCL)).
  web.ece.main.edu/~weaver/projects/perf_events/

- **PerfUserGuide - kernel - a user guide to Linux performance**
  Perf is a profiler tool for Linux 2.6+ based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple command line interface. Perf is based on the perf_events interface exported by recent versions of the Linux kernel.
  code.google.com/p/kernel/wiki/PerfUserGuide

- **Brendan’s blog » Linux Kernel Performance: Flame Graphs**
  Linux Kernel Performance: Flame Graphs. To get the most out of your systems, you want detailed insight into what the operating system kernel is doing.
  dtrace.org/blogs/brendan/2012/03/17/linux-kernel-p...

- **Perf events - KVM**
  This page describes how to count and trace performance events in the KVM kernel module. There are two tools, kvm_stat and kvm_trace, which were previously used for these tasks.
  linux-kvm.org/page/Perf_events

- **perf (Linux) - Wikipedia**
  perf (sometimes “Perf Events” or perf tools, originally “Performance Counters for Linux”, PCL) - is a performance analyzing tool in Linux, available from kernel version 2.6.31.
  en.wikipedia.org/wiki/Perf_...(Linux)

- **Linux PERF_EVENTS Local Root - EXPLOIT, LOCAL, SICUREZZA**
  linux perf_events local root, exploit, linux, local, local root, perf_events, privilege escalation, root, security, stocressa, vulnerabilities, vulnerability
  mondounix.com/linux-perf-events-local-root/

- **arighi’s blog: Linux PERF_EVENTS root exploit - CVE-2013-2094**
  Linux PERF EVENTS root exploit - CVE-2013-2094 (quick way to fix it) Recently a quite critical flaw has
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 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.

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 Sappie

There are no restrictions on opening; anyone can open 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/r (No such device) error. My call looks like this.

stackoverflow.com/questions/5215328/mmap-on-pr

Advisory: Linux kernel 2.2.x /proc/pi

Details: The /proc/pid/mem interface is designed to access the memory of another application in a convenient way.

securityfocus.com/advisories/4797 More from sec

Vulnerability Note VU#470151 - ...pr

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...
How popular are those features?

A `/proc/PID/mem` vulnerability [LWN](http://lwn.net/Articles/476947/)

It was part of a patch set that was specifically large processes easily via the `/proc/PID/mem` file. [More from lwn.net](http://lwn.net/Articles/476947/)

Linux Local Privilege Escalation via Sappie

There are no restrictions on opening; anyone can ordinary VFS restrictions. [More from lwn.net](http://lwn.net/Articles/437061/)

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 commodity OSes 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 \( \sim \) 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

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

\[ \Sigma = 370 \text{ SLOC} \]
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)
- Attacker controls unprivileged process
- System call interface
- LKM (on-demand loadable)
- LKM (driver)
- LKM (other)
- Hardware interface
- Hardware
- attacker entry
- partial a.s.
- running kernel
IsolSec Linux Kernel Security Model

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

**System call interface**

**Application (privileged)**

**Application (unprivileged)**

**Attacker controls unprivileged process**

**LKM (on-demand loadable)**

**LKM (driver)**

**LKM (other)**

**Core Kernel**

**Hardware interface**

**Hardware**

- 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)

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

Drivers and non-ODL LKMs are not considered

Attacker controls unprivileged process

Attacker entry
can run partial a.s.

running kernel
IsolSec Linux Kernel Security Model

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

**Attacker controls unprivileged process**

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

**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

- Application (privileged)
- Application (unprivileged)
- System call interface
- Core Kernel
- 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
  - (procfs, sysfs, debugfs)

attacker entry
partial a.s.
running kernel

© 2014 IBM Corporation
IsolSec Linux Kernel Security Model

- 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

- Attacker controls unprivileged process

- Drivers and non-ODL LKMs are not considered
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 entry
- partial a.s.
- running kernel
- Attacker controls unprivileged process
- LKMs cannot be On-demand loaded
GenSec 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

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 (IRC) 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.
“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 (lfc) 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

Tailored kernel

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**
Automatic Kernel-Configuration Tailoring

Distribution kernel and use case

Tailored kernel

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**
Resulting kernel

- Removed files from tailored kernel compared to Ubuntu standard
- Source files in both kernels

Legend:
- Blue: 33%
- Orange: 15%
- Dark blue: 71%
- Medium orange: 86%
- Yellow: 38%
- Green: 34%
- Red: 25%
- Pink: 8%
- Brown: 87%
- Light gray: 100%
- Gray: 62%
Resulting kernel

![Kernel Comparison Diagram]

- **Drivers**: 95%
- **FS**: 86%
- **Networking**: 87%
- **Others**: 62%

- **Red** indicates removed files from tailored kernel compared to Ubuntu standard.
- **Blue** indicates source files in both kernels.
### Resulting kernel

<table>
<thead>
<tr>
<th>Category</th>
<th>Attack Surface Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>arch</td>
<td>33%</td>
</tr>
<tr>
<td>block</td>
<td>15%</td>
</tr>
<tr>
<td>crypto</td>
<td>71%</td>
</tr>
<tr>
<td>drivers</td>
<td>95%</td>
</tr>
<tr>
<td>fs</td>
<td>86%</td>
</tr>
<tr>
<td>ipc</td>
<td>38%</td>
</tr>
<tr>
<td>kernel</td>
<td>34%</td>
</tr>
<tr>
<td>lib</td>
<td>25%</td>
</tr>
<tr>
<td>mm</td>
<td>8%</td>
</tr>
<tr>
<td>net</td>
<td>87%</td>
</tr>
<tr>
<td>sound</td>
<td>100%</td>
</tr>
<tr>
<td>others</td>
<td>62%</td>
</tr>
</tbody>
</table>

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

- Typical server use case: LAMP
Results: tracing

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

No new features

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

85%
Results: attack surface reduction

85%
Results: attack surface reduction

- GenSec
  - Drivers
  - fs
  - net
  - Sound
  - Kernel
  - Others

- IsolSec
  - Drivers
  - fs
  - net
  - Sound
  - Kernel
  - Others

85% attack surface reduction
Results: attack surface reduction

- GenSec: 85% reduction
- IsolSec: 82% reduction
Results: attack surface reduction
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
The false positive challenge

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

The graph shows the number of remaining kernel functions over time elapsed since the first system call of the application. The time axis is non-linear, with time points marked as 1e+06 and 1e+07.
Run-time kernel attack surface reduction
Run-time kernel attack surface reduction

Performance
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)

0 10 20 30

0 10 20 30 40 50 60 70 80 90 100

Attack Surface Reduction in SLOC (%)

No grouping
File grouping
Cluster grouping
Directory grouping
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 exploits:
  - Compile-time Tailoring
    - Prevents 285 CVEs out of 485.
  - Run-time Trimming
    - Prevents up to 184 out of 262 CVEs.
    - In general, better ASR but lower convergence rate
- Both mechanism aim to be practical
  - no significant overhead
  - non-intrusive
References


