kGuard

Lightweight Kernel Protection against Return-to-user Attacks

Vasileios P. Kemerlis Georgios Portokalidis
Angelos D. Keromytis

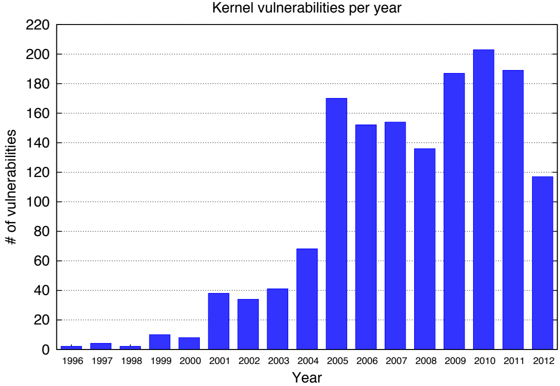
Network Security Lab
Department of Computer Science
Columbia University
New York, NY, USA

USENIX Security Symposium

Outline

Kernel vulnerabilities

Current state of affairs



Linux alone had $\geq$ 140 assigned CVE numbers in 2010!

- 12 privilege escalation exploits
- 13 bugs that can be triggered **remotely**
- kernel memory leaks, auth{entication, orization} bypass, DoS, ...

Kernel vulnerabilities (cont'd)

Why care?

Kernel attacks are becoming more common

- High-value asset → Most **privileged** piece of code
  - Responsible for the security of the OS → Reference monitor
- Large attack surface (syscalls, device drivers, pseudo fs, ...)
  - Big codebase → More bugs?
- Exploiting privileged userland processes has become harder → canaries+ASLR+NX+Fortify source+RELRO+BIND_NOW+Ptr. mangle+, ...

Return-to-user (ret2usr) attacks

Attacking the Core

Traditional kernel exploitation

- Kernel-level memory corruption $\rightsquigarrow$ code-injection, code-reuse (ROP)

Return-to-user (ret2usr) attacks

- Kernel-level memory corruption $\rightsquigarrow$ run userland code

- Attacks against OS kernels that have shared address spaces
- Overwrite kernel-level control data with **user space** addresses
  - return addresses
  - dispatch tables
  - function pointers
- Facilitate privilege escalation (arbitrary user-provided code execution)
  - ✗ `http://www.exploit-db.com/exploits/20201/` released last week!

Return-to-user (ret2usr) attacks

Why do they work?

Weak kernel/userland separation

- Shared process/kernel model → Performance
- Kernel entrance is hardware-assisted → The opposite is not true
- While executing kernel code complete and unrestricted access to all memory and system objects is available
- The attacker completely controls user space memory (both in terms of contents & perms.)

kGuard

Versatile & lightweight protection against ret2usr

- Defensive mechanism that builds upon inline monitoring and code diversification
- Cross-platform solution that enforces address space separation between user and kernel space
  - x86, x86-64, ARM
  - Linux, Android, {Free, Net, Open}BSD, ...
- Non-intrusive & low overhead

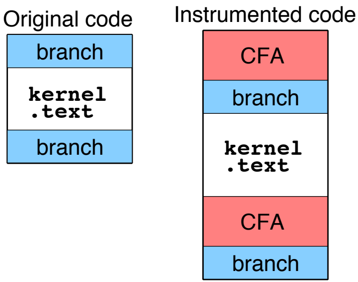
Goal

✓ Cast a **realistic** threat ineffective

kGuard design

Control-flow assertions (key technology #1)

- Compact, inline guards injected at compile time
  - Two flavors → CFA_R & CFA_M
- Placed before every exploitable control transfer
  - `call`, `jmp`, `ret` in x86/x86-64
  - `ldm`, `blx`, ..., in ARM



- Verify that the target address of an *indirect* branch is always inside kernel space
- If the assertion is true, execution continues normally; otherwise, control is transfered to a runtime violation handler

kGuard design (cont'd)

CFA_R example

```
     cmp  $0xc0000000,%ebx
     jae  lbl
     mov  $0xc05af8f1,%ebx
lbl: call *%ebx
```

```
if (reg < 0xc0000000)
  reg = &<violation_handler>;
call *reg
```

Indirect call in `drivers/cpufreq/cpufreq.c` (x86 Linux)

kGuard design (cont'd)

CFA_M examples (1/2)

```
       push %edi
       lea  0x50(%ebx),%edi
       cmp  $0xc0000000,%edi
       jae  lbl1
       pop  %edi
       call 0xc05af8f1
lbl1:  pop  %edi
       cmpl $0xc0000000,0x50(%ebx)
       jae  lbl2
       movl $0xc05af8f1,0x50(%ebx)
lbl2:  call *0x50(%ebx)
```

```
if (&mem < 0xc0000000)
  call <violation_handler>;
if (mem < 0xc0000000)
  mem = &<violation_handler>;
call *mem ;
```

Indirect call in `net/socket.c` (x86 Linux)

kGuard design (cont'd)

CFA_M examples (2/2) & optimizations

```
    cmpl    $0xc0000000,0xc123beef
    jae     lb
    movl    $0xc05af8f1,0xc123beef
lb: call    *0xc123beef
```

~~if (&mem < 0xc0000000)~~
  ~~call <violation_handler>;~~

```
if (mem < 0xc0000000)
  mem = &<violation_handler>;
call *mem ;
```

Optimized *CFA_M* guard (x86 Linux)

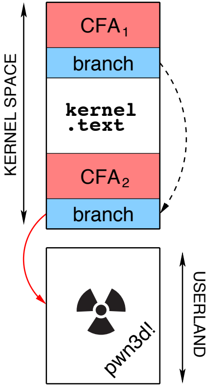
Bypassing kGuard

Bypass trampolines

- CFAs provide reliable protection *iff* the attacker partially controls a computed branch target
- What about vulnerabilities that allow overwriting kernel memory with **arbitrary** values?

Attacking kGuard

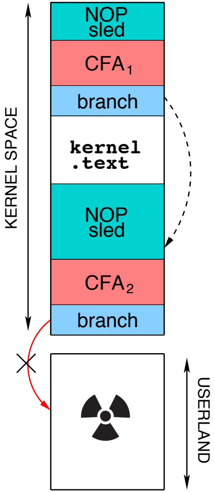
1. Find **two** computed branch instructions whose operands can be reliably overwritten
2. Overwrite the value (branch target) of the first with the address of the second
3. Overwrite the value of the second with a user-space address



Countermeasures

Code inflation (key technology #2)

- *Reshape* kernel's text area
  - Insert a random NOP sled at the *beginning* of the text
  - Inject a NOP sled of random length *before* every CFA
- Each NOP sled "pushes" further instructions at higher memory addresses (cumulative effect)



Result

- The location of each indirect control transfer is randomized (per build)

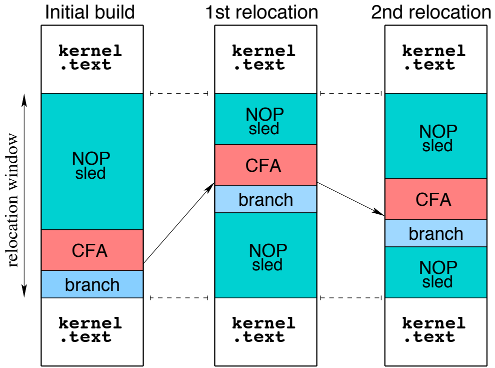
Important assumption

- Kernel text & symbols secrecy
  (proper fs privs., dmesg, /proc)

Countermeasures (cont'd)

CFA motion (key technology #3)

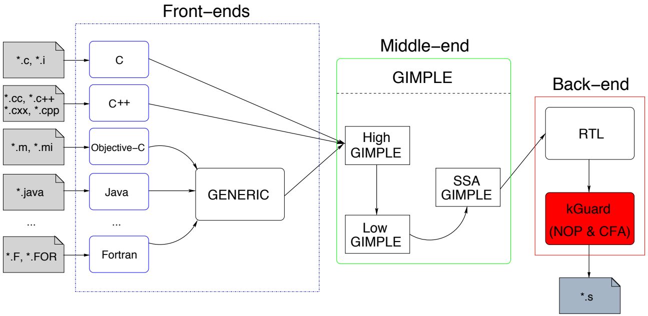
- Relocate the injected guards & protected branches
- Make it harder for an attacker to find a bypass trampoline



Implementation

kGuard as a GCC plugin

- Implemented kGuard as a set of modifications to the pipeline of GCC ("de-facto" compiler for Linux, BSD, Android, ...)
- Back-end plugin $\rightarrow \sim$ 1KLOC in C



Evaluating kGuard

Testbed & methodology

- Single host
  - 2.66GHz quad core Xeon X5500
  - 24GB RAM
- Debian Linux v6 ("squeeze" with kernel v2.6.32)
- GCC v4.5.1, MySQL v5.1.49, Apache v2.2.16
- `NOP` sled size $\rightsquigarrow$ [0 – 20]
- 10 repetitions of the same experiment
- 95% confidence intervals (error bars)

Effectiveness

Vulnerability	Description	Impact	Exploit x86	Exploit x86-64
CVE-2009-1897	NULL *function* pointer	2.6.30–2.6.30.1	$\surd$	—
CVE-2009-2692	NULL *function* pointer	2.6.0–2.6.30.4	$\surd$	$\surd$
CVE-2009-2908	NULL *data* pointer	$\leq$ 2.6.31	$\surd$	$\surd$
CVE-2009-3547	*data* pointer corruption	$\leq$ 2.6.32-rc6	$\surd$	$\surd$
CVE-2010-2959	*function* pointer overwrite	2.6.{27.x, 32.x, 35.x}	$\surd$	—
CVE-2010-4258	*function* pointer overwrite	$\leq$ 2.6.36.2	$\surd$	$\surd$
EDB-15916	NULL *function* pointer overwrite	$\leq$ 2.6.34	$\surd$	$\surd$
CVE-2009-3234	ret2usr via kernel stack buffer overflow	2.6.31	$\surd$	$\surd$

$\surd$: detected and prevented successfully —: exploit unavailable

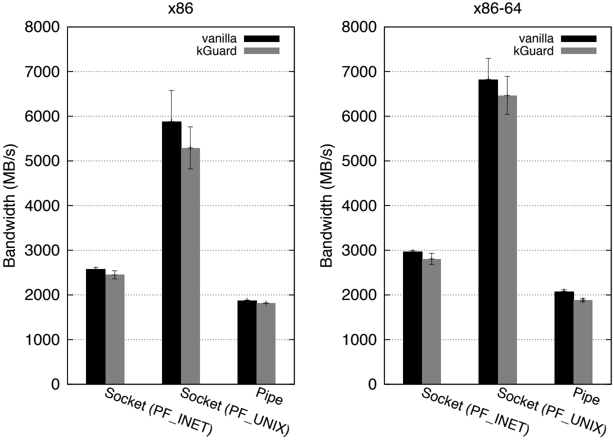
Macro benchmarks

App. (Bench.)	**x86**	**x86-64**
Kernel build (`time(1)`)	1.03%	0.93%
MySQL (`sql-bench`)	0.92%	0.85%
Apache (`ApacheBench`)	$\leq$0.01%	$\leq$0.01%

Impact on real-life applications: $\leq 1\%$

Micro benchmarks (1/3)

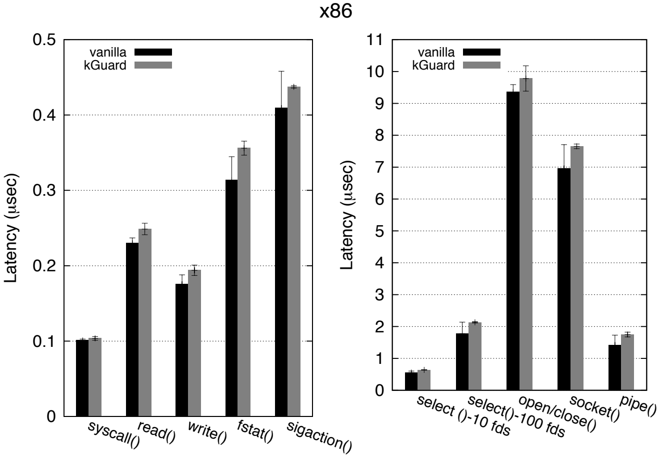
Bandwidth (lmbench)



Arch.	Slowdown
x86	3.2% – 10% (avg. 6%)
x86-64	5.25% – 9.27% (avg. 6.6%)

Micro benchmarks (2/3)

Latency x86 (lmbench)

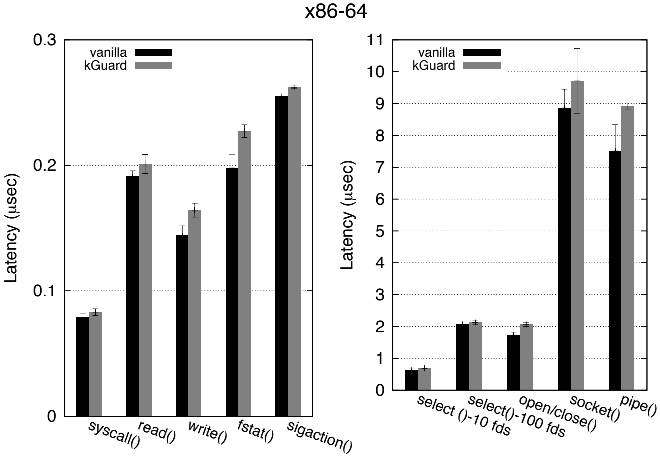


Overhead

2.7% – 23.5% (avg. 11.4%)

Micro benchmarks (3/3)

Latency x86-64 (`lmbench`)



Overhead

2.9% – 19.1% (avg. 10.3%)

Translation overhead

Arch.	Build time (inc.)	Size (inc.)
x86	0.3%	3.5%/0.43%
x86-64	0.05%	5.6%/0.56%

- Additional time needed for CFA-based confinement & Code inflation
- Kernel image/modules size increase

Conclusion

kGuard

- Versatile & lightweight mechanism against ret2usr attacks
- Builds upon inline monitoring and code diversification
- Cross-platform solution
  - x86, x86-64, ARM, ...
  - Linux, Android, {Free, Net, Open}BSD, ...
- Non-intrusive & low overhead
  - 11.4%/10.3% on syscall & I/O latency on x86/x86-64
  - $\sim$ 6% on IPC bandwidth
  - 3.5% – 5.6% size overhead
  - $\leq$ 1% on real-life applications

Try it!

http://www.cs.columbia.edu/~vpk/research/kguard/

Bonus Slides

Current defences

Issues & limitations (1/2)

Restricting `mmap`

- Restricts the ability to map the first pages of the address space (typically 4KB – 64KB)
- Mitigation strategy → Protection scheme against NULL ptr. exploits
- Does **not** protect against exploits that redirect control to memory pages above the forbidden region
- Breaks compatibility with applications that rely on having access to low logical addresses
- Circumvented repeatedly

Current defences (cont'd)

Issues & limitations (2/2)

PaX `UDEREF/KERNEXEC`

- Patch for hardening the Linux kernel against user space pointer dereferences & code execution
- In x86 it relies on memory segmentation
- In x86-64, where segmentation is not available, it resorts in user space remapping (temporarily)
- Requires patching, works only on x86/x86-64, incurs non-negligible performance overhead

Macro benchmarks

- Kernel build (`time(1)`)

Arch.	PaX (inc.)	kGuard (inc.)
x86	1.26%	1.03%
x86-64	2.89%	0.93%

- MySQL (`sql-bench`)

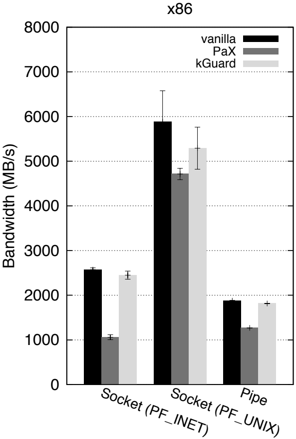
Arch.	PaX (inc.)	kGuard (inc.)
x86	1.16%	0.92%
x86-64	2.67%	0.85%

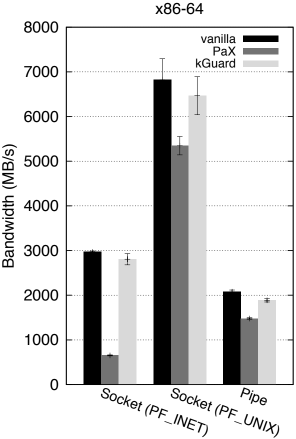
- Apache (`ApacheBench`)
  - PaX: 0.01% – 1.27%
  - kGuard: $\leq$ 0.01%

Micro benchmarks (1/3)

Bandwidth (lmbench)

Arch.	PaX	kGuard
x86	19.9% – 58.8% (avg. 37%)	3.2% – 10% (avg. 6%)
x86-64	21.7% – 78% (avg. 42.8%)	5.25% – 9.27% (avg. 6.6%)

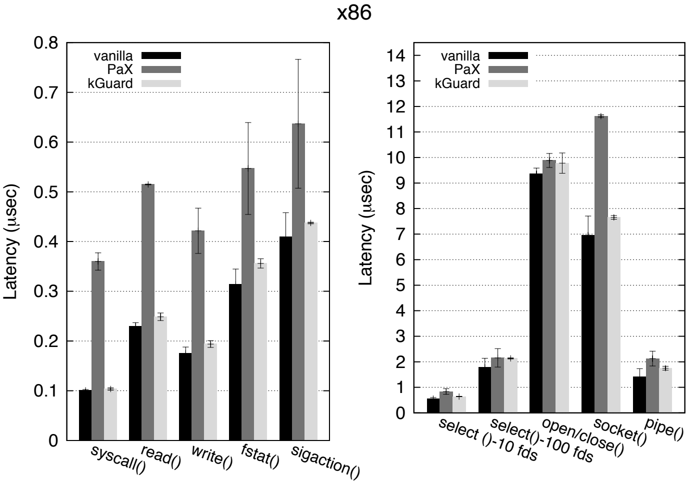




Micro benchmarks (2/3)

Latency x86 (lmbench)

PaX	**kGuard**
5.6% – 257% (avg. 84.5%)	2.7% – 23.5% (avg. 11.4%)



Micro benchmarks (3/3)

Latency x86-64 (`lmbench`)

PaX	**kGuard**
19% – 531% (avg. 172.2%)	2.9% – 19.1% (avg. 10.3%)