Linux Kernel Self Protection Project

Linux Security Summit, Los Angeles
September 14, 2017

Kees ("Case") Cook
keescook@chromium.org

https://outflux.net/slides/2017/lss/kspp.pdf

Agenda

- Background
  - “Security” in the context of this presentation
  - Why we need to change what we’re doing
  - Just fixing bugs isn’t sufficient
  - Upstream development model
- Kernel Self Protection Project
  - Who we are
  - What we’re doing
  - How you can help
- Challenges

I. Lietuvių Susivažiavimas Vilniuje.

Lapkričio 21—22 (Gruodžio 4—5) d. 1905.

PROGRAMA:

1. Lietuvystės praeitis ir dabartis.
2. Lietuviai, latviai ir įvairios tautos Lietuvoje gyvenančios.
3. Caro manifestas nuo Spalinio 17 (30) dienos.
4. Lietuvos autonomija ir prikergimas Suvalkų gubernijos prie autonomiškosios Lietuvos.
5. Rinkimai į Viešpatystės seimą.
6. Lietuvos sodiečiai po rusų valdžia.
7. Lietuvos mokyklos.
8. Apie įvairius mokesčius.
9. Apie žemiečių įstaigas.
10. Apie išeivystę (emigraciją).
11. Apie kareiviavimą.

Gali būti įnešami ir kitoki klausimai.

Organizacijos Komitetas.

Kernel Security

- More than access control (e.g. SELinux)
- More than attack surface reduction (e.g. seccomp)
- More than bug fixing (e.g. CVEs)
- More than protecting userspace
- More than kernel integrity
- This is about *Kernel Self Protection*

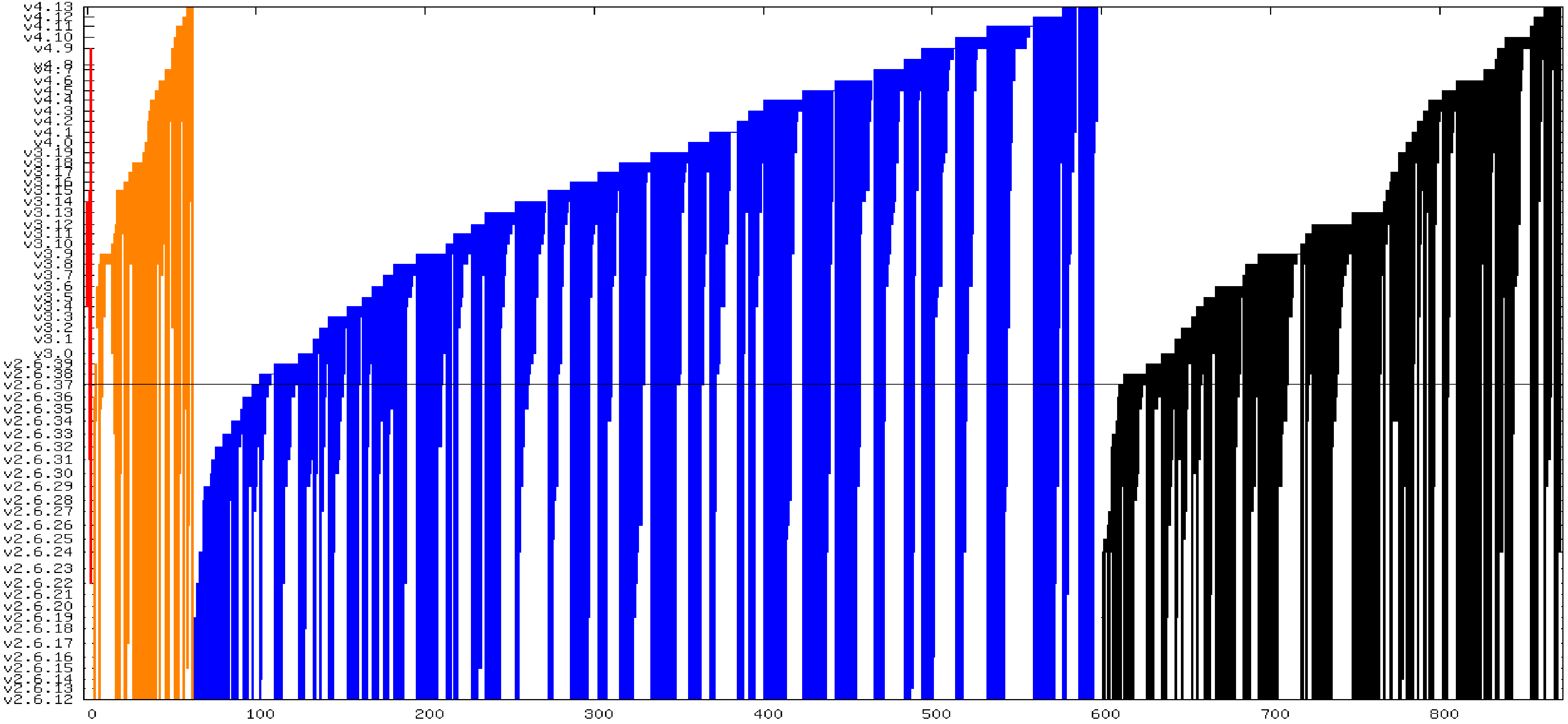
Devices using Linux

- Servers, laptops, cars, phones, …
- >2,000,000,000 active Android devices in 2017
- Vast majority are running v3.4 (with v3.10 slowly catching up)
- Bug lifetimes are even longer than upstream
- “Not our problem”? None of this matters: even if upstream fixes every bug found, and the fixes are magically sent to devices, bug lifetimes are still huge.

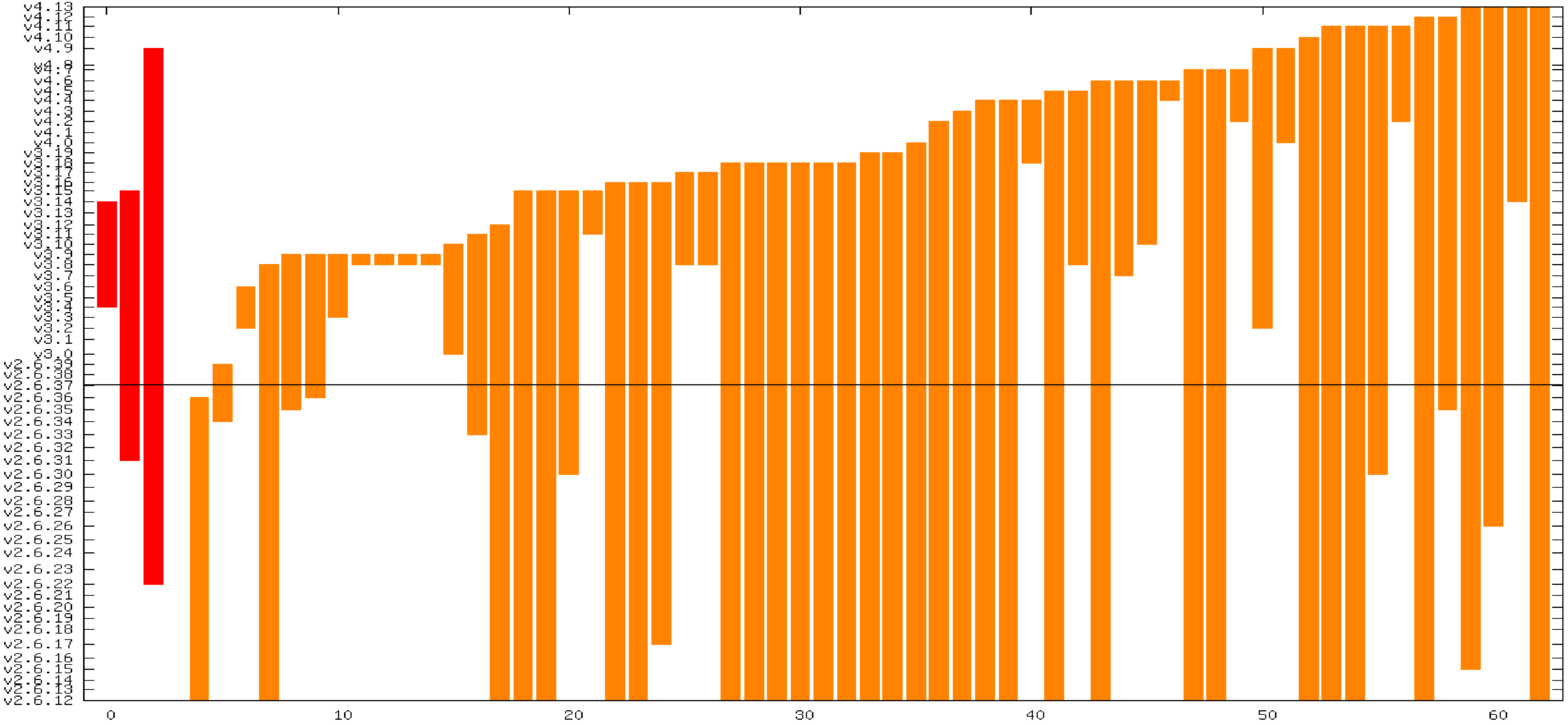
Upstream Bug Lifetime

- In 2010 Jon Corbet researched security flaws, and found that the average time between introduction and fix was about 5 years.
- My analysis of Ubuntu CVE tracker for the kernel from 2011 through 2017:
  - Critical: 3 @ 5.3 years
  - High: 59 @ 6.4 years
  - Medium: 534 @ 5.6 years
  - Low: 273 @ 5.6 years

CVE lifetimes



critical & high CVE lifetimes



Upstream Bug Lifetime

- The risk is not theoretical. Attackers are watching commits, and they are better at finding bugs than we are:
  - http://seclists.org/fulldisclosure/2010/Sep/268
- Most attackers are not publicly boasting about when they found their 0-day...

Fighting Bugs

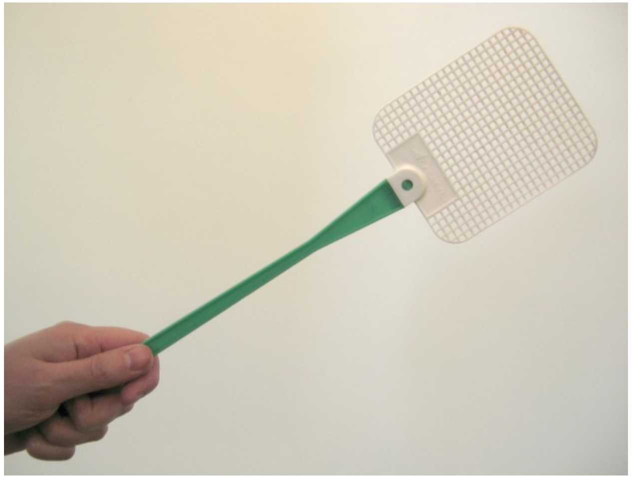
- We’re finding them
  - Static checkers: compilers, coccinelle, sparse, smatch, coverity
  - Dynamic checkers: kernel, trinity, syzkaller, KASan-family
- We’re fixing them
  - Ask Greg KH how many patches land in -stable
- They’ll always be around
  - We keep writing them
  - They exist whether we’re aware of them or not
  - Whack-a-mole is not a solution

Analogy: 1960s Car Industry

- @mricon's presentation at 2015 Linux Security Summit
  - http://kernsec.org/files/lss2015/giant-bags-of-mostly-water.pdf
- Cars were designed to run, not to fail
- Linux now where the car industry was in 1960s
  - https://www.youtube.com/watch?v=fPF4fBGNK0U
- We must handle failures (attacks) safely
  - Userspace is becoming difficult to attack
  - Containers paint a target on kernel
  - Lives depend on Linux

Killing bugs is nice

- Some truth to security bugs being “just normal bugs”
- Your security bug may not be my security bug
- We have little idea which bugs attackers use
- Bug might be in out-of-tree code
  - Un-upstreamed vendor drivers
  - Not an excuse to claim “not our problem”



Killing bug classes is better

- If we can stop an entire kind of bug from happening, we absolutely should do so!
- Those bugs never happen again
- Not even out-of-tree code can hit them
- But we'll never kill all bug classes

Killing exploitation is best

- We will always have bugs
- We must stop their exploitation
- Eliminate exploitation targets and methods
- Eliminate information leaks
- Eliminate anything that assists attackers
- *Even if it makes development more difficult*

Typical Exploit Chains

- Modern attacks tend to use more than one flaw
- Need to know where targets are
- Need to inject (or build) malicious code
- Need to locate malicious code
- Need to redirect execution to malicious code

What can we do?

- Many exploit mitigation technologies already exist (e.g. grsecurity/PaX) or have been researched (e.g. academic whitepapers), but many not present in upstream Linux kernel
- There is demand for kernel self-protection, and there is demand for it to exist in the upstream kernel
- http://www.washingtonpost.com/sf/business/2015/11/05/net-of-insecurity-the-kernel-of-the-argument/

The Washington Post

Out-of-tree defenses?

- Some downstream kernel forks:
  - RedHat (ExecShield), Ubuntu (AppArmor), Android (Samsung KNOX), grsecurity (so many things)
- If you only *use* the kernel, and don't *develop* it, you're in a better position
  - But you're depending on a downstream fork
  - Fewer eyeballs (and less automated testing infrastructure) looking for vulnerabilities
  - Developing the kernel means using engineering resources for your fork
    - e.g. Android deals with multiple vendor forks already
    - Hard to integrate multiple forks
- Upstreaming means:
  - No more forward-porting
  - More review (never perfect, of course)

Digression 1: defending against email Spam

- Normal email server communication establishment:

Client	Server
[connect]	
	[accept]220 smtp.some.domain ESMTP ok
EHLO my.domain	
	250 ohai
MAIL FROM:<me@my.domain>	
	250 OK
RCPT TO:<you@your.domain>	
	250 OK
DATA	

Spam bot communication

- Success, and therefore timing, isn't important to Spam bots:

Client	Server
[connect]	
	[accept]220 smtp.some.domain ESMTP ok
EHLO my.domain	
MAIL FROM:<me@my.domain>	
RCPT TO:<you@your.domain>	
DATA	
	250 ohai
	250 OK
	250 OK

Trivially blocking Spam bots

- Insert a short starting delay

Client	Server
[connect]	
	[accept]
EHLO my.domain	
MAIL FROM:<me@my.domain>	
RCPT TO:<you@your.domain>	
DATA	
	554 smtp.some.domain ESMTP nope

Powerful because it's not the default

- If everyone did this (i.e. it was upstream), bots would adapt
- If a defense is unexamined and/or only run by a subset of Linux users, it may be accidentally effective due to it being different, but may fail under closer examination
- Though, on the flip side, heterogeneous environments tend to be more resilient

Digression 2: Stack Clash research in 2017

- Underlying issues were identified in 2010
  - Fundamentally, if an attacker can control the memory layout of a setuid process, they may be able to manipulate it into colliding stack with other things, and arranging related overflows to gain execution control.
  - Linux tried to fix it with a 4K gap
  - grsecurity (from 2010 through at least their last public patch) took it further with a configurable gap, defaulting to 64K

A gap was not enough

- In addition to raising the gap size, grsecurity sensibly capped stack size of setuid processes, just in case:

```
do_execveat_common(...) {
    ...
    /* limit suid stack to 8MB
     * we saved the old limits above and will restore them if this exec fails */
    if (((!uid_eq(bprm->cred->euid, current_euid())) ||
         (!gid_eq(bprm->cred->egid, current_egid()))) &&
           (old_rlim[RLIMIT_STACK].rlim_cur > (8 * 1024 * 1024)))
               current->signal->rlim[RLIMIT_STACK].rlim_cur = 8 * 1024 * 1024;
    ...
```

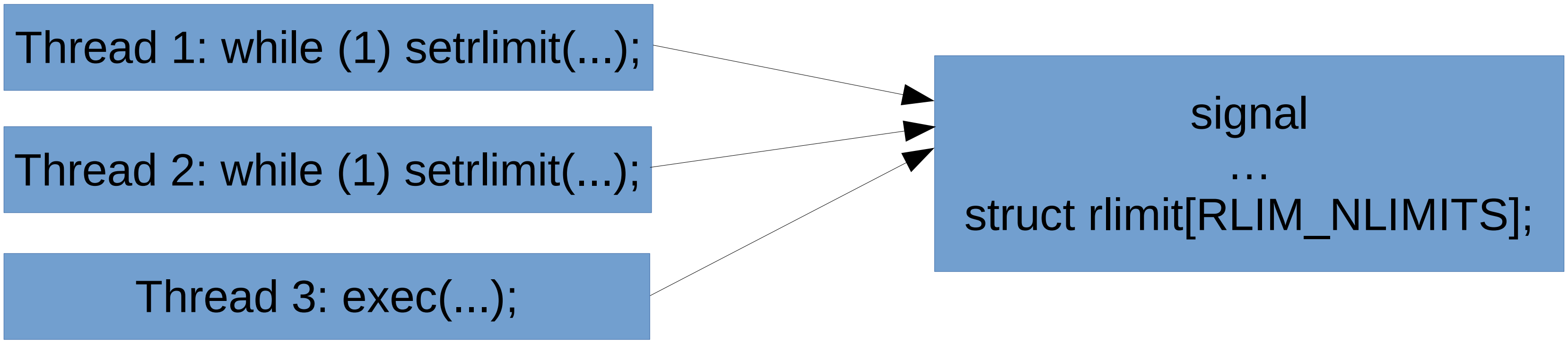
Upstreaming the setuid stack size limit

- Landed in v4.14-rc1
- 15 patches
- Reviewed by at least 7 other people
- Made the kernel smaller
- Actually keeps the stack limited for setuid exec

```
16 files changed, 91 insertions(+), 159 deletions(-)
```

Important detail: threads

- Stack rlimit is a single value shared across entire thread-group
- Exec kills all other threads (part of the “point of no return”) as late in exec as possible
- If you check or set rlimits before the point of no return, you're racing other threads



Un-upstreamed and unexamined for seven years

```
$ uname -r
4.9.24-grsec+
$ ulimit -s
unlimited
$ ls -la setuid-stack
-rwsrwxr-x 1 root root 9112 Aug 11 09:17 setuid-stack
$ ./setuid-stack
Stack limit: 8388608
$ ./raise-stack ./setuid-stack
Stack limit: 18446744073709551615
```

Out-of-tree defenses need to be upstreamed

- While the preceding example isn't universally true for all out-of-tree defenses, it's a good example of why upstreaming is important, and why sometimes what looks like a tiny change turns into **much** more work.

- How do we get this done?

Kernel Self Protection Project

- http://www.openwall.com/lists/kernel-hardening/
  - http://www.openwall.com/lists/kernel-hardening/2015/11/05/1
- http://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project
- People interested in coding, testing, documenting, and discussing the upstreaming of kernel self protection technologies and related topics.

Kernel Self Protection Project

- There are other people working on excellent technologies that ultimately revolve around the kernel protecting *userspace* from attack (e.g. brute force detection, SROP mitigations, etc)
- KSPP focuses on the kernel protecting the *kernel* from attack
- Currently ~12 organizations and ~10 individuals working on about ~20 technologies
- Slow and steady

Developers under KSPP umbrella

- LF’s Core Infrastructure Initiative funded: Emese Revfy, with others pending
- Self-funded: Andy Lutomirski, Russell King, Valdis Kletnieks, Jason Cooper, Daniel Micay, David Windsor, Richard Weinberger, Richard Fellner, Daniel Gruss, Jason A. Donenfeld, Sandy Harris, Alexander Popov
- ARM: Catalin Marinas, Mark Rutland
- Canonical: Juerg Haefliger
- Cisco: Daniel Borkmann
- Docker: Tycho Andersen
- Google: Kees Cook, Thomas Garnier, Daniel Cashman, Jeff Vander Stoep, Jann Horn, Eric Biggers
- Huawei: Li Kun
- IBM: Michael Ellerman, Heiko Carstens, Christian Borntraeger
- Imagination Technologies: Matt Redfearn
- Intel: Elena Reshetova, Hans Liljestrand, Casey Schaufler, Michael Leibowitz, Dave Hansen, Peter Zijlstra
- Linaro: Ard Biesheuvel, David Brown, Arnd Bergmann
- Linux Foundation: Greg Kroah-Hartman
- Oracle: James Morris, Quentin Casasnovas, Yinghai Lu
- RedHat: Laura Abbott, Rik van Riel, Jessica Yu, Baoquan He

Probabilistic protections

- Protections that derive their strength from some system state being unknown to an attacker
- Weaker than “deterministic” protections since information exposures can defeat them, though they still have real-world value
- Familiar examples:
  - stack protector (canary value can be exposed)
  - Address Space Layout Randomization (offset can be exposed)

Deterministic protections

- Protections that derive their strength from organizational system state that always blocks attackers
- Familiar examples:
  - Read-only memory (writes will fail)
  - Bounds-checking (large accesses fail)

Bug class: stack overflow and exhaustion

- Exploit example:
  - https://jon.oberheide.org/files/half-nelson.c
- Mitigations:
  - **stack canaries, e.g. gcc's -fstack-protector (v2.6.30) and -fstack-protector-strong (v3.14)**
  - guard pages (e.g. GRKERNSEC_KSTACKOVERFLOW)
    - **vmap stack (v4.9 x86, v4.14 arm64), removal of thread_info from stack (v4.9 x86, v4.10 arm64)**
  - *alloca checking (e.g. PAX_MEMORY_STACKLEAK): Alexander Popov*
  - shadow stacks (e.g. Clang SafeStack)

Bug class: integer over/underflow

- Exploit examples:
  - https://cyseclabs.com/page?n=02012016
  - http://perception-point.io/2016/01/14/analysis-and-exploitation-of-a-linux-kernel-vulnerability-cve-2016-0728/
- Mitigations:
  - check for atomic overflow (e.g. PAX_REFCOUNT)
    - *refcount_t: Elena Reshetova, David Windsor, Kees Cook, Ard Biesheuvel, Li Kun*
  - compiler plugin to detect multiplication overflows at runtime (e.g. PAX_SIZE_OVERFLOW)

Bug class: buffer overflows

- Exploit example:
  - http://blog.includesecurity.com/2014/06/exploit-walkthrough-cve-2014-0196-pty-kernel-race-condition.html
- Mitigations:
  - runtime validation of variable size vs copy_to_user / copy_from_user size (e.g. PAX_USERCOPY)
    - **CONFIG_HARDENED_USERCOPY (v4.8)**
    - *Usercopy whitelisting: David Windsor, Kees Cook*
    - *Usercopy slab segregation: David Windsor, Kees Cook*
  - metadata validation (e.g. glibc's heap protections)
    - **linked-list hardening (from grsecurity) CONFIG_DEBUG_LIST (v4.10)**
    - *CONFIG_SLUB_HARDENED, heap freelist obfuscation (from grsecurity): Daniel Micay, Kees Cook*
    - *Heap canaries: Daniel Micay*
  - FORTIFY_SOURCE (inspired by glibc), check buffer sizes of str*/mem* functions at compile- and run-time
    - **CONFIG_FORTIFY_SOURCE (v4.13)**
    - *Intra-object checking: Daniel Micay*

Bug class: format string injection

- Exploit example:
  - http://www.openwall.com/lists/oss-security/2013/06/06/13
- Mitigations:
  - **Drop %n entirely (v3.13)**
  - detect non-const format strings at compile time (e.g. gcc's -Wformat-security, or better plugin)
  - detect non-const format strings at run time (e.g. memory location checking done with glibc's -D_FORITY_SOURCE=2)

Bug class: kernel pointer leak

- Exploit examples:
  - examples are legion: /proc (e.g. kallsyms, modules, slabinfo, iomem), /sys, **INET_DIAG (v4.1)**, etc
  - http://vulnfactory.org/exploits/alpha-omega.c
- Mitigations:
  - **kptr_restrict sysctl (v2.6.38)** too weak: requires dev opt-in
  - remove visibility to kernel symbols (e.g. GRKERNSEC_HIDESYM)
  - detect and block usage of %p or similar writes to seq_file or other user buffers (e.g. GRKERNSEC_HIDESYM + *PAX_USERCOPY*)

Bug class: uninitialized variables

- This is not just an information leak!
- Exploit example:
  - https://outflux.net/slides/2011/defcon/kernel-exploitation.pdf
- Mitigations:
  - *GCC plugin, stackleak: clear kernel stack between system calls (from PAX_MEMORY_STACKLEAK): Alexander Popov*
  - **GCC plugin, structleak: instrument compiler to fully initialize all structures (from PAX_MEMORY_STRUCTLEAK): (__user v4.11, by-reference v4.14)**

Bug class: use-after-free

- Exploit example:
  - http://perception-point.io/2016/01/14/analysis-and-exploitation-of-a-linux-kernel-vulnerability-cve-2016-0728/
- Mitigations:
  - clearing memory on free can stop attacks where there is no reallocation control (e.g. PAX_MEMORY_SANITIZE)
    - **Zero poisoning (v4.6)**
  - segregating memory used by the kernel and by userspace can stop attacks where this boundary is crossed (e.g. *PAX_USERCOPY*)
  - randomizing heap allocations can frustrate the reallocation efforts the attack needs to perform (e.g. OpenBSD malloc)
    - **Freelist randomization (SLAB: v4.7, SLUB: v4.8)**

Exploitation: finding the kernel

- Exploit examples (see “Kernel pointer leaks” above too):
  - https://github.com/jonoberheide/ksymhunter
- Mitigations:
  - hide symbols and kernel pointers (see “Kernel pointer leaks”)
  - kernel ASLR
    - text/modules base: **x86 (v3.14), arm64 (v4.6), MIPS (v4.7),** *ARM: Ard Biesheuvel*
    - memory: **x86 (v4.8)**
    - PIE: **arm64 (v4.6)**, *x86: Thomas Garnier*
  - runtime randomization of kernel functions
  - executable-but-not-readable memory
    - **x86 (v4.6), arm64 (v4.9)**
  - per-build structure layout randomization (e.g. GRKERNSEC_RANDSTRUCT)
    - **manual (v4.13), automatic (v4.14)**

Exploitation: direct kernel overwrite

- How is this still a problem in the 21st century?
- Exploit examples:
  - Patch setuid to always succeed
  - http://itszn.com/blog/?p=21 Overwrite vDSO
- Mitigations:
  - Executable memory cannot be writable (CONFIG_STRICT_KERNEL_RWX)
    - **s390: forever ago**
    - **x86: v3.18**
    - **ARM: v3.19**
    - **arm64: v4.0**

Exploitation: function pointer overwrite

- Also includes things like vector tables, descriptor tables (which can also be info leaks)
- Exploit examples:
  - https://outflux.net/blog/archives/2010/10/19/cve-2010-2963-v4l-compat-exploit/
  - https://blogs.oracle.com/ksplice/entry/anatomy_of_an_exploit_cve
- Mitigations:
  - read-only function tables (e.g. PAX_CONSTIFY_PLUGIN)
  - make sensitive targets that need one-time or occasional updates only writable during updates (e.g. PAX_KERNEXEC):
    - **__ro_after_init (v4.6)**

Exploitation: userspace execution

- Exploit example:
  - See almost all previous examples
- Mitigations:
  - hardware segmentation: **SMEP (x86), PXN (ARM, arm64)**
  - emulated memory segmentation via page table swap, PCID, etc (e.g. PAX_MEMORY_UDEREF):
    - **Domains (ARM: v4.3)**
    - **TTBR0 (arm64: v4.10)**
    - *PCID (x86): Andy Lutomirski*
  - compiler instrumentation to set high bit on function calls

Exploitation: userspace data

- Exploit examples:
  - https://github.com/geekben/towelroot/blob/master/towelroot.c
  - http://labs.bromium.com/2015/02/02/exploiting-badiret-vulnerability-cve-2014-9322-linux-kernel-privilege-escalation/
- Mitigations:
  - hardware segmentation: **SMAP (x86), PAN (ARM, arm64)**
  - emulated memory segmentation via page table swap, PCID, etc (e.g. PAX_MEMORY_UDEREF):
    - **Domains (ARM: v4.3)**
    - **TTBR0 (arm64: v4.10)**
    - *PCID (x86): Andy Lutomirski*
  - *eXclusive Page Frame Ownership: Tycho Andersen, Juerg Haefliger*

Exploitation: reused code chunks

- Also known as Return Oriented Programming (ROP), Jump Oriented Programming (JOP), etc
- Exploit example:
  - http://vulnfactory.org/research/h2hc-remote.pdf
- Mitigations:
  - JIT obfuscation (e.g. BPF_HARDEN):
    - **eBPF JIT hardening (v4.7)**
  - compiler instrumentation for Control Flow Integrity (CFI):
    - Clang CFI https://clang.llvm.org/docs/ControlFlowIntegrity.html
    - kCFI https://github.com/kcfi/docs
    - GCC plugin: Return Address Protection, Indirect Control Transfer Protection (e.g. RAP) https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf

Added in v4.10

- PAN emulation, arm64
- thread_info relocated off stack, arm64
- Linked list hardening
- RNG seeding from UEFI, arm64
- W^X detection, arm64

Added in v4.11

- refcount_t infrastructure
- read-only usermodehelper
- structleak plugin

Added in v4.12

- read-only and fixed-location GDT, x86
- usercopy consolidation
- read-only LSM structures
- KASLR enabled by default, x86
- stack canary expanded to bit-width of host
- stack/heap gap expanded

Added in v4.13

- CONFIG_REFCOUNT_FULL
- CONFIG_FORTIFY_SOURCE
- randstruct (manual mode)
- ELF_ET_DYN_BASE lowered

Challenge: Culture

- Conservatism
  - 16 years to accept symlink restrictions upstream
- Responsibility
  - Kernel developers must accept the need for these changes
- Sacrifice
  - Kernel developers must accept the technical burden
- Patience
  - Out-of-tree developers must understand how kernel is developed

Challenge: Technical

- Complexity
  - Very few people are proficient at developing (much less debugging) these features
- Innovation
  - We must adapt the many existing solutions
  - We must create new technologies
- Collaboration
  - Explain rationale for new technologies
  - Make code understandable/maintainable by other developers and accessible across architectures

Challenge: Resources

- People
  - Dedicated developers
- People
  - Dedicated testers
- People
  - Dedicated backporters

Thoughts?

Kees ("Case") Cook
keescook@chromium.org
keescook@google.com
kees@outflux.net

https://outflux.net/slides/2017/lss/kspp.pdf

http://www.openwall.com/lists/kernel-hardening/
http://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project