Linux Kernel Self Protection Project

UC Santa Cruz, Open Source Programming
Feb 16, 2016

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

https://outflux.net/slides/2017/uc/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
- 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*

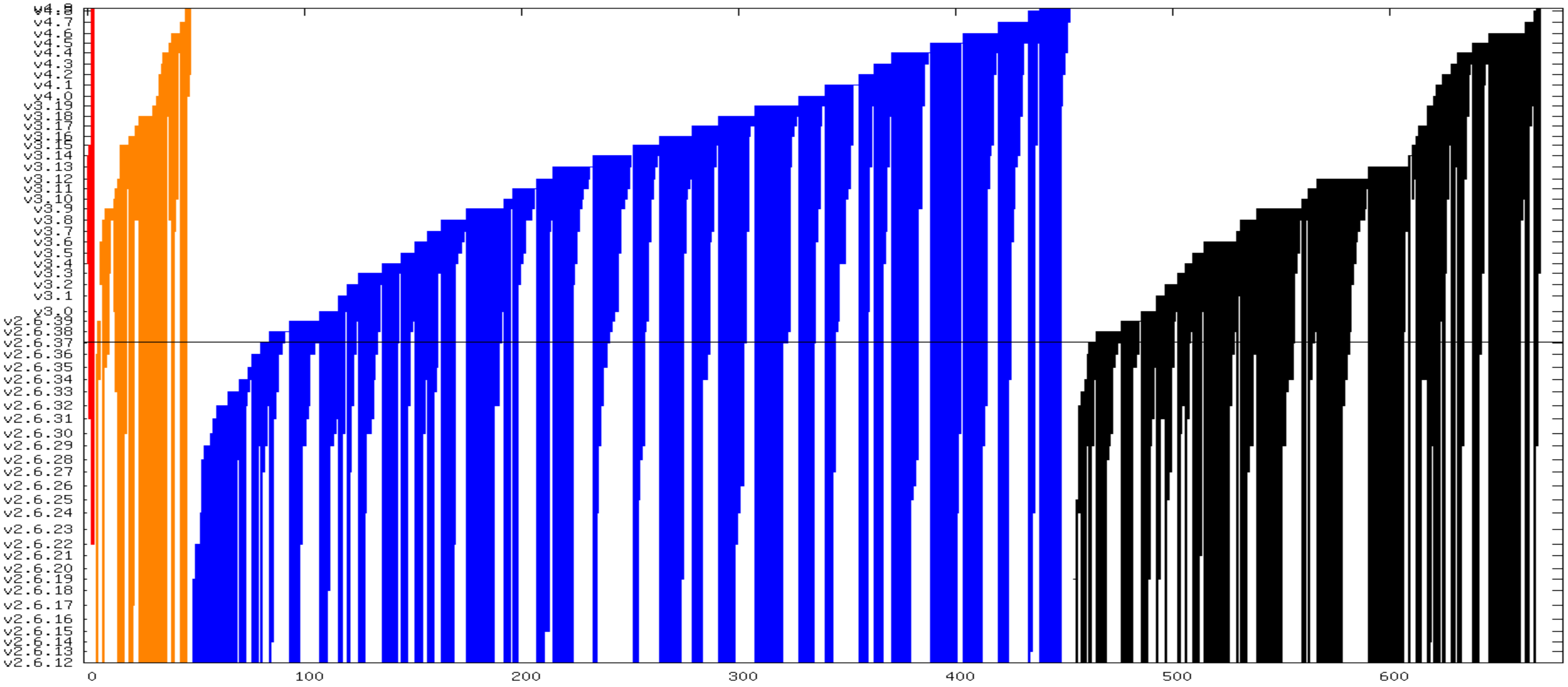
Devices using Linux

- Servers, laptops, cars, phones, …
- >1,400,000,000 active Android devices in 2015
- Vast majority are running v3.4 (with v3.10 a distant second)
- 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.

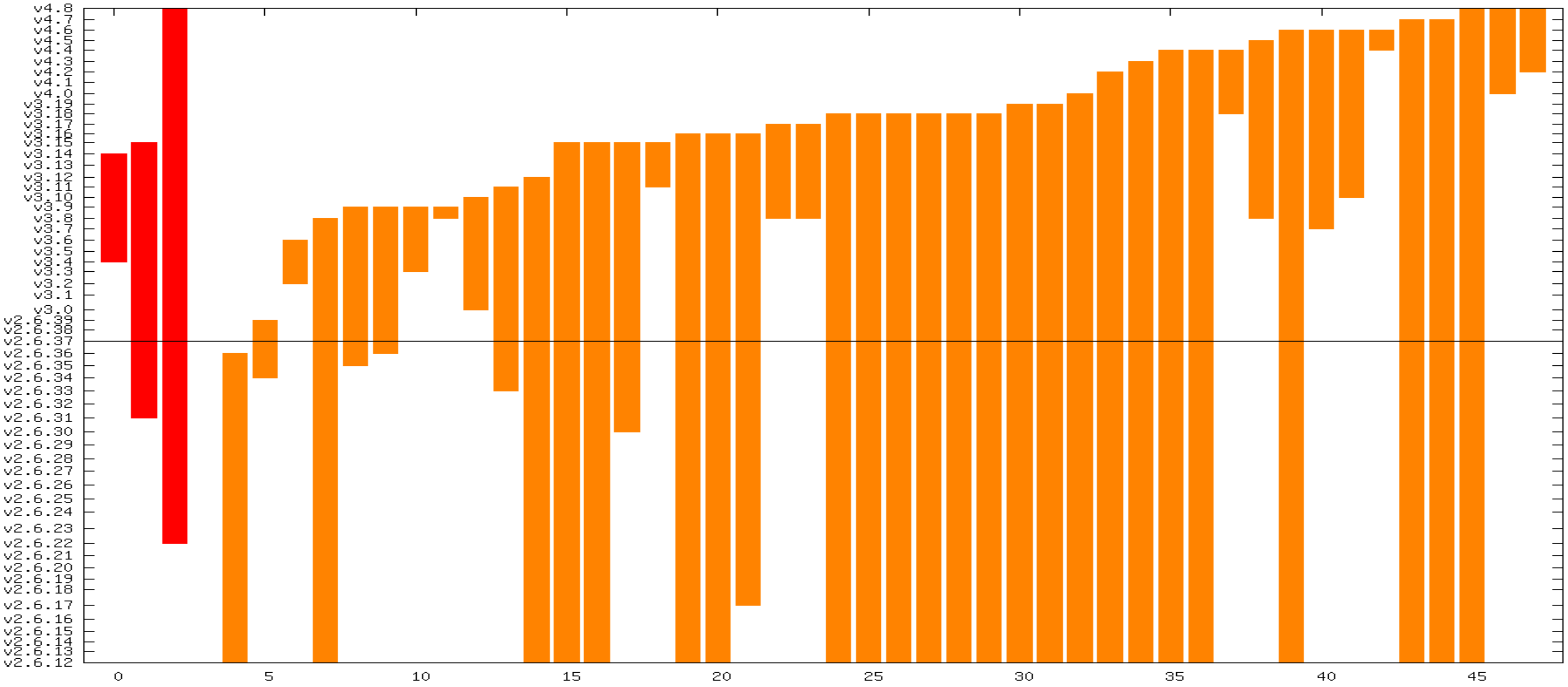
Upstream Bug Lifetime

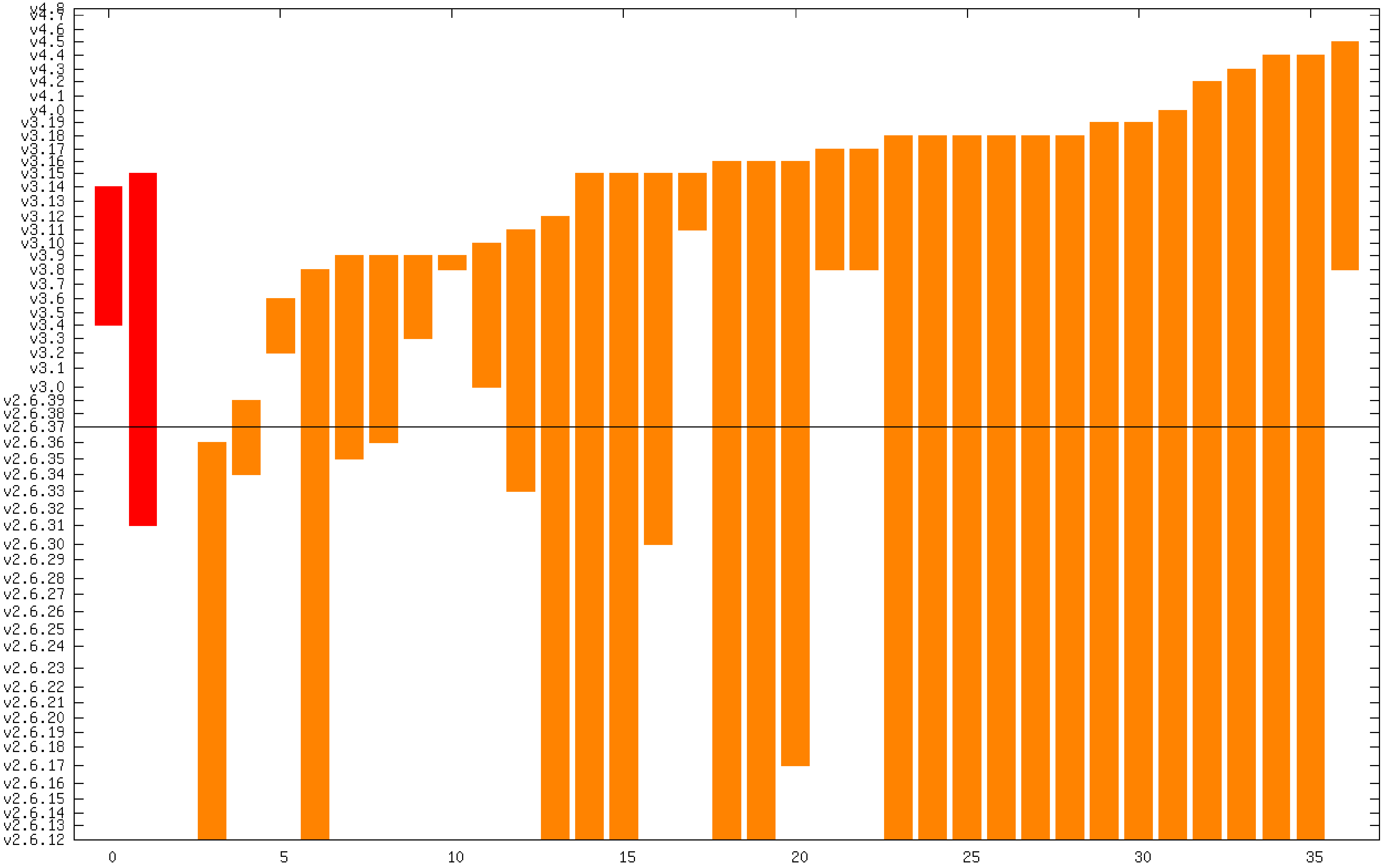
- In 2010 Jon Corbet researched security flaws, and found that the average time between introduction and fix was about 5 years.
- My analysis of Ubuntu CVE tracker for the kernel from 2011 through 2016:
  - Critical: 2 @ 3.3 years
  - High: 34 @ 6.4 years
  - Medium: 334 @ 5.2 years
  - Low: 186 @ 5.0 years

CVE lifetimes



critical & high CVE lifetimes





v4.8
v4.7
v4.6
v4.5
v4.4
v4.3
v4.2
v4.1
v4.0
v3.19
v3.18
v3.17
v3.16
v3.15
v3.14
v3.13
v3.12
v3.11
v3.10
v3.9
v3.8
v3.7
v3.6
v3.5
v3.4
v3.3
v3.2
v3.1
v3.0
v2.6.39
v2.6.38
v2.6.37
v2.6.36
v2.6.35
v2.6.34
v2.6.33
v2.6.32
v2.6.31
v2.6.30
v2.6.29
v2.6.28
v2.6.27
v2.6.26
v2.6.25
v2.6.24
v2.6.23
v2.6.22
v2.6.21
v2.6.20
v2.6.19
v2.6.18
v2.6.17
v2.6.16
v2.6.15
v2.6.14
v2.6.13
v2.6.12
0
5
10
15
20
25
30
35

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

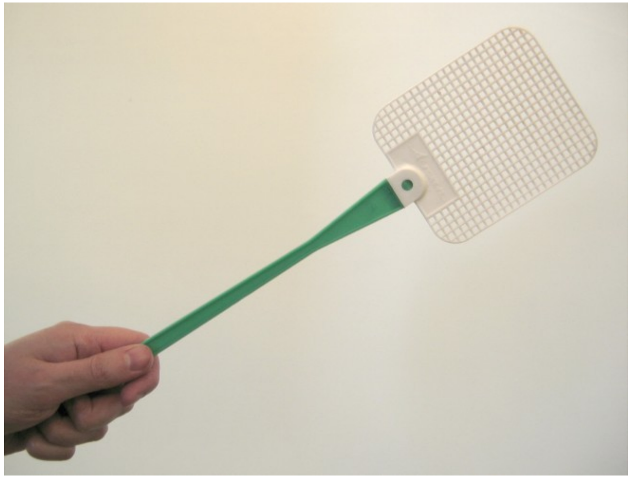
- We’re finding them
  - Static checkers: compilers, smatch, coccinelle, coverity
  - Dynamic checkers: kernel, trinity, KASan
- We’re fixing them
  - Ask Greg KH how many patches land in -stable
- They’ll always be around
  - We keep writing them
  - They exist whether we’re aware of them or not
  - 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 are not present in the 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

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 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 ~10 organizations and ~5 individuals working on about ~20 technologies
- Slow and steady

Developers under KSPP umbrella

- LF’s Core Infrastructure Initiative funded: Emese Revfy
- Self-funded: Andy Lutomirski, Russell King, Valdis Kletnieks, Jason Cooper, Jann Horn, Daniel Micay, David Windsor, Richard Weinberger
- ARM: Catalin Marinas, Mark Rutland
- Cisco: Daniel Borkmann
- Google: Kees Cook, Thomas Garnier, Daniel Cashman, Jeff Vander Stoep
- HP Enterprise: Juerg Haefliger
- IBM: Michael Ellerman, Heiko Carstens, Christian Borntraeger
- Imagination Technologies: Matt Redfearn
- Intel: Elena Reshetova, Casey Schaufler, Michael Leibowitz, Dave Hansen
- Linaro: Ard Biesheuvel, David Brown
- Oracle: 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 (cookie 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

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)
    - *vmalloc stack, removal of thread_info (x86)*: Andy Lutomirski
  - alloca checking (e.g. PAX_MEMORY_STACKLEAK)
  - kernel stack location randomization
  - shadow stacks

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 refcount overflow (e.g. PAX_REFCOUNT)
    - *PAX_REFCOUNT port*: David Windsor, Elena Reshetova
  - compiler plugin to detect multiplication overflows at runtime (e.g. PAX_SIZE_OVERFLOW)

Bug class: Heap overflow

- 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*: Kees Cook, Rik van Riel, Laura Abbott, Casey Schaufler
  - guard pages
  - metadata validation (e.g. glibc's heap protections)
    - *CONFIG_DEBUG_LIST hardening*: Kees Cook

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:
  - clear kernel stack between system calls (e.g. PAX_MEMORY_STACKLEAK)
  - instrument compiler to fully initialize all structures (e.g. PAX_MEMORY_STRUCTLEAK)

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)**: Laura Abbott
  - 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)**: Thomas Garnier

Exploitation: finding the kernel

- Exploit examples:
  - See “Kernel pointer leaks” above
  - 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)**: Ard Biesheuvel, **MIPS (v4.7)**: Matt Redfearn
    - *memory (x86)*: Thomas Garnier
  - runtime randomization of kernel functions
  - executable-but-not-readable memory
    - **x86 (v4.6)**: Dave Hansen, *arm64*: Catalin Marinas
  - per-build structure layout randomization (e.g. GRKERNSEC_RANDSTRUCT)
    - *RANDSTRUCT port*: Michael Leibowitz

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 should not be writable (e.g CONFIG_DEBUG_RODATA)
    - **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)*: Kees Cook, David Brown, Jessica Yu, Heiko Carstens

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)**: Russell King
    - *TTBR0 (ARMv8.0)*: Catalin Marinas
  - 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)**: Russell King
    - *TTBR0 (ARMv8.0)*: Catalin Marinas

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)**: Daniel Borkmann, Elena Reshetova
  - compiler instrumentation for Control Flow Integrity (CFI)
  - Return Address Protection, Indirect Control Transfer Protection (e.g. RAP)
    - https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf

Added in v4.3

- CONFIG_CPU_SW_DOMAIN_PAN (PAN Emulation on ARM)
- Ambient capabilities (notable userspace protection)
- Seccomp support on PowerPC (notable attack surface reduction)

Added in v4.4

- x86_64 vsyscall CONFIG option (notable attack surface reduction)

Added in v4.5

- ASLR entropy size sysctl (notable userspace protection)

Added in v4.6

- KASLR text and module base on arm64
- RODATA by default on ARMv7+, arm64
- RODATA mandatory on x86
- Zero-poisoning for heap memory
- __ro_after_init basic infrastructure
- execute-only memory on x86

Added in v4.7

- KASLR text/module base for MIPS
- SLAB freelist randomization
- eBPF JIT constant blinding

Added in v4.8

- SLUB freelist randomization
- KASLR of full range of physical memory on x86_64
- KASLR of kernel memory base on x86_64
- hardened usercopy
- gcc plugin infrastructure
- ptrace before seccomp (notable attack surface reduction)

Added in v4.9

- latent_entropy gcc plugin
- vmalloc stack on x86
- PAN emulation for arm64

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
kees@outflux.net

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

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