Improving Kernel Security

Kernel Summit 2015, Seoul

Kees (“Case”) Cook
keescook@chromium.org

James Morris
james.l.morris@oracle.com

https://outflux.net/slides/2015/ks/security.pdf

What I mean by “Security”

- More than access control (SELinux)
- More than attack surface reduction (seccomp)
- More than bug fixing (CVEs)
- Must develop “Kernel Self-Protection”

Background: devices using Linux

- Servers, laptops, cars, phones ...
- >1,000,000,000 active Android devices in 2014
- 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 we fix every bug we find, and they magically get fixed downstream, bug lifetimes are still huge

Upstream Bug Lifetime

- In 2010 Jon Corbet showed average security bug lifetime to be about 5 years from introduction to fix
- My analysis of Ubuntu CVE tracker covering 2011 through 2015:
  - critical: 2 @ 3.3 years
  - high: 31 @ 6.3 years
  - medium: 297 @ 4.9 years
  - low: 172 @ 5.1 years
- http://seclists.org/fulldisclosure/2010/Sep/268

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 you're aware of them or not
  - Whack-a-mole is not a solution

Analogy: 1960s Car Industry

- @mricon's presentation at the Linux Security Summit
- Cars were designed to run, not to fail
- Linux now where the car industry was in 1960s
- We must handle failures safely
  - Lives depend on Linux

Killing bugs is nice

- Some truth to security bugs being “just normals 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 for us to claim “not our problem”

Killing bug classes is better

- If we can stop an entire kind of bug from happening, we absolute 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 to malicious code

What do we do?

- What follows is not an exhaustive list, but I don't have much time...

Bug class: Stack overflow

- The traditional attack is saved return address overwrite, but there are data-only attacks possible too.
- exploit example:
  - https://jon.oberheide.org/files/half-nelson.c
- Mitigations:
  - stack canary (e.g. gcc's -fstack-protector(-strong))
  - kernel stack location randomization
  - shadow stacks

Bug Class: Integer over/underflow

- exploit example:
  - https://jon.oberheide.org/blog/2010/09/10/linux-kernel-can-slub-overflow/
- Mitigations:
  - instrument compiler to detect overflows at runtime

Bug class: Heap overflow

- exploit example:
  - http://blog.includesecurity.com/2014/06/exploit-walkthrough-cve-2014-0196-pty-kernel-race-condition.html
  - https://github.com/jonoberheide/kstructhunter
- Mitigations:
  - runtime validation of variables sizes vs copy_*_user
  - guard pages
  - linked list validation

Bug class: format string injection

- exploit example:
  - http://www.openwall.com/lists/oss-security/2013/06/06/13
- Mitigations:
  - drop %n entirely
- Still potentially an info leak

Bug class: kernel pointer leak

- exploit example:
  - examples are legion
  - http://vulnfactory.org/exploits/alpha-omega.c
  - /proc entries, INET_DIAG, slabinfo
- Mitigations:
  - kptr_restrict too weak
  - detect seq_file + %p and block output

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

Exploitation: finding the kernel

- exploit example:
  - so many ways, see “kernel pointer leak” above
  - /proc/kallsyms, /proc/modules
  - https://github.com/jonoberheide/ksymhunter
- Mitigations:
  - hide symbols and kernel pointers
  - kernel ASLR
  - runtime randomization of kernel functions
  - X^R memory
  - structure layout randomization
- Can this class of exploit ever be killed? Have to take it in pieces.

Exploitation: Direct text overwrite

- I shouldn't even have to mention this!
- exploit example:
  - patch setuid to always succeed
- Mitigations:
  - W^X kernel page table permission

Exploitation: Function ptr overwrite

- This includes things like vector tables, descriptor tables (which should also be hidden to avoid information leaks)
- exploit example:
  - 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:
  - constify function pointer tables
  - temporarily make targets writable

Exploitation: Userspace execution

- exploit example:
  - see almost all previous examples
- Mitigations:
  - hardware segmentation: SMEP, PXN
  - instrument compiler to set high bit on function calls
  - emulate memory segmentation via separate page tables

Exploitation: Userspace data

- exploit example:
  - 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, Domains
  - emulate memory segmentation via separate page tables

Exploitation: Reused code chunks

- Also known as Return Oriented Programming, Jump Oriented Programming, etc
- exploit example:
  - http://vulnfactory.org/research/h2hc-remote.pdf
- Mitigations:
  - compiler instrumentation for Control Flow Integrity
  - Return Address Protection, Indirect Control Transfer Protection

Challenge: Culture

- Conservatism
  - 16 years to get symlink protections in, and that was just a userspace defense
- Responsibility
  - We must accept the need for these features
- Sacrifice
  - We must accept the technical burden

Challenge: Technical

- Complexity
  - Very few people are proficient at developing (much less debugging) these features
- Innovation
  - We must adapt the many existing solutions
  - We can still innovate

Challenge: Resources

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

Thoughts?

Kees (“Case”) Cook
keescook@chromium.org

James Morris
james.l.morris@oracle.com

https://outflux.net/slides/2015/ks/security.pdf