PaX - kernel self-protection

PaX Team

H2HC 2012.10.09

Overview

- Host Intrusion Prevention System
- Focus: exploit techniques against memory corruption bugs
- Threat model: arbitrary read-write memory access
- Bugs vs. Exploits vs. Exploit techniques
- Performance vs. Usability
- 2000-2012, linux 2.2-3.6

Bugs

- Buffer overflows (stack/heap/static data)
- Heap object mismanagement (double free, use-after-free, etc)
- Integer overflows (underallocation, buffer overflow, reference counts, etc)
- see http://cwe.mitre.org/
- Known/unknown (0-day)
- None of this matters :)

Exploit Techniques

- Execute new (injected) code (shellcode)
- Execute existing code out-of-(intended)-order (return-to-libc, ROP/JOP)
- Execute existing code in-(intended)-order (data-only attacks)
- Increasing order of difficulty
- Decreasing amount of control

Introduction

Kernel

- Non-executable pages (KERNEXEC)
- Userland/kernel separation (UDEREF)
- Userland/kernel copying (USERCOPY/STACKLEAK/SANITIZE)
- Reference counter overflows (REFCOUNT)

Toolchain

Future

Overview

- Implements non-executable page behaviour (where there is no direct hardware support)
- Makes data pages non-executable (anything but kernel and module code)
- Makes read-only data actually read-only in the page tables
- Makes some important kernel data read-only (IDT, GDT, some page tables, CONSTIFY, __read_only, etc)
- i386: segmentation
- amd64: NX bit (except very early Intel P4 Xeon CPUs)

KERNEXEC/i386 Overview

- Idea: have `__KERNEL_CS` cover only kernel code
  - base: `__PAGE_OFFSET+__LOAD_PHYSICAL_ADDR`
  - limit: 4GB during init, `_etext` after `free_initmem`
- Excludes userland for free (unlike on amd64, but there is SMEP to the rescue :)
- Special problems: logical/linear translations, relocations, modules and .init code

KERNEXEC/i386 Problems

- Problem: kernel assumes logical address == linear address, no longer true for code (function pointers)
- Needs translation (`ktla_ktva` and `ktva_ktla`) for:
  - runtime patching and probing (alternatives, backtrace, ftrace, kprobes, lockdep, perf, etc)
  - module loading (relocation)
- Relocatable kernel does no longer need relocations for code

KERNEXEC/i386 Module handling

- Module code must be allocated within `__KERNEL_CS`
  - Module data is a separate allocation
  - Module read-only data is allocated with the code
- Preallocated area in vmlinux at compile/link time, size is configurable
  - Fragmentation can be a problem
  - Consumes (reserves) physical memory in the direct mapping even when no modules are loaded
- Module loader allocates code and data separately, special allocator for code under KERNEXEC/i386

KERNEXEC/i386 Initialization code handling

- Kernel initialization code is discarded at runtime
  - Still must be within `__KERNEL_CS` during init
  - Placed along with the other init code/data
- Its memory is freed/reused

KERNEXEC/amd64 Overview

- Idea: remove `rwx` mappings from the kernel virtual address range
- Establishes more control over kernel page tables (per-cpu pgd)
- kmaps: tool for auditing a page table hierarchy
  - More details than `CONFIG_X86_PTDUMP`
- Special problems: modules, vsyscall, BIOS/ACPI (ioremap)
- Does not prevent userland code execution per se
  - KERNEXEC gcc plugin
  - `CR4.SMEP`

KERNEXEC/amd64 Problems

- Module problem: one contiguous rwx allocation (before the days of `CONFIG_DEBUG_SET_MODULE_RONX`)
- Reuses KERNEXEC/i386 module loader logic (was already in generic kernel code anyway)
  - Module code/rodata vs. data
- vsyscall problem: page mapped twice (`rw-`, `r-x`), abused by an exploit for CVE-2009-0065
- ioremap problem: too easy access to physical memory
  - No access allowed above 1MB
  - No more sensitive data in the first 1MB

Overview

- Prevents unintended userland accesses from the kernel
- Intended accesses via explicit accessors only: `*copy*user*`, `*{get,put}_user`
  - NULL pointer dereferences
  - 'magic' poison value dereferences
  - AMD catalyst bug
- i386: segmentation
- amd64: paging
- Haswell and CR4.SMAP, see our blog

UDEREF/i386

- Expand down kernel data segment (`__KERNEL_DS`) prevents userland access
  - Intended accesses use segment override prefix (gs) with `__USER_DS`
  - Segment prefix collides with stack smashing protector (SSP)
- Default userland code/data segments prevent kernel access
  - TLS/LDT still have to be allowed, kernel doesn't use them
- kernel-to-kernel copying (`set_fs`)
  - Used to patch `__USER_DS` to change its limit
  - Nowadays switches gs between 0, `__USER_DS` or `__KERNEL_DS`

UDEREF/amd64

- Idea: unmap userland while executing in the kernel
  - Remaps it elsewhere (shadow) as non-executable (KERNEXEC)
  - Obvious performance impact, thanks to AMD for killing segmentation
  - Requires per-cpu top-level page directory (pgd) for SMP/multi-threaded apps
- Performance optimizations
  - Context switch copies a few pgd entries only, one cacheline's worth
  - No TLB flush on kernel->userland transitions

per-cpu pgd concept

- Idea: instead of a single per-process pgd have one per-cpu
  - Allows local (per-cpu) changes to the process memory map
- `swapper_pg_dir` (`init_level4_pgt` on amd64) is kept as master pgd for the kernel
- `cpu_pgd[NR_CPUS][PTRS_PER_PGD]` array
  - Invariant: `cr3` on cpuN must always point to `cpu_pgd[N]`
- Reduces number of userland pgd entries (256 vs. 8 on amd64), reduces ASLR (5 bits less)
  - 8 entries occupy one cache line
- Future optimization: per-process pgd can be reduced to the userland pgd entries only

per-cpu pgd management

- When allocating the per-process pgd:
  - pud tables (8 of them on amd64) are allocated as well
  - They are never freed while the process is alive
- The per-process pgd does not have the kernel pgd entries
  - Prevents its accidental use in `cr3`
- `switch_mm`: calls `__clone_user_pgds` and `__shadow_user_pgds`
  - clone: sets up the normal userland pgd entries in `cpu_pgd[N]`
  - shadow: sets up the shadow userland mapping in `cpu_pgd[N]`

Overview

- Bounds checking for copying from kernel memory to userland (info leak) or vice versa (buffer overflow)
- spender's idea: `ksize` can determine the object's size from the object's address
- Originally heap (slab) buffers only
- Limited stack buffer support (see Future section)
- Disables SLUB merging
- Data lifetime reduction: STACKLEAK and SANITIZE
- Process kernel stack clearing (STACKLEAK)
  - Enhanced with a gcc plugin
- Freed page clearing in the low level page allocator (SANITIZE)

USERCOPY

- Instruments `copy*user` functions to call `check_object_size` when size is not a compile-time constant
- `check_object_size` is implemented for SLAB/SLUB/SLOB
- Only slabs marked with the `SLAB_USERCOPY` flag are let through
  - `cifs_request`, `cifs_small_rq`, `jfs_ip`, `kvm_vcpu`, `names_cache`, `task_xstate`
  - All `kmalloc-*` slabs (for now)
  - Some kernel code is patched to reduce flag proliferation
- Limited stack buffer checking (`object_is_on_stack`)
  - Current function frame under `CONFIG_FRAME_POINTER`
  - Current kernel stack without `CONFIG_FRAME_POINTER`

STACKLEAK

- Idea: reduce lifetime of data on process kernel stacks by clearing the stack on kernel->user transitions
  - Per-arch hooks in the low-level kernel entry/exit code
  - Moved `thread_info` off the stack
- Initially blind memset on the entire kernel stack (8 kbytes)
  - Too slow (unused part of the stack is cache cold)
- Refinement: detect/clear only the used part of the stack
  - Looks for memset pattern from stack bottom to top
  - Optimization: check only a certain length (cache line)
- Needs to record stack depth in functions with a big stack frame
  - Manual inspection and patching
  - Instrumentation by a gcc plugin

STACKLEAK

- Special paths for ptrace/auditing
  - Low-level kernel entry/exit paths can diverge for ptrace/auditing and leave interesting information on the stack for the actual syscall code
- Problems: still considerable overhead, races, leaks from a single syscall still possible
- Solution: dual process kernel stack, one used only for copying to/from userland
  - Needs static analysis to find all local variables whose address is sunk into `copy*user`
  - New gcc plugin, LTO

SANITIZE

- Reduces potential info leaks from kernel memory to userland
- Freed memory is cleared immediately in `free_pages_prepare`
- Optimization: `prep_new_page` does not need to handle `__GFP_ZERO`
- Low-level page allocator, not slab layer
- Works on whole pages, not individual heap objects
  - Kernel stacks on task death
  - Anonymous userland mappings on munmap
- Anti-forensics vs. privacy

Overview

- Detects reference counter overflows
- Idea: detect signed overflow (in the middle of the counter space, `INT_MAX+1`)
- Linux refcounts are based on `atomic_t` and `atomic64_t`
- Per-arch assembly accessors, access to CPU flags
- False positives: not all variables of these types are refcounts (statistics, unique ids, bitflags)
  - Manual auditing, should be automated (gcc plugin)
- armv6+/sparc64/x86, soon powerpc

Introduction

Kernel

Toolchain
- GCC plugins
- Kernel stack information leak reduction (STACKLEAK)
- Read-only function pointers (CONSTIFY)
- KERNEXEC/amd64 helper plugin
- Integer (size) overflows (SIZE_OVERFLOW)
- Latent Entropy Extraction (LATENT_ENTROPY)

Future

Overview

- Idea: add special instrumentation during compilation to detect/prevent entire bug classes at runtime
- Loadable module system introduced in gcc 4.5
- Loaded early right after command line parsing
- No well defined API, all public symbols available for plugin use
- Typical (intended :) use: new IPA/GIMPLE/RTL passes
  - Plugins can sign up for events, insert/remove/replace passes
  - No (easy) access to language frontends

Introduction to gcc

- Compilation process is a pipeline, driven by the compiler driver
- Language frontend parses the source code and produces GENERIC/GIMPLE
  - Plugins can implement new attributes and pragmas, inspect structure declarations and variable definitions (gcc 4.6+)
  - Static Single Assignment (SSA) based representation
- First set of optimization/transformation passes runs on GIMPLE (`-fdump-ipa-all`, `-fdump-tree-all`)
  - Data structures: `gimple`, `tree`
- GIMPLE is lowered to RTL (pre-SSA gcc had only this)
- Second set of optimization passes runs on RTL (`-fdump-rtl-all`)
  - Data structures: `rtx`, `tree`

Overview

- First plugin :)
- Idea: insert function call to `pax_track_stack` if local frame size is over a specific limit
  - `pax_track_stack` records deepest used kernel stack pointer
- Problem: frame size info is available at the last RTL pass only, too late to insert complex code like a function call
- New strategy: instrument every function first and remove unneeded instrumentation later
  - Also finds all (potentially exploitable :) `alloca` calls

STACKLEAK

- GIMPLE pass: inserts call to `pax_track_stack` into every function prologue
  - unless alloca is in the first basic block
  - alloca is bracketed with a call to `pax_check_alloca` and `pax_track_stack`
- RTL pass: removes unneeded `pax_track_stack` calls
  - if the local frame size is below the limit
  - if alloca is not used

Overview

- Automatic constification of ops structures (200+ types in linux)
  - Structures with function pointer members only
  - Structures explicitly marked with a `do_const` attribute
- `no_const` attribute for special cases
  - Unfortunately many ops structures want to be written at runtime
- Local variables not allowed (compiler error generated)

CONSTIFY

- `PLUGIN_ATTRIBUTES` callback: registers `do_const` and `no_const` attributes
  - Linux code patched by hand
  - Could be automated (static analysis, LTO)
- `PLUGIN_FINISH_TYPE` callback: sets `TYPE_READONLY` and `C_TYPE_FIELDS_READONLY` on eligible structure types
  - Only function pointer members, recursively
  - `do_const` is set, `no_const` is not set
- End result is that the frontend will do the dirty job of enforcing C variable constness
- GIMPLE pass: constified types cannot be used for local variables (stack is writable :)

Overview

- Goal: prevent executing userland code on amd64
- Idea: set most significant bit in all function pointers
  - Userland addresses become non-canonical ones, GPF on any dereference
- GIMPLE pass: handles C function pointers (`execute_kernexec_fptr`)
- RTL pass: handles function return values (`execute_kernexec_retaddr`)

KERNEXEC/amd64 helper plugin

- Two methods: `bts` vs. `or` (reserves `%r10` for bitmask)
- Compatibility vs. performance
- Special cases: vsyscall, assembly source, asm()
  - `kernexec_cmodel_check` to exclude code in vsyscall sections
  - Manual verification/patching
  - GIMPLE pass to reload `r10` when clobbered by asm()

Overview

- Detects integer overflows in expressions used as a size parameter: `kmalloc(count * sizeof...)`
- Written by Emese Révfy, extends spender's old idea (preprocessor trick)
- Initial set of functions/parameters marked by the `size_overflow` function attribute
- Walks use-def chains and duplicates statements using a double-wide integer type
- SImode/DImode vs. DImode/TImode
- Special cases: asm(), function return values, constants (intentional overflows), memory references, type casts, etc
- More in our blog

SIZE_OVERFLOW

- PLUGIN_ATTRIBUTES callback: size_overflow attribute, takes arbitrary arguments (size parameter index)
  - Only a handful of functions are marked by hand
  - Hash table lookup for the rest (could be automated with LTO)
- GIMPLE pass: handle_function enumerates all function calls looking for the size_overflow attribute (or hash table)
- handle_function_arg starts the real work
  - Manually walks the use-def chain of the given function argument
  - Walk forks on binary/ternary operations and phi nodes
  - Walk stops at asm/call stmts, function parameters, globals, memory references, constants, etc

SIZE_OVERFLOW

- When a walk stops, stmt duplication begins
  - New variable is created with `signed_size_overflow_type`
  - DImode or TImode (signed)
- When stmt duplication reaches the original function call, the duplicated result is bounds checked
  - Against `TYPE_MAX_VALUE`/`TYPE_MIN_VALUE`
  - Optimization: check omitted if the walk did not find any stmt that could cause an overflow

Overview

- Goal: extract entropy from kernel state during boot
- Inspired by https://factorable.net/
- USENIX Security Symposium, August 2012
- Problem: much less entropy after boot than needed
- Result: vulnerable RSA and DSA keys used for SSH/TLS
- Some fixes in Linux but can we do better?

LATENT_ENTROPY

- Idea: compute a hash-like function embedded in the control flow graph of kernel boot code
- Similar and also simpler approach already in Phrack 66
- Insert a random combination of ADD/XOR/ROL insns into every basic block
- Mix end state into a global variable in the function epilogues
- Feed global variable (entropy) into the kernel entropy pools after each initcall
- Entropy is not actually accounted for until someone cryptanalyzes this whole thing :)
- More info on our mailing list

Future

- LLVM/Clang
- Link-Time Optimization (LTO)
- Control flow enforcement (CFE)
- Data flow enforcement (KERNSEAL)
- Miscellaneous

Overview

- http://llvm.org and http://clang.llvm.org
- Mostly works with linux-side patches only
- clang 3.1 and `-integrated-as`, `.code16gcc`/`.code16`
- `-fcatch-undefined-behavior` (ext4 triggers it on mount)
- LTO
- Port the gcc plugins to llvm
- New plugins for clang (not really feasible with gcc)

Overview

- Idea: run optimization passes on one big translation unit combined from all source files
  - Allows whole program analysis
- Mostly works with gcc 4.7
- Takes 5 minutes and 4GB RAM on a quad-core Sandy Bridge
- Problems: KALLSYMS, tracing, initcalls, section attributes
- Better support for other plugins (CONSTIFY, REFCOUNT, SIZE_OVERFLOW, STACKLEAK, USERCOPY)
- New plugins: static stack overflow checking, sparse attributes, etc

LTO plans

- CONSTIFY: find all non-constifiable types/variables
- REFCOUNT: find all non-refcount `atomic_t`/`atomic64_t` uses
- SIZE_OVERFLOW: walk use-def chains across function calls, eliminate the hash table
- STACKLEAK: find all local variables whose address sinks into `copy*user`
- USERCOPY: find all `kmalloc-*` slab allocations that sink into `copy*user`

Overview

- Against the "execute existing code out-of-(intended)-order" exploit technique
- Compiler plugin to instrument all function pointer dereferences
- (No) support for binary-only modules
- Assembly source: manual instrumentation
- Runtime code generation (JIT compiler engines) support
- Performance impact is critical (<5% desired), very hard problem

Overview

- Against the "execute existing code in-(intended)-order" technique
- Ensures that certain kernel data cannot be modified unintentionally (arbitrary write bug)
- Credential structures, memory management data, filesystem metadata/data (page cache), etc
- Needs lots of infrastructure:
  - Read-only slab and kernel stacks (except for the current one :)
  - Efficient hardware support is missing (SMAP v2?)

Overview

- Android port
- Better virtualization support
  - virtualbox, vmware, xen
- More architecture support (especially kernel self-protection)
- Heap (slab) hardening
- CPU cores dedicated to security
- Your ideas :)

http://pax.grsecurity.net

http://grsecurity.net

irc.oftc.net #pax #grsecurity