"Sections are types, linking is policy"

Intra-Process Memory Protection for Applications on ARM and x86: Leveraging the ELF ABI

Sergey Bratus
Julian Bangert
Maxwell Koo

SEL
SCHWEITZER ENGINEERING LABORATORIES, INC.

NARF INDUSTRIES

DARTMOUTH
VOX CLAMANTIS IN DESERTO
1769

The Problem

- A buggy library can read or corrupt **any** of your process memory
  - "An image parser just stole my private keys"

"What's your angle?"

- Software is **already** split into parts
  - Libraries, compilation units, functions, ...
    - Their **interactions** tell a lot about them
- Linkers/binary toolchains already know a **lot** about intended & unintended interactions between these parts
  - **But**: runtime **discards** all this information, wastefully

With ELFbac, you can describe how parts of your application interact (via ELF metadata)

"Sections are types, linking is policy"

Key architectural idea

- **ELF sections** describe **identities** & layout of program's code & data parts in memory
  - Great for policy, but discarded by loaders :(
- Kernel's **virtual memory structures** describe layout of process' parts in memory
  - Intent (r?,w?,x?) is enforced via PTEs & page faults
- Connect *ELF structs -> VM structs* via a "non-forgetful" loader! Enforce intended code & data interaction

Outline

- Why use ELF ABI for policy
    - ***Unforgetful loader*** for intra-memory ACLs
- Case studies:
    - OpenSSH policy vs CVE-2016-0777 (roaming bug)
    - ICS protocol proxy
- Internals
    - Linux x86 prototype (Julian)
    - ARM prototype (Max)

Background/Motivation

- File-level policies (e.g., SELinux) fail to capture what happens **inside** a process (cf. Heartbleed, etc.)
- CFI, DFI, SFI, etc. are good *mitigations*, but they aren't policy: they don't describe **intended** operation of code
- **ELF ABI** has plenty of structure to encode **intent** of a process' parts: libraries, code & data sections
  - Already supported by the GCC toolchain!
  - Policy is easy to create, intuitive for C/C++ programmers

Policy vs mitigations

- Both aim to block unintended execution (exploits)
- Mitigations attempt to **derive** intent
  - E.g., no calls into middles of functions, no returns to non-call sites, etc.
- Policy attempts to **express** intent explicitly
  - E.g., no execution from data areas, no syscalls beyond a whitelist, no access to files not properly labeled
- Policy should be **relevant** & **concise** (or else it's ignored)

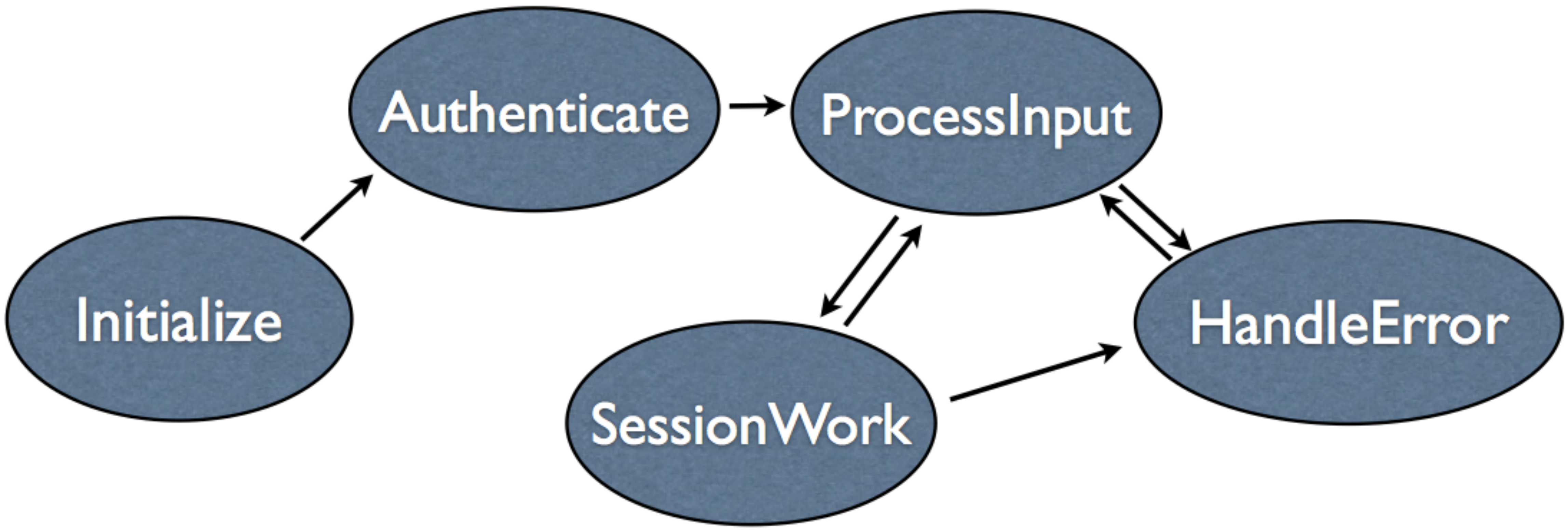
Policy wish list

- Relevance: describe what **matters**
  - E.g.: SELinux is a "bag of permissions" on file ops. Can't describe **order** of ops, number of ops, **memory accesses**, any **parts** of a process
  - Once your key is in memory, its file label is irrelevant
- Brevity: describe **only** what matters
  - E.g.: SELinux makes you describe **all** file ops; you need tools to **compute** allowed data flows

What matters?

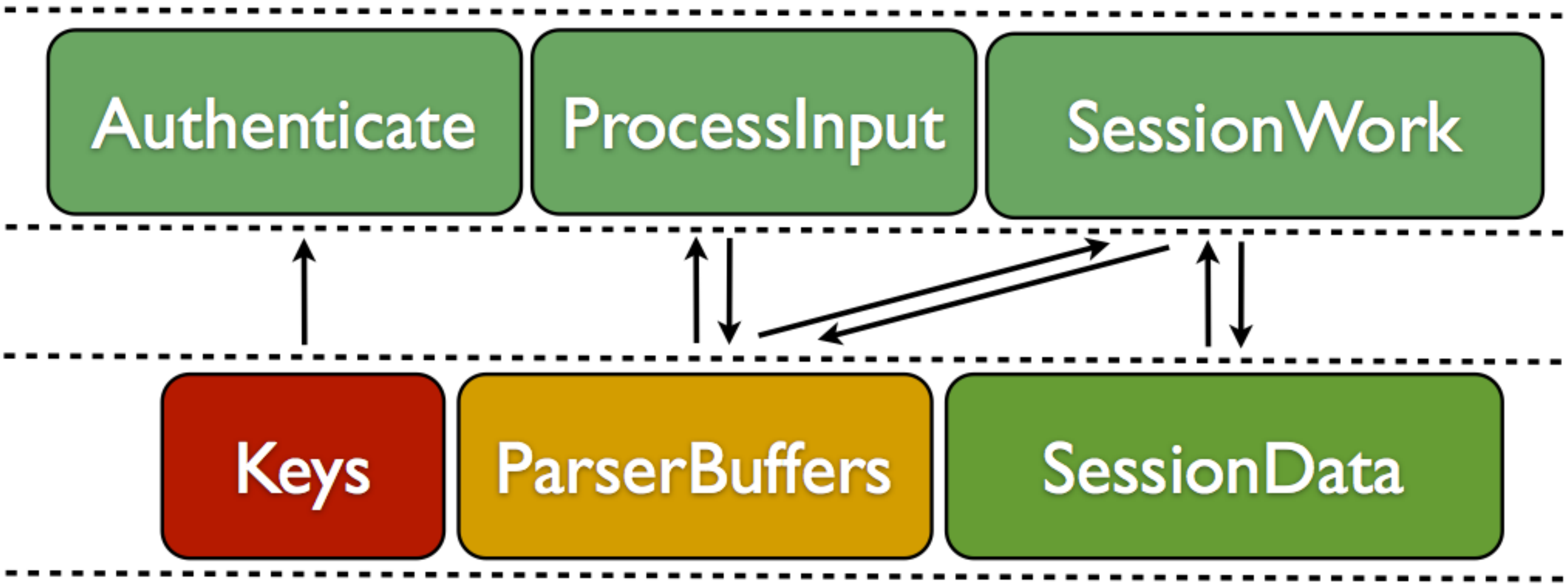
- *Composition*: a process is no longer "a program"; it's also many different **components** & libraries, all in one space, but with very different purposes & intents
- *Order of things*: a process has **phases**, which have different purposes & intents
- *Exclusive relationships*: pieces of code and data have **exclusive relationships** by function & intent
  - **"This is *my* data, only *I* should be using it"**

Process phases



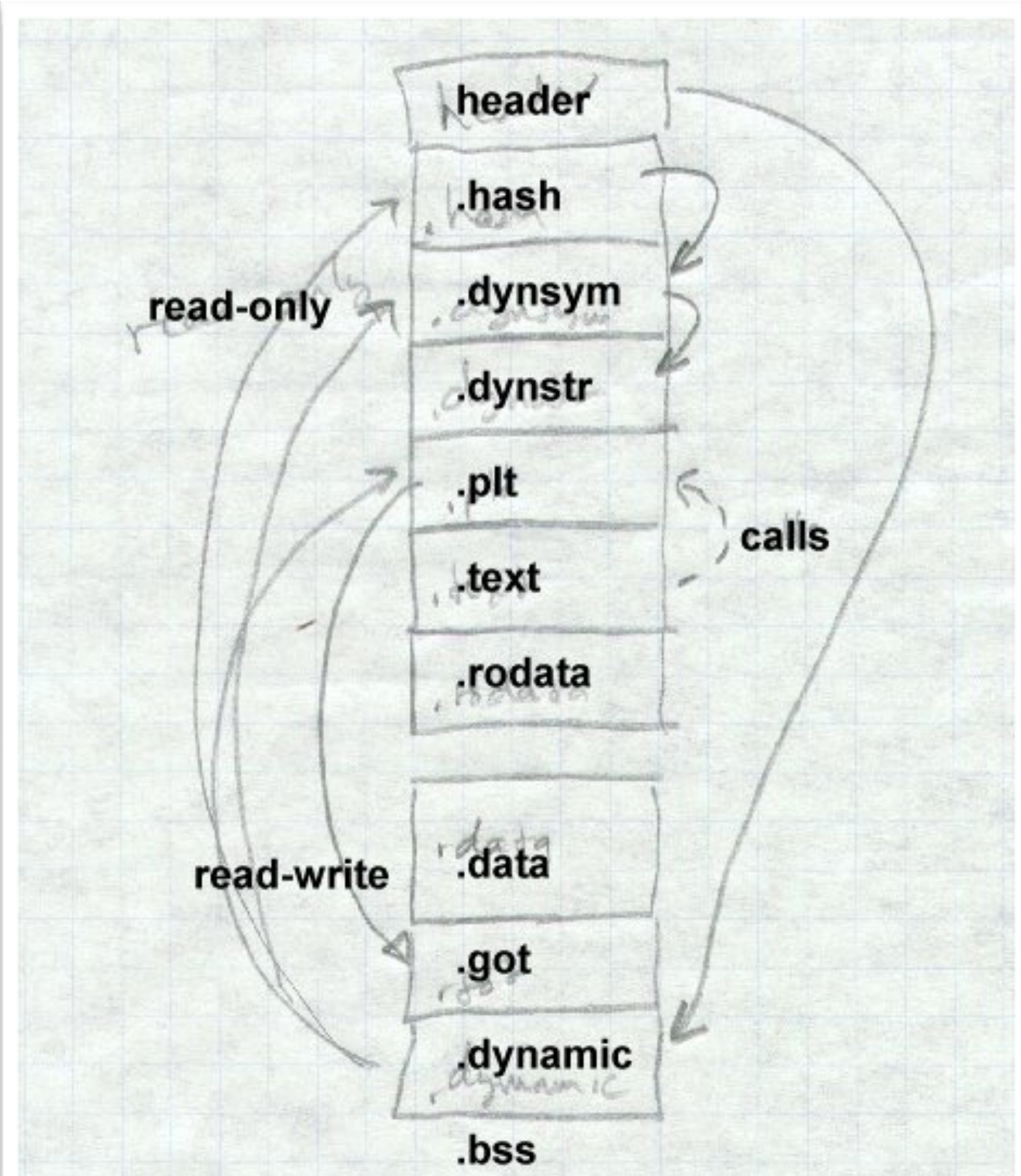
- "Phase" ~ code unit ~ EIP range ~ memory section

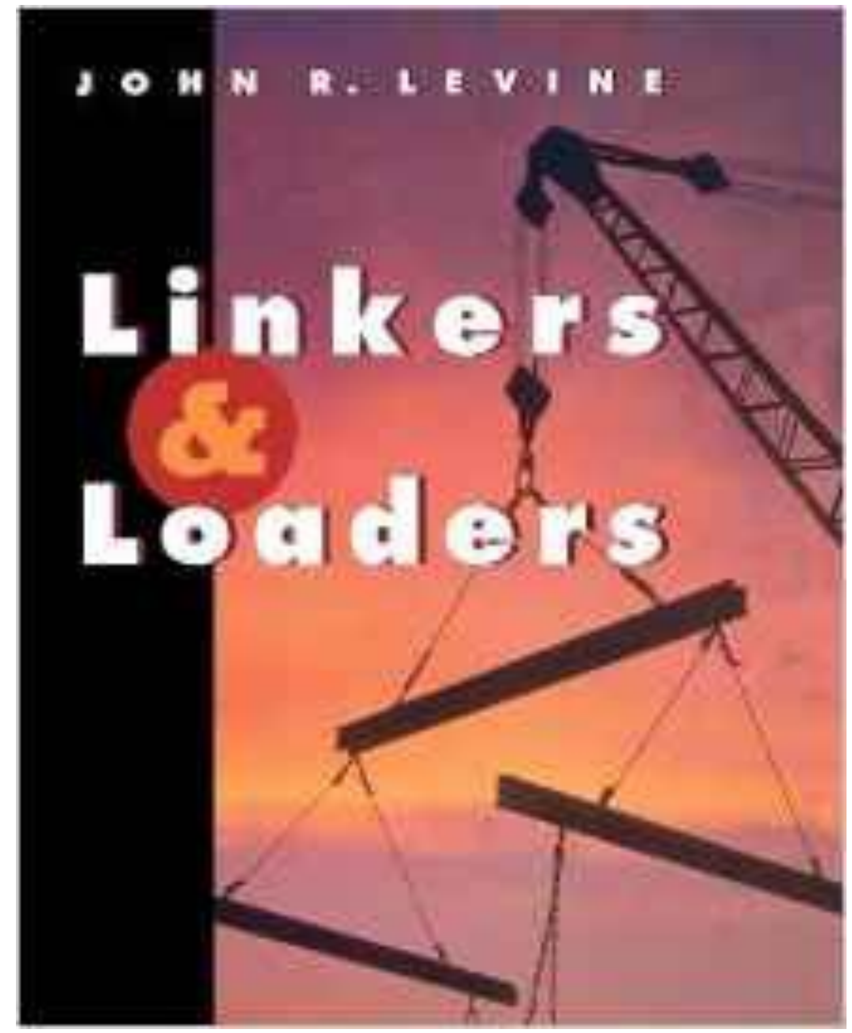
Access relationships are key to programmer intent



- Unit semantics ~ Explicit data flows (cf. *qmail*)

An inspiration: ELF RTLD

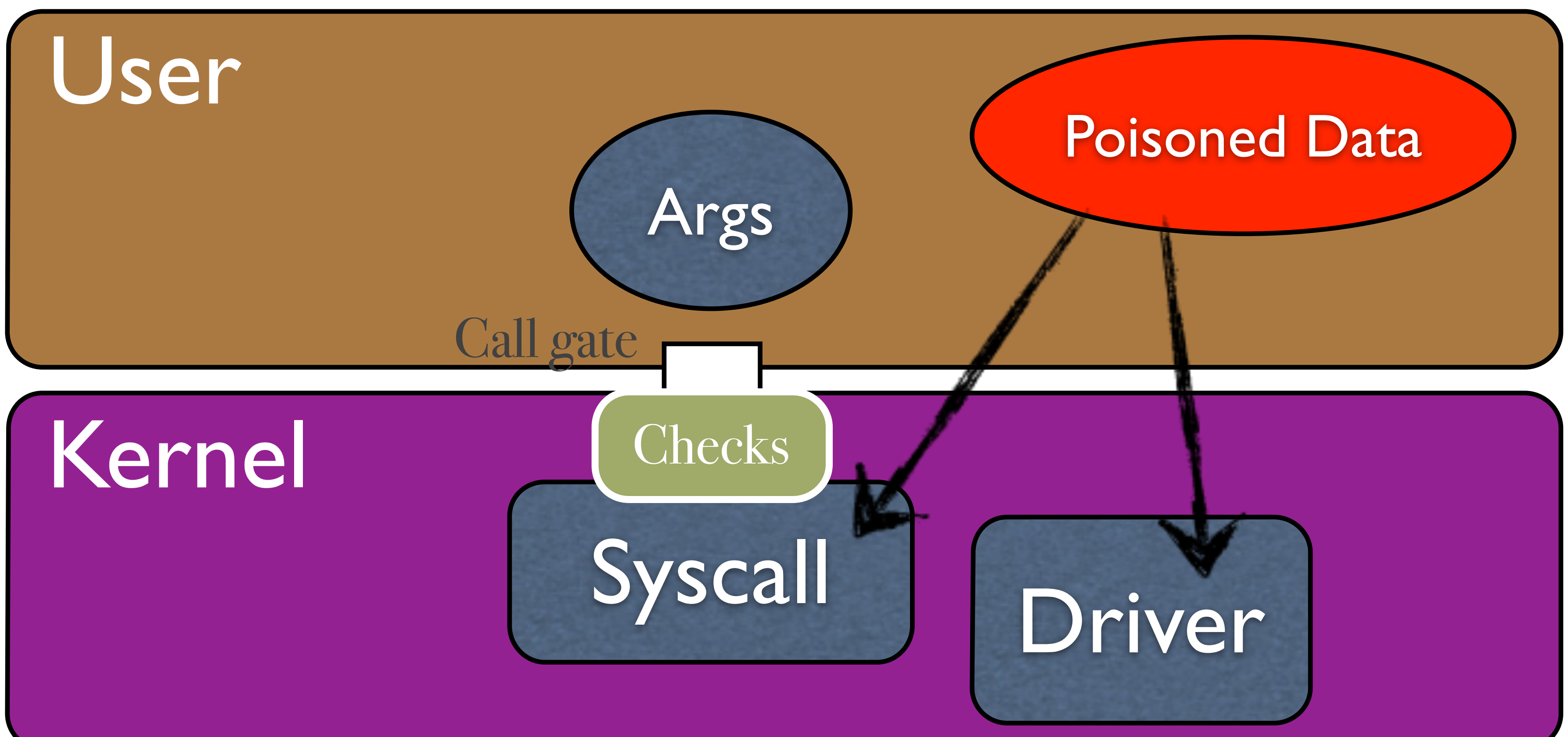




John Levine, "Linkers & loaders"

An inspiration: PaX/GrSec UDEREF

UDEREF prevents kernel code from accessing userland data it wasn't meant to access

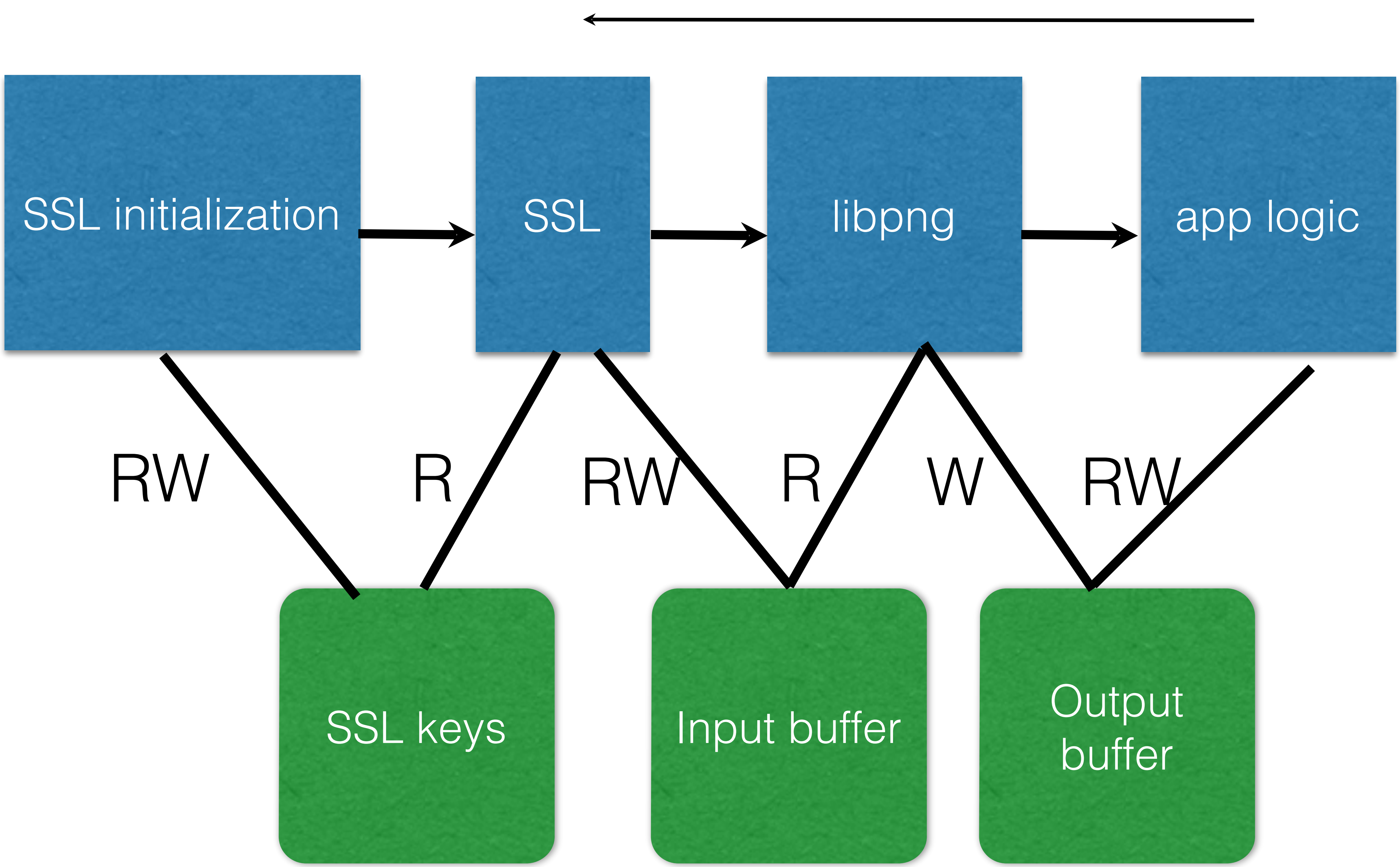


"Some thoughts on security after ten years of qmail", D.J. Bernstein, 2007

- Used process isolation as security boundaries
  - Split functionality into many per-process pieces
- Enforced **explicit data flow** via process isolation
- ***"Least privilege was a distraction, but <u>isolation</u> worked"***

http://cr.yp.to/qmail/qmailsec-20071101.pdf

Back to our example



"Sections are types, linking is policy"

- The idea of a *type* is "objects with common operations"
  - Methods of a class in OOP, typeclasses in FP, etc.
- For data sections, their dedicated code sections are their operations
  - It's dual: data accessed by code tells much about code
- Linkers collect similar sections into contiguous pieces
  - Linkers see much info, but discard it all

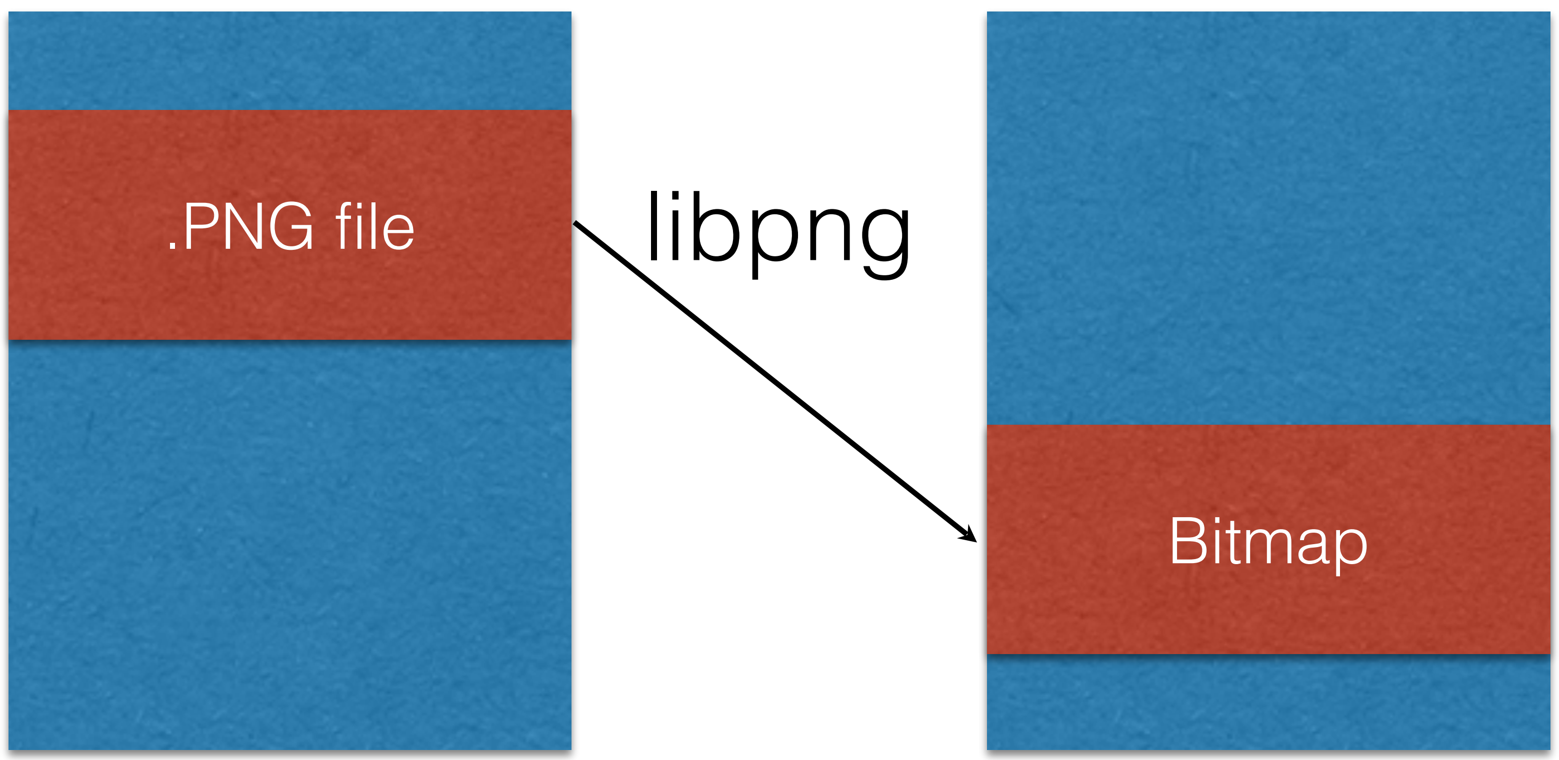
Enforcing: Unforgetful loader

- Modern OS loaders **discard** section information
- New architecture:
  - '**Unforgetful** loader' preserves section identity after loading
  - Enforcement scheme for **intent-level semantics**
  - Better tools to capture semantics in ABI

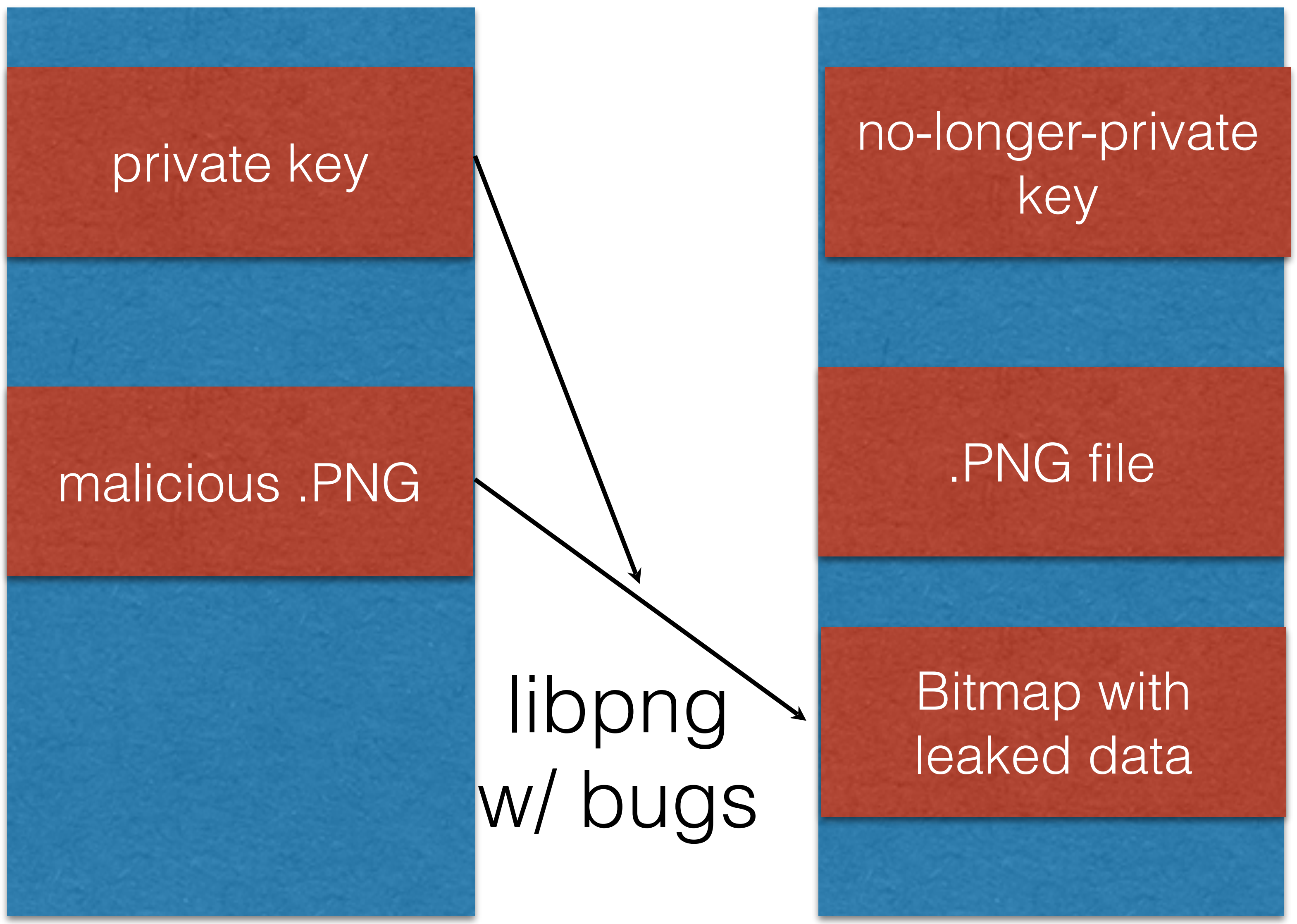
Motivating Example

Example policies

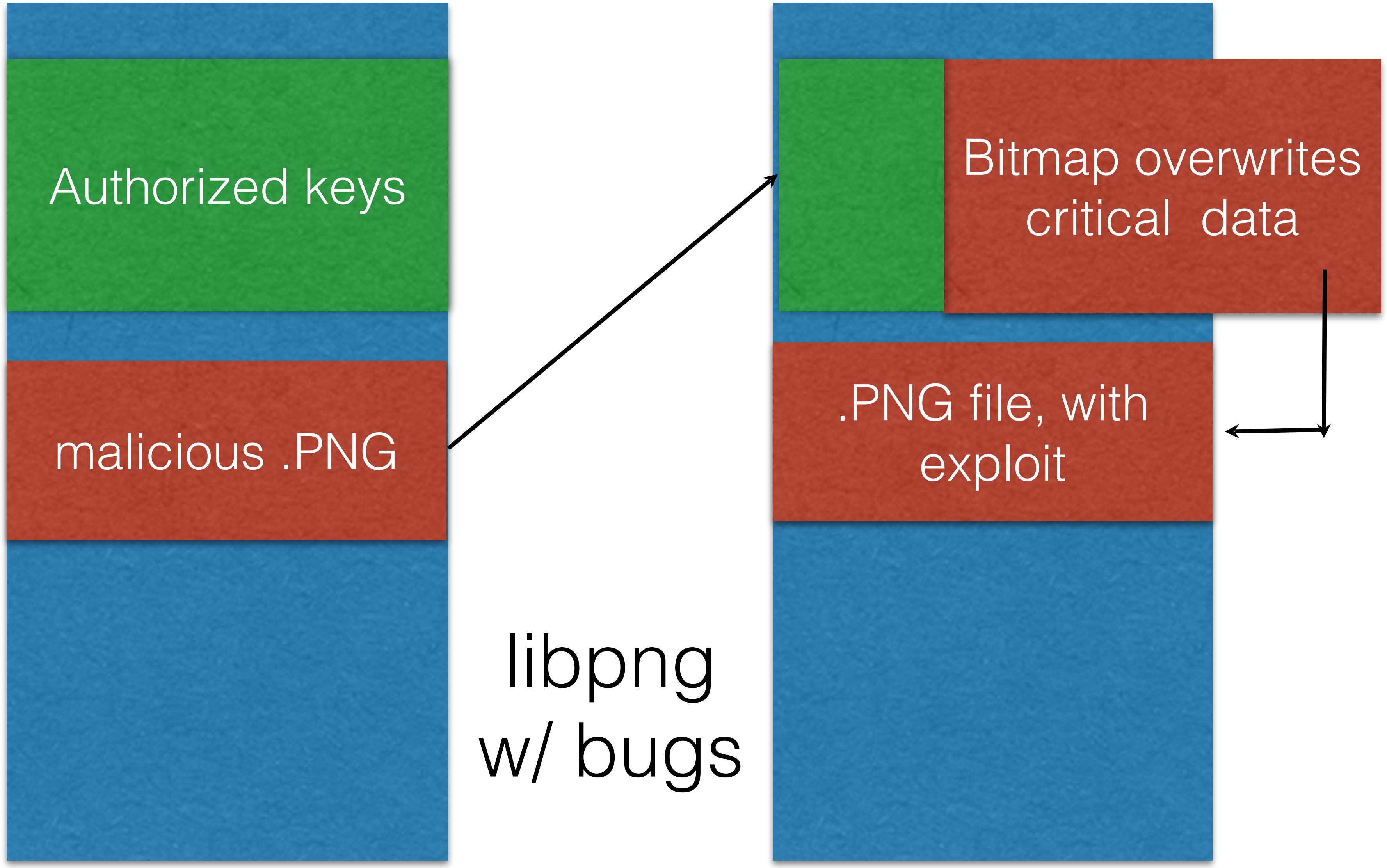
- Web application decompresses a PNG file
- Mental model



What attackers see

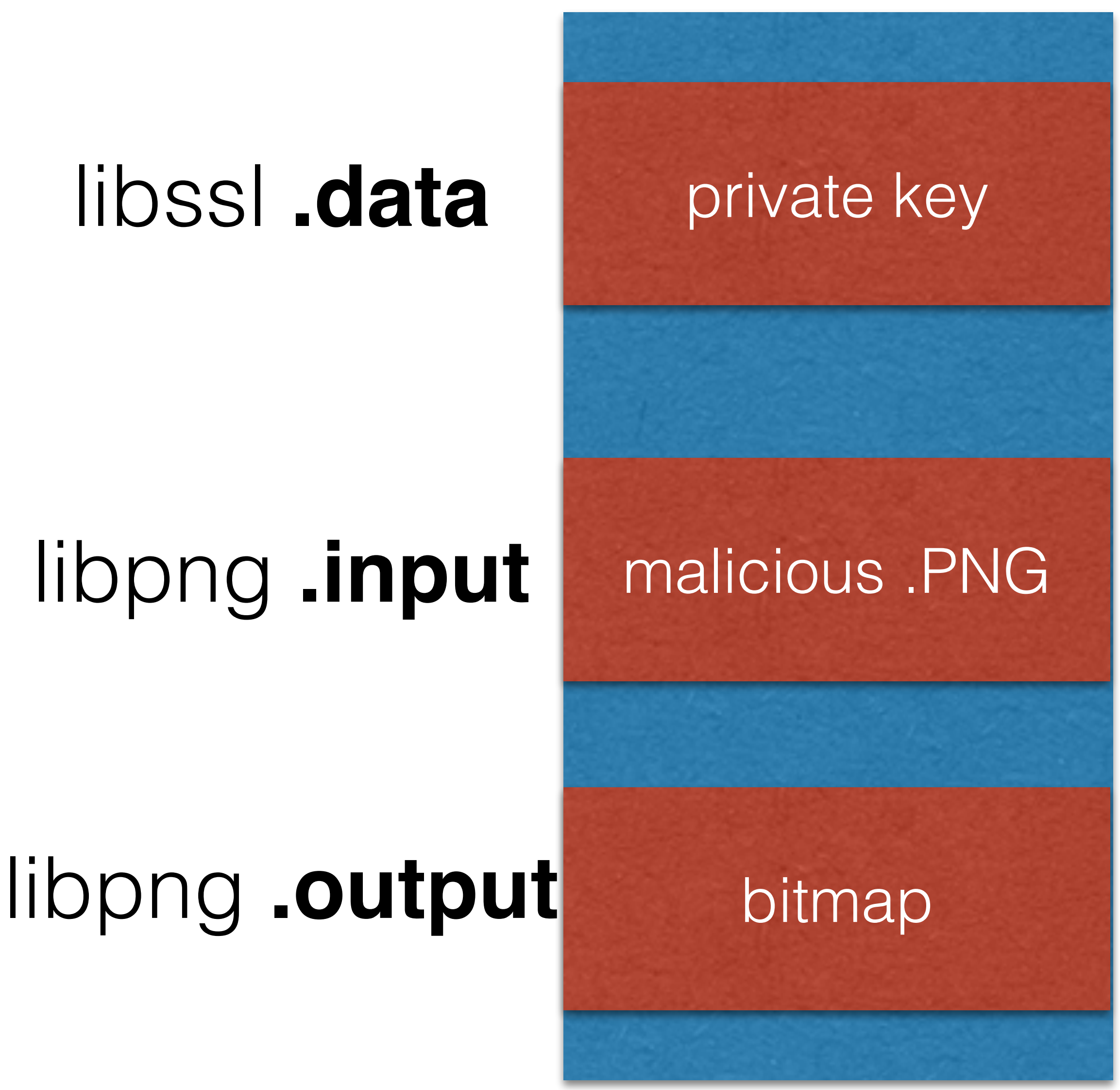


Or



Authorized keys
malicious .PNG
libpng
w/ bugs
Bitmap overwrites
critical data
.PNG file, with
exploit

Mapping it into the ABI



- Easy to introduce new sections
- Each code segment can get different permissions
- Only libssl.text can access libssl.data
- libpng.text can only access libpng.input and libpng.output
- And libpng.input can only be read by libpng.

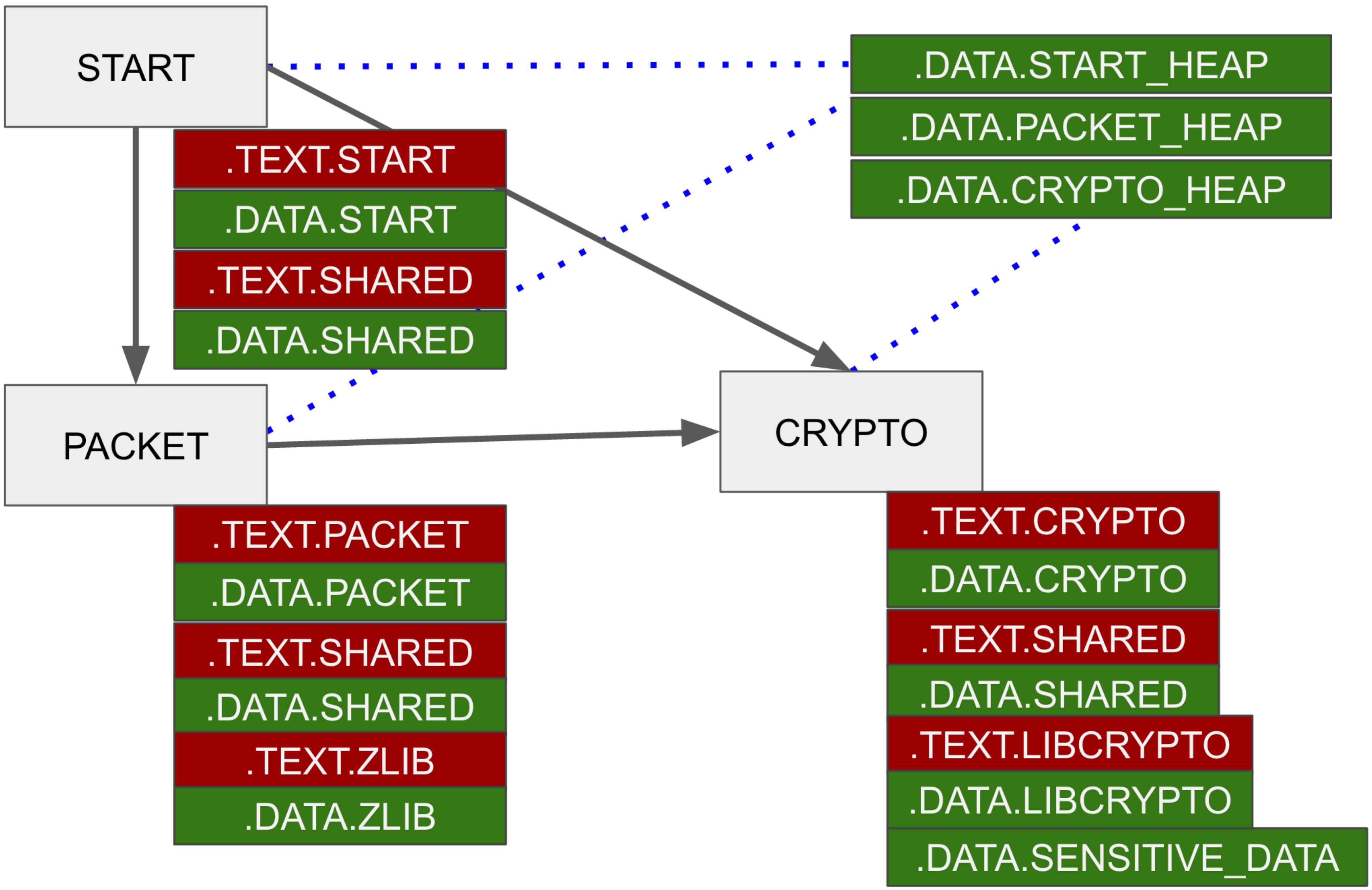
ELFbac Policy Case Studies

I. OpenSSH

OpenSSH policy

- OpenSSH attacked via crafted inputs
  - GOBBLES pre-auth RCE 2002 -- CVE-2016-077{7,8}
- OpenSSH introduced the original **privilege drop** as a **policy** primitive
  - "If the process asks for a privileged op after ***this point***, it's no longer trustworthy; kill it"
- But accesses to (a) non-raw data by a parser (b) raw data beyond the parser are **also** privilege!

OpenSSH policy at a glance



OpenSSH demo

ELFbac vs CVE-2016-0777

ELFbac for OpenSSH

- Policies for both the OpenSSH **client** and **server**
  - Isolate portions of OpenSSH responsible for crypto/key management from those responsible for processing & parsing packets
  - Create separate sections for sensitive data blobs, allowing for finer-grained access control
  - Control access to libraries used by OpenSSH based on **where** used
- Prevent direct leaking of sensitive data like private keys from, e.g., ***CVE-2016-0777*** (roaming vuln)
- Separate heaps for dynamic allocations, with specific access permissions across process phase boundaries

II. ICS/SCADA proxy

ELFbac for SCADA/ICS

- **DNP3** is a complex ICS protocol; prone to parser errors
  - S4x14: "Robus Master Serial Killer", Crain & Sistrunk
- Only a small subset of the protocol is used on any single device. Whitelisting this syntax is natural.
  - A filtering proxy is a DNP3 device's best friend
  - "**Exhaustive syntactic inspection**": *langsec.org/dnp3/*
- ELFbac policy: isolate the parser from the rest of the app

Parser isolation

- **Raw data is (likely) poison**; parsing code is the riskiest part of the app & its only defense
- Parser must be separated from the rest of the code
    - **No other section** touches raw input
    - Parser touches no memory outside of its output area, where it outputs checked, well-typed objects
- *Input* => Parser => *Well-typed data* => Processing code

Our ARM target

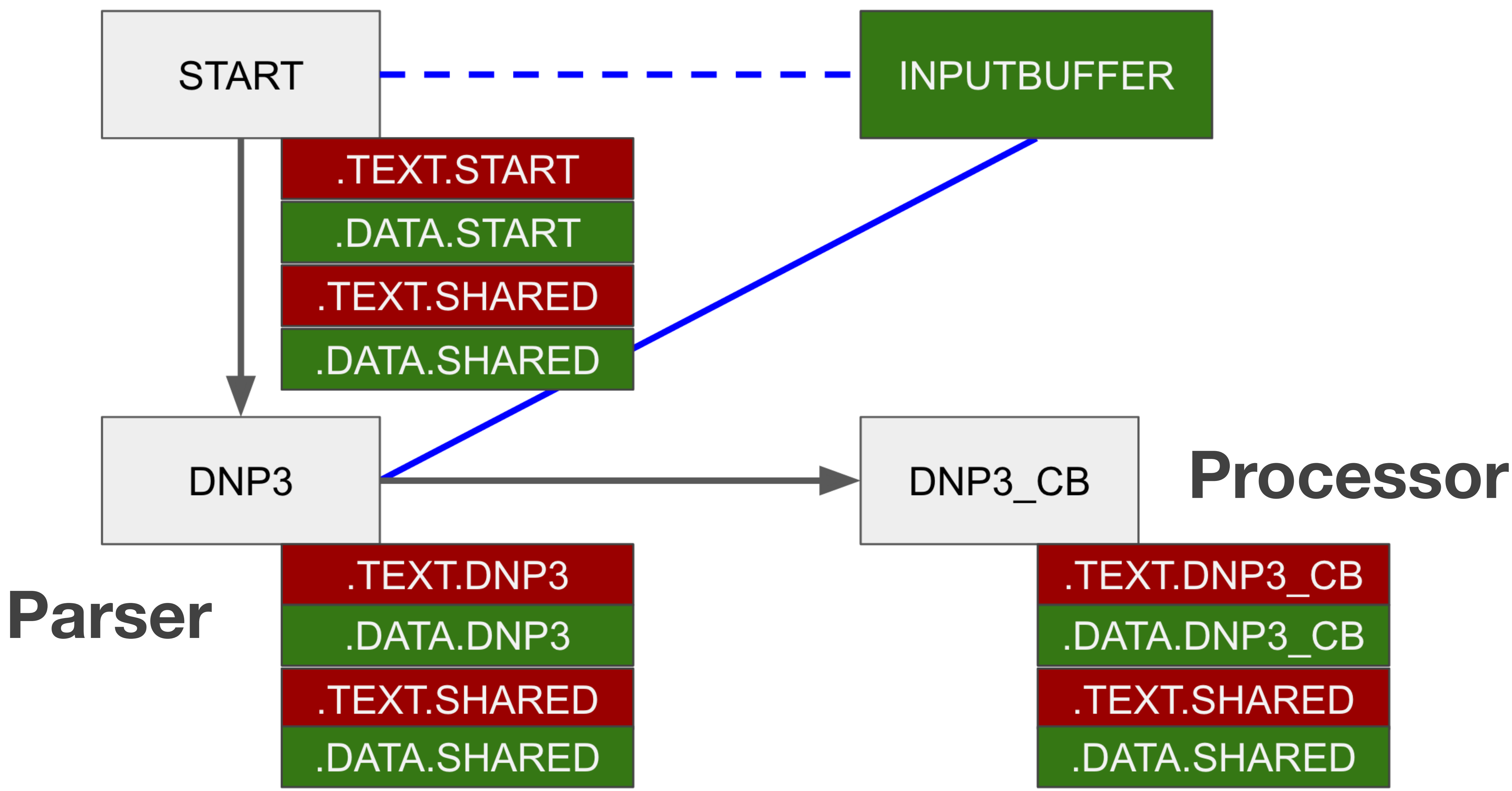
UC-8100 Series

Communication-centric RISC computing platform

- ARMv7 Cortex-A8 300/600/1000 MHz processor
- Dual auto-sensing 10/100 Mbps Ethernet ports
- SD socket for storage expansion and OS installation
- Rich programmable LEDs and a programmable button for easy installation and maintenance
- Mini PCIe socket for cellular module
- Debian ARM 7 open platform
- Cybersecurity

c UL us LISTED

ICS proxy policy at a glance



ELFbac & Grsecurity/PaX for ARM

- We worked with the Grsecurity to integrate ELFbac on ARM with **Grsecurity for ICS** hardening:
  - Cohesive set of protections for ICS systems on ARM
    - PAX_KERNEXEC, PAX_UDEREF, PAX_USERCOPY, PAX_CONSTIFY, PAX_PAGEEXEC, PAX_ASLR, and PAX_MPROTECT
  - Available from ***https://grsecurity.net/ics.php***
- ELFbac + Grsecurity ICS tested with our DNP3 proxy on a common industrial computer Moxa UC-8100, ARM v7 (Cortex-A8)

Implementation internals

Linux x86 prototype sketch

- Prototype on Linux via virtual memory system
- Each **phase** of execution (=policy-labeled code section) sees a different **subset of the address space** (=labeled data sections)
- **Traps** handle phase transitions by changing *CR3*
- Each phase has its own **page tables** that cache part of the address space, reusing existing TLB invalidation primitives.
- Use **PCID** on newer processors to reduce TLB misses

Life of a program:
from ELF file to a process

Bridging the gap between ELF program metadata and kernel's virtual memory structs

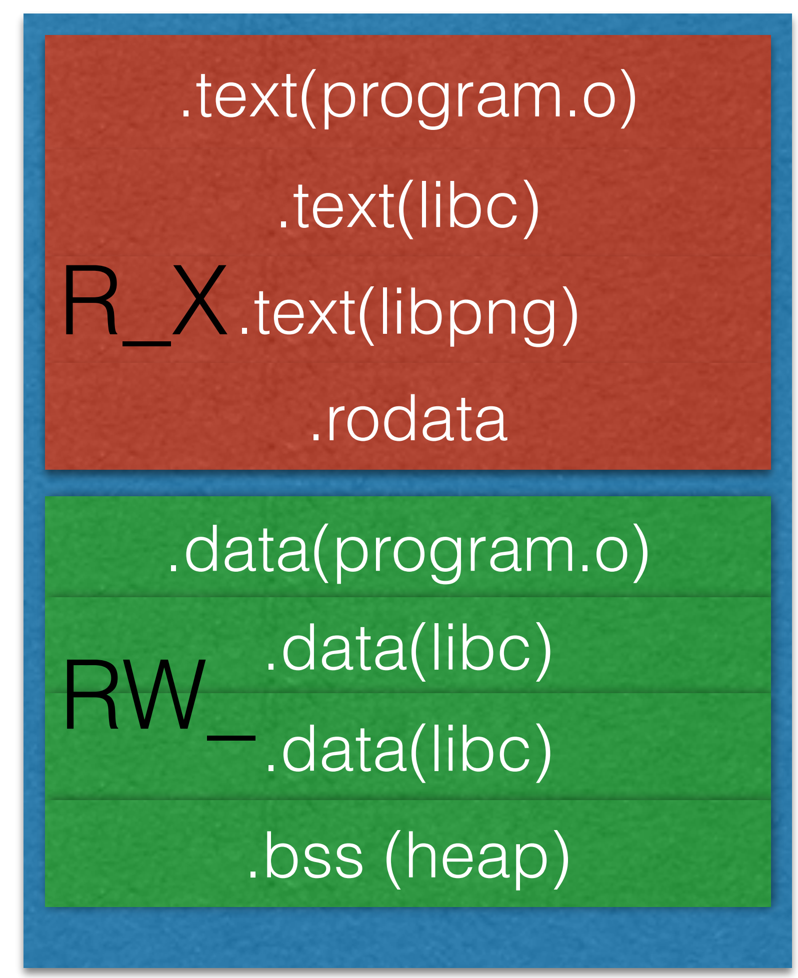
ELF sections

ELF consists of sections:

- Code
- Data (RW/RO)
- GOT/PLT jump tables for dynamic linking
- Metadata: Symbols, ...
- Can be controlled from C:
  __section__(section_name)
- Flexible mechanism
- ~30 sections in typical file

Sections turn into segments

Linker combines sections & groups them into segments:



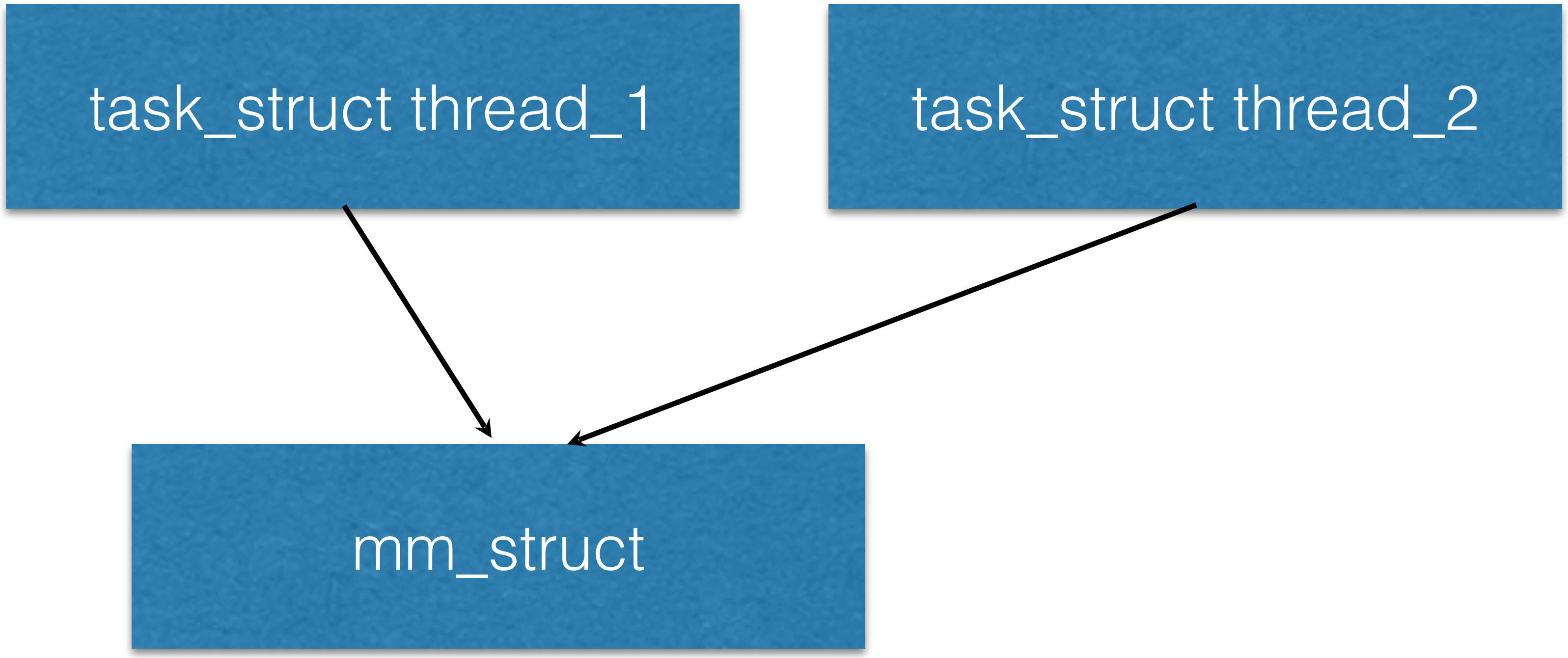
Only RWX bits enforced

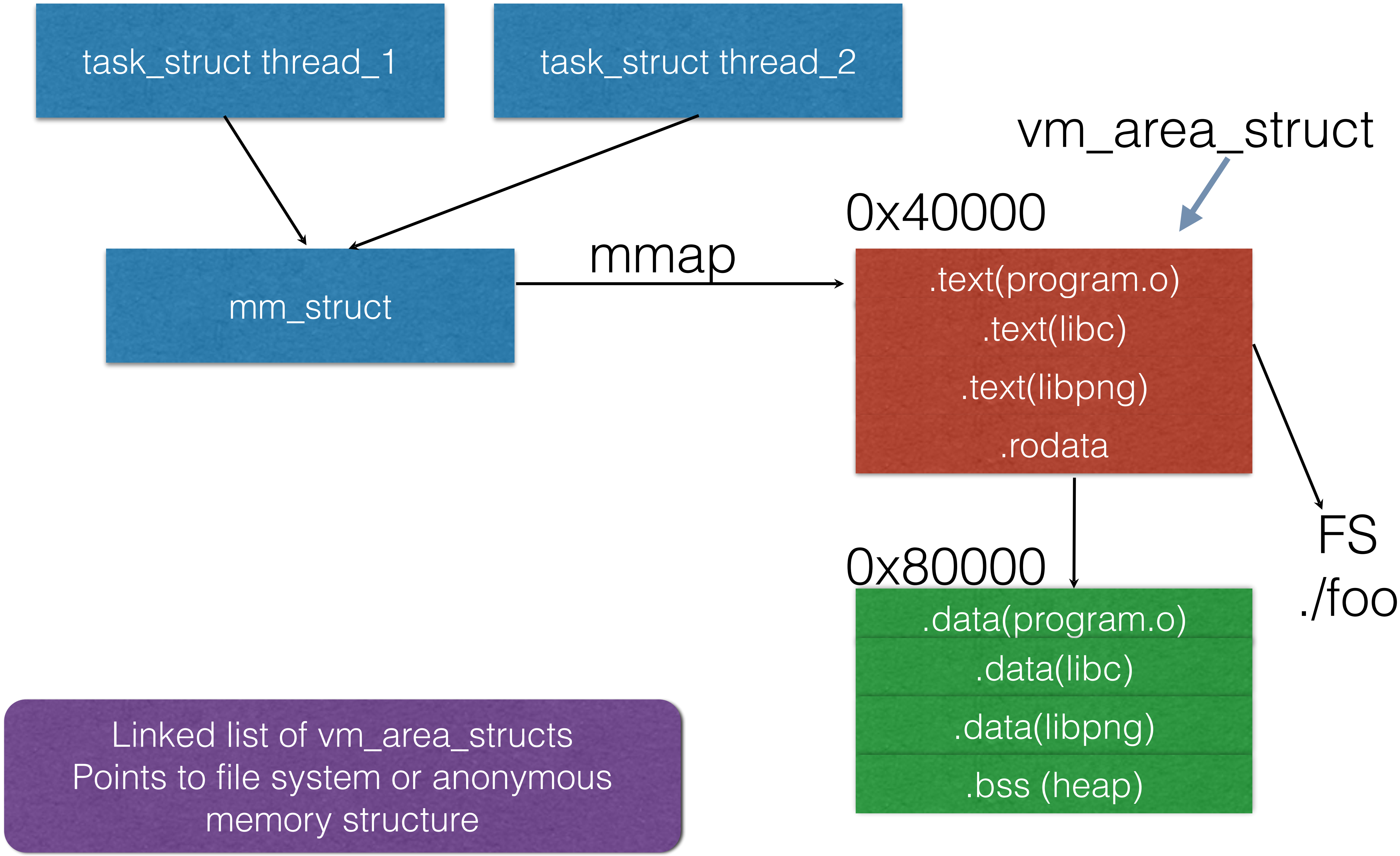
How a process is set up

- Static linking:
  - kernel (**binfmt_elf.{c,ko}**) reads segments
  - calls **mmap**() for each segment
  - jumps to the entry point
- Dynamic linking
  - Kernel loads **ld.so** (as in the above)
  - ld.so parses ELF file again (bugs happen here)
  - ld.so opens shared libraries, mmaps and maintains **.PLT** / **.GOT** tables
- One mmap() call per segment

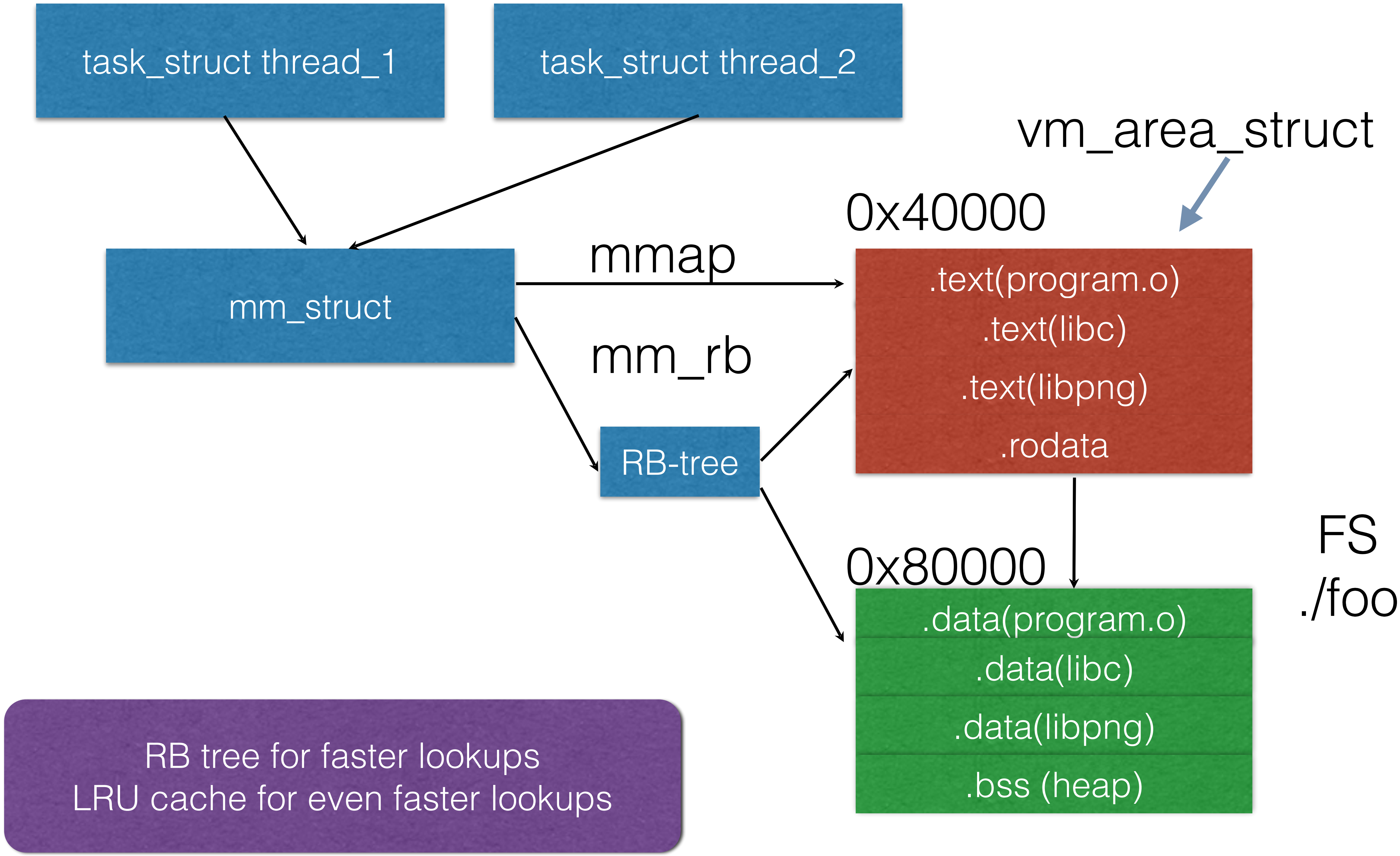
What the kernel does:

- Kernel:
  - task_struct for each thread
    - registers, execution context => state
    - pid, uid, capabilities => identity of the process
  - mm_struct for address space

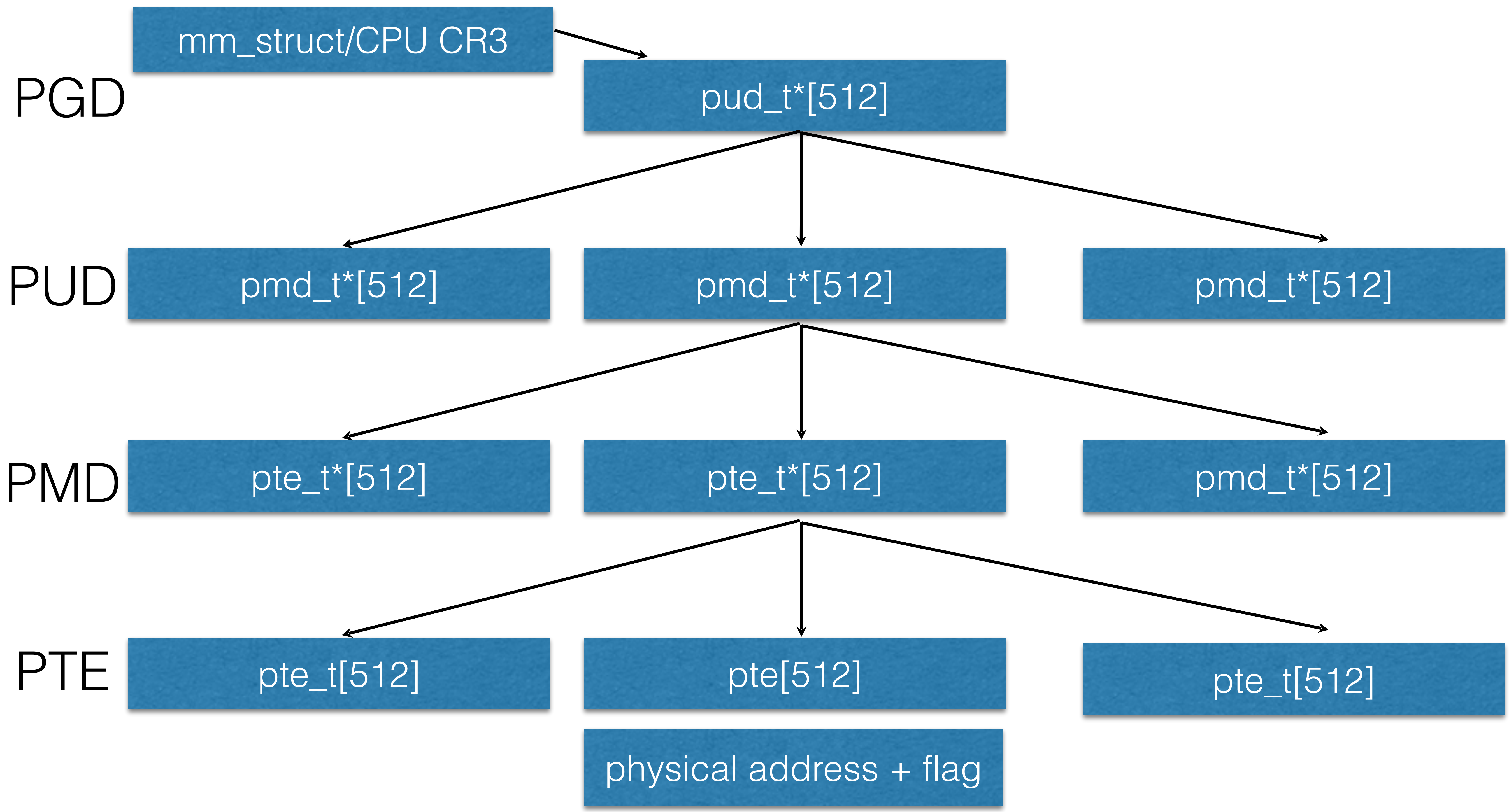

task_struct thread_1
task_struct thread_2
mm_struct



task_struct thread_1
task_struct thread_2
vm_area_struct
0x40000
mmap
mm_struct
.text(program.o)
.text(libc)
.text(libpng)
.rodata
FS
./foo
0x80000
.data(program.o)
.data(libc)
.data(libpng)
.bss (heap)
Linked list of vm_area_structs
Points to file system or anonymous
memory structure


task_struct thread_1
task_struct thread_2
vm_area_struct
0x40000
mmap
mm_struct
mm_rb
.text(program.o)
.text(libc)
.text(libpng)
.rodata
RB-tree
0x80000
FS
./foo
.data(program.o)
.data(libc)
.data(libpng)
.bss (heap)
RB tree for faster lookups
LRU cache for even faster lookups

What the CPU sees



All three structures have to be kept in sync

Caching

- Walking these structures on every memory access would be prohibitively slow
- TLBs cache every level of this hierarchy
- Originally invalidated on reload
- **Tagged** TLBs (PCID on intel). ELFbac also had the first PCID patch for linux. Transparent on AMD

Caches enforce policy!

- NX bit is seen as a mere mitigation
- Actually it is **policy** that express **intent**
- First implementations of NX used cache state (split TLB) meant for performance to add semantics
- ELFbac does the same with TLBs and PCID

It's all about caching

- Each VM system layer is a **cache**
- And performs checks
  - Checks get semantically less expressive as you get closer to hardware
- ELFbac adds another layer of per-phase **caching**
- Allows us to enforce a semantically rich **policy**

Example: Page faults

- If the page table lookup fails, CPU calls the kernel
- Kernel looks for the **vm_area_struct** (rb_tree)
- **Check:** If not present, SIGSEGV
- Fill in page table, with **added semantics**
  - Swap-in
  - Copy-on-write
  - Grow stacks

ELFbac execution model

- Old **n-to-1 relationship**:
  - task_struct (n threads) **<->** mm_struct (1 process)
- New **n-to-m relationship:**
  - task_struct (n threads) **<->** mm_struct (m ELFbac phases)
- A lot of kernel code would have to change to update m copies

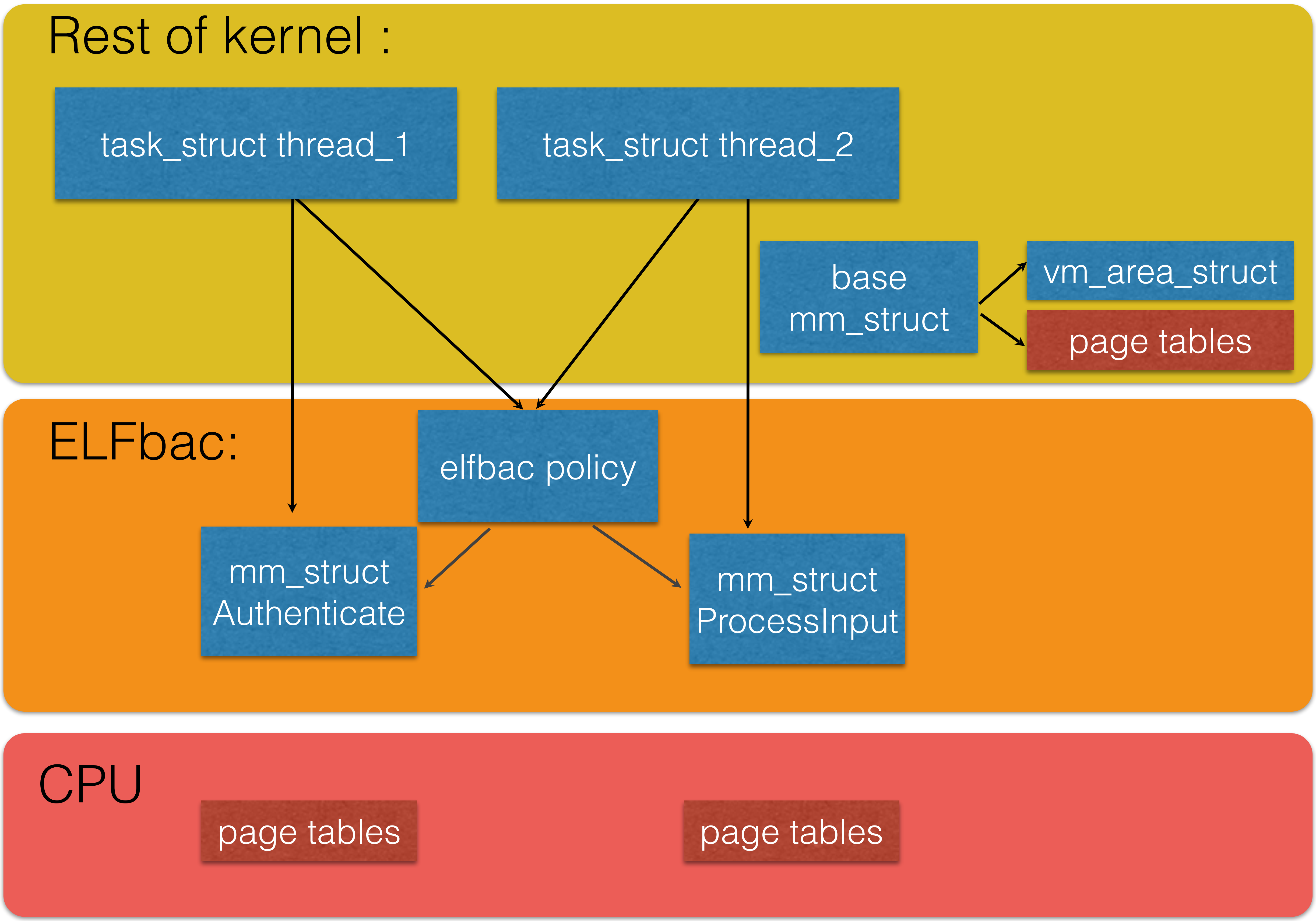
Caching as a solution

- ELFbac states are **subsets** of the base address space
  - Base address space still represented by mm
- Squint enough, and a subset is like a **cache**
- Only need **invalidation** instead of mutation
- Caches already have to be invalidated (TLB)
- Linux: mm_notifier plug-in API (virtualization)

ELFbac page fault handler

- If the access would fault on the base page tables
  - Fall back to the old page fault handler
- Look up the address in ELFbac policy
  - Move process to new phase if necessary
  - Otherwise copy page table entry to allow future accesses

What each part sees:



Performance overheads

- NGINX benchmarked with a policy isolating **all** libraries from the main process:
  - Best case: around ~5% (AMD Opteron Piledriver)
    - worst case: ~30% on some Intel platforms
  - Too many state transitions on the hot path
  - Policy must be adapted to the application structure
- Average ~15% when running on KVM
  - KVM already incurs performance costs
  - KVM optimizes virtual memory handling

Porting to embedded ARM

- Focused on compartmentalizing ELF binaries under **static** linking
  - Dynamic linking case supportable by creating an ELFbac-aware ld.so, left to future work
- Policies generated from a JSON descriptor file
  - tool produces both the **linker script** and the **binary policy**
- Binary policy is packed into a special segment, loaded by the kernel during ELF loading time

Internals of ARM port

- **Page fault** handler enforces state & transition rules
  - Changed to accommodate simpler binary policy
- ARM **ASIDs** (tagged TLB) reduce overhead between state transitions
  - Essential to reduce overhead

Binary Rewriting Tools

- Storing policy in an ELF executable as a section requires binary rewriting
  - Made our own tool ***Mithril***, currently only implemented for ELF (***github.com/jbangert/mithril***)
- Translates binaries into a *canonical form* that is less context-dependent and can be easily modified
- Tested on the **entire** Debian x86_64 archive, producing a bootable system
  - ~25GB of packages rewritten, 260 core hours on S3

Drawbacks and TODOs

- Significant performance tuning still outstanding
- Implement an ELFbac-aware **malloc**
  - Methods for easy labeling of anonymous allocations
- Integration with system call policy mechanisms (e.g. Capsicum)
- Provide rich policies for many standard libraries
  - ELFbac is not a mitigation, it's a way to design policies and resilient applications

ELFbac is a design style

- "Who cares? That's not how code gets written"
- Availability of enforcement mechanisms **reshapes** programming practice
  - C++ took over the world by making contracts (e.g., encapsulation) enforceable (weakly, at compile time)
  - Non-enforceable designs are harder to adopt & check
- **Only enforceable separation matters; ELFbac makes program separation into units enforceable**

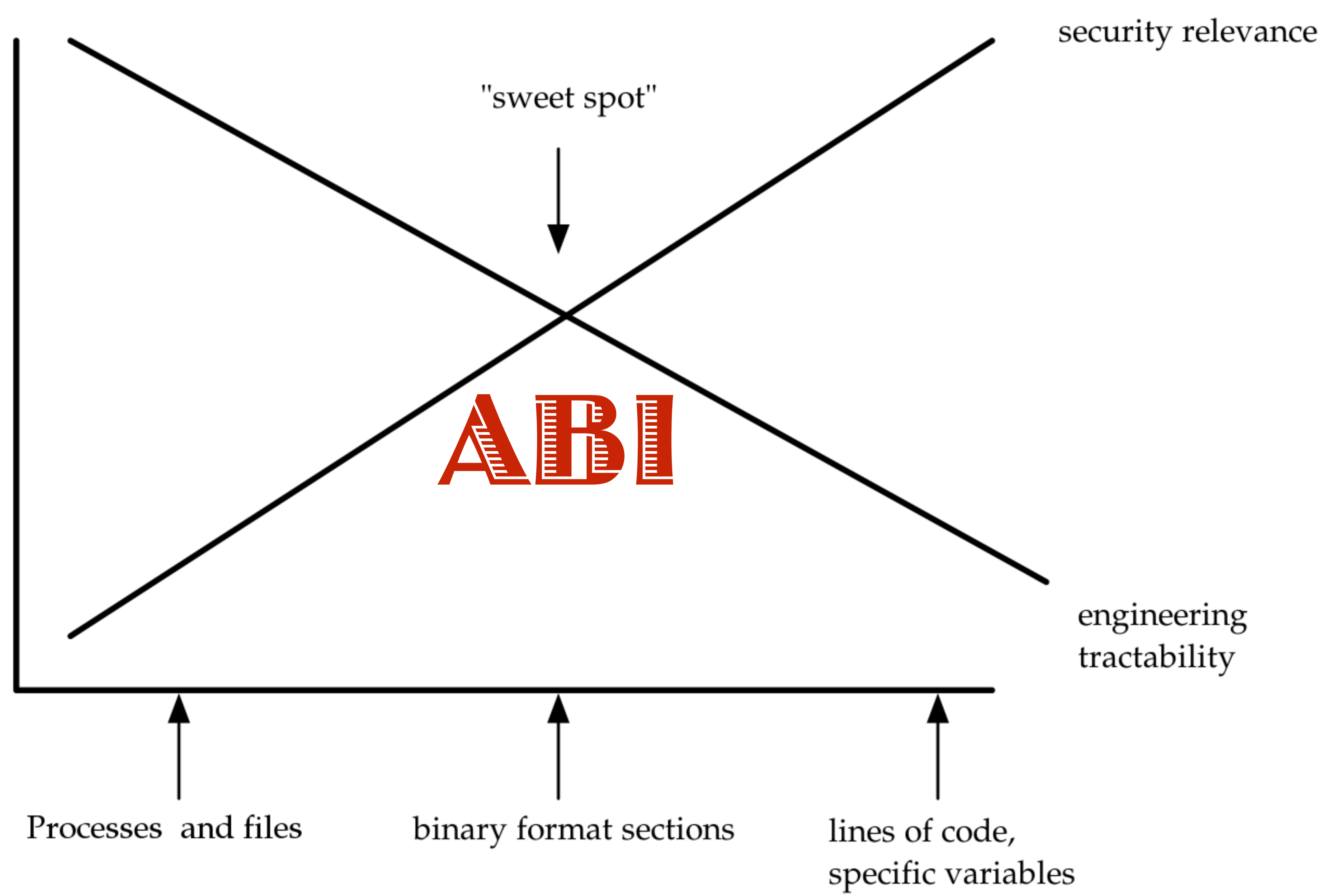
Application design considerations

- "Separating concerns" is good engineering, but has limited security pay-offs
    - All concerns still live in the **same address space**
- Separating heaps without ELFbac has **limited** returns:
    - Proximity obstacles to overflows/massaging, but still the same address space, accessible by all code
    - Mitigation, not policy
- With ELFbac, keeping marked, separate heaps becomes policy: clear **intent**, enforced w.r.t. code units

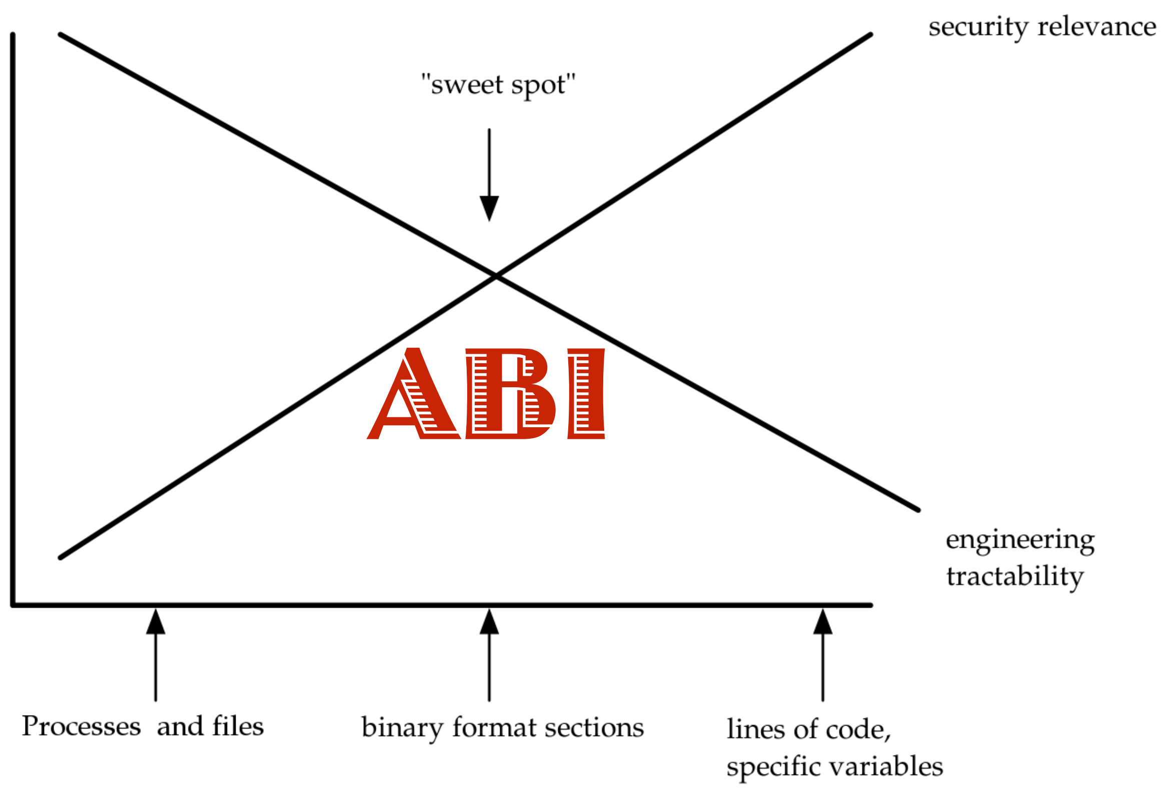
Takeaway

- Per-process bags of permission are no longer a suitable basis for security policy
- Instead, **ABI-level memory objects** at process **runtime** are the sweet spot for policy
- Modern ABIs provide enough granularity to capture **programmers intent** w.r.t. **code** and **data units**
    - ELFbac: Intent-level semantics compatible with ABI, standard build/binary tool chains

Policy Granularity: ABI is the Sweet Spot



Thank you



- *http://elfbac.org/*
- *https://github.com/sergeybratus/elfbac-arm/*