Lightweight Capability Domains:
TOWARDS DECOMPOSING THE LINUX KERNEL

Charles Jacobsen  Muktesh Khole  Sarah Spall  Scott Bauer  Anton Burtsev
Decades of work to make kernels more secure

- Decompose and isolate subsystems
- Formal verification/static analysis
- Stack guards
- Address space layout randomization
- NX, SMEP protection
- Control flow integrity
- Software fault isolation
- Safe languages
Linux Kernel Vulnerabilities by Year
Example

- Remote exploit in Linux network firewall
  - Arbitrary code execution
  - Linux Kernel v 3.0 (June, 2011) – 3.13.6 (March, 2014)
  - CVE-2014-2523

static bool dccp_new (...) {
    struct dccp_header _dh, *dh;

    - skb_header_pointer(skb, dataoff, sizeof(_dh), &dh);
    + skb_header_pointer(skb, dataoff, sizeof(_dh), &_dh);
};
Why haven’t things changed?

1. Attackers circumvent runtime security mechanisms  
   *Stack guards, ASLR, NX, SMEP*

2. Kernels are big, complex, evolving organisms  
   *Formal verification, safe languages, decompose and isolate*

3. Other techniques introduce too much overhead  
   *Decompose and isolate, SFI, strong CFI*
Linus:

“Any time you try to make things be about just security, you’re missing some other part of the equation.”

Boy Genius Report, September 2015
Reconsidering decomposition

- Modular kernel opens up new design possibilities
- Strong isolation more realistic on current hardware
- More rigorous formal verification, testing becomes possible
End goals

1. Performance and capabilities that beats the current kernel
2. Strong isolation of code and resources
3. Explicit resource access control
History

*These focus on security and reliability.*

- Sawmill
  - Abandoned, details not fully published

- Nooks
  - Handwritten wrappers and object tracker code
  - IPC overhead significant for some benchmarks

- User-level device drivers
  - Automation, code reuse

- virtuOS
  - Coarse “vertical” slicing of system stacks
Lessons learned

1. It’s got to be incremental
2. We must start with unmodified code
3. Decomposition must be automated
4. The resulting system must be fast
Outline

- Lightweight Capability Domains
- Breaking the code apart
- Automating Decomposition
- Making it fast
Lightweight Capability Domains
Isolate unmodified code
Benefits

- Trusted, non-isolated code runs as before
- No de-privileging of isolated code, runs in supervisor level
- Isolated address spaces, memory, and devices
Breaking the code apart
Small library kernel for common functions like memcpy and malloc
Shared data

- Use private copies, like Nooks
- Synchronized during cross-domain invocations
Decentralize object tracking

- Domains are “microkernels” for their resources
- Cross-domain references are capabilities
Some domains only use a subset of fields
Use same struct for backward compatibility, but glue code only synchronizes fields in projection
Revisit this in IDL

```
struct inode {
    umode_t i_mode;
    kuid_t i_uid;
    kgid_t i_gid;
    unsigned int i_flags;
    unsigned long i_ino;
    dev_t i_rdev;
    ...
}
```
Automating Decomposition
Scalar functions

- Nothing new here

```go
interface {
    rpc int scalar_func(int a, int b);
}
```
Example

- User module invokes functions in filesystem module to mount a filesystem
The original interface

```c
struct super_block {
    char* name;
    type1 field1;
    type2 field2;
    type3 field3;
    struct block_device* bdev;
};

struct block_device {
    type1 field1;
    type2 field2;
};

// Mount a filesystem
int mount(struct super_block* sb);

// Look up a mount instance
struct super_block* lookup(char* name);
```
Complexities

- Shared objects
- Stateful interaction – user expects pointer to same struct it invoked mount with when it invokes lookup
- Object hierarchy
Projections, revisited

```c
projection super_block <struct super_block> {
    [in] char* name;
    [in,out] type1 field1;
    [in] type2 field2;
    projection block_device <struct block_device> *bdev;
};

projection block_device <struct block_device> {
    [in] type1 field1;
    [out] type2 field2;
};

struct super_block {
    char* name;
    type1 field1;
    type2 field2;
    type3 field3;
    struct block_device *bdev;
};

struct block_device {
    type1 field1;
    type2 field2;
};
```
Declare `mount`

- Uses super block projection

```c
interface {

    rpc int mount(projection super_block
                   <struct super_block> *sb);

}
```
Private copy lifetimes

```c
rpc int mount([alloc(caller)] projection super_block
               <struct super_block> *sb);

projection super_block <struct super_block> {
    ...
    [alloc(caller)] projection block_device
                   <struct block_device> *bdev;
};
```
Future IDL work

- Data structure analysis for assisting with IDL writing
- Security policies?
- Locks
  - Some locking internal to subsystems
  - Other locking is cross-domain
- Object-oriented interfaces nearly complete
Making it fast
Assign subsystems to cores

- Uninterrupted access to CPU
- Improve memory locality
- Passive code → active code
Fast, cross-core async IPC

- Baremetal pipelines
- Minimizes synchronization of threads across domains
- Decentralizes communication
Making async feasible

- Unmodified code not designed for asynchronous function invocations
- Want to minimize number of threads (stacks, etc.) that service a domain
- Use AC language from Barrelfish
  - Cooperative, event-driven execution
  - No stack ripping
do {
    async write();
    async read();
} finish();
Conclusions

- It’s time to try again!
  - Current trends in hardware
  - Abstract and automate decomposition

- Design may not be feasible for embedded systems, but may facilitate re-writing kernel in safe language

- Result: Secure, scalable, modular kernel