Lightweight Capability Domains:
TOWARDS SECURE OPERATING SYSTEM KERNELS

Anton Burtsev
University of California, Irvine
aburtsev@ics.uci.edu
OS kernels are ubiquitous
12,000,000 LoC
40 subsystems
3,200 device drivers
Technology hasn’t changed for decades

- Unsafe C language
- Primitive testing
- No verification
- 50,000 commits a year
Kernel bugs

Logical
- Missing pointer and permission checks
- Buffer and integer overflow
- Uninitialized data
- Null dereference
- Divide by zero
- Infinite loop
- Data races
- Memory mismanagement

Semantic
- High-level security policies
- Protocol violations
Example

```c
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);
};
```

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

- Exploit
- Persistence
- Rootkit
In a modern system, an attacker is one kernel vulnerability away from gaining complete control of the entire machine.

- Not going to change
Can we make our systems secure?
Isolation
Isolation
Isolation

Diagram showing layers of the system with arrows indicating connections and potential vulnerabilities.
Isolation is effective

- Web browsers
- Mashup web pages
- Application containers

- Example: Google Chrome exploit
  - 9 steps
  - 6 vulnerabilities
Isolation is hard

- Commodity kernels are not built for isolation
  - Shared memory
  - Call/return programming model
  - Complex interfaces

- Microkernels tried and failed
  - Massive engineering effort
  - Performance issues
Our goal: decomposed kernel

- Strongly isolated environment
- Explicit access control for each resource
- Reuse of unmodified code
- Fast
Outline

- Patterns of decomposition
  - General techniques to break the kernel apart
- Language support
  - Automating the effort
- Performance
  - Making decomposed environment practically fast
Patterns of Decomposition
Commodity OS: Shared Object Space

- write_inode()
- ext_write_inode()
- submit_bio()
- file
- super block
- block device
LCDs: Isolated object spaces

ext3_write_inode()
Example: kernel interfaces

```c
int register_filesystem(struct file_system_type * fs);

struct super_operations {
    struct inode *(*alloc_inode)(...);
    void (*destroy_inode)(...);
    void (*dirty_inode)(...);
    int (*write_inode)(...);
    ...
};

struct super_block {
    dev_t s_dev;
    unsigned char s_blocksize_bits;
    void *s_fs_info;
    ...
};
```
Patterns of decomposition

- General techniques for decomposing common patterns of kernel code
  - Imported and exported functions
  - Function pointers (interfaces)
  - Shared data structures
Exported functions

```
int foo(int a, int b);
```

Original code

```
interface bar(channel foo_chnl) {
  rpc int foo(int a, int b);
}
```

IDL
int foo(int a, int b) {
    foo_caller(foo_chnl, a, b);
}

int foo_caller(chnl_t *chnl, int a, int b) {
    msg.type = FOO;
    msg.reg[0] = a;
    msg.reg[1] = b;
    ipc_send(chnl, &msg);
}

Generated glue code
Object synchronization

- Explicit subset of fields
- Synchronized upon function invocations

```c
struct inode {
    umode_t i_mode;
    kuid_t i_uid;
    kgid_t i_gid;
    unsigned int i_flags;
    unsigned long i_ino;
    ...
}
```

```idl
projection <struct inode> inode {
    unsigned short [in] i_mode;
    unsigned int [in, out] i_flags;
    unsigned long [in, out] i_ino;
}
```

```c
rpc int foo(projection inode *inode);
```
Remote references

```
void write_inode() {
    container_of(...);
    ...
}
```

```
void write_inode() {
    ...
}
```
Function pointers and interfaces

Virtual File System

File System

Asynchronous Message Channel

Remote Reference

Virtual File System

File System
libKernel: common kernel functions

- Memory management
  - kmalloc()

- Synchronization
  - RCU, spin_lock()

- Common utilities
  - memcpy(), printk()
Access control and information flow
Information flow

```python
foo(in, out)
```
Capabilities

```c
{...
  foo(a,b);
  ...
}

foo(a,b) {
  msg = {FOO, a, b};
  ipc_send(msg);
}

msg = ipc_recv();
...
foo_stub(msg);
```
Language Support
module vfs (channel vfs_chnl) {
    projection <struct fs> fs {
        int [in] id;
        int [in, out] size;
        // Construct new async channel for fs_ops
        channel [alloc(caller, callee)] container->chnl;
    }

    projection fs_ops [alloc(callee)]
        *fs_ops (container->chnl);
}

rpc int register_fs(projection fs [alloc(callee)]
        *fs);
Data Structure Analysis (DSA)

```c
struct super_block {
    struct list_head s_list;
    dev_t s_dev;
    unsigned char s_blocksize_bits;
    unsigned long s_blocksize;
    loff_t s_maxbytes;
    struct file_system_type *s_type;
    const struct super_operations *s_op;
    const struct dquot_operations *dq_op;
    ...
}
```

- Part of LLVM
  - Track objects and fields accessed inside each function
  - Generates suggestions for IDL interfaces
Performance
Domain boundaries are expensive

Round-trip, call/reply invocation

- x86 32bit
  - 661 cycles (3.4Ghz i7) [seL4 team, APSys’15]
  - Maybe ~500 cycles on 64bit with tagged TLBs

- ARM
  - 663 cycles (1Ghz Cortex-A9) [APSys’15]

- x86_64 VT-x non-root to VT-x root
  - 2000 cycles
Insights

- Asynchronous runtime
- Cross-core invocations
Threads vs messages
while (msg = recv(chnl)) {
  dispatch(msg);
}

Message based systems
Problem: blocking I/O
Add more threads?

Application

Virtual File System

File System

Device Drivers
do{
  async foo();
  async bar();
}finish();
async
do{
    async foo();
    async bar();
}finish();
Asynchronous message loop

do{
    while (msg = receive(chnl)) {
        async dispatch(msg);
    }
}finish();

- Remains source code compatible
- Sane programming model
- No stack ripping
Overheads of async

- Macro-based
  - No compiler changes
  - Credit Barrelfish

```c
DO_FINISH(
{
    ASYNC(do_something());
    ASYNC(do_something_else());
});
```

<table>
<thead>
<tr>
<th></th>
<th>Cycles(ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a new async thread</td>
<td>24 cycles</td>
</tr>
<tr>
<td>Context-switch between two threads</td>
<td>83 cycles</td>
</tr>
</tbody>
</table>
Insight: cross-core invocations are faster
Cross-core IPC performance

- Call reply invocation
  - 4 cache coherence transactions
  - Each requires 80 cycles within one socket
- HW pipelining and prefetching for larger message queues

<table>
<thead>
<tr>
<th>Latency tests</th>
<th>Cycles(ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send/receive a message</td>
<td>384 (160)</td>
</tr>
<tr>
<td>Send/receive a message (queue of 4)</td>
<td>159 (67)</td>
</tr>
</tbody>
</table>
Status, conclusions and future work
- Working
  - LCD domains
  - Load and run Linux kernel modules
  - Capabilities, synchronous IPC
- In progress
  - Decomposing the file system interface
  - IDL compiler
  - Async
  - Fast asynchronous IPC
  - Data structure analysis
Conclusions

It’s time to try this again!

- Decomposed kernels are feasible
  - New hardware (cores, multi-queue I/O devices)
  - Much better static analysis tools

- Result
  - Better security
  - Cleaner, more scalable kernels
Revisiting security
Kernels done right (more conclusions)

- Spatial scheduling
  - Core specialization

- Asynchronous cross-core IPC
  - Lightweight on-the-core invocations

- Composable asynchronous I/O
  - Event programming without stack ripping
Decomposed kernels matter in a long run
Practical kernel verification

- Isolated object spaces
- Restricted concurrency
- Explicit interfaces and protocols
Deker: decomposed verified kernels

Joint work with Zvonimir Rakamaric
(University of Utah)
Kernel bypass for low-latency applications

Legacy Applications

VFS

Memory Mgmt

Linux Kernel

LCD Microkernel

Low-latency network application (e.g., key-value store, SDN controller, big-data analytics)

Unmodified NIC driver

I/O MMU
Thank you!

This work is supported by NSF, Google, and NetApp.

aburtsev@ics.uci.edu