Enforcing Kernel Security Invariants with Data Flow Integrity

Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, Wenke Lee

Institute for Information Security & Privacy
Georgia Tech
Kernel Memory Corruption Vulnerability

• Kernel is important
  • The de-facto trusted computing base (TCB)
  • Foundation of upper level security mechanisms (e.g., app sandbox)

• Kernel vulnerabilities are not rare
  • Written in C
  • Emphasize on performance
Privilege escalation attacks

• One of the most powerful attacks
• Most popular attack against kernel
• Hard to prevent
  • Chrome sandbox bypass
  • iOS jailbreak
  • Android rooting
Challenge 1: many ways to exploit

```c
static int acl_permission_check
(struct inode *inode, int mask)
{
    unsigned int mode = inode->i_mode

    if (likely(uid_eq(current_fsuid(), inode->i_uid)))
        mode >>= 6;
    else if (in_group_p(inode->i_gid))
        mode >>= 3;

    if ((mask & ~mode &
         (MAY_READ | MAY_WRITE | MAY_EXEC)) == 0)
        return 0;

    return -EACCES;
}
```

**Code Injection Attack**
Disable the check

**Control-flow hijacking**
Bypass the check

**Data-oriented attacks**
Manipulate the check
Challenge 2: performance

- Protecting all data is not practical
  - Secure Virtual Architecture (SVA) [SOSP’07]
    - Enforces kernel-wide memory safety
    - Performance overhead: 5.34x ~ 13.10x (LMBench)
Our approach

• Only protects a subset of data that is large enough to enforce **access control invariants** [NTIS AD-758 206]
  • Complete mediation
    • **Control-data** → Code Pointer Integrity [OSDI’14]
  • Tamper proof
    • **Non-control-data** used in **security checks** → this work
  • Correctness
Step 1: discover all related data

• Observation: OS kernels have well defined error code for **security checks** (when they fail)
  • POSIX: EPERM, EACCESS, etc.
  • Windows: ERROR_ACCESS_DENIED, etc.

• Solution: leverage this implicit semantic to automatically infer **security checks**

• Benefits
  • **Soundness**: capable of detecting all security related data (as long as there is no semantic errors)
  • **Automated**: no manual annotation required
A simple example

```c
static int acl_permission_check
  (struct inode *inode, int mask)
{
    unsigned int mode = inode->i_mode;

    if (likely(uid_eq(current_fsuid(), inode->i_uid)))
      mode >>= 6;
    else if (in_group_p(inode->i_gid))
      mode >>= 3;

    if ((mask & ~mode &
         (MAY_READ | MAY_WRITE | MAY_EXEC)) == 0)
      return 0;
    return -EACCES;
}
```
A simple example

```c
static int acl_permission_check
    (struct inode *inode, int mask)
{
    unsigned int mode = inode->i_mode;

    if (likely(uid_eq(current_fsuid(), inode->i_uid)))
        mode >>= 6;
    else if (in_group_p(inode->i_gid))
        mode >>= 3;

    if ((mask & ~mode &
         (MAY_READ | MAY_WRITE | MAY_EXEC)) == 0)
        return 0;
    return -EACCES;
}
```
A simple example

Step 2: collect conditional branches

Collect Dominators

```c
if (condition1 || condition2)
    return 0;
else
    return -EACCESS;
```
A simple example

Step 2: collect conditional branches

Avoid Explosion

```c
if (uid_eq)
    mode >> 6;
else
    mode >> 3;

if (condition)
    return -EINVAL;
```
A simple example

```c
static int acl_permission_check
    (struct inode *inode, int mask)
{
    unsigned int mode = inode->i_mode;
    if (likely(uid_eq(current_fsuid(), inode->i_uid)))
        mode >>= 6;
    else if (in_group_p(inode->i_gid))
        mode >>= 3;
    if ((mask & ~mode &
        (MAY_READ | MAY_WRITE | MAY_EXEC)) == 0)
        return 0;
    return -EACCES;
}
```
Be complete

• Collects data- and control-dependencies \textit{transitively}
• Collects sensitive pointers \textit{recursively}
Step 2: protect the integrity of data

- Data-flow integrity [OSDI’06]
  - Runtime data-flow should not deviate from static data-flow graph (similar to control-flow integrity)
    - For example, string should not flow to return address or uid
  - How
    - Check the last writer at every memory read
  - Challenge
    - Performance! (104%)
How to reduce performance overhead

• Observation 1: reads are more frequent than writes
  • Check write instead of read

• Observation 2: most writes are not relevant
  • Use isolation instead of inlined checks

• Observation 3: most relevant write are safe
  • Use static analysis to verify

Write Integrity Test [S&P’08]
Two-layered protection

• Layer one: data-flow isolation
  • Prevents unrelated writes from tampering sensitive data
  • Mechanisms: segment (x86-32), WP flag (x86-64), access domain (ARM32), virtual address space, virtualization, TrustZone, etc.

• Layer two: WIT
  • Prevents related but unrestricted writes from tampering sensitive data
Additional building blocks

• Shadow objects
  • Lacks fine-grained isolation mechanisms
  • Sensitive data is mixed with non-sensitive data

• Safe stack
  • Certain critical data is no visible at language level, e.g., return address, register spills
  • Access pattern of stack is different
  • Safety is easier to verify
Prototype

• ARM64 Android
  • For its practical importance and long updating cycle
  • Enough entropy for stack randomization

• Data-flow isolation
  • Heap: virtual address space based, uses ASID to reduce overhead
  • Stack: randomization based

• Shadow objects
  • Modified the SLUB allocator
Implementation

• Kernel
  • Nexus 9 lollipop-release + LLVMLinux patches
  • Our modifications: 1900 LoC

• Static Analysis
  • Framework: KINT [OSDI’12]
  • Point-to analysis: J. Chen’s field-sens [GitHub]
    • Context sensitive from KOP [CCS’09]
  • Safe stack: CPI [OSDI’14]
  • Our analysis + modifications: 4400 LoC
  • Instrumentation: 500 LoC
How many sensitive data structures

- Control data: 3699 fields (783 structs), 1490 global objects
- Non-control data: **1731** fields (**855** structs), **279** global objects
  - False positives: 491 fields (221 structs) / 61 fields (25 structs)
How secure is our approach

• Inference
  • Sound $\rightarrow$ no false negatives
  • Catch: no semantic errors

• Data-flow (point-to) analysis
  • Sound but not complete $\rightarrow$ over permissive
  • Improve the accuracy with context and field sensitivity

• Against existing attacks
  • All prevented
Performance impact

• Write operations
  • 26645 (4.30%) allowed, 2 checked

• Context switch
  • 1700 cycles

• Benchmarks
  • LMBench (syscalls): 1.42x ~ 3.13x (0% for null syscall)
  • Android benchmarks: 7% ~ 15%
Conclusion

- Data-oriented attacks are very practical, especially in kernel
- Leveraging implicit semantics to avoid annotation
- Combining program analysis with system design is a great way to build principled and practical security solution
Thank you!

Q & A