Improving Integer Security for Systems with KINT

Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, Frans Kaashoek
MIT CSAIL Tsinghua IIIS
Integer error

- Expected result goes out of bounds

- Math (∞-bit): \(2^{30} \times 2^3 = 2^{33}\)

- Machine (32-bit): \(2^{30} \times 2^3 = 0\)

- Can be exploited by attackers
Example: Buffer Overflow

- Array allocation
- `malloc(n * size)`
- Overflow: $2^{30} \times 2^3 = 0$
- Smaller buffer than expected

- Memory corruption that leads to privilege escalation. One famous example is the iPhone jailbreak (CVE-2011-0226)
Example: logical bug

- Linux kernel OOM killer (CVE-2011-4097)

Computes “memory usage score” for a process and then kills the one with the highest score.

- Score is calculated by:

\[
\text{nr\_pages} \times 1000 \div \text{nr\_totalpages}
\]

- A malicious process can be consuming too much memory, while having a low score and trick the kernel into killing some other process.
Why integer errors become a threat

- 2007 CVE survey:
  “Integer overflows, barely in the top 10 overall in the past few years, are number 2 for OS vendor advisories, behind buffer overflows.”

- 2010-early 2011 CVE survey: Linux kernel
  More than 1/3 of serious bugs are integer errors
Why is it hard to prevent integer errors

- **Arbitrary-precision integers (Python/Ruby)**
  - Performance: require dynamic storage
  - Their implementation (in C) have overflows
- **Trap on every overflow**
  - False positives: overflow checks intentionally incur overflow
  - Linux kernel requires overflow to boot up
- **Memory-safe languages (C#/Java)**
  - Performance concerns: runtime checks
  - Not enough: integer errors show up in logical bugs
Contributions

- A case study of 114 bugs in the Linux kernel
- **KINT**: a static analysis tool for C programs
  - Used to find the 114 bugs
- **kmalloc_array**: overflow aware allocation API
- **NaN integer**: automated overflow checking
Case study: Linux kernel

- Applied KINT to Linux kernel source code
  - November 2011 to April 2012
- Inspect KINT's bug reports & submit patches
- 114 bugs found by KINT
  - confirmed and fixed by developers
- 105 exclusively found by KINT
- 9 simultaneously found by other developers
Most are memory and logic bugs

- Error analysis showed that:
  - 42% of the total bugs were logical.
  - 37% of the total bugs were buffer overflows.
  - 21% of the total bugs were other kind of bugs.

- 67% of the above errors had incorrect checks.
Example: wrong bounds

net/core/net-sysfs.c

Struct flow_table {
...
struct flow entries[0];
};

unsigned long n=/*from user space*/;
if(n>1<<30) return -EINVAL;
table = vmalloc ( sizeof ( struct flow_table) +
    n * sizeof ( struct flow ) );
for(i=0;i<n;++i)
table->entries[i] =....;
Example: wrong type

Drivers/gpu/drm/vmwgfx/vmwgfx_kms.c

    u32 pitch = /*from user space*/
    u32 height = /*from user space*/

- Patched once:
  
    u32 size = pitch \times height;
    if (size > vram_size) return;

- Patched twice:
  
    u64 size = pitch \times height;
    if (size > vram_size) return;

- Patched for a 3\textsuperscript{rd} and final time:
  
    u64 size = (u64)pitch \times (u64)height;
    if( size > vram_size) return;
Writing correct checks is non-trivial

- 2/3 of the 114 integer errors have checks
- One check was fixed 3 times and still buggy
- Even two CVE cases were fixed incorrectly
  - Each received extensive review

So.... how do we find them???
Finding integer errors

- Random testing... is not enough
  - Low coverage as it's hard to trigger corner cases
- Symbolic model checking
  - Path explosion
- Environment modeling
- KINT: static analysis for bug detection
KINT Overview

- LLVM IR (from C code)
- Range analysis (whole-program)
- Taint analysis (whole-program)
- User annotations
- Per-function analysis
- Solving & classification
- Possible bugs
Per-function analysis
Solving boolean constraints
KINT Overview
Checks in caller
A whole-program range analysis
Taint analysis for bug classification
KINT Implementation
KINT Usage
Evaluation

- Effectiveness in finding new bugs
- False negatives (missed errors)
- False positives (not real errors)
- Time to analyze Linux kernel
KINT finds new bugs
KINT finds most known integer errors
False positives (CVE)
False positives (entire kernel)

- Linux kernel 3.4-rc1 in April 2012
- 125,172 possible bugs in total
- 741 ranked as “risky”
  - Allocation size computed from untrusted input
- Skimmed the 741 bugs in 5 hours
- Found 11 real bugs
- We don’t know if the rest are real bugs
KINT analysis time
Summary of finding bugs with KINT
Mitigating allocation size overflow
Generalized approach: NaN integer
Verbose manual check
NaN integer example
Conclusion
Finale

- THE
  - END