Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences

Kangjie Lu, Aditya Pakki, Qiushi Wu

28th USENIX Security Symposium, August 16, 2019
Contributions

- **Missing-check** bug detection for OS kernels
  - Scalable and context-aware interprocedural static analysis techniques
- Identification of critical variables, peers, and indirect-call targets (additional 93% reduction)
- **278** new bugs in Linux 4.20
  - 151 patches confirmed with 134 in mainline
Security checks safeguard the OS kernel state

- Kernels are critical, complex, and error prone
- Developers enforce numerous (>400k) security checks
- What is a security check? [LRSan CCS’18]
  - Conditional statements (e.g., if statement) with at least two branches
    - At least one branch has error handling and
    - At least one branch does NOT have error handling
  
  
  Not a security check if all branches have or do not have error handling
Common classes of security checks

/* Input validation */
res = get_user(size, buf);
if (size > MAX)
    return -EINVAL;

/* Check operation result */
*vaddr = pic_alloc(...);
if (*vaddr == NULL)
    return -ENOMEM;

/* Missing permission check */
if (!access_ok(VERIFY_WRITE, addr, size))
    return -EFAULT;

/* Check system state */
if (!PGE_EMPTY(agp_bridge))
    return -EBUSY;
Who guards the guardians?

- Security checks themselves are buggy
- The most common case is - a security check itself is missing
Missing-check bugs: Not enforcing a required security check on critical variable

Source of critical variable

Present or absent?

Intended security check

Use of critical variable
Importance of detecting missing-check bugs

- Adding/updating checks constitute 59.5% of vulnerability patches
- Security impacts of missing-check
  - Denial of Service
  - Memory Corruption
  - Information leakage
  - ...

Possible detection approaches and challenges

● Rule-based approach to identify these bugs
  ○ Challenges
    ■ Require semantic understanding
    ■ Hard to generalize
● Cross-checking (i.e., statistical analysis)
  ○ Our choice
The idea of cross-checking

- Statistical model that avoids computing ground truth
  - **Majority** decision is applied to the group
  - **Minority** cases are likely bugs
  - **Assumption:** majority kernel code is good
- 9 out of 10 doors use deadbolts for security
  - A door without deadbolt is likely unsecure
Challenges in cross-checking

- **Scalability**: Can’t cross check every variable
  - Focus on critical variables only
- **Similarity**: Generate statistically significant peers
  - Find sufficient semantically-similar code
- **Granularity**: Optimize the comparison levels
  - Not too coarse-grained or too fine-grained
High-level overview of Crix

- **Crix -** Criticalness and constraints Inferences for detecting missing checks

**Identify critical variables** ➔ **Find peers** ➔ **Model checks** ➔ **Detect deviations as bugs**
Critical variable identification solves scalability

**Insights** (1) a variable is critical if it is validated in a security check; (2) Checked variables can propagate criticalness

- Collect checked variables as initial seed
- Collect sources and propagated variables of each critical variable

```c
// Allocate a netlink msg
skb = genlmsg_new(...);
// Allocation success check
if (!skb)
    return -ENOMEM;
// skb criticality propagated
nla_put(skb,...);
```
Tackling similarity by identifying peers in kernels

- **Requirement:** A large set; similar context & semantics
- **Observation:** indirect calls, return inst, direct callers generate peers
- **Approach:** Slice from critical variable to src & use

![Diagram](image)
Precisely identifying indirect call targets

- **Challenge:** scalability, callgraph precision
- Indirect calls peers share similar arguments
  - Count & type
- Currently, indirect call targets are identified via
  - Points-to analysis or Function-type matching
- Our new approach - **Two-layer type analysis**
Two-layer Type Analysis for accurate indirect call peers identification

- First layer - function type matching
  - \texttt{add(int a, int b)} vs \texttt{add1(int a, int b, int c)}

- Second layer - struct type matching
  - Function pointers are stored in a struct field &
  - Loaded from this struct field during dereference

- Uses escape analysis for soundness
Cross checking peers to detect deviations

- Use global threshold, Relative Frequency (RF = 0.15)

\[
RF = \frac{N_{nc}}{N_t}
\]

- RF: ratio of slices missing a check (\(N_{nc}\)) to total number of slices (\(N_t\))
- RF determined via empirical study of security patches
Implementation of Crix

- Multiple LLVM 8.0 passes
- 4.5K lines C++ code
- 64 minutes to complete
- 17,343 modules for x86 *allyesconfig*
- Uses threshold (RF) to prioritize 804 bugs
  - Ranking as a heuristic
Evaluating Crix on Linux Kernel 4.20

- 278 new bugs
  - 134 applied to mainline
  - 99 bugs fixed within one week of submission
  - 195 bugs in driver modules
    - 27 driver modules have >1 bug
  - Latent period of 4 years 7 months
    - 10% have latent period over 10 years
Besides Alias Analysis, Crix has more limitations

- Context determination is not comprehensive
  - In error paths, missing checks are often considered unnecessary

https://www.nationalgeographic.org/media/sinking-of-the-titanic/
Conclusion

- Security checks are critical but buggy
- Finding, modeling, cross-checking peer slices for semantic- & context-aware detection
- 93% more reduction in indirect call targets compared to existing techniques
- Code @ [https://github.com/umnsec/crix](https://github.com/umnsec/crix)