Lecture Notes:
Unleashing MAYHEM on Binary Code

Rui Zhang

February 22, 2017

1 Finding Exploitable Bugs

1.1 Main Challenge in Exploit Generation

Exploring enough of the state space of an application to find exploitable paths.

1.2 Four main principles

1. make forward progress for arbitrarily long times
2. should not repeat work
3. should not throw away any work
4. reason about symbolic memory

1.3 Key points for finding exploitable bugs

1. Low-level details matter (like return addresses, stack pointers, motivation for focusing on binary-level techniques)
2. Enormous number of paths (if statement, long input)
3. The more checked paths, the better
4. Execute as much natively as possible

2 Background

2.1 Offline symbolic execution

1. Reason about a single execution path at a time
2. Satisfy principle 1: iteratively picking new paths to explore
3. Satisfy principle 3: every run is independent and can be immediately reused
4. Does not satisfy 2: every run needs to restart execution from the very beginning

5. Drawback: for every explored execution path, we need to first re-execute a very large number of instructions until we reach the symbolic condition where execution forked, and then begin to explore new instructions

2.2 Online executors

1. Execute all possible paths in a single run of the system
2. Forks at each branch point, previous instructions never re-executed
3. Principle 1 and 3 are not met, interesting paths are potentially eliminated
4. Drawback: quickly strain the memory, causing the entire system to grind to a halt

2.3 Symbolic Index Modeling

1. A symbolic index occurs when the index used in a memory lookup is not a number, but an expression
2. Previous tools:
    - Concretizing the index: reduce complexity, but may cause us to mis paths, natural choice for offline executors
    - Allowing memory to be fully symbolic: formulas involving symbolic memory are more expressive, solving/exploration times are usually higher

3 Design of MAYHEM

3.1 Combine Concrete Execution and Symbolic Execution

1. Concrete Executor Client (CEC)
    - Takes in a binary program along with the potential symbolic sources (input specification) as a input, begins communication with the SES
    - Handles multiple concrete execution states simultaneously, execution state includes current register context, memory and OS state (snapshot of the virtual file system, network and kernel state)
    - Context switches between different concrete execution states depending on the symbolic executor
    - Virtualization layer mediates all system calls to the host OS and emulates them
2. Symbolic Executor Server (SES)
    - Symbolically executes blocks that the CEC sends, outputs several types of test cases including normal test cases, crashes, exploits
    - Manages the symbolic execution environment and decides which paths are executed by the CEC

- Maintains two contexts: variable context (all symbolic register values and temporaries); a memory context (keeping track of all symbolic data in memory)
- Keep a priority queue for path selector
- Preconditioned symbolic execution: a user can give a partial specification of the input to reduce the range of search space
- Path selection (three heuristic ranking rules, designed to prioritize paths that are most likely to contain a bug)
  - Executors exploring new code have high priority
  - Executors that identify symbolic memory accesses have higher priority
  - Execution paths where symbolic instruction pointers are detected have the highest priority

3. Steps followed by MAYHEM
   (a) Specify which input sources are potentially under attacker control
   (b) Initialization; CEC instruments the code and perform dynamic taint analysis (checks if a block contains tainted instruction, where a block is a sequence of instructions that ends with a conditional jump or a call instruction)
   (c) CEC encounters a tainted branch condition or jump target, it suspends concrete execution. CEC sends the instructions to the SES, SES determines which branches are feasible
   (d) SES jits the instructions to an intermediate language, symoblically executes the corresponding IL, maintains two types of formulas
       - Path Formula: reflects the constraints to reach a particular line of code
       - Exploitability Formula: determines whether the attacker can gain control of the instruction pointer and execute a payload
   (e) When MAYHEM hits a tainted branch point, the SES queries the SMT solver and decides whether to fork execution or not. New forks are sent to the path selector to be prioritized, and corresponding execution state is restored
   (f) SES switches context between executors, CEC checkpoints/restores the provided execution state and continues execution
   (g) When MAYHEM detects a tainted jump instruction, it builds an exploitability formula and queries an SMT solver
   (h) The above steps are performed until an exploitable bug is found, MAYHEM hits a user-specified maximum runtime, or all paths are exhausted.

3.2 Hybrid Symbolic Execution

1. Instead of running in pure online or offline execution mode, MAYHEM can alternate between modes.

2. Crux: distribute the online execution tasks into subtasks without losing potentially interesting
3. Overview:
   - Execution alternates between online and offline symbolic execution runs
   - When memory is under pressure, the hybrid engine picks a running executor, saves the current execution state and path formula. The thread is restored by restoring the formula, concretely running the program up to the previous execution state
   - Efficiently reasoning about symbolic memory
   - Generates exploits for several security vulnerabilities: buffer overflows, function pointer overwrites, format string vulnerabilities
4. Four main phases:
   (a) Initialization: Starts analysis in online mode
   (b) Online Exploration: Symbolically executes the program in an online fashion, context-switching between current active execution states, and generating test cases
   (c) Checkpointing: When the system reaches a memory cap, it switches to offline mode. It will select and generate a checkpoint for an active executor.
   (d) Checkpoint Restoration: The checkpoint manager selects a checkpoint based on a ranking heuristic and restores it in memory.
5. Performance tuning:
   - Independent formula
   - Algebraic simplifications
   - Taint analysis

3.3 Memory modeling in MAYHEM

1. 40% examples require handling symbolic memory, simple concretization was insufficient
2. MAYHEM models memory partially
3. Memory Objects: Whenever a symbolic index is used to read memory, MAYHEM generates a fresh memory object M that contains all values that could be accessed by the index – M is a partial snapshot of the global memory
4. Memory Object Bounds Resolution
   (a) Worst case: require up to 2^{32} queries to the solver
   (b) Resolve the bounds of the memory region, the bounds need to be conservative , but not continuous

    (c) Use solver to perform binary search

5. Problems
    (a) Querying the solver on every symbolic memory dereference is expensive
    (b) Memory region may not be continuous
    (c) The values within the memory object might have structure
    (d) In the worst case, a symbolic index may access any possible location in memory

6. Optimization
    (a) Value Set Analysis: returns a strided interval for the given symbolic index, then refined by the solver
    (b) Refinement Cache: keeps a cache mapping intervals to potential refinements
    (c) Lemma Cache: uses another level of caching to avoid repeatedly querying a-equivalent formulas
    (d) Index Search Trees: replaces memory object lookup expressions with index search trees
    (e) Bucketization with Linear Functions: an extra preprocessing step before passing the object to the IST. The idea is to use the memory object structure to combine multiple entries into a single bucket

7. Prioritized Concretization
    (a) Above optimizations are only activated when the size is below a threshold
    (b) When the memory object size exceeds the threshold, MAYHEM will concretize the index used to access it
    (c) Prioritize the possible concretization values
        - check if it is possible to redirect the pointer to unmapped memory
        - check if it is possible to redirect the symbolic pointer to symbolic data

4 Exploit Generation

1. MAYHEM checks two exploitable properties: a symbolic (tainted) instruction pointer (buffer overflow), and a symbolic format string (format string attack)

2. Can generate both local and remote attacks

5 Discussion

1. Do not bypass OS defenses such as ASLR and DEP

2. Limitations: does not have models for all system/library calls; only analyze a single execution thread on every run; only executes tainted instructions

6 Thoughts

Exploit generation is a straightforward application of symbolic execution. This paper's main contribution is to leverage the state-of-the-art techniques and also provide to their own solutions to improve the performance in exploit generation. Their efforts made the symbolic execution of real-world programs feasible.

7 Questions

Their methods seem to be tuned for buffer overflow and format string attacks. If we need to add other types of attacks into MAYHEM, what efforts will be involved?