(State of) The Art of War
Offensive Techniques in Binary Analysis

Authored By:
Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, Giovanni Vigna
UC Santa Barbara

Presented By:
Nitesh Dagli and Subin Samuel
Introduction

- Two major issues in binary analysis at the time of this paper
  - Many binary analysis implementations don’t ever leave the research phase
  - Amount of work required to reproduce systems and unavailability to the public
- So how do we solve this?
- The answer to the first problem is angr
- The answer to the second problem is the DARPA hosted Cyber Grand Challenge(CGC)
Cyber Grand Challenge

● DARPA Cyber Grand Challenge 2016
● How does the CGC solve our second problem?
● Top 7 of Qualifying round: $750K:
● Finals: First: $2M, Second: $1M, Third: $750k
● Autonomously:
  ○ Identify
  ○ Exploit
  ○ Patch vulnerabilities
● DECREE OS - Simple OS that has 7 system calls
  ○ transmit, receive, waitfd, random, allocate, deallocate, and terminate.
Automated Binary Analysis

- Very difficult to implement and deploy in the real world
- In order to maintain feasibility of offensive binary analyses, trade-offs have to be made
- Two major trade-offs to be made
  - Replayability
  - Semantic insight
- Attempting both Replayability and high amount of semantic insight suffer from scalability.
- Offensive Binary Analysis techniques
Static Techniques reason without executing

Two major drawbacks
- Results not replayable
- Lower semantic insight

Must be able to recover a CFG
- One major challenge: Indirect Jumps
- Another is code coverage

Three types of indirect jumps
- Computed, Context-sensitive, and Object-sensitive

CFG recovery analysis properties
- Soundness
- Completeness
Dynamic Vulnerability Discovery

- Two main categories
  - Concrete and Symbolic execution
- Dynamic Concrete execution
  - Main application is fuzzing
- Types of fuzzing
  - Coverage based fuzzing
  - Taint-based fuzzing
- Dynamic Symbolic Exploitation
  - Will track state of registers and memory, high semantic insight
  - One main issue: Path Explosion
- Symbolic-assisted Fuzzing
  - Attempts to answer the path explosion problem
- Under-Constrained symbolic execution
  - Effective at finding potential bugs but suffers from false positives and no replayability
Exploitation

- Three steps for exploitation
  - Reproduce an identified crash
  - Generating an exploit to verify the impact of the crash
  - Hardening the exploit
- Crash reproduction is very tricky
  - Many fuzzers will hardcode sources of randomness
- Two cases of non-trivially replayable inputs
  - Missing Data and Missing relationships
- Exploit Generation
  - Not all found crashes are exploitable
- Exploit Hardening
  - DEP and ASLR are rendering automatic exploit generates useless
  - Convert these exploits into a ROP equivalent
Angr was created in order to implement many of the analyses discussed in the previous sections.

Design Goals for angr:
- Cross-architecture support
- Cross-platform support
- Support for Different types of analysis
- Usability
- Multiple submodules to support implementation of various analyses

Submodule: Intermediate Representation:
- Supports multiple architectures
- Uses libVEX, the IR lifter of Valgrind

Submodule: Binary Loading:
- Handled by a CLE module
Analysis Engine cont...

- Submodule: Program State Representation/Modification
  - SimuVEX module will represent the program state
  - SimuState is implemented as a collection of state plugins
  - Plugins are:
    - Registers, Symbolic memory, Abstract memory, POSIX, Log, Inspection, Solver, and Architecture
Analysis Engine cont...

- Submodule: Data Model
  - Register and memory values are represented by Claripy
  - Claripy supports the concrete domain, symbolic domain and the value-set abstract domain

- Submodule: Full-Program Analysis
  - Creates a Project object, from which all other submodules can be accessed.

- Two main interfaces
  - Path Groups
  - Analyses

- Angr is Open Source!
  - Over 65,000 lines of code
  - Useable via IPython shell or as a python module
  - Is part of pip
Implementation: Control Flow Graph Recovery

CFGAccurate
- Forced Execution: Identifies basic blocks and direct jumps.
- Symbolic Execution: Backtrack to merge point then perform symbolic execution into indirect jump.
- Backward Slicing: Context-sensitive. Backtrack to previous call context then perform symbolic execution into indirect jump.

CFGFast
- High code coverage(blocks) without concern for reachability of functions(jumps)
- Uses recursive disassembly and heuristics to solve some cases
Implementation: Value Set Analysis

- Static Analysis Technique
- Combines numeric analysis + pointer analysis
- Until approximate possible values of registers/abstract locations at every point in the program.
- Improvements on original design:
  - Set of strided-intervals
  - Algebraic solver
  - Signedness-agnostic domain
- Used VSA for memory corruption detection in 3 phases:
  - Variable recovery
  - Finds abnormal buffers (overlapping, out-of-bounds)
  - All abnormal buffers are potential memory corruptions
Implementation: Dynamic Symbolic Execution

- Based off of Mayhem, binary analysis tool.
- Uses Claripy, angr’s solver engine, which interfaces with Z3 to populate symbolic memory
- Each execution path is a *Path* object. Groups of these are *PathGroup*
- Uses *Veritesting* to merge paths to try to prevent/mitigate path explosion
Implementation: Under-Constrained Symbolic Execution

- Analysis on each function separately (Contextless and non-replayable)
- Tags missing contexts in the state as under-constrained
- If a security violation shows up but all of the values are under-constrained, this violation is a false positive
- Improved the original design by:
  - All global data as under-constrained
  - Hard-coded a path depth limit to prevent path explosions (64 paths)
  - False-positive filtering
Implementation: Symbolic-Assisted Fuzzing

- Full approach: Driller.
- American Fuzzy Lop and angr
- If the AFL gets no progress after a round of mutations, then angr is invoked on all paths that are deemed unique
- When AFL is stuck, angr checks the constraints to new indirect jump and provides starting value to AFL to start mutating
- Balances expensive symbolic execution time with cheap fuzzing time
- Mitigates fuzzing’s low semantic insight
Implementation: Crash Reproduction

- Implemented Replayer: recovers missing relationships between input and output.
- Symbolic Execution of path from start state to crash state to find input constraints of path.
- Two main limitations:
  - Some crashes don’t get all of the data/context to replay the crash.
  - Replayer follows exact path using crashing inputs. May be problems if some random number determines a different path to get to the same crash state.
Implementation: Exploit Generation

• Concolic Execution: Hybrid-Symbolic Execution on concrete paths.
• Reaches a crash state, can inspect the symbolic state to measure exploitability.
• *Instruction pointer Overwrite*. If symbolic bits are found at instruction pointer at crash, we can constrain the instruction pointer to point at controlled sequence of instructions (shellcode).
• For CGC there are 2 types of exploits:
  ○ Controls general purpose register + instruction pointer register
  ○ Controlled read from process memory space
• Out of 126 binaries - they exploited 4
• Main challenge: They found many vulnerabilities but few exploitable ones.
Implementation: Exploit Hardening

- Implemented ROP chain compiler based on the ideas in Q.

Approach:
- Gadget discovery: scan executable code then at every byte offset to identify ROP gadget.
- Gadget arrangements: ROP chain compiler determines arrangements of ROP gadgets to perform high-level actions.
- Payload generation: combine gadgets into a chain to perform higher-level actions.

Improvements from Q
- Uses stack as scratch storage space.
- Q used value sampling method to identify ROP gadgets - misses gadget. angr symbolically analyzed every gadget with caching techniques to keep analysis fast.
Comparative Evaluation

- First of a kind: previous comparisons of binary analysis techniques were on different implementations
- (Except AFL) All analyses are implemented on the same analysis engine and share over 90% of the same code
- Use CGC binaries, to keep the environment the same + simple
Comparative Evaluation: CFG Recovery

- Compared \textit{CFGAccurate} and \textit{CFGFast} with state-of-the-art commercial tool IDA Pro 6.9
- IDA is closed source: believe it recovers CFG by symbols+heuristics to determine functions, and lightweight data-flow analyses to solve for targets of indirect jumps.
- Measure results by recovered basic blocks and control flow transfers
  - \textit{CFGFast}
    - CFGFast code coverage \( > \) (slightly) IDA Pro
    - CFGFast better edge coverage than IDA
    - CFGFast \( >> \) IDA 😎
  - \textit{CFGAccurate} code coverage \( < \) IDA’s.
## Angr vs IDA Pro’s CFG’s

<table>
<thead>
<tr>
<th>Approach</th>
<th>Functions</th>
<th>Function Edges</th>
<th>Blocks</th>
<th>Block Edges</th>
<th>Bytes</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M</td>
<td>A</td>
<td>M</td>
<td>A</td>
<td>M</td>
<td>A</td>
</tr>
<tr>
<td>IDA Pro 6.9</td>
<td>48</td>
<td>52.96</td>
<td>76.5</td>
<td>99.62</td>
<td>829</td>
<td>3589.93</td>
</tr>
<tr>
<td>angr - CFGFast</td>
<td>61</td>
<td>70.08</td>
<td>88</td>
<td>118.74</td>
<td>843</td>
<td>3609.45</td>
</tr>
<tr>
<td>IDA Pro 6.9 - reachability</td>
<td>37</td>
<td>40.96</td>
<td>74</td>
<td>90.76</td>
<td>496</td>
<td>1043.81</td>
</tr>
<tr>
<td>angr - forced execution</td>
<td>31</td>
<td>33.24</td>
<td>48</td>
<td>55.22</td>
<td>349.5</td>
<td>413.85</td>
</tr>
<tr>
<td>angr - symbolic back traversal</td>
<td>32</td>
<td>33.76</td>
<td>50</td>
<td>56.28</td>
<td>368</td>
<td>635.41</td>
</tr>
<tr>
<td>angr - backward slicing</td>
<td>30</td>
<td>32.80</td>
<td>47.5</td>
<td>53.89</td>
<td>344.5</td>
<td>653.56</td>
</tr>
</tbody>
</table>

**TABLE II**

**Evaluation of CFGFast’s and CFGAccurate’s recovered CFG versus the CFG recovered by IDA Pro.** The median number (M) and average number (A) of each value across all binaries are shown.
Comparative Evaluation: Vulnerability Analysis Techniques

Dynamic Symbolic Execution:

- Regular DSE: 16 vulnerabilities
- Veritesting found 11.
- Tradeoff:
  - Path merging introduces complex expressions (when 2 paths merge a register for creates a complex expression) which overloads the constraints solver.
  - Veritesting finds shallow bugs overwhelms the constraint solver for longer paths

Symbolic-assisted fuzzing

- Individual inputs traced by DSE don’t branch
- AFL filters input that doesn’t increase code coverage
- Symbolic-assisted fuzzer: 77 vulnerabilities
- AFL: 68.
Comparative Evaluation: Vulnerability Analysis Techniques

DSE vs fuzzing
- Fuzzing found 3x more vulnerabilities.
- DSE is good at finding shallow crashes.

Under-constrained symbolic execution
- Contextless analysis
- Not replayable and high false positives
- False positive rate of 93%.

Static buffer overlap detection:
- VSA-based memory corruption.
- Not replayable and high false positives
- False positive rate of 82.8%
Non-replayable vs Replayable analyses

- Surprisingly low performance of non-replayable techniques.
- Expected can achieve more coverage.
- Instead context they lacked lead to high false positives.
- Implemented aggressive false positive filtering (ended up filtering true positives as well).
<table>
<thead>
<tr>
<th>Technique</th>
<th>Replayable</th>
<th>Semantic Insight</th>
<th>Scalability</th>
<th>Crashes</th>
<th>False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Symbolic Execution</td>
<td>Yes</td>
<td>High</td>
<td>Low</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>Veritesting</td>
<td>Yes</td>
<td>High</td>
<td>Medium</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>Dynamic Symbolic Execution + Veritesting</td>
<td>Yes</td>
<td>High</td>
<td>Medium</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>Fuzzing (AFL)</td>
<td>Yes</td>
<td>Low</td>
<td>High</td>
<td>68</td>
<td>0</td>
</tr>
<tr>
<td>Symbolic-Assisted Fuzzing</td>
<td>Yes</td>
<td>High</td>
<td>High</td>
<td>77</td>
<td>0</td>
</tr>
<tr>
<td>VSA</td>
<td>No</td>
<td>Medium</td>
<td>High</td>
<td>27</td>
<td>130</td>
</tr>
<tr>
<td>Under-constrained Symbolic Execution</td>
<td>No</td>
<td>High</td>
<td>High</td>
<td>25</td>
<td>346</td>
</tr>
</tbody>
</table>

**TABLE IV**

Evaluation results across all vulnerability discovery techniques.
Critique

- Still have a long long way to go
- Angrs future value
  - Angr is open source
  - Can implement techniques quite easily on the framework
- Many of the more complicated analyses options ended up being worse
  - AFL > DSE
  - CFGFast > IDA Pro > CFGAccurate
Conclusions

- The main focus of this paper is broken up into three parts
  - First is the discussion on various binary analysis techniques focusing on vulnerability detection
  - Second is on the binary analysis system called angr which implements said techniques
  - Third is on the comparison of the various implementations through angr