REVERY
FROM POC TO EXPLOITABLE

Rafael

INTRODUCTION TO THIS PAPER

- CCS 18
- One step towards Automatic Exploit Generation (AEG)
  based on Angr
- For AEG
  generation working exploit
- Contribution
  transfer PoC input to Exploit input
  layout-contributor digraph
  layout-oriented fuzzing
  control-flow stitching
- No need for Source code or debug symbols
- Limited case: ASLR
  need extra information

WHY WE NEED AEG?

- Automatic accessment of Vulnerabilities
- Take priority on which could be exploited

HISTORY FOR AEG

- Based on patch analysis
  S&P 2008, APEG
- Dynamic analysis && Constraint Solver
  Sean Healan, 2019
- Based on Source code && Symbolic Execution && Dynamic analysis
  NDSS 2011, AEG
- Based on Binary && Symbolic Execution && Dynamic analysis
  S&P 2012, Mayhem
- Based on crash analysis
  SERE 2012, CRAX
- Data-Only
  USENIX Security 2015, FlowStitch

HISTORY FOR AEG

- DARPA Cyber Grand Challenge (CGC)
  released in 2013
  started in 2014
  qualified-test in 2015
  final in 2016
  First prize attended the Defcon CTF 2016
- RHG in 2017
  based on Linux
- DEFCON China in 2018
  based on Linux

AEG

- Analyze crashing path with dynamic analysis
- Use symbolic execution to collect cnstraints of exploitable states
- Solve with constraint solver
  generate possible input
- Analysis and Exploit Automation [1]
  Compilers (Program transformation)
  Fuzz testers (Input generation)
  SMT solvers (Symbolic reasoning)
  Model checkers (state space exploration)
  Symbolic Execution (path generation)
  Emulators (Machine modeling)
  Abstract interpreters (Abstraction)

PROBLEMS FOR AEG [1]

- Exploit Specification problem (A, H)
  H. Privilege Separation Inference
- Input generation problems (B, C, D, E)
  B. Pre/post-conditions inference
  C. Loop assertion inference
  D. Exploit input definability
  E. Expliot derivability
- Exploit primitive composition problem (F)
  F. Multi-interaction exploit
- Environment determination (I, J, K)
  I. Heap likelihood inference
  J. Generalized program timing attack
  K. Indirect information disclosures
- State space representation (G)
  G. Multi-concurrent exploit

Of course not all problems need to be resolved,
but different problems in different scenarios

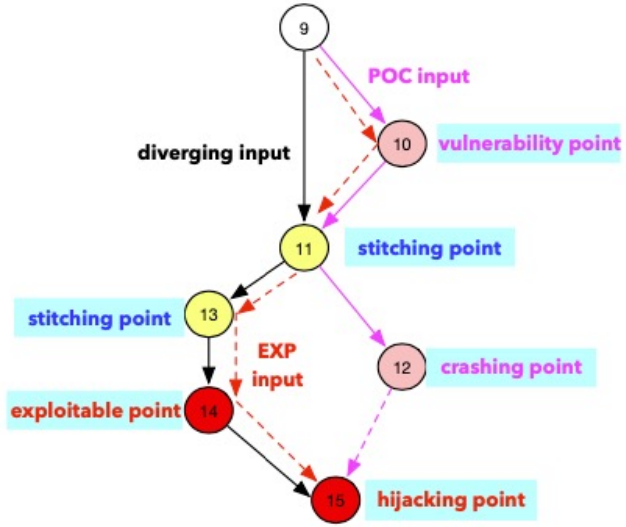
CHALLENGES RESOLVED IN THIS PAPER

- Exploit derivability [1]
- Symbolic execution bottleneck
  path explosion
  concretized values could lead to non-exploitable states
- Exploiting heap-based vulnerabilities
  too complicated for program analysis

TRIGGER VUL V.S. EXPLOIT VUL

- To trigger vulnerabilities
  path constraints
  vulnerabilities constraints
- To exploit vulnerabilities
  shellcode constraints
  EIP constraints
  memory alignment constraints
  Bypass constraints

EXPLOIT DERIVABILITY

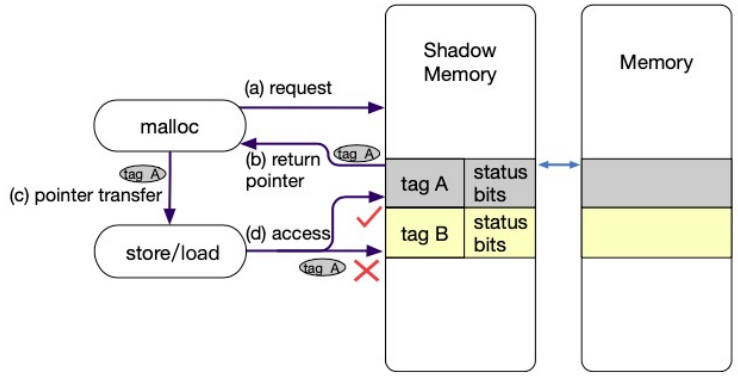


- Given PoC input,
  turned into Weird Machine (WM) with non-exploitable ini state
  we need to search for exploit in diverging path
- Given a concrete program input
  and associated program crash log,
  find the longest crash trace prefix from which the desired exploitable program state can be reached.
- What are stitching points?
  (x + y) > 4 and (x + y) <= 4
- Stitch points together to generate exploitaion

```
F(int x, int y)
{
 int loc[4];
 if ((x + y) > 4)                    // buggy check
     return (loc[x]);                // program crash here
 else if ((x + y) <= 4) {            // still buggy check
    x = G(x, y);
    loc[x] = 0;                      // how to reach here?
    return (0);
  }
}
Int G(int x, int y) { while (x < y) x++; return (x); }
```

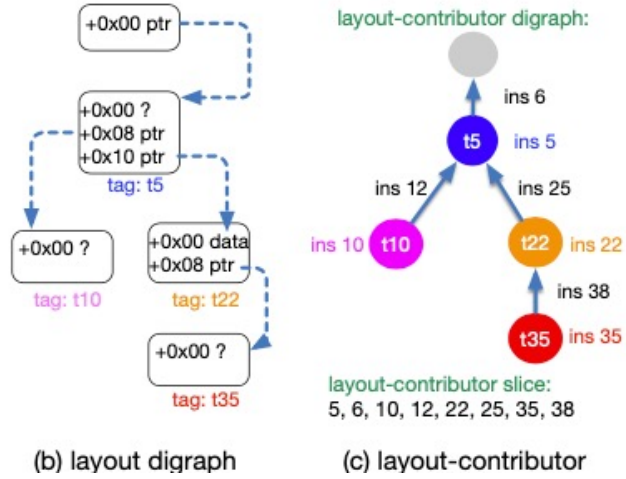
WORKFLOW 1.

- Abstract, Vulnerability Analysis -> identify vulnerability point
  uses dynamic analysis to test with PoC input
  tracks states of pointers and memory objects
  catches security violation along crashing path
  retrieves layout-contributor digraph



- Vulnerability Identification
  memory tagging on shadow-memory (each heap objects)
  use security rules (v1: access intended objects, v2: read busy objects, v3: write alive objects) to identify vulnerability
  e.g. Buff overflow -> v1, UAF -> v1, v2, v3

- Layout Analysis
  point-to replationship between indexing objects and exceptional objects
  Given a exceptional objects, we could use backward-slicing to construct this digraph



(b) layout digraph (c) layout-contributor

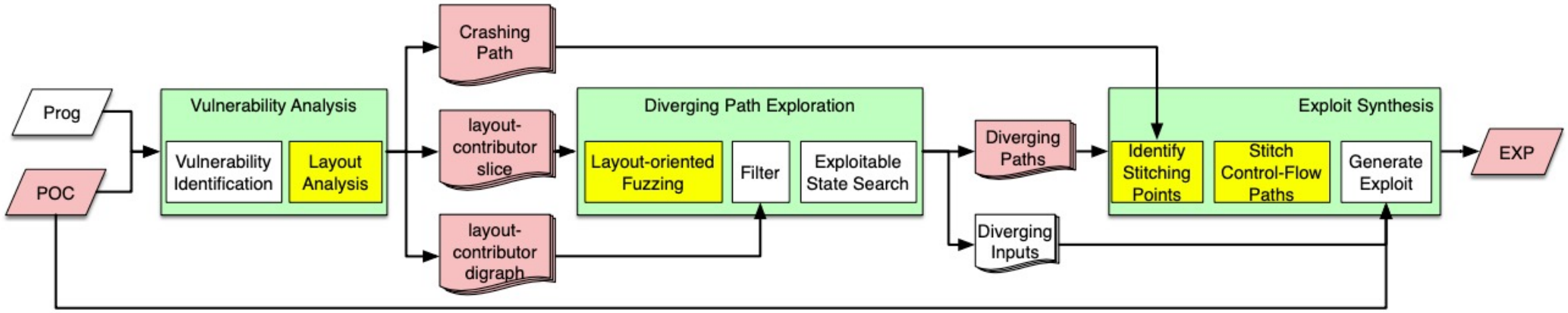
WORKFLOW 2.

- Abstract, Diverging Path Exploration -> solve problem of Exploit Derivability
  Use Fuzzing instead of Symbolic Execution
  (Just a story, we can use Concolic Testing to explore diverging paths)
  fuzzer would explore paths close to the crashing path
  directed fuzzing with guildance of layout-contributor
  identify exploitable states
- Fuzzing is more effective for exploring paths and program states
- Layout-Oriented Fuzzing > find diverging paths
  extended from AFL with 3 intuitions (1. better if more hitting, 2. better if longer, 3. chose which would not cause to duplicate operations)
- Diverging Path Filtering
  construct new layout-contributor digraph of the exceptional objects with backward-slicing
  Compare digraph and filter out if not same
- Exploitable States Searching < taint analysis
  find sensitive instructions which contains taint labels of exceptional objects

WORKFLOW 3.

- Abstract, Exploit Synthesis
  a new input triggering both vulnerabilities and exploitable states
  data flow analysis to find stitching points in crashing path and diverging path
  stitch them (symbolic execution) to synthesize exploitation path
- Identify Stitching Points
  matches data flow in diverging path against crashing path, locate the difference
  locations where exceptional objects are corrupted in crashing path
  data flow after locations in diverging path should have intersections with data flow before the stitching points in crashing path
- Control-Flow Path Stitching
  Symbolic Execution with heuristics (reusing sub-path of diverging path)
- Exploit Generation
  Constraint solving for vul constraints, path constraints, and exploit constraints
  add constraint of memory allocation size or memory address as diverging path

WORKFLOW


Prog
POC
Vulnerability Analysis
Vulnerability Identification
Layout Analysis
Crashing Path
layout-contributor slice
layout-contributor digraph
Diverging Path Exploration
Layout-oriented Fuzzing
Filter
Exploitable State Search
Diverging Paths
Diverging Inputs
Exploit Synthesis
Identify Stitching Points
Stitch Control-Flow Paths
Generate Exploit
EXP

LIMITATION

- Cannot bypass advanced defense, e.g. ASLR
- Heap-based exploits are limited to specific memory layout
- We would need to assemble different vulnerabilities at the same time
- Limited program analysis to make comprehensive understand

CONCLUSION

- Solve issues of Symbolic Execution bottleneck
- Exploits for heap-based vulnerabilities
- Search exploitable states in diverging path rather than crashing path

REFERENCE

1. https://openwall.info/wiki/_media/people/jvanegue/files/aegc_vanegue.pdf