1 AEG via Symbolic Methods

Various variants of AEG (Automatic Exploit Generation) have been studied, most often by Brumley's team at CMU. Nonexhaustively:

- Automatic Exploit Generation [2]
  - Demonstrates end-to-end system for generating exploits from the source of an application, ran on various small-medium applications successfully
  - Entirely based on (dynamic) symbolic (and concolic) execution, leveraging KLEE
  - Preconditioned symbolic execution, i.e. using heuristics to prune symbolic branches, to mitigate path explosion
  - Environment modelling to handle syscalls and other interactions
  - They have a short updated article in 2014 [1] that presents their binary AEG tool Mayhem. Report also includes a short history of AEG
- Automatic patch-Based Exploit Generation is Possible: Techniques and Implications [4]
  - Generating exploit for binaries that have been patched, by examining differences
  - Assumes that patches for (input sanitization) vulnerabilities typically add some new check, so generates candidate exploits for the unpatched version that fail these checks
  - Handles path explosion by doing dynamic analysis up to some instruction i on the path to the new check, and then does static analysis (i.e. model checking) for the remainder of the path to the new check. Tries to find a combination of the two solutions that are compatible
- CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations [6]
  - Uses crashing executions to guide concolic execution to find bugs (primarily control hijack attacks) in binaries
  - Concretization of irrelevant functions (standard practice in dynamic symbolic execution/concolic execution)
  - Uses S^2E for system simulation
  - Runs on large applications (approx. 1-2M LOC)

2 AEG via Hybrid Fuzzing/Symbolic Execution

More recently, esp. wrt. CGC [5], combinations of fuzzing and symbolic execution techniques have become more popular. Certainly worth looking at all the CGC finalists. CGC also involved automated patching of vulnerabilities, which is something interesting to look at as well.

- Cyber Grand Shellphish [11]
  - A report by the CGC Shellphish team in Phrack
  - Bunch of explanation of the contest format as well as their approach, and a number of other ideas
  - TODO: Finish reading, probably has lots of references as well
- Driller: Augmenting Fuzzing Through Selective Symbolic Execution [9]
  - Important paper that improved the state-of-the-art by augmenting fuzzing with symbolic execution in a novel manner
  - Key idea: there are broadly two types of inputs to conditional branches, namely 'general input' of which many different inputs could work, and 'specific input', of which only a tiny fraction of inputs pass the check. Specific input checks logically separate the application into compartments. Exploration within a compartment is easily handled by a fuzzer, while pushing exploration between compartments is best handled by symbolic execution. This helps mitigate path explosion in symbolic execution as well as we are usually looking just one part of the application, and lets us have the benefits of fuzzing.
  - The basic idea of combining the two has been done before, but usually in the form of fuzz, then symbolically execute, and negate the accumulated precondition to gain maximum code coverage. However it lacks the key insight above, and ends up wasting time by exploring branches within a single compartment using symbolic execution.
  - This paper was the basis for the Shellphish team, they compare it to other techniques as well, a lot of references to look at. Have to look a bit more at this paper as well.

3 Background and Theory

There's a fair amount of background theory that goes into these things, but the basic ideas are fairly simple. Here are some topics to look out for, feel free to add more:

- Basic first-order logic concepts, e.g. models, satisfiability, entailment provability, soundness, completeness, etc.. Most texts on first order logic will do. [3] is a fairly good one and also talks a little bit about decision procedures which are used in SMT solvers, although that is a little bit less relevant for us (probably)
- Satisfiability Modulo Theories (SMT) solvers. The canonical example of an SMT solver is Microsoft's Z3 [8]
- (Dynamic) Symbolic Execution, and Concolic Execution. There are many resources on this with varying degrees of clarity, here is one possibility [10]
- "Static" Symbolic Execution, a.k.a. (Bounded) Model Checking. Once again there are many resources on this. There are many different ways to do model checking, [7] looks at some of the ideas, though probably only some of the key ideas are relevant

References

References

[1] Thanassis Avgerinos et al. "Automatic Exploit Generation". In: *Commun. ACM* 57.2 (Feb. 2014), pp. 74–84. ISSN: 0001-0782. DOI: 10.1145/2560217.2560219. URL: http://doi.acm.org/10.1145/2560217.2560219.

[2] T. Avgerinos et al. "Automatic Exploit Generation". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS'11)*. 2011.

[3] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007. ISBN: 3540741127.

[4] D. Brumley et al. "Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications". In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. May 2008, pp. 143–157. DOI: 10.1109/SP.2008.17.

[5] DARPA. *Cyber Grand Challenge*. 2016. URL: http://archive.darpa.mil/cybergrandchallenge/ (visited on 05/28/2017).

[6] S. K. Huang et al. "CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations". In: *2012 IEEE Sixth International Conference on Software Security and Reliability*. June 2012, pp. 78–87. DOI: 10.1109/SERE.2012.20.

[7] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. New York, NY, USA: Cambridge University Press, 2004. ISBN: 052154310X.

[8] Microsoft. *Z3*. URL: https://github.com/Z3Prover/z3 (visited on 05/28/2017).

[9] Nick Stephens et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution". In: *NDSS*. 2016.

[10] Martin Vechev. *Program Analysis Lecture 8*. URL: http://www-verimag.imag.fr/~mounier/Enseignement/Software_Security/ConcolicExecution.pdf (visited on 05/28/2017).

[11] zardus et al. *Cyber Grand Shellphish*. 2017. URL: http://phrack.org/papers/cyber_grand_shellphish.html (visited on 05/28/2017).