"Offensive Techniques in the Binary Analysis"

Binary analysis is an approach to check vulnerabilities in computer programs through their binary executable that runs on the computer. This is important for two reasons. First, there is sometimes no access to the source code, and second, some of the properties associated with the source code of the program may not be held after compilation. For example, it is possible to have some vulnerabilities or malicious behavior generated or injected to the binary during the compilation process (with or without purpose). The authors of this article, presented a tool named "Angr", which automatically performs the process of binary code analysis, and can also automatically perform "Exploit Generation" and "Exploit Hardening". This tool uses a variety of dynamic and static analysis techniques. For instance in static analysis, they use static symbolic execution and in dynamic analysis some variety of techniques like Fuzzing.

The authors have two reasons for developing this tool and making it open source. First, without such a tool, researchers need to develop their tools each time and implement their methods of examining the security vulnerabilities, and this comes at the cost of time. Second, the difference in tools among researchers and the way they implement them, makes it nearly impossible to correctly repeat their results (replayability of the results is one of the main requirements of a good research).

To develop angr and test its results, researchers have used the DARPA dataset released after the Cyber Grand Challenge. The advantage of this dataset is that the designers of this contest developed a very simple operating system with only seven simple system calls to simplify the environment in which the program runs. So the environment and software collaboration is very simple; also, several software ranging from a complex web server to image processing tools developed for this operating system by the designers and evaluated by the contestants.

Among the trade-offs that can be taken into account in the design of such a tool, the authors point out the trade-off between replayability and code coverage, and also the trade-off between the semantic insight and the scalability of techniques. Replayability means that it must be possible to correctly describe the path leading up to the crash in the software; on the other hand, to cover as much as code possible, all the code must be examined, that is, a technique which is replayable and has a high code coverage, should check all possible paths in program, and it's very expensive because of exponential growth in number of paths. In order to have full semantic insight, the analytical tool must have a processing power equal to the power which is required to run the software in all possible cases to have a full insight into all of its components, and such a tool is not scalable at all.

Several techniques have been used to discover vulnerabilities in programs. These techniques include static analysis including CFG[1] recovery, flow modeling, data modeling and dynamic analysis including concrete execution, Fuzzing, and Symbolic execution. Due to the 2 pages limit of this summary, only the results of this tool will be noted and analyzed.

The authors of this paper have applied their tool on the DARPA programs; results are shown in Table 1.

[1] Control Flow Graph

Table 1 Results of applying all the angr analysis techniques to the DARPA dataset

Technique	Replayable	Semantic Insight	Scalability	Crashes	False Positives
Dynamic Symbolic Execution	Yes	High	Low	16	0
Veritesting	Yes	High	Medium	11	0
Dynamic Symbolic Execution + Veritesting	Yes	High	Medium	23	0
Fuzzing (AFL)	Yes	Low	High	68	0
Symbolic-Assisted Fuzzing	Yes	High	High	77	0
VSA	No	Medium	High	27	130
Under-constrained Symbolic Execution	No	High	High	25	346

As you can see, the best results are achieved from symbolic-assisted fuzzing technique. One of the most important reasons for the higher number of vulnerabilities discovered by Fuzzing is that contrary to the symbolic analysis methods, it doesn't suffer from path explosion problem, since a path is taken by the algorithm through mutating the input each time. So, it is able to search much more paths than symbolic analysis in general. On the other hand, one of Fuzzing's problems is that it doesn't know which part of input it should mutate to activate paths that has not been taken yet; to solve this problem, Symbolic-Assisted Fuzzing is used which has more semantic insight into the program and its branches. This way, it can generate an input which actives paths other than those that are examined by the Fuzzing algorithm. Then hands this input to the Fuzzing algorithm to use it, and then mutate it to activate other paths. As a result, Symbolic-Assisted Fuzzing has more code coverage and can detect more vulnerabilities because of its higher semantic insight.

One of the interesting results that appears in this article is the Fuzzing code coverage. I should note that the designers used two CFGAccurate and CFGFast algorithms to recover control flow graph. The two specifications of generated graph which is more important to the authors is *soundness* and *completeness*. A CFG is sound if it includes all the paths which the software can take in real execution (a full mesh graph of basic-blocks is sound). A CFG is complete, if all edges it has can be actually enabled by the software (an empty graph with this definition is complete).

After final analysis, they concluded that if they use the paths recovered in the Fuzzing algorithm to form a control flow graph, the resulting graph would have more code coverage than the CFG recovered by CFGAccurate and CFGFast. Also this new graph is complete because all of its paths are generated by a dynamic algorithm which activated them before.

References

[1] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel and G. Vigna, "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, USA, 2016.