Review on AEG: Automatic Exploit Generation

Summary:

An attacker can take the control over a computer program using the control flow exploits, and its prerequisite step is to identify a bug which might be exploitable. Unlike the automatic bug finding techniques, automatic exploit generation was less practical when the paper was first published in 2011. This paper presented the first fully automatic exploit generation (AEG) technique along with efficient automated exploitable bug identification approach in real programs. Using AEG, the authors analyzed fourteen open-source projects and generated sixteen control flow hijacking exploits including two zero-day exploits against previously unknown security flaws. The significant contributions are- development of preconditioned symbolic execution to minimize the search state of potential bug identification and building the first end-to-end system for demonstrating the AEG technique.

Overview of AEG:

In a single sentence, the AEG is a command line tool which examines source code, produces symbolic execution formulas, performs binary analysis to accumulate runtime information, and generates an exploit string as output which can be injected into the target vulnerable program. From the given source code to the expected exploit generation, we can break the AEG functioning process into six different steps.
1. Pre-Process: All we require to generate an exploit using AEG is the source code of the application. However, AEG necessitates two inputs: program binary and LLVM bytecode. So, the user needs to generate the target binary (B_{gcc}) and LLVM bytecode (B_{llvm}) of the program using GCC and LLVM compiler respectively before starting the analysis.
2. Source Analysis: Next step is to search for the buggy path of the program. To make the buggy pathfinding approach better AEG takes help of the maximum size of symbolic data- max. AEG heuristically assumes that the value of *max* would be 10% larger than the largest statically allocated buffers of the target program.
3. Bug Finding: The system has to explore an infinite number of paths for finding the potential buggy ones. Preconditioned symbolic execution is an excellent addition by the authors which narrows down the state space to search. Still, AEG needs to determine which one it should explore first. Buggy-Path-First and Loop Exhaustion are two distinct versions of path prioritization algorithm which assist in choosing the proper one from among the shortlisted paths.
4. Dynamic Binary Analysis: The buggy path alone is not enough to generate and verify an exploit. It demands a set of runtime information, e.g., the address to overwrite and stack memory contents (for the illustrated return-to-stack example of the paper), and AEG collects all the information through dynamic binary analysis.

5. Exploit Generation: Gathering the buggy path constraints and runtime information from the previous two steps, AEG generates two exploits: return-to-stack and return-to-libc, two of the most popular control hijacking attack techniques.
6. Exploit Verification: Finally, AEG verifies that the generated exploit is an actual working exploit. For the explained example in this paper (stack-overflow return-to-stack), it runs the executable with the exploit and spawns a shell for successful exploitation.

Preconditioned Symbolic Execution:

From a limitless number of program execution paths, preconditioned symbolic execution constrains the paths considered to those that would most likely include exploitable bugs. In traditional symbolic execution, for every program branches, the interpreter conceptually "forks off" two different interpreters; thus, the total number of interpreters is exponential in the number of branches. Depending upon a condition called path predicate, the interpreter takes its decision whether or not to explore the current path. In the newly developed preconditioned symbolic execution, the authors initialized the concept of precondition predicate which can minimize the state space by specifying exploitability conditions as a precondition. AEG uses four types of preconditions- None, Known Length, Known Prefix, and Concolic Execution.

Path Prioritization:

Though preconditioned symbolic execution limits the search space, the user yet requires prioritizing the shortlisted paths for exploring. AEG resolves this issue with path ranking heuristics where all the unexplored paths are pushed into a priority queue based on their ranking and pulled back from the queue when exploring the next. To rank the path AEG applies two different algorithms- Buggy-Path-First and Loop Exhaustion. The theme of buggy-path-first is that one bug on a path implies subsequent statements are also expected to be buggy (and hopefully exploitable). The loop exhaustion method gives higher priority to the interpreters which exploring the maximum number of loop iterations, believing that more computations increase the probability of bug producing.

Exploit Generation:

The AEG exploit generator generates two types of exploits: return-to-stack and return-to-libc. The return-to-stack technique overwrites the return address of the buggy function to enforce the program counter pointing back to the injected input, which is attacker defined shellcode for the demonstrated example in this paper. AEG maintains a shellcode database to support various types of exploits, and it holds two kinds of shellcode classes: standard shellcodes for local exploits, and binding and reverses binding shellcodes for remote exploits. The return-to-libc attack technique replaces the return address so that the *execev* function points to *libc*. This method is valid only for local attack scenario, and the attacker must know the address of "/bin/sh" string in the binary.

Implementation of AEG:

Authors preferred the combination of C++ and Python to develop the AEG which consists of four major components: symbolic executor, dynamic binary evaluator, exploit generator, and constraint solver to verify the exploits. They had to add 5000 lines of code on top of KLEE to make the AEG algorithm and heuristics working and support various input sources. KLEE is a symbolic virtual machine built on top of the LLVM compiler infrastructure, and its distributed under the UIUC open source license. The entire dynamic binary evaluator was implemented in Python. For constraint solving they used STP, a famous constraint solver which can solve numerous kinds of problems generated by program analysis tools, automated bug identifiers, cryptographic algorithms, and intelligent *fuzzers*.

Evaluation:

Authors chose 14 open source programs from different popular advisories: Common Vulnerabilities and Exposures (CVE), Open Source Vulnerability Database (OSVDB), and Exploit-DB (EDB), and tested them with AEG. The fastest generated exploit was for *iwconfig* which took 0.5s and required to explore a single path. For the full test suite, it took 114.6s on an average to generate and verify an exploit. The experiment also demonstrated that supplying more information as the precondition for the symbolic executor significantly advances the performance, e.g., providing the length precondition identified almost three times more exploitable bugs. Additionally, buggy-path-first prioritization technique supported to identify an exploitable bug followed by an un-exploitable bug which might not be possible with the standard method.

Advancement of This Work:

Identification of the problem, technical contributions, and the organization of the paper to resolve the issues for automatic exploit generation are convincing. Some of the factors which make this paper a great one in my eye are depicted below-
1. It breaks the whole solution process into six different components and details the fundamental concepts separately, and thus, it presents itself as more accessible to grasp for the reader.
2. It applies the unique idea of combining the source-code analysis with binary level analysis to improve scalability in discovering bugs and gather runtime information for exploit generation.
3. The paper introduces and outlines the concept of preconditioned symbolic execution which significantly improves the performance for pruning off the state space search. Additionally, it considers modifying the KLEE to make the symbolic execution working for AEG techniques.

4. It explains the full “stack-overflow return-to-stack” exploit generation from the beginning with a simple example and provides the video link from their website which makes the paper complete. Description of the formation and positioning process of attacker defined program (shellcode) into the target program binary is another notable contribution.
5. The evaluation section precisely explicates the advantages of preconditioned symbolic execution and path prioritizing algorithm appended by the authors.

Later Works:

In 2014, the refined version of AEG [1] was published in the Communications of the ACM where it described the algorithmic part in detail for similar works. Before that in 2012, the authors introduced Mayhem [2], a tool and set of techniques for AEG on executable code. With Mayhem, they proposed techniques for actively managing symbolically executed program paths without exhausting memory and reasoning about symbolic memory addresses efficiently. Unlikely the first introduced AEG, Mayhem generates 7 exploits for Windows applications along with 22 for Linux applications. Additionally, as of July 2013, Mayhem was re-designed to generate exploits for buffer overflows, format strings, command injection, and some information-leak vulnerabilities. Using the core concept of AEG Joshua Garcia et. all designed Automatic generation of inter-component communication exploits for Android applications [3]. System Service Call-oriented Symbolic Execution of Android [4] is another variation of AEG.

Further Improvement Opportunity:

Unquestionably the paper is a great one to address the problem and come up with the solution for automatic exploit generation. However, according to me, it would be the best if the authors would focus on the following issues.
1. AEG always requires the source code to generate an exploit, and they left AEG on binary-only as future work. However, providing the central idea about what causes binary-only instrumentation challenging for AEG would be more convincing.
2. The unsafe path predicate represents the path predicate of an execution which violates the safety property φ of a program. The authors claim that they use popular safety properties for C programs including checking for out-of-bounds writes and unsafe format strings. Anyway, providing the whole list of considered safety properties in AEG would help the users to weigh the exact enormity.
3. For preconditioned symbolic execution AEG supports four types of preconditions: None, Known Length, Known Prefix, and Concolic Execution. However, I believe, there is a lack of information about when to apply which precondition, and why they developed only four preconditions.
4. In Loop Exhaustion path prioritization algorithm, it gives higher priority to the interpreter which explores the maximum number of loop iterations. Does it always yield a positive result? In my thinking, it seeks more attention to verify with a good number of test cases.

5. The exploit generation algorithms are at the paper's heart, and the paper presents only the "stack-overflow return-to-stack" from among four algorithms in section 6.2. However, a comprehensive explanation of the demonstrated algorithm is highly expected in the paper which is missing.
6. It illustrates only the local exploit generation with the stack-overflow example. Demonstrating another full example for generating an exploit in a remote system would be excellent.
7. The paper describes the whole process for C programs. Merest information about the possible scenarios for other languages like C#, PHP, etc. would create more research opportunities.
8. As Android OS is designed based on Linux kernel, the implication of AEG on Android applications should be more straightforward with minimum modifications. Discussion of this point might be an enchanting addition to pave the way for future research in mobile application security.

References:

1. Avgerinos, T., Cha, S.K., Rebert, A., Schwartz, E.J., Woo, M. and Brumley, D., 2014. Automatic exploit generation. Communications of the ACM, 57(2), pp.74-84.

2. Cha, S.K., Avgerinos, T., Rebert, A. and Brumley, D., 2012, May. Unleashing mayhem on binary code. In Security and Privacy (SP), 2012 IEEE Symposium on (pp. 380-394). IEEE.

3. Garcia, J., Hammad, M., Ghorbani, N. and Malek, S., 2017, August. Automatic generation of inter-component communication exploits for Android applications. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (pp. 661-671). ACM.

4. Luo, L., Zeng, Q., Cao, C., Chen, K., Liu, J., Liu, L., Gao, N., Yang, M., Xing, X. and Liu, P., 2017, June. System service call-oriented symbolic execution of android framework with applications to vulnerability discovery and exploit generation. In Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (pp. 225-238). ACM.