Review

The motivation for writing this paper is to discuss Security-critical bugs that pave the way for attackers to steal information, monitor content, introduce vulnerabilities and damage the behavior of software. Implementing the system that performs automated vulnerability detection and exploit generation (AEG) will make computers more secure by finding security-critical bugs before they are exploited by attackers.

Initially, AEG performs a mix of binary- and source-level analysis in order to scale to larger programs. In addition, this automatic end-to-end system utilizes preconditioned symbolic execution explore and uses path prioritization techniques until an exploitable path is detected. Then, while executing the vulnerable function, performs dynamic binary analysis on the target binary with a concrete buggy input and extracts low-level runtime information. Furthermore, with the path predicate of the bug and runtime information constructs a formula for a control flow hijack exploit. In the end, AEG has to verify that the generated exploit is a working exploit for a given system. AEG implementation is a single command line that analyzed 14 open source projects and generates working exploits for target applications. AEG generated 16 control flow hijack exploits, two of which were against previously unknown vulnerabilities.

Important Points

The primary domain where AEG has shone, from my personal standpoint, is the combination of static binary analysis and dynamic binary analysis techniques. There's a fairly obvious reason why this is the case; Static binary analysis can be slow, and it has limitations when dealing with indirect jump statements. In other words, indirect jump statements are harder than direct jump statements to resolve because the application is passed control to a target whose value, for example, could be arbitrarily calculated or dependent on the context of the application. Thus, the approximations about the control flow of an application have a risk of not resolves indirect jump statements at all. It suffices to perform dynamic binary analysis examines a program's behavior while it is running in a given environment that allows you to explore individual paths which makes it very precise. AEG performs a mix of binary- and source-level analysis in order to improve scalability in finding bugs and binary and runtime information to exploit programs.

Additionally, a more powerful, dynamic analysis technique that is implemented in AEG is preconditioned symbolic execution to determine what inputs cause each part of a program to execute. In classical symbolic execution where variables and application input are modeled using symbolic values instead of using concrete values, during execution, both memory and register state are tracked and are also modeled symbolically. Symbolic execution is typically used to dynamically generate test cases which are used to drive path exploration, unlike traditional fuzzing techniques where test cases must be manually generated to seed the system. However, dynamic symbolic execution suffers from a problem known as path explosion, whereby new paths are created at every new branch. This can lead to an exponential number of paths to be explored and hence makes dynamic symbolic analysis computationally expensive, hence limiting the scalability of analysis systems that use this technique as it's the only mechanism of path exploration. The paper approach to combat these limitations is developed and implemented 4 different preconditions for efficient exploit generation and with a combination of both concrete and symbolic execution crafts a technique known as concolic execution.

Limitations and improvements

First, with reference to the bug categories considered in the excerpt, this viewpoint is not correct. By looking at the bug classes that delineated in this article, hardly anyone can deny the fact that these bugs exploited with lacking individuality. The system found and exploited the bugs which are easily achievable and do not need tons of effort. It's at this point the distance between these bugs and the realities of industrial vulnerability detection and exploit development can be seen. Moreover, for more complex vulnerabilities and exploits that require a skilled attacker, this AEG system doesn't change the threat model.

Furthermore, Memory errors continue to compromise the security of today's systems. One of the things I really want to support is the automatic generation of exploits for modern heap-based vulnerabilities, although this article totally ignored what has been required to write a heap exploit. AEG was limited to stack-based buffer overflows and format string exploits because it did not have semantic information about user bytes in memory. For example, it could not explicitly assert the constraint on user input that the user input be of a certain length. Automatically exploit generation of heap exploits requires one to be able to discover and trigger heap manipulation primitives as well as whatever else must be done. This is a difficult problem to solve automatically and one that is completely ignored. To automatically generate a Linux heap exploit we must have some information on the relationship between user input such as from sockets, files or command line, and the structure of the process's heap. When manually writing an exploit we will often want to force the program to allocate huge amounts of memory, and the usual way to do this involves jumping into the code/disassembly and poking around for a while until you find a memory allocation dependent on the size of some user-supplied field, or a loop doing memory allocation with a user influenceable bound.

Finally, to keep up with the hardware innovation, many modern software systems are intrinsically concurrent, consisting of multiple executions flows that progress simultaneously. Personally, I think a serious problem in this research area is neglecting such trivial issues. Concurrency introduces new challenges to the testing process in many respects such as the increase in complexity, nonrepeatability, Heisenbugs such as data races, which are non-deterministic concurrency errors, pervasively infect the concurrent software and the lack of a synchronized global clock. Huge amount of interleaving space is the essential problem of concurrent program bug detection. Thus, testing concurrent systems requires not only to explore the space of possible inputs, but also the space of possible interleaving, which can be intractable also for small programs. In order to systematically explore the interleaving space and effectively expose concurrent bugs, good coverage criteria are desired.