System to automatically find a common type of programming bug significantly outperforms its predecessors

March 23 2015

Integer overflows are one of the most common bugs in computer programs—not only causing programs to crash but, even worse, potentially offering points of attack for malicious hackers. Computer scientists have devised a battery of techniques to identify them, but all have drawbacks.

This month, at the Association for Computing Machinery's International Conference on Architectural Support for Programming Languages and Operating Systems, researchers from MIT's Computer Science and Artificial Intelligence Laboratory (CSAIL) will present a new algorithm for identifying integer-overflow bugs. The researchers tested the algorithm on five common open-source programs, in which previous analyses had found three bugs. The new algorithm found all three known bugs—and 11 new ones.

The variables used by computer programs come in a few standard types, such as floating-point numbers, which can contain decimals; characters, like the letters of this sentence; or integers, which are whole numbers. Every time the program creates a new variable, it assigns it a fixed amount of space in memory.

If a program tries to store too large a number at a memory address reserved for an integer, the operating system will simply lop off the bits that don't fit. "It's like a car odometer," says Stelios Sidiroglou-Douskos,

a research scientist at CSAIL and first author on the new paper. "You go over a certain number of miles, you go back to zero."

In itself, an integer overflow won't crash a program; in fact, many programmers use integer overflows to perform certain types of computations more efficiently. But if a program tries to do something with an integer that has overflowed, havoc can ensue. Say, for instance, that the integer represents the number of pixels in an image the program is processing. If the program allocates memory to store the image, but its estimate of the image's size is off by several orders of magnitude, the program will crash.

Charting a course

Any program can be represented as a flow chart—or, more technically, a graph, with boxes that represent operations connected by line segments that represent the flow of data between operations. Any given program input will trace a single route through the graph. Prior techniques for finding integer-overflow bugs would start at the top of the graph and begin working through it, operation by operation.

For even a moderately complex program, however, that graph is enormous; exhaustive exploration of the entire thing would be prohibitively time-consuming. "What this means is that you can find a lot of errors in the early input-processing code," says Martin Rinard, an MIT professor of computer science and engineering and a co-author on the new paper. "But you haven't gotten past that part of the code before the whole thing poops out. And then there are all these errors deep in the program, and how do you find them?"

Rinard, Sidiroglou-Douskos, and several other members of Rinard's group—researchers Eric Lahtinen and Paolo Piselli and graduate students Fan Long, Doekhwan Kim, and Nathan Rittenhouse—take a

different approach. Their system, dubbed DIODE (for Directed Integer Overflow Detection), begins by feeding the program a single sample input. As that input is processed, however—as it traces a path through the graph—the system records each of the operations performed on it by adding new terms to what's known as a "symbolic expression."

"These symbolic expressions are complicated like crazy," Rinard explains. "They're bubbling up through the very lowest levels of the system into the program. This 32-bit integer has been built up of all these complicated bit-level operations that the lower-level parts of your system do to take this out of your input file and construct those integers for you. So if you look at them, they're pages long."

Trigger warning

When the program reaches a point at which an integer is involved in a potentially dangerous operation—like a memory allocation—DIODE records the current state of the symbolic expression. The initial test input won't trigger an overflow, but DIODE can analyze the symbolic expression to calculate an input that will.

The process still isn't over, however: Well-written programs frequently include input checks specifically designed to prevent problems like integer overflows, and the new input, unlike the initial input, might fail those checks. So DIODE seeds the program with its new input, and if it fails such a check, it imposes a new constraint on the symbolic expression and computes a new overflow-triggering input. This process continues until the system either finds an input that can pass the checks but still trigger an overflow, or it concludes that triggering an overflow is impossible.

If DIODE does find a trigger value, it reports it, providing developers with a valuable debugging tool. Indeed, since DIODE doesn't require

access to a program's source code but works on its "binary"—the executable version of the program—a program's users could run it and then send developers the trigger inputs as graphic evidence that they may have missed security vulnerabilities.

Provided by Massachusetts Institute of Technology