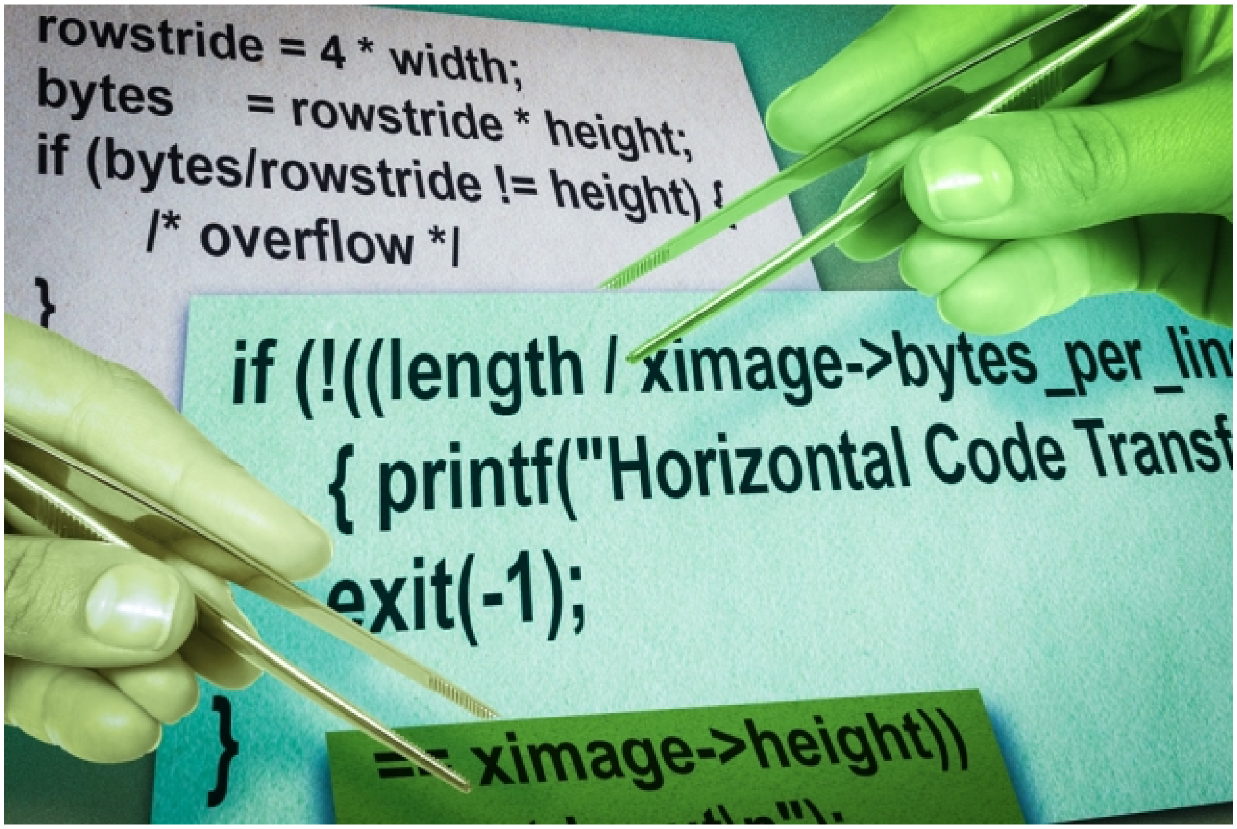
System fixes bugs by importing functionality from other programs—without access to source code

June 29 2015, by Larry Hardesty


rowstride = 4 * width;
bytes = rowstride * height;
if (bytes/rowstride != height)
/* overflow */
}
if (!((length / ximage->bytes_per_lin
{ printf("Horizontal Code Transf
exit(-1);
}
== ximage->height))

Credit: Jose-Luis Olivares/MIT

At the Association for Computing Machinery's Programming Language Design and Implementation conference this month, MIT researchers presented a new system that repairs dangerous software bugs by automatically importing functionality from other, more secure applications.

Remarkably, the system, dubbed CodePhage, doesn't require access to the source code of the applications whose functionality it's borrowing. Instead, it analyzes the applications' execution and characterizes the types of security checks they perform. As a consequence, it can import checks from applications written in programming languages other than the one in which the program it's repairing was written.

Once it's imported code into a vulnerable application, CodePhage can provide a further layer of analysis that guarantees that the bug has been repaired.

"We have tons of source code available in open-source repositories, millions of projects, and a lot of these projects implement similar specifications," says Stelios Sidiroglou-Douskos, a research scientist at MIT's Computer Science and Artificial Intelligence Laboratory (CSAIL) who led the development of CodePhage. "Even though that might not be the core functionality of the program, they frequently have subcomponents that share functionality across a large number of projects."

With CodePhage, he says, "over time, what you'd be doing is building this hybrid system that takes the best components from all these implementations."

Sidiroglou-Douskos and his coauthors—MIT professor of computer science and engineering Martin Rinard, graduate student Fan Long, and Eric Lahtinen, a researcher in Rinard's group—refer to the program CodePhage is repairing as the "recipient" and the program whose functionality it's borrowing as the "donor." To begin its analysis, CodePhage requires two sample inputs: one that causes the recipient to crash and one that doesn't. A bug-locating program that the same group reported in March, dubbed DIODE, generates crash-inducing inputs automatically. But a user may simply have found that trying to open a particular file caused a crash.

Carrying the past

First, CodePhage feeds the "safe" input—the one that doesn't induce crashes—to the donor. It then tracks the sequence of operations the donor executes and records them using a symbolic expression, a string of symbols that describes the logical constraints the operations impose.

At some point, for instance, the donor may check to see whether the size of the input is below some threshold. If it is, CodePhage will add a term to its growing symbolic expression that represents the condition of being below that threshold. It doesn't record the actual size of the file—just the constraint imposed by the check.

Next, CodePhage feeds the donor the crash-inducing input. Again, it builds up a symbolic expression that represents the operations the donor performs. When the new symbolic expression diverges from the old one, however, CodePhage interrupts the process. The divergence represents a constraint that the safe input met and the crash-inducing input does not. As such, it could be a security check missing from the recipient.

CodePhage then analyzes the recipient to find locations at which the input meets most, but not quite all, of the constraints described by the new symbolic expression. The recipient may perform different operations in a different order than the donor does, and it may store data in different forms. But the symbolic expression describes the state of the data after it's been processed, not the processing itself.

At each of the locations it identifies, CodePhage can dispense with most of the constraints described by the symbolic expression—the constraints that the recipient, too, imposes. Starting with the first location, it translates the few constraints that remain into the language of the recipient and inserts them into the source code. Then it runs the recipient again, using the crash-inducing input.

If the program holds up, the new code has solved the problem. If it doesn't, CodePhage moves on to the next candidate location in the recipient. If the program is still crashing, even after CodePhage has tried repairs at all the candidate locations, it returns to the donor program and continues building up its symbolic expression, until it arrives at another point of divergence.

Automated future

The researchers tested CodePhage on seven common open-source programs in which DIODE had found bugs, importing repairs from between two and four donors for each. In all instances, CodePhage was able to patch up the vulnerable code, and it generally took between two and 10 minutes per repair.

As the researchers explain, in modern commercial software, security checks can take up 80 percent of the code—or even more. One of their hopes is that future versions of CodePhage could drastically reduce the time that software developers spend on grunt work, by automating those checks' insertion.

"The longer-term vision is that you never have to write a piece of code that somebody else has written before," Rinard says. "The system finds that piece of code and automatically puts it together with whatever pieces of code you need to make your program work."

"The technique of borrowing code from another program that has similar functionality, and being able to take a program that essentially is broken and fix it in that manner, is a pretty cool result," says Emery Berger, a professor of computer science at the University of Massachusetts at Amherst. "To be honest, I was surprised that it worked at all."

"The donor program was not written by the same people," Berger explains. "They have different coding standards; they name variables differently; they use all kinds of different variables; the variables could be local; or they could be higher up in the stack. And CodePhage is able to identify these connections and say, 'These variables correlate to these variables.' Speaking in terms of organ donation, it transforms that code to make it a perfect graft, as if it had been written that way in the beginning. The fact that it works as well as it does is surprising—and cool."

More information: "Automatic error elimination by horizontal code transfer across multiple applications." Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. dl.acm.org/citation.cfm?id=2737988

This story is republished courtesy of MIT News (web.mit.edu/newsoffice/), a popular site that covers news about MIT research, innovation and teaching.

Provided by Massachusetts Institute of Technology

Citation: System fixes bugs by importing functionality from other programs—without access to source code (2015, June 29) retrieved 20 March 2024 from https://phys.org/news/2015-06-bugs-importing-functionality-programswithout-access.html