The Real Cost of Software Errors

The Real Cost of Software Errors

Michael Zhivich and Robert K. Cunningham
MIT Lincoln Laboratory

Software is no longer creeping into every aspect of our lives—it's already there. In fact, failing to recognize just how much everything we do depends on software functioning correctly makes modern society vulnerable to software errors.

Estimating the cost of these errors is difficult because the effects of a single critical system's failure can have profound consequences across various sectors of the economy. The US National Institute of Standards and Technology (NIST) estimates that the US economy loses $60 billion each year in costs associated with developing and distributing software patches and reinstalling systems that have been infected, as well as cost from lost productivity due to computer malware and other problems that software errors enable (see www.nist.gov/public_affairs/releases/n02-10.htm). The amount spent cleaning up from any particular virus is staggering—the Love Bug virus (2000) cost an estimated US$8.75 billion worldwide, whereas CodeRed (2001) weighed in at $2.75 billion and Slammer (2003) at $1.5 billion (see www.computereconomics.com/article.cfm?id=936).

Worse yet, software problems don't stop with crashing browsers, zombie computers sending spam, or credit-card number theft. Critical infrastructure systems, including the power grid, petroleum refineries, oil and gas pipelines, water treatment plants, and nuclear power plants, all rely on industrial automation systems to perform data acquisition and real-time control. The software operating these systems suffers from similar errors as its enterprise counterparts; however, failures in these domains are much more severe. The August 2003 blackout in the northeastern US occurred, in part, because of a software fault in GE's XA/21 alarm-management system. A memory corruption error triggered by a race condition sent the system into an infinite loop, thus leaving the operator without updated information about its state. Had the system been working correctly, the operator could have prevented the cascading failures and minimized the damage.[1] Estimated costs associated with this blackout were between US$7 and $10 billion (see www.icfi.com/Markets/Energy/doc_files/blackout-economic-costs.pdf).

Given that software errors are so costly, an astute reader might ask why the software industry isn't doing more to prevent them. A common answer, even from some software security professionals, is that software errors aren't sufficiently important because they "don't kill people." Unfortunately, this statement indicates a misunderstanding of just how much we rely on software's correct operation. Here, we've compiled several examples from diverse critical systems that demonstrate just how deadly software errors can really be.

Patriot Missile Defense System Failure

Patriot is the US Army's mobile surface-to-air missile defense system, which was designed to defend against aircraft, cruise missiles, and short-range ballistic missiles. On 25 February 1991, a Patriot system failed to track and intercept an incoming Iraqi Scud missile at an army base in Dhahran, Saudi Arabia. The failure let the Scud missile reach its target, killing 28 American soldiers and wounding roughly 100 others (see www.fas.org/spp/starwars/gao/im92026.htm).

The failure's cause was a rounding error resulting in a clock drift that worsened with increased operational time between system reboots. The original design assumed that the Patriot system would be fielded in a mobile fashion and thus frequently moved—the operational cycle was supposed to be 14 hours. The system in Dhahran was running for far longer—roughly 100 hours, resulting in a clock skew of 0.34 seconds. Although this percentage error might seem small, it was sufficient to miscalculate the incoming missile's location. The Patriot system verifies that an incoming object is a target for interception by computing a "box" based on the first radar contact with the object, a known missile speed, and a time counter. A radar contact within the predicted

box would confirm target detection, but the clock skew caused the system to miscalculate the box's boundaries, so it didn't register the second radar contact with the high-speed missile.

The ultimate irony in this disaster is that the army had already worked out a software fix for this problem—the updated software arrived at the base just one day too late.

Radiation Treatment Overdoses

Although we might expect deaths due to military systems (faulty or otherwise), fatalities resulting from medical systems designed to heal are more concerning. A fairly well-known failure in a linear accelerator, known as Therac-25, resulted in several cancer patients receiving deadly radiation overdoses during their treatment between June 1985 and January 1987 at several oncology clinics in the US and Canada. The dosages were later estimated to be more than 100 times greater than those typically used for treatment.

A race condition was partly to blame for these accidents. Therac-25 was an improvement over the Therac-20 model; it was smaller and cheaper, and it utilized fewer hardware components. One change from the Therac-20 model replaced hardware circuitry that acted as a safety interlock—ensuring proper positioning of shielding surfaces and preventing the electron beam power level from exceeding a predefined maximum regardless of operator input—with a software implementation. The software bug was triggered when a quick-fingered operator issued a specific set of commands at a control terminal; the display information on the operator's screen was inconsistent with the device's actual operation and prevented the operator from recognizing that an error had occurred. The same bug existed in the Therac-20 model's software, but the hardware safety interlock prevented the system from delivering fatal radiation dosages.[2]

A more recent case of deadly overdoses in radiation treatment occurred at the Instituto Oncologico Nacional in Panama City in 2001. Treatment-planning software from Multidata Systems International resulted in incorrectly calculated radiation dosages. Twenty-eight patients received excessive amounts of radiation, with fatal consequences for several. Operators triggered the software error by attempting to overcome the system's limitation in the number and configuration of shielding surfaces used to isolate an area for irradiation. The operators found that by drawing an area with a hole in it, they could get the system to dispense the right dosage in the correct location. However, unknown to them, drawing such a surface in one direction resulted in a correct calculation, whereas drawing the surface differently resulted in an overdose. We can't blame the software alone for these incidents—the operators were supposed to perform manual calculations to ensure that the dosage the software computed was appropriate. They ignored this important check due to lax administrative procedures at the medical institution (see www.fda.gov/cdrh/ocd/panamaradexp.html or www.fda.gov/bbs/topics/NEWS/2003/NEW00903.html).

Bellingham, WA, Pipeline Rupture

Critical infrastructure systems, like military and medical systems, also depend on software for robust and secure operation. Operators rely on supervisory control and data acquisition (SCADA) systems to provide accurate, real-time information in order to assess the system state correctly and operate it reliably. Although critical infrastructure systems include physical safeguards in conjunction with computerized SCADA systems, a software failure that prevents the operator from seeing the system's actual state can result in a catastrophic failure.

On 10 June 1999, a 16-inch-diameter pipeline ruptured in Bellingham, Washington, releas-

ing 237,000 gallons of gasoline into a creek that flowed through Whatcom Falls Park. The gasoline pools ignited and burned an area covering approximately 1.5 miles along the creek. Two 10-year-old boys and an 18-year-old man died as a result of this accident, and an additional eight injuries were documented. The failed pipeline's owner, the Olympic Pipeline Company, estimated the total property damages to be at least US$45 million.

The rupture itself occurred due to a combination of causes. The pipeline was weakened by physical damage from nearby construction. High pressures on the pipeline during subsequent operation exacerbated this damage, resulting in the rupture and release of gasoline. However, an investigation determined that an unresponsive SCADA system prevented the operator from recognizing that the pipeline had ruptured and taking appropriate action to limit the amount of gasoline spilled. The report authors believe that shutting down the pipeline earlier could have prevented the resulting explosion (see www.ntsb.gov/publictn/2002/PAR0202.pdf).

Finding Solutions

As these examples demonstrate, software errors affect many critical systems, from military installations to medical systems to critical infrastructure. Moreover, in some cases, software faults result in fatal consequences because software is integral to how we assess and respond to situations in the physical domain. Whether software faults are triggered by operator error, an unlikely sequence of events, or a malicious adversary matters little—the results can be equally disastrous. Robustness and security are two sides of the same coin because both can be compromised by failures in a system's availability, reliability, or integrity.

Although we can't blame software errors entirely for the incidents we present here, our reliance on software must come with an understanding of the consequences of software failures. These examples further support the NIST report's conclusions—software is inadequately tested for supporting critical systems reliably. Academic work in several disciplines, such as model checking, software verification, static analysis, and automated testing, exhibits promise in delivering software systems that are free from certain defects.

Model checking and software verification rely on mathematically modeling a program's computation to verify that a certain fault (such as a division by zero) can't occur at any point in the program. These methods identify not the problem's location but rather parts of the code that are guaranteed to be problem-free. Although this metric turns the usual bug-finding task on its head, it lets us make certain statements about the quality of the code that the system can prove to be correct. Unfortunately, conducting such proofs for large code bases becomes impractical because an accurate model of software behavior becomes unwieldy quickly, forcing the system to make approximations that affect the analysis's precision.

Static analysis has become rather popular, and several commercial companies sell tools that can analyze C, C++, Java, and C# software. Static analysis encompasses several different techniques, the most prominent being lexical analysis and abstract interpretation. A lexical analysis tool scans the software to identify usages and language idioms that are frequently misused and thus likely indicate that the programmer has made an error. However, these tools tend to have high false-alarm rates, requiring developers to evaluate many false warnings in search of real bugs. Abstract interpretation tools, on the other hand, perform a symbolic execution of the program that treats all inputs and unknown memory locations as containing a symbolic value that the program can manipulate. Such a technique tends to arrive at more precise results than lexical analysis; however, program size and complexity can force the tool to use approximations that adversely affect both the false-positive and false-negative rates.[3]

Automated testing has also received a lot of attention in the past several years. With the advent of *fuzzing*, a technique that uses mutation or other mechanisms to send random-looking inputs to the program in attempts to crash it, many developers are looking to augment their functional testing infrastructure to include automated fuzz testing. Incorporating such testing into the software development process provides some confidence in software's ability to handle even unexpected inputs robustly. Such techniques also benefit from source or binary program instrumentation that can detect subtle errors that might not result in a crash.[4,5]

Although much academic research has focused on software analysis and testing, the software industry has been slow to adopt these tools. A quick back-of-the-envelope calculation presented as part of the Quality of Protection Workshop indicates that industry spending on protecting software assets (estimated at 0.2 percent of the value of the software market) is an order of magnitude smaller than

Further Reading

The ACM Forum on Risks to the Public in Computers and Related Systems (RISKS) is an excellent resource for examples of computer failures and their effects on the public. You can find an archive of this mailing list at http://catless.ncl.ac.uk/Risks/.

spending to protect network assets (estimated at 2.3 percent of the value of the network equipment market; see http://1raindrop.typepad.com/1_raindrop/2008/11/the-economics-of-finding-and-fixing-vulnerabilities-in-distributed-systems-.html). On the other hand, the market for software tools and services is growing—in 2007, it was worth US$275 to $300 million, with 50 to 83 percent increases in static analysis and black-box testing sectors.[6] We can only hope that this trend continues, as more robust and secure software can help us prevent tragedies similar to the ones described here from happening in the future. □

References

1. US–Canada Power System Outage Task Force, *Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations*, tech. report, US Dept. of Energy, Apr. 2004.
2. G. Williamson, "Software Safety and Reliability," *IEEE Potentials*, vol. 16, no. 4, 1997, pp. 32–36.
3. R. Lippmann, M. Zitser, and T. Leek, "Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code," *ACM SIGSOFT Software Eng. Notes*, vol. 29, no. 6, 2004, pp. 97–106.
4. T. Leek, M. Zhivich, and R. Lippmann, "Dynamic Buffer Overflow Detection," *Proc. Workshop Evaluation of Software Defect Detection Tools*, 2005; www.cs.umd.edu/~pugh/BugWorkshop05/papers/61-zhivich.pdf.
5. M. Zhivich, *Detecting Buffer Overflows using Testcase Synthesis and Code Instrumentation*, master's thesis, Dept. of Electrical Eng. and Computer Science, Massachusetts Inst. of Tech., May 2005.
6. G. McGraw, "Software [In]Security: Software Security Demand Rising," *InformIT*, 11 Aug. 2008; www.informit.com/articles/article.aspx?p=1237978.

Michael Zhivich *is an associate member of the technical staff at the MIT Lincoln Laboratory. His research interests include program analysis, automated testing, cryptography, and usability aspects of security. Zhivich has Sc.B. and M.Eng. degrees in computer science and electrical engineering from MIT. He's a member of the ACM and Eta Kappa Nu. Contact him at mzhivich@ll.mit.edu.*

Robert K. Cunningham *is the associate leader of the Information Systems Technology group at the MIT Lincoln Laboratory. His research interests include detection and analysis of malicious software and automatic detection of software faults in mission-critical software. Cunningham has an Sc.B. in computer engineering from Brown University, an MS in electrical engineering, and a PhD in cognitive and neural systems from Boston University. He's a member of Sigma Xi and a senior member of the IEEE. Contact him at rkc@ll.mit.edu.*