Recent Progress in Software Security

Edward Amoroso

EXACTLY 50 YEARS ago, Edsger Dijkstra sent the article “A Case against the GOTO Statement” to the Communications of the ACM, explaining why GOTO introduced too much complexity and should thus be avoided. Given the urgency of Dijkstra’s message, Pascal inventor Niklaus Wirth made the prescient decision to recast the article as a letter to the editor with the now-iconic title, “Go To Statement Considered Harmful.”

In the years since, our community has, sadly, lost Dijkstra, but the debate he sparked has remained active. The cybersecurity community in particular has been vocal about finding ways to improve software, because most vulnerabilities involve exploitable weaknesses introduced through badly written code. Unfortunately, the rush to modern DevOps coding and the demands of software marketing have tended to overshadow most correctness concerns.

Instead, the cybersecurity community has widely adopted an approach to reduce cybersecurity risk in software that involves a collage of techniques, tools, and methods, each addressing some aspect of the threat implications of bad code. Here, I briefly survey recent progress in each element of this combined approach, including the pros and cons for reducing cybersecurity risk.

Advanced Malware Detection
Although improved programming methodology continues to influence software security, the cybersecurity software community has focused mostly on malware detection. This situation is curious, because while it’s in everyone’s interest in cybersecurity to prevent exploitable bugs, agreement exists that this is basically impossible for nontrivial code. Vendors have thus built small empires based on this (so far) correct assumption.

Whereas the original methods of malware detection were built on matching application code (or operating systems) to signatures, more modern methods review behaviors for evidence of unacceptable runtime activity. Behavioral investigation is enabled by dynamic provision of virtual machines for safe detonation of executables. Without such virtual contained environments, behavioral analysis would be too dangerous for production systems.

Modern research in malware detection employs machine learning to help train security tools to identify bad code on the basis of samples. So, just as AI-powered systems are fed pictures of cats for learned recognition, comparable systems are fed “pictures” of files containing malware. Deep-learning techniques use massive parallelism to improve such algorithms’ efficiency.

Perhaps the unifying aspect of this evolving space is that malware detection tools presume the continued existence of problems, which helps justify business investment by start-ups and other security vendors. The likelihood is thus low that software professionals will advance our art to the point at which no malware exists. So, the antimalware industry should expect to see continued vibrancy of its collective offerings in terms of sales, revenue, and growth.

Software Process Maturity
Another focus in modern software security involves inferring code security through its associated software process. That is, many security experts have suggested that, rather than directly inspecting software for evidence of malware or vulnerabilities, you examine that software’s development process. This is like determining patients’ health by asking them about their behaviors rather than testing their blood.

The theory supporting this approach is largely empirical—namely, that good code has tended to come from well-trained developers working with world-class tools in modern, well-organized development environments. In contrast, exploitable vulnerabilities frequently have been found in code written by poorly
trained developers using make- 
shift tools in ad hoc development 
environments.

So, maturity models have emerged 
that let you link the degree of soft-
ware security to the quality of the 
process. This has the useful side ef-
fect of driving improved security for 
all code that emerges from a given 
vendor’s or team’s software process. 
Common methods demanded in such 
processes include automation, peri-
odic penetration testing, and proper 
software updating and maintenance 
procedures.

One excellent benefit of process 
maturity approaches is that little 
downside exists in any effort to im-
prove the steps taken to create code. 
If the underlying rubric is sound, the 
associated effort to bring the soft-
ware process in line with accepted 
best practices will have benefits far 
beyond just improved protection. 
Code reduction, time-to-market im-
provements, and quality increases 
will all result from improved soft-
ware processes.

Software Review 
and Scanning

The most traditional means for im-
proving software security involve 
direct inspection of code, some-
times using code-scanning tools. 
The tools’ earliest use seems to have 
been at Bell Labs in the 1970s, with 
the introduction of the lint prepro-
cessing program, which scanned C 
code and recommended improve-
ments. All subsequent code-scanning 
tools trace their lineage to this early 
concept.

The ongoing use of manual code 
reviews is much debated in the soft-
ware community, with traditional-
ists insisting that human inspection 
remains essential to high-quality, 
secure products. The challenge is 
that with the rapid cycle times in a 
DevOps environment, little time ex-
ists for human review of source code. 
Automated scans thus have become 
the norm in such environments; this 
has its pros and cons.

Software security will always in-
clude some degree of review and 
scans, presumably done properly 
once for reusable components, thus 
precluding the need for repeat se-
curity analysis. Critics claim that 
reusable componentry has been an 
elusive goal for decades. How-
ever, few would argue that modern 
DevOps and cloud-based software 
process environments are fertile 
ground for standard, well-reviewed 
components.

Runtime Software Controls

Perhaps the most promising advance 
in software security involves using 
runtime controls that are embedded 
in the execution environment. This 
technique is sometimes called runtime 
application self-protection (RASP). 
Through the integration of behav-
ioral and even machine-learning con-
trols into and around an executable, a 
programmed protection environment 
emerges—one that can compensate 
for code weaknesses.

RASP controls, cloud develop-
ment, and DevOps are all tightly 
 woven in most software develop-
ment organizations. All three aim to 
 increase delivered code’s speed and 
 flexibility. However, a somewhat 
open question is whether these three 
 initiatives result in more secure code. 
Certainly, RASP will reduce the risk 
of any application good or bad, but 
 it’s unclear whether programmers 
 write better code in the presence 
of RASP.

Nevertheless, runtime software 
controls will continue to influence 
software security, especially in the 
context of new self-learning meth-
ods. Machine-learning techniques 
have advanced to the point at which 
observed behaviors can serve as 
training data to label new variants 
of software exploits. This is an ex-
citing new way to drive improved, 
autonomous software control using 
platform automation.

Deep-learning advances are es-
pecially promising for software se-
curity. This is because the improved 
efficiency and massive parallelism 
that characterize the approach are 
perfectly suited to the large number 
of combinations that must be exam-
ined in typical software execution. 
We might hope that deep-learning 
algorithms would be a superior way 
to review code for unused execution 
paths, dead code, logic errors, race 
conditions, and the like.

O ur industry’s early focus on 
methodology, as evidenced 
by Edsger Dijkstra’s teach-
ings on software, remains an im-
portant consideration in the assurance 
of secure software. However, the 
community has taken many practi-
cal steps to improve code quality and 
security in the absence of any real 
correctness progress by program-
mers. Bugs still abound in nontrivial 
software, and security teams must 
be practical in their risk reduction 
efforts.

We can hope that in the coming 
years, these methods will synthe-
size with improved programming 
languages and ever-improving pro-
gramming techniques into an eco-
system that reduces risk by improving 
software. Given modern infrastruc-
ture’s dependency on well-designed 
code with a minimum of exploitable 
flaws, this is certainly a welcome 
goal.

©
Reference


ABOUT THE AUTHOR
EDWARD AMOROSO is the founder and chief executive officer of TAG Cyber. He’s also a Distinguished Research Professor in the New York University Tandon School of Engineering’s Computer Science Department, an adjunct professor of computer science at the Stevens Institute of Technology, and a senior advisor at Johns Hopkins University’s Applied Physics Laboratory. Contact him at eamorosotag-cyber.com.