Editorial

When Tradecraft Includes Code

Kaukab Jamal Zuberi

For decades, we treated programming as a specialist skill. A few “technical people” built tools, while everyone else focused on sources, languages, relationships, and judgment. That separation no longer holds. Modern security and intelligence work is saturated with digital traces—messages, metadata, location signals, transactions, device artifacts, and online influence campaigns. If you cannot work with data at speed, you often cannot work at all.

A recent public statement by a national intelligence service captured the shift bluntly: officers should be as comfortable with “lines of code” as they are with human sources and use artificial intelligence to augment—not replace—human skill.

The striking part is not the specific programming language. It is the admission that operational advantage is now created inside pipelines: collection to triage, triage to analysis, analysis to action, and action to learning.

There is a practical reason for this. The volume is too large for manual work. Even a straightforward task—de-duplicating records, correlating timestamps, identifying anomalies, mapping a network, or testing a hypothesis—requires automation. Code is how you turn a question into a repeatable method instead of a one-off effort. In digital forensics and threat intelligence, code is also how you standardize extraction, reduce human error, and document what you did so another analyst can reproduce it.

But there is a second reason that matters even more: adversaries live in code. Disinformation is automated. Scams are industrialized. Intrusions are scripted and scaled. Synthetic media is used to impersonate, recruit, extort, and misdirect. Major law-enforcement threat assessments now describe how criminal organizations exploit artificial intelligence to increase scale and realism, including impersonation and synthetic content.

When the threat is automated, a purely manual defense becomes a slow defense.

This is where the “learn to code” message can be misunderstood. The goal is not to turn every officer into a software engineer. The goal is baseline code literacy across the workforce, with deeper engineering depth in dedicated teams. Code literacy means you can read a script and see what it really does. You can validate an analysis rather than accepting it. You can spot when a tool is producing a “clean” output from messy assumptions. You can ask better questions of your technical colleagues—and recognize when the tool, not the adversary, is the problem.

That said, making code a default part of tradecraft introduces risk if it is not governed. A spreadsheet macro, a quick script, or a “helpful” automation can quietly become operational infrastructure. If it is not reviewed, version-controlled, and tested, it becomes a hidden vulnerability. Secure development guidance exists precisely because modern environments are targeted through the build process and toolchain, not only through frontline systems. The same lesson applies inside any sensitive organization: treat internal tooling as a defended asset.

There are well-established secure software practices that fit this moment. NIST’s Secure

Software Development Framework emphasizes secure build processes, auditing unexpected changes to tools, documenting lessons learned from code review, and validating authenticity and integrity of development tools.

In plain terms: scripts need peer review; dependencies need control; environments need hardening; logs need to exist; and someone needs to be accountable for what gets deployed and why. "Everyone can code" without these guardrails becomes "everyone can accidentally create risk."

Artificial intelligence raises the stakes further. If analysts begin relying on AI-assisted tooling for triage, summarization, translation, or pattern discovery, they must also understand AI-specific failure modes and attacks. Government guidance on AI cyber security highlights risks such as data poisoning and indirect prompt injection—cases where malicious content can steer an AI-enabled system through the data it consumes. This is not an abstract concern. It is a reminder that the model is part of the attack surface, and that "automation" can be manipulated if you do not design for residual risk.

So, what does a sensible code shift look like?

1. First, teach code as disciplined thinking, not as a trendy skill. The core is logic, verification, and documentation: "What did we do, on what data, with what assumptions, and can someone else reproduce it?"

2. Second, standardize safe ways to work. Approved libraries, controlled environments, and templates matter. They reduce both mistakes and improvisation.

3. Third, reward "boring" engineering. Reviews, tests, and audit trails rarely look heroic, but they prevent operational embarrassment and real-world harm.

4. Fourth, keep humans responsible for decisions. When AI is used, the human must remain answerable for the outcome—because accountability is not something you can outsource to a tool.

5. Finally, do not lose the older skills. Code is not a replacement for cultural understanding, source handling, or investigative judgment. It is a force multiplier. The best practitioners will be bilingual: fluent in people and fluent in systems.

In the end, this shift is less about modernizing an organization's image and more about reducing the distance between reality and response. The world now generates more digital "heat than light," as one public speech put it. Code, used responsibly, helps separate signals from noise. Used carelessly, it simply automates confusion. The differences will be governance, discipline, and the humility to remember that tools are only as trustworthy as the methods behind them.

.