kiuwan
by Sembi

Application Security Guide for COBOL

Protecting Legacy Systems Against Today's Security Threats

COBOL

Executive Summary

COBOL is more than 65 years old and nowhere close to retirement. An estimated 220 to 850 billion lines of it run in production today—in banks, insurance companies, government agencies, and retailers. It processes $3 trillion in transactions every single day, handles 95% of ATM transactions, and supports roughly 40% of global banking systems. The organizations that depend on it aren't going to rewrite it. They need to secure it.

COBOL's security risks don't come from the language being inherently broken. They come from the same factors that define any legacy system: accumulated technical debt, a shrinking pool of developers with deep COBOL expertise, and decades of updates layered onto infrastructure that wasn't designed for today's threat environment. The 2026 Sembi Software Quality Pulse Report found that legacy systems rank among security professionals' top challenges across industries—a signal that the COBOL problem is the norm, not an exception.

This guide covers six of the most common security vulnerabilities in COBOL applications: SQL injections, insufficient input validation, code smells, security validation control failures, supply chain and modernization risks, and defective code. Each section includes a checklist and concrete next steps. A final assessment checklist at the end can serve as your team's starting baseline.

Kiuwan, a Sembi company, is built for legacy language environments, including COBOL. Where relevant, this guide shows where static analysis fits into your security workflow.

The COBOL Legacy Language

> *COBOL processes more of the world's critical transactions than most people realize. Understanding why it's still here is the first step toward understanding why securing it matters.*

Common Business-Oriented Language (COBOL) is one of the oldest programming languages still in widespread use. Its origins go back to the 1950s, when it was developed through the Conference on Data Systems Languages (CODASYL) as an offshoot of the FLOW-MATIC language pioneered by Grace Hopper, with direct involvement from the U.S. Department of Defense. The goal was a portable, scalable language suited to data processing at scale, and by that measure, COBOL delivered.

Given its design for business and government rather than academic computer science, COBOL was quickly adopted across industries. Banks, insurers, government agencies, and large enterprises standardized on it fast. Newer languages arrived, but COBOL stayed. For most organizations, the cost and complexity of replacing deeply embedded COBOL systems have consistently outweighed the appeal of doing so. Today, COBOL is classified as a legacy language not because it has stopped working, but because it outlasted the generation of developers who built it. Many businesses and government agencies prefer to maintain COBOL applications that still meet their needs rather than invest in migrating to a newer language.

$3 Trillion USD

processed in COBOL-based transactions every single day

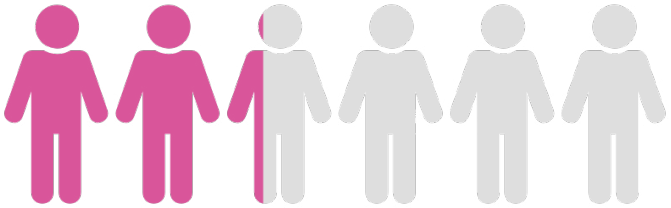
COBOL is widely used in mainframe computing, particularly for batch and financial transaction processing. It supports 80% of in-person credit card transactions and handles over 1.5 million banking operations per second with 99% accuracy. Given how deeply embedded it is in critical infrastructure, COBOL's security risks are everyone's problem, and addressing them starts with understanding what those risks actually are.

COBOL and Software Security

> *COBOL's design strengths—like user-friendliness, business orientation, and ease of implementation—are also the source of its most persistent security challenges.*

COBOL's development outside academic computer science has real implications for security. On one hand, it makes COBOL genuinely user-friendly for the business and government professionals who depend on it. On the other hand, that same design philosophy means COBOL can be more vulnerable to certain security risks than languages built with stricter engineering discipline from the start.

38.8% of security teams report being understaffed



Talent shortages are slowing security operations across the industry. In COBOL environments, where the pool of specialists is already narrow and retiring out of the workforce, this gap is even more pronounced.
(2026 Sembi Software Quality Pulse Report)

The talent problem compounds the risk. Fewer new developers are trained in COBOL, and the engineers who built and understand these systems are retiring. Organizations that still rely on COBOL applications are finding it increasingly difficult to locate IT professionals with the specific expertise to identify and address COBOL-specific vulnerabilities. The sections below cover the six most common vulnerabilities COBOL teams need to address, and how to take steps to remediate each one.

Common COBOL Security Vulnerabilities

1. SQL Injections

> *SQL injection is among the most common vulnerabilities in COBOL applications, and one of the most preventable. Fixing it means acknowledging how the openings for SQL injection are created.*

SQL injections exploit weaknesses in how an application processes SQL input. An attacker accesses the application from the client side and inserts malicious SQL code into an entry field, using it to access, modify, or destroy data within the system.

The consequences can be severe. A successful SQL injection on a bank's COBOL transaction system could allow an attacker to alter financial records, impersonate authorized users, or escalate to administrator privileges—locking out legitimate users, changing security settings, and exfiltrating data at scale.

COBOL systems are particularly exposed because they rely heavily on SQL statements for data input and output, and often lack sufficient parameter constraints around those statements. Applications that build SQL statements from raw user input rather than parameterized values carry the most risk. The same design choices that make COBOL accessible to non-specialist users create gaps that attackers can exploit.

The fix is parameterized inputs, in which the application's logic determines what goes into SQL statements rather than user-supplied data. Web application firewalls add a layer of friction for attackers, though they don't substitute for fixing the underlying code

SQL Injection Checklist

- [ ] All SQL statements use parameterized inputs, not raw user data
- [ ] Input validation controls are enforced at all SQL entry points
- [ ] Web application firewall is configured to cover COBOL-adjacent services
- [ ] SAST scans are run on all database-interfacing COBOL modules
- [ ] Access controls limit which users can execute high-privilege SQL operations

What to Do Next

- [ ] Run a SAST scan using Kiuwan Code Security targeting your batch transaction and database-interfacing COBOL modules
- [ ] Audit SQL statements currently built from user input and convert them to parameterized queries
- [ ] Review your WAF configuration to confirm COBOL-adjacent services are in scope

2. Insufficient Input Validation

> *Insufficient input validation is often framed as a training problem. It's a systems problem—and it opens the door to a range of attacks beyond SQL injection.*

Any application that accepts user input needs controls to sanitize that data before it touches the system. Without them, COBOL applications are open to SQL injections, cross-site scripting, and code injection—each of which can allow attackers to access data, bypass authentication, or shut out legitimate users.

Long-running COBOL environments are often exposed to this risk because of the language's age, its heavy reliance on user-input, and the limited pool of developers who understand COBOL validation patterns. Gaps often go undetected, not because teams are negligent, but because no one with the right knowledge is actively looking for them.

The fix is achievable without a full system rewrite. Ensuring developers and administrators understand which validations need to be in place, combined with a code analysis tool that surfaces gaps at scale, significantly reduces exposure.

Input Validation Checklist

- [ ] Input validation controls are defined and enforced for all user-facing fields
- [ ] XSS protections are in place for any web-facing COBOL interfaces
- [ ] Code injection protections are in place for all code-facing interfaces
- [ ] Development team is trained on COBOL input sanitization
- [ ] SAST tool is used to surface unvalidated input paths across the codebase

What to Do Next

- [ ] Run Kiuwan's Code Quality analysis to find unvalidated input paths across your COBOL modules
- [ ] Conduct a team training session on COBOL input validation patterns
- [ ] Prioritize interfaces that interact directly with financial or personally identifiable data

3. Non-Critical Code Smells

> *Code smells don't crash systems, and that's exactly what makes them dangerous. They persist quietly until the underlying issues they signal become exploitable.*

In programming, code smells are symptoms of a deeper problem, not critical failures in themselves. Non-critical smells are in some ways more persistent than critical ones: a system crash demands immediate attention, but a nuisance that doesn't break anything is easy to rationalize away. These issues don't always create immediate exploitabilities, but they make vulnerabilities harder to find, fix, and prevent. The underlying issue festers while the surface looks fine.

COBOL systems are particularly prone to code smells, partly because of the language's origins outside of computer science, and partly because of the decades of accumulated maintenance they've seen. Common examples include:

- **Long Parameter List**
  When a block of code has more than five or six parameters, it becomes harder to read, test, and validate—meaning security edge cases are more likely to slip through. A long parameter list usually signals that a method is handling too many responsibilities and needs to be broken down.

- **Duplicate Code**
  Code copied and pasted across an application rather than abstracted into reusable functions creates a critical maintenance problem: a fix applied to one instance doesn't propagate to the others. Residual vulnerabilities survive even after a remediation pass.

- **Bloaters**
  Methods or code classes that have expanded beyond their intended scope, doing too many things at once or operating without a clear function. Bloated code causes delays and system errors that can interfere with security controls and make the system more vulnerable to attack.

- **Dead Code**
  Unused variables, abandoned methods, or unexecuted code that clutters the codebase. Dead code makes review and debugging harder and can mask real vulnerabilities. If it serves no purpose, remove it.

- **Lazy Classes**
  Classes built in anticipation of features that were never implemented. They slow down the system and can compromise features such as input validation and parameter enforcement. Like dead code, they should be cataloged and removed.

The solution to all of these is treating code smells as early warnings rather than acceptable background noise. That requires visibility into the codebase at scale, which is where automated scanning becomes essential.

Code Smells Checklist

- [ ] Codebase is scanned for duplicate code blocks
- [ ] Long parameter lists are identified, and refactoring is scheduled
- [ ] Bloated methods are cataloged and a refactoring plan is in place
- [ ] Dead code is removed or documented
- [ ] Lazy/orphaned classes are cataloged and addressed
- [ ] Code smell findings are tracked and addressed in the regular development cycle, not deferred indefinitely

What to Do Next

- [ ] Run Kiuwan's Code Quality across your COBOL modules and generate a prioritized remediation list
- [ ] Establish a code review gate that flags duplicate code before it's committed
- [ ] Schedule a dedicated refactoring sprint for your highest-risk bloaters

4. Strict Input/Output Validation

> *When authentication controls fail, the instinct is to disable them rather than fix them. That instinct, understandable under pressure, is exactly what attackers count on.*

Users accessing data within a COBOL application should be authenticated. That's simple in principle. In practice, when authentication systems malfunction (and aging COBOL systems often do), the path of least resistance is to bypass or temporarily disable them. "Temporarily" has a way of becoming permanent.

COBOL applications face two specific authentication failure modes. The first is duplicate validation forms: multiple forms sharing the same name within the security protocol. When this happens, the system typically selects one arbitrarily and discards the other, creating authentication gaps that attackers can exploit to access critical data or shut down the system.

The second is erroneous validation methods; failures in the authentication process that block legitimate users from the application. The risk is that administrators disable security validation procedures entirely to restore access, rather than investigating the root cause. This leaves the system far more exposed than the original malfunction.

Both failure modes require the same fix: robust monitoring of validation controls, a clear process for investigating and resolving authentication errors, and a policy that treats "disable it for now" as a last resort with mandatory follow-up.

Security Validation Checklist

- [ ] All validation form names are unique, with no duplicates in the security validation layer
- [ ] Authentication failure alerts are routed to the security team in real time
- [ ] A documented process exists for investigating and resolving authentication errors
- [ ] No security validation controls have been disabled without a documented exception and remediation timeline
- [ ] Authentication and authorization logs are reviewed on a regular, documented schedule

What to Do Next

- [ ] Audit the names of your current validation forms and resolve any duplicates
- [ ] Review your incident log for any instances where validation controls were disabled, and confirm they were re-enabled
- [ ] Use Kiuwan to detect improper data validation patterns in your COBOL authentication modules

5. Supply Chain and Modernization Vulnerabilities

> *COBOL systems don't exist in isolation. The modern tools built around them, like APIs, build pipelines, logging gateways, and AI migration agents, create attack surfaces that the original language was never designed to address.*

Supply chain vulnerabilities weren't historically associated with COBOL, but the OWASP Top 10 for 2025 elevated supply chain failures to a standalone high-risk category—and for good reason. COBOL code is most at risk not through direct compromise but through the modernization scaffolding that surrounds it.

Consider how a modern financial application connects to legacy COBOL code: it needs APIs, build dependencies, logging services, and possibly AI-assisted migration layers. If a bad actor can infiltrate any of those—the build pipeline, a third-party dependency, a gateway service—the entire system is compromised, regardless of how secure the COBOL core is.

AI-assisted migration can introduce a specific and growing risk if not reviewed by COBOL experts. Teams migrating COBOL codebases using AI agents may encounter translation errors. A coding agent may misread COBOL's copybook structure or its complex nested logic (such as `GOTO` statements that govern critical batch processes), producing code that looks correct but behaves incorrectly.

Undetected before deployment, these errors can spread through the ecosystem, creating regulatory exposure, financial losses, or data breaches.

Supply Chain Checklist

- [ ] Build pipeline access controls are reviewed and restricted to authorized users only
- [ ] Third-party dependencies and surrounding components are scanned with SCA
- [ ] APIs used to bridge COBOL to modern systems are reviewed for known vulnerabilities
- [ ] AI-assisted code migration is reviewed by a human with COBOL expertise before deployment
- [ ] COBOL logic translation is reviewed for structural errors, particularly `GOTO` and copybook handling

What to Do Next

- [ ] Run Kiuwan Code Security on COBOL systems, and Kiuwan Insights on the surrounding third-party dependencies (i.e. APIs, surrounding libraries, etc.)
- [ ] Review and restrict access to your build pipeline—treat it as a high-value attack target
- [ ] Require human sign-off on any AI-generated migration code before it's promoted to production

6. Security Vulnerabilities Due to Defective Code

> *Decades of code accumulation mean decades of accumulated defects. In a codebase the size of the COBOL ecosystem, even a small defect rate represents a significant attack surface.*

An estimated 220 to 850 billion lines of COBOL are in use today, with roughly 1.5 billion new lines added annually. This code manages approximately $3 trillion in daily commerce, supports 95% of ATM transactions, and underpins roughly 40% of global banking systems. Experts estimate that 60% of COBOL systems carry hidden defects—the result of poor documentation, outdated business logic, and the ongoing talent shortage as experienced developers retire.

Not every defect is an exploitable security vulnerability. But the volume means treating defective COBOL code as a low-priority background problem is a gamble most organizations can't afford. Data-centric security is the most practical approach at this scale: rather than auditing every line of code, it separates the protection of sensitive data from the software it runs on, reducing exposure even when vulnerabilities can't be remediated immediately.

SAST is the most widely adopted type of security testing

Static application security testing (SAST) was identified as the top security testing method in the 2026 Sembi Software Quality Pulse Report, surpassing API security testing, penetration testing, and SCA. Teams using SAST detect vulnerabilities earlier and more reliably than those depending on manual reviews.

Kiuwan surfaces the defects most likely to create exploitable vulnerabilities and prioritizes them for remediation, giving security teams a manageable starting point rather than an overwhelming backlog.

Defective Code Checklist

- [ ] SAST scans are run across the full COBOL codebase, not just recent changes
- [ ] Known defects are tracked in a central register with an owner and a remediation SLA assigned
- [ ] Data-centric security controls are in place to protect sensitive data independent of the software layer
- [ ] Third-party libraries and dependencies in COBOL systems are scanned for known vulnerabilities (SCA)
- [ ] Continuous scanning is configured so that new code changes are checked before deployment

What to Do Next

- [ ] Start with a Kiuwan Code Security scan of your highest-risk COBOL modules—those handling financial transactions, authentication, or sensitive PII
- [ ] Implement a data-centric security layer to protect sensitive data even where vulnerabilities in the underlying code aren't immediately remediable
- [ ] Set up continuous scanning to check new code changes against your security baseline before deployment

The Cost of Not Securing COBOL Applications

> *COBOL's decades of deployment across banking, insurance, and government don't make it immune to security breaches. The financial and operational consequences of COBOL security failures are well-documented and growing.*

According to IBM's 2025 Cost of a Data Breach report, the financial sector remains among the top three industries most affected by breaches, with average damages of approximately $5.56 million. Federal agencies—many of which still run COBOL for Social Security, Medicare, tax processing, and other sensitive operations—face comparable exposure. The Government Accountability Office has repeatedly flagged these systems as critical, yet several agencies have been slow to implement modernization plans despite longstanding recommendations.

13.7% of security professionals named legacy systems as a top security challenge

In the 2026 Sembi Software Quality Pulse Report, legacy systems ranked alongside evolving threats and DevOps integration gaps as the most frequently cited pressures on security teams. This isn't a fringe concern—it sits at the center of how the industry is struggling to modernize fast enough.

The compounding effect of talent shortages makes the cost picture starker. Understaffed security teams don't just move slower, they cost more. IBM research cited in the 2026 Sembi Software Quality Pulse Report found that organizations with understaffed security functions face an average of $1.76 million in additional breach costs compared to those adequately resourced. For teams already stretched thin on COBOL expertise, that gap is even more pronounced.

The math cuts both ways. A targeted investment in COBOL security—like SAST, SCA, secure code review, and developer training—costs a fraction of what a single breach costs to remediate. For private sector organizations, the cost of inaction is direct: regulatory fines, litigation, and customer trust that's difficult to rebuild. For government agencies, it's taxpayer data and operational continuity. Investing in COBOL security today is a far more cost-effective strategy than responding to a preventable incident tomorrow.

Securing COBOL: *A Path Forward*

> *The vulnerabilities in this guide are serious. They're also addressable, without replacing your systems or rebuilding from scratch.*

Securing COBOL requires tooling that understands legacy language environments, a team that knows what to look for, and a structured process for finding and fixing problems before they become incidents. None of that requires a full migration.

Kiuwan's Code Security analyzes COBOL code at scale, identifying the vulnerability patterns covered in this guide—SQL injection risks, input validation gaps, code smells, authentication control failures, supply chain exposure, and defective code—and surfacing them in a prioritized, actionable format. Its code analysis capabilities extend that visibility to the third-party libraries and components surrounding your COBOL systems. Every scan, finding, and remediation is documented, giving compliance and risk management teams the audit trail they need.

Use the Final Security Assessment Checklist on the next page to baseline your current posture before you begin remediation.

Final COBOL Security Assessment Checklist

Use this checklist to baseline your current security posture. Work through critical items first.

CRITICAL — ADDRESS IMMEDIATELY

- [ ] SAST has been run across all COBOL modules handling transactions, authentication, or PII
- [ ] All SQL entry points use parameterized inputs—no SQL statements built from raw user data
- [ ] No security validation controls have been disabled without a documented exception and timeline
- [ ] Authentication failure alerts are routed to the security team in real time
- [ ] Build pipeline access is restricted to authorized users only

HIGH — ADDRESS WITHIN 30 DAYS

- [ ] Input validation controls are enforced at all user-facing fields
- [ ] All validation form names are unique, with no duplicates in the security validation layer
- [ ] Third-party dependencies and surrounding components are scanned (SCA)
- [ ] The web application firewall is configured and covers COBOL-adjacent services
- [ ] Authentication and authorization logs are reviewed on a regular, documented schedule
- [ ] Any AI-assisted COBOL migration code is reviewed by a human expert before deployment

MEDIUM — ADDRESS WITHIN 90 DAYS

- [ ] Codebase is scanned for duplicate code, bloaters, dead code, and lazy classes
- [ ] Long parameter lists are identified and refactoring is scheduled
- [ ] Continuous scanning is configured so that new code changes are checked before deployment
- [ ] Data-centric security controls are in place to protect sensitive data
- [ ] The development team is trained on COBOL-specific security and input sanitization patterns
- [ ] Code smell findings are tracked in the regular development cycle, not deferred

ONGOING — BUILD INTO YOUR DEVELOPMENT CYCLE

- [ ] COBOL codebase inventory is maintained and updated as new code is added
- [ ] Security testing is included in the release process, not run ad hoc
- [ ] Known defects are tracked with ownership and remediation SLAs assigned
- [ ] APIs bridging COBOL to modern systems are reviewed for vulnerabilities regularly
- [ ] Security posture is reported to leadership on a quarterly basis

Ready to Secure Your COBOL Systems?

Kiuwan is built for legacy language environments, including COBOL, JCL, and mainframe systems.

You don't have to modernize your entire stack to improve your security posture. You just have to know where the risks are.

See COBOL Security in Action

Discover hidden COBOL vulnerabilities with your first SAST scan—no commitment required.

Start a Free Trial

Talk to a Security Specialist

Not sure where to start? Our team works with legacy environments across financial services, government, and insurance.

Book a Live Demo

kiuwan
by Sembi