Carnegie Mellon University
Software Engineering Institute

EXAMPLES OF TECHNICAL DEBT'S CYBERSECURITY IMPACT

Robert Nord, Ipek Ozkaya, Carol Woody
July 2021

Executive Summary

In the course of development, evolution, and sustainment of software-intensive systems, accumulation of **technical debt** is common. Development teams often make tradeoff decisions among competing solutions, some of which leads to technical debt. When the path taken is expedient in the short term (e.g., deploying a feature more quickly with suboptimal architectural choices), but more expensive in the long term (e.g., work will have to be undone to "do it right), a project has usually taken on technical debt. Not all debt is bad, but unmanaged debt often leads to a number of avoidable situations, such as

- escalating sustainment costs
- increasing delays in delivering new features
- inability to fix software defects, vulnerabilities, and design issues due to growing complexity
- operational problems that degrade system qualities

Often driven by a schedule rush, many causes can contribute to accumulation of technical debt, which can adversely impact not only quality, cost, and schedule but also cybersecurity. Technical debt affects the design of the system, making it more difficult to locate and fix vulnerabilities.

Organizations have turned to methods such as DevSecOps to reduce cybersecurity risk, with varying results. DevSecOps seeks ways to reduce total cost, deliver on time, and improve productivity, all while improving quality and security. Although the term "DevSecOps" is often linked to tools and automation, experienced DevSecOps practitioners understand that tools and automation alone cannot ensure security, particularly where there is substantial technical debt, nor can tools and automation ensure that practitioners will recognize and eliminate technical debt in the first place.

Avoiding the negative outcomes associated with accumulating technical debt requires organizations to embrace technical debt management as a core software engineering activity and incorporate technical debt management with other project management and cybersecurity practices. A proactive focus on design and architecture provides a practical way to optimize the use of DevSecOps, and identify, prioritize, and mitigate technical debt-related cybersecurity risks that might otherwise be missed.

Managing technical debt is about how much risk and liability an organization is willing to take. Well managed technical debt can enable organizations to navigate the impact of challenging design tradeoffs during a systems development and sustainment. Unmanaged technical debt increases the overall cost of ownership and liability by increasing the risk of security issues, bugs, and sustainability costs in addition to opening up the system to quality issues. This document offers representative examples of technical

debt to provide guidance on how organizations can utilize and adapt their existing practices, development environments, and DevSecOps tooling to recognize and record technical debt in an effort to help supplement their cybersecurity and software quality management practices. These examples reinforce that technical debt is neither a special kind of defect or vulnerability, but it is a third kind of concept that is critical to recognize and manage to improve the overall quality, security, cost, and schedule posture of software system development. While existing processes, tools, and DevSecOps approaches help, they do not solve the problem without proactively managing technical debt and embracing related practices.

Understand the Consequences of Technical Debt

All organizations with long-lived software-intensive systems have to deal with technical debt. And not surprisingly, the longer the intended life of a system, the more important it becomes to manage technical debt as over the course of many modifications, the system will be exposed to opportunities to take on new technical debt and reduce the existing debt. Understanding that all systems have some level of technical debt is a critical first step to successfully managing it. Issues characterized as technical debt have a significant bearing on cybersecurity concerns and their resolution as well. Especially concerning vulnerabilities, developers often opt to deploy an update quickly to mitigate a security problem. This approach may result in developers failing to take the time to analyze the problem and get to the root cause of the issue. In cases where the update fails to fix the issue at the root cause, vulnerabilities will reoccur. The patches will introduce additional complexity, degrade the architecture of the system, and increase the time developers need to understand that area of the code. This is a canonical, yet far too common example of technical debt expanding cybersecurity risk because far-reaching effects of the bad fix and the rework it introduces are not recognized and the update does not fix the cause. Failing to recognize the technical debt increases both the total cost of ownership and cybersecurity risk.

A common operating definition of technical debt is essential for assessing, quantifying, and reducing technical debt, especially for those organizations that operate their systems in a large ecosystem where multiple internal teams and external organizations contribute to the development of the system. We recommend the following definition that highlights the riskier aspects of systems which accumulate added rework cost as an outcome of design decisions that impact systems' structure and behavior:

> *In software-intensive systems,* ***technical debt*** *consists of design or implementation constructs that are expedient in the short term but that set up a technical context that can make future change more costly or impossible [Kruchten 2019].*

Indicators of technical debt include symptoms (e.g., the situations listed above) and causes (e.g., friction related to processes, people, and the development infrastructure). Neither the causes nor the symptoms are the technical debt, but they are useful in detecting, prioritizing, monitoring, and preventing it. Causes and symptoms should be analyzed for actual technical debt so that strategies can be developed for its long-term management.

Sound software engineering practice includes the design and implementation of an architecture that meets the "-ilities" or quality attribute requirements. Internal software quality determines the maintainability and evolvability of the system. Other relevant quality attributes often include a combination of availability, performance, reliability, security, and interoperability, in addition to any other organization-specific quality concerns. The decisions made when designing an architecture are critical to achieve its quality attribute goals. And the cost associated with correcting them at a later time can be significant. Failing to address these architectural concerns will result in technical debt. Consequently, these flaws related to technical debt will increase the likelihood that security vulnerabilities will be inserted when the software is changed. Finally, technical debt when not managed escalates the total cost of ownership. When technical debt problems remain in production code, they potentially cause damaging operational events such as outages, data corruption, performance degradation, and security breaches in addition to increased rework and development costs.

While the software industry increasingly recognizes the importance of managing technical debt, organizations need specific guidance and examples to assist their effective tracking and quantification, in particular for understanding how technical debt may expand exposure to cybersecurity risk. This document summarizes representative examples of technical debt to guide organizations in utilizing and adapting their existing software engineering practices, development environments, and DevSecOps tooling to recognize and record technical debt as a first step in establishing practices for managing technical debt.

Build on Software Quality Management Practices

Three categories of issues must be managed to deliver high-quality software successfully: defects, vulnerabilities, and technical debt [Figure 1].

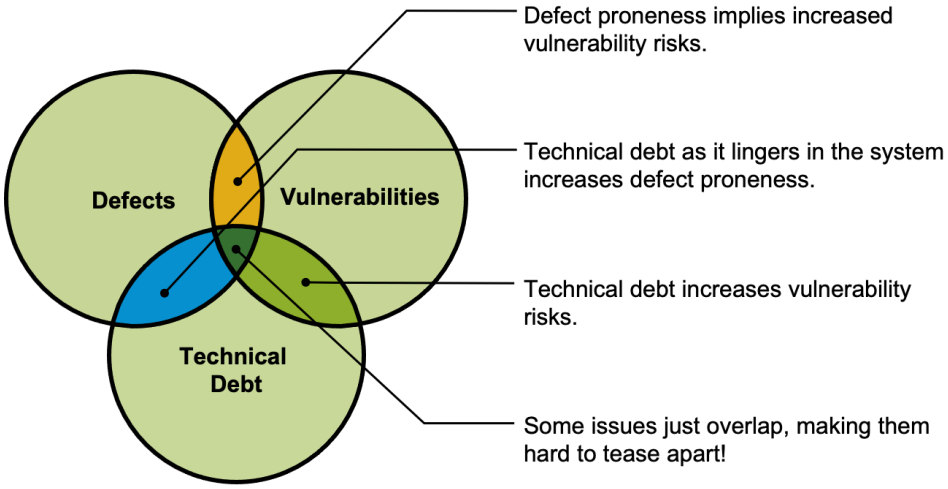

Figure 1: Categories of issues that need to be managed in software system development

Defects are errors in coding or logic that cause a program to malfunction or to produce incorrect and unexpected results. Most, if not all, defects should be caught through routine testing and code analysis practices including unit and acceptance tests. Vulnerabilities are weaknesses that can be accessed and exploited by a capable attacker. The criticality of a vulnerability is assessed by determining the risk it presents, where risk is a measure of the likelihood that a threat will exploit the vulnerability coupled with the magnitude of the resultant impact. The higher the risk, the higher the criticality. And lastly, technical debt consists of design or implementation constructs that make future changes more costly, issues that neither defects nor vulnerabilities effectively address.

There are subtleties in these definitions that drive the reasons why they need to be explicitly managed and overlaps where common quality management practices can help. Technical debt as it lingers makes it more difficult to modify the system which indirectly affects security by making it more difficult or time-consuming to fix vulnerabilities due to increased complexity. It might also make changing the

software more error prone and amplify defect and vulnerability risks. Research has identified that managing technical debt also assists in managing vulnerabilities [Nord 2016, Izurieta 2019].

Software engineering practices across the development life cycle support teams working together to plan, develop, deploy, and operate systems that meet the organization's programmatic, business, and mission goals. Developing and deploying high-quality software necessitates accepting that defects, vulnerabilities, and *technical debt items* all need to be actively managed to improve both the quality and the delivery tempo of a system. Consequently, the approach enumerated below for uncovering technical debt follows that for detecting any other issue in your system that may affect software quality and security. Uncovering technical debt, however, further emphasizes design and architecture choices and cost of change.

1. *Understand key programmatic, business, and mission goals.* Any issue occurs in the context of addressing a goal about market position, quality, productivity, or cost originating from a program's business and mission goals. Some organizations clearly communicate short-term and long-term goals, while some development teams have to infer the goals through the pain felt across their organization as a consequence of the debt they carry. The associated pain points are symptoms that can inform software analysis for identifying technical debt.
2. *Identify key concerns/questions about the system related to your goals.* A clear understanding of the goals will help identify the criteria you need to measure the concerns against. For technical debt, these will relate to the cost of change to enhance the software.
3. *Define observable qualitative and quantitative criteria related to your questions and goals.* Technical debt as a design or code construct is traceable to a concrete system artifact, such as code, build scripts, and automated test suites. Therefore, such criteria include the quantifiable effect on system attributes that worsens over time. Examples include increasing numbers of defects and security vulnerabilities, decreasing maintainability and code quality, and propagation of changes that trace to several locations in the system.
4. *Select and apply one or more techniques or tools to analyze your software for the criteria defined.* Selecting which tools to use is not a trivial process. The nature of the technical debt felt, as well as the key business and mission goals of the system, will often drive this selection. For example, while government programs are mandated to conduct automated security analysis, only some commercial organizations incorporate this into their basic testing practices.
5. *Document the issues you uncover that meet the criteria as technical debt items.* While development efforts expect to track security-related issues and defects explicitly, the same is not true for technical debt. An effective technical debt management practice starts with emphasizing the importance of tracking the uncovered technical debt issues.

The approach to uncovering and managing technical debt is not a distinct, independent, one-time activity but iterative and continuous and should be integrated with existing practices. Moreover, software programs that have decades of envisioned operational future must revisit their programmatic, business, and mission goals periodically.

Security initiatives strive to introduce processes to seamlessly monitor and mitigate cybersecurity risk across the application life cycle, utilizing automation where feasible. As the capabilities of automating

testing, integration, and conformance tools improve, DevSecOps will fulfill its promise of achieving faster and more reliable software delivery. However, DevSecOps is not a magic solution to resolving your technical debt. There are kinds of technical debt that an automated pipeline will not be able to detect and solve. For example, architecture decisions can be tough to automate and monitor. A DevSecOps pipeline, no matter how smooth the automation process, will not tell you whether you have selected the UI framework that best fits the user interaction you need to implement. While you can push patches and upgrades to runtime, these can actually accumulate technical debt rather than fix the problem at its source, increasing cybersecurity risks. An automated tool chain will not help you detect major re-architecting that may need to be done, as the software will continue to work. DevSecOps is a practice for improving software development quality and timeliness and can be an effective approach for intentional management of technical debt. However, DevSecOps does not replace a holistic technical debt management practice. It is only when supplemented with architecture analysis, the DevSecOps continuous analysis mindset will facilitate technical debt detection and management.

The next sections provide representative categories of examples of technical debt, including a discussion of how they can be identified and characterized using detection approaches that focus on a variety of artifacts.

Identify Technical Debt Items

An organization needs to actively monitor four categories of technical debt to ensure that existing DevSecOps, software quality, and security management practices are well aligned to also support technical debt management as part of the five-step process noted above. We organize these categories based on the artifact they are detected from.

1. *Detect technical debt from code*, where code-level conformance and structural analysis indicate maintainability and concerns related to the structure of the system and the codebase.
2. *Detect technical debt from symptoms* that signal architecture issues.
3. *Detect technical debt from architecture* during design reviews and analysis of decisions.
4. *Detect technical debt from development and deployment infrastructure*, which are not typically part of the delivered system but may impact its delivery, security, and quality.

To reason about technical debt, estimate its magnitude, and offer information on which to base decisions, you must anchor technical debt to explicit technical debt items that identify parts of the system: code, design, test cases, or other artifacts. A technical debt item is a single issue that connects affected development artifacts with consequences for the quality, value, and cost of the system triggered by one or more causes related to business, change in context, development process, and people and teams.

We demonstrate each of the four categories of technical debt detection with examples next. These examples of criteria, techniques, and technical debt item descriptions are from actual systems and developer discussions, drawing on the concepts of secure design [IEEE 2014], and abstracted for a general

audience. In order to exemplify the relationship between technical debt and cybersecurity in some of the examples, we refer to the Common Weakness Enumeration (CWE™). CWE is a categorized, publicly accessible list of software and hardware weakness types (https://cwe.mitre.org). The CWE was recently expanded to include quality characteristics such as maintainability that impact security [CISQ 2019].

Detect Technical Debt from Code

Technical debt takes different forms in different types of development artifacts. The source code embodies many design and programming decisions. The code can be subjected to review, inspection, and analysis with static checkers to find issues of finer granularity: while such analysis can detect some types of technical debt such as code clones and unnecessary complexity, almost all other violations detected will be symptoms that require some further analysis [ISO/IEC 2021, OMG 2018].

Static analysis checkers that are part of DevSecOps tool chains assist with detecting growing complexity, business logic nonconformances, and some basic classes of design issues such as very large classes and single points of failure. When not actively managed, all of these issues start accumulating unintended future rework, resulting in technical debt. Furthermore, typical examples of technical debt, such as greater complexity, increase opportunities for vulnerabilities.

Static analysis is not the only approach to examine code for technical debt and its symptoms. Examining the code at a high level with a focus on architecture is another approach to surface code conformance issues that result in technical debt. To understand the impact of change driven by technical debt, developers need to identify the modules of a system that are the focus of a change and follow the dependencies to the modules that will be affected by the change. Relevant characteristics for analyzing individual elements and their dependencies include complexity of individual software elements, interfaces of software elements, interrelationships among the software elements, system-wide properties, and interrelationships between software elements and stakeholder concerns.

Here is an example of a technical debt item that signals accumulating system complexity and uncovers needed design analysis and rearchitecting using static code analysis, which alerts for CWEs. In this example shown in Table 1, static code analysis the team regularly runs reveals many small, avoidable coding issues related to reliability, security, performance efficiency, and maintainability that were never addressed due to schedule pressure and lack of coding guidelines. Together they have caused the modifiability of the codebase to degrade.

™ CWE is a trademark of The MITRE Corporation.

Table 1: Example for recognizing technical debt with static code analysis

Name	Accumulated CWEs from violating maintainability quality rules resulted in technical debt.
Summary	Automated static source code analysis revealed an increasing number of issues with the following weaknesses and security implications of maintenance and evolution: CWE-561 Dead Code, CWE-1047 Modules with Circular Dependencies (120 issues), CWE-1074 Class with Excessively Deep Inheritance (37 issues). Due to the severe number of these issues, system modifiability has degraded significantly.
Consequences	We have already received two vulnerability reports in the dead code area; more are likely to emerge. There are increasing numbers of defects at the area of the codebase with the deep inheritance hierarchy. Modules with circular dependencies also take longer to incorporate new capabilities, increasing maintenance and evolution costs. In general, these areas of the codebase are difficult to maintain, which affects security by making it more difficult or time-consuming to find and fix vulnerabilities.
Remediation approach	Dead code: Remove the dead code. • Address during local refactoring within an iteration. Circular dependencies and excessive inheritance: These will require rearchitecting. • Designers need to understand how the architecture and evolution of the software influence security considerations under many circumstances. Address this in the next architecture review. The addition of continuous integration processes creates a requirement for architecture modularity and flexibility to support security, as changes to systems are pushed automatically and at ever shorter periodicity. • Understanding and restructuring module dependencies to eliminate circular dependencies and excessive inheritance will require planning across iteration boundaries.
Reporter/ assignee	The dead code and inheritance hierarchy issues were automatically reported as a result of the static code analysis scan: As the software development lead, I am reporting this as a composite technical debt item. I have also created two related issues in the backlog and linked to this issue: 1. Remove dead code (assigned to the developer team for the next iteration). 2. Remove circular dependencies and deep inheritance (assigned to the architect to resolve as part of the architecture refactoring effort).

It is important to recognize that there is no one-size-fits-all tool that automatically uncovers single instances of such technical debt items. Running a static analysis tool for the first time can yield thousands of issues. Recording all individual issues that tools identify as separate technical debt items or composing them as one major technical debt item is unwieldy and an incorrect approach. Furthermore, such an approach often leads to these issues lingering in the backlog as they are perceived as false positive noise, and developers might disable the rules for detecting them during future checks. Following the five-step process noted above to understand system quality goals provides a focus for the development team to create a manageable number of issues. They record the relevant results as technical debt items so they can start managing them. As this example highlights, identification of such violations will point to areas of further analysis to look at clusters of technical debt. Areas where large clusters of technical debt issues accumulate are good candidates for rework and architectural changes.

Going forward, the organization can address how to ensure that the team does not inject new debt into the source code so no one has to deal with these many issues again. The causes can be identified and process improvement practices put in place to address them. Creating coding guidelines and providing training for developers improves their savviness at recognizing when they potentially inject technical debt in the code. Running static analysis in a continuous integration environment promotes clean code

where developers get immediate feedback on the issues during a commit and are required to fix them before acceptance.

Detect Technical Debt from Symptoms

Technical debt symptoms are not always simple to recognize. Automated tools, such as tools that check for code quality or secure coding violations, can uncover some symptoms that signal technical debt. As seen in the previous section, one step in the right direction is to use agreed-upon CWEs associated with maintainability checks as a basis for identifying technical debt related to security issues [CISQ 2019]. This approach also helps with concrete quantification.

Other symptoms such as major faults or delivery delays in the system can also signal technical debt. Establishing continuous monitoring for such symptoms and reacting promptly will prevent technical debt from accumulating in the first place. For example, symptoms of technical debt can be exposed using metrics that indicate recurring defects and vulnerabilities, increasing number of defects and vulnerabilities in one particular area of the system, or defects that have not been possible to resolve, reducing delivery tempo. These should stimulate further analysis.

Repeated security breaches traced to security-related bugs, such as a crash or exploit enabled by an out-of-bounds number, are additional examples of technical debt items that can be detected by their symptoms. Table 2 summarizes such an example of a technical debt issue that increases vulnerabilities.

Table 2: Example for recognizing technical debt from observable symptoms

Name	Screen spacing creates numerous unexpected crashes across the codebase due to API incompatibility.
Summary	The source code uses a very large negative letter-spacing in an attempt to move the text offscreen. The system handles up to -186 em fine, but crashes on anything larger. A similar issue was fixed with a patch, but there were several other similar reports. Time permitting, I'm inclined to want to know the root cause of this. My sense is that if we patch it here, it will pop up somewhere else later.
Consequences	We already had 28 reports from seven clients. And it definitely leaves the software vulnerable. Finding the root cause can be time-consuming given that existing patches did not resolve the issue.
Remediation approach	We already patched this twice. The responsible thing to do is to first find the root cause and create a fix at the source. My previous experience tells me that the external Web client and our software again has an API incompatibility, but further analysis is needed. The course of action is to verify where the root of this is and see if we can fix it on our side. If the external Web client team needs to fix it, we would need to negotiate.
Reporter / assignee	DevSecOpsTeam / External WebClientTeam

While patches provide immediate relief, tracing interconnections in the design revealed a dependency on an external library maintained by another group, as the developer suspected. The dependencies to external software elements were not analyzed and designed for security issues, which resulted in multiple crashes across the system with the same root cause. Repeated crashes are symptoms pointing to the

technical debt in this example. They should trigger further architecture analysis and identification of the external dependency which, if not fixed, will widen security risk exposure. The additional rework is caused here by creating multiple patches, which increase system complexity without resolving the security issue.

A tendency sometimes exists to immediately categorize all of such symptomatic defects and vulnerabilities as technical debt. This approach results in an artificial increase in the number of issues while hindering the opportunity to do deep analysis and find the root cause. In a highly dynamic DevSecOps environment where organizations are under attack, the symptoms of vulnerabilities associated with an attacker's behavior need to be communicated between operations and development teams to trace operational weaknesses to root cause vulnerabilities in the source code. This further refines the goal of using a static analysis tool to address the vulnerability associated with attackers' behavior, rather than executing static analysis tools out of context and trying to deal with the myriad results [Izurieta 2019]. The same mindset needs to be embraced when using such tools to detect symptoms to identify and mitigate technical debt.

Detect Technical Debt from Architecture

The key difference between detecting technical debt using code analysis and detecting it at the architecture level is that the code is more concrete, tangible, and visible. It can be explored using software tools, but that provides information at a lower level of granularity, sometimes giving the impression that fixing local issues will eliminate technical debt. Code analysis does not reveal system-wide, systematic architecture issues which hint at technical debt. Architecture analysis reveals technical debt that is more encompassing and pervasive. It involves choices about the structure or the architecture of the system: choice of platform, middleware, technologies for communication, user interface, or data persistency. It is typically more difficult to detect and assess architectural decisions resulting in debt with tools, and the cost associated with repaying the debt is larger and intertwined in a complex network of structural dependencies.

Architecture analysis allows a team to assess whether design decisions will meet the quality attribute requirements early in development. Malicious external attacks that expose the vulnerabilities of a system at runtime are lagging indicators of the failure to meet a security quality attribute requirement. As operations staff employ countermeasures, development staff trace the cause to the source code vulnerability to aid in patching the system in a first response. Tracing further to the root cause when there is a design or architecture issue and remediating the technical debt can prevent the issue or related issues from resurfacing and benefit the business by positioning the system to make it easier to analyze, maintain, and evolve over its life span.

Lightweight architecture analysis techniques surface risks in design decisions that can lead to technical debt. A number of analysis techniques have proven useful for examining the architecture as it is being designed and used throughout the software development life cycle: thought experiments, reflective

questions, checklists, scenario-based analysis and walkthroughs, analytics models, prototypes, and simulations. Developers often use existing frameworks and components to provide some of the structure and behavior of the system. The choices made to use these frameworks and components are design decisions that affect the quality and security of the system. In this example of Table 3, a design decision made early in the development effort has resulted in a security breach.

Table 3: Example of recognizing technical debt requiring architecture rework to enhance security

Name	Missing Authentication for Critical Function (CWE: 306) requires significant architectural rework.
Summary	The authentication for functionality for user identity management had been assumed out of scope in the first release. This resulted in the recent security breach and compromised the data in the system. No critical information was compromised; however, we cannot continue to operate before adding authentication.
Consequences	Given the number of features that depend on this, we are looking at significant rearchitecting. The consequences will depend on the associated functionality, but we will have to reassess read/write accesses to our sensitive data and recreate administrative and other privileged functionality.
Remediation approach	Divide the software into anonymous, normal, privileged, and administrative areas. Identify which of these areas require a proven user identity, and use a centralized authentication capability. Identify all potential communication channels, or other means of interaction with the software, to ensure that all channels are appropriately protected. Our developers sometimes perform authentication at the primary channel but open up a secondary channel that is assumed to be private. For example, a login mechanism may be listening on one network port, but after successful authentication, it may open up a second port where it waits for the connection but avoids authentication because it assumes that only the authenticated party will connect to the port. In general, if the software or protocol allows a single session or user state to persist across multiple connections or channels, authentication and appropriate credential management need to be used throughout.
Reporter / assignee	Reported by a Dev engineer during system integration test. Remediation assigned to multiple team members including the DevSecOps team and lead architect.

CWE-306, Missing Authentication for Critical Functionality, is a vulnerability that enables attackers to gain the privilege level of the exposed functionality. The technical impact of the weakness can be used to determine the cost to the development team of carrying the technical debt and the risk exposure to the business. Manual analysis is needed to understand the underlying design issue and the cost of remediating the debt by improving the design.

Tradeoffs made among system qualities to meet the organization's mission or business goals may lead to such technical debt. For example, since authentication consumes system resources and results in timing lags that can degrade performance, the decision may be made to omit reauthentication given the context (e.g., authentication occurs in the control panel software but not in the vehicle it is operating). As hardware performance improves over time and software changes enlarge the attack surface, this decision should be revisited. Whether it is easy or difficult to reinsert authentication depends on whether architecture decisions made early on will support this kind of evolution. Recording this as a technical debt issue proactively gives the team an opportunity to revisit the decision as hardware and software assumptions evolve, and resolve it in a timely fashion. Even better, if the technical debt item is acknowledged and recorded at the time the decision is made—that is, when the decision to skip authentication

was agreed upon—architects and designers could consider other choices that would simplify reintroducing authentication at a later time.

Detect Technical Debt from Development and Deployment Infrastructure

Technical debt also occurs in the development and deployment infrastructure. This section describes two examples of technical debt, one related to the suboptimal design and coding of test infrastructure (Table 4) and another to misalignment between the infrastructure and the code itself (Table 5).

Infrastructure has become a key software development artifact. Analyzing for technical debt in the infrastructure that serves the completed code to a running system in operation encompasses issues in build, test, and deployment code. Current DevSecOps trends are increasing automation capabilities and tool support, and these trends have exposed deficiencies in the production process used by development organizations. Infrastructure-related technical debt impedes a team's ability to evolve a system or fix known issues. These problems often influence an organization's ability to achieve business goals, particularly if they slow velocity or hinder the ability to release in small rapid increments. Analysis techniques for code and design can be applied to build scripts, test suites, and deployment scripts to detect the presence of technical debt.

Consider the following first example of test suites. Test suites are, in effect, code. Suboptimal design and coding of tests also lead to the same weaknesses as with the product code that have security implications related to maintenance and evolution. In this example, the development team would like to reuse new Test Helper modules for a legacy test framework. The development team is migrating integration tests to the new test framework. There are two parallel sets of Test Helper modules to maintain during migration. Duplication is a source of technical debt and requires changes in two places. Often, changes are not synced, resulting in unintended drift between frameworks. The remediation approach allows the legacy test framework to reuse the new test framework's Test Helper modules, which are cleaner (better documentation, linted, obvious errors fixed). The technical debt item exemplified in Table 4 shows the team's analysis to get insight into the maintainability of the test framework.

Table 4: Recognizing technical debt in the test infrastructure

Name	Maintaining two parallel Test Helper modules results in inconsistencies.
Summary	While the DevTeam has been migrating its integration tests to the new test framework, there have been two parallel Test Helper modules to maintain, one for the new framework and another for the legacy framework. The redundancy is resulting in inconsistencies and unneeded work.
Consequences	This test code is a source of technical debt and requires team members to make changes in two places. Often, they forget, which leads to unintended drift between the two frameworks. Scaling this infrastructure to dozens of teams will magnify the challenges as we roll out the testing framework.
Remediation approach	Reuse the new test framework's Test Helper modules. The goal isn't 100% code reuse between the old and new test framework, but 80–90%. The test methods from the legacy module that remain are here for three reasons:

	• When ported to the new test framework, the test methods were refactored into different modules and will require updating legacy tests to load new modules. • Navigating the page in the old test framework is hacky and has been cleaned up in the new test framework so they won't ever share implementations. • Subtle refactoring changes make the new implementation fail certain tests. This test failure should be followed up by using the old implementation and then refactoring once all tests have been migrated.
Reporter / assignee	DevTeam / QATeam

The misalignment of the build, test, deployment, and delivery strategies and accompanying tools is another area where technical debt appears in the development and deployment infrastructure. Technical debt can appear in the misalignment between the infrastructure and the code in the following ways:

- *Testing.* As software evolves rapidly, new tests may be missing, may test an older interpretation of the requirements, or may interact with other tests in unknown ways.
- *Infrastructure of the operational system.* Deferred binding generates a responsibility for the development team to make architecture decisions to accommodate the change during deployment, delivery, and runtime and a responsibility for the staff of the operational system to make the change.

In the second example in Table 5, the security implications of a change request impact not only the code but also its alignment with test, deployment, and delivery. These issues are documented as a technical debt description and included in the backlog.

Table 5: Recognizing technical debt within infrastructure misalignment

Name	Database misalignment with continuous delivery pipeline impacts security during upgrade.
Summary	A database engine upgrade reveals that the security implications of the upgrade are not well understood and controlled. Secondary and tertiary dependencies are not well documented or understood. These dependencies are presently precluding us from completing the upgrade because we are constantly running into issues.
Consequences	Designers need to understand how change influences security considerations under these secondary and tertiary dependencies. The need for security considerations will appear during continuous delivery in • testing, since all possible variations of states will need to be verified to guarantee that they uphold the security posture of the system (among, of course, other tested behavior) • deployment, when permissions, access control, and other security-related activities and decisions need to take place • delivery and runtime, in the form of configuration changes, enabling and disabling of features, and sometimes dynamic loading of objects The addition of continuous integration processes creates a requirement for security flexibility, as changes to systems are pushed automatically and at ever shorter periodicity.
Remediation approach	Analyze for the database and infrastructure dependencies and rework the design for secure updates.
Reporter / assignee	Reported by the Ops engineer doing the upgrade. Remediation assigned to multiple team members including the DevSecOps team and lead architect.

The organization needs a deliberate strategy for managing technical debt not only for development but also for testing and production. An agile or flexible architecture complements continuous integration processes and allows the team to explore technical options rapidly with minimal ripple effect. The architecture can be understood in terms of design decisions that influence the time and cost to implement, test, and deploy changes and operate the software without introducing bugs and vulnerabilities.

When there are distributed teams, coordination issues can create misaligned assumptions about design decisions, which can cause technical debt. Distributed teams face coordination challenges as the architecture is apportioned to them for implementation and again when they hand off their implementations to an integrated testing environment. Tests and infrastructure should be designed and aligned for their purpose, implemented following sound coding practices, and executed in alignment with the functionality and attributes they are meant to support.

Prioritize and Monitor Technical Debt at All Levels of Planning

The technical debt items identified through analyzing the code, symptoms of technical debt, architecture, and development and deployment infrastructure are managed with the rest of the tasks and stories in the backlog. A *technical debt item* description provides a systematic way of capturing a technical debt item and its properties. The examples summarized in Tables 1-5 demonstrate some typical technical debt items. Recording these examples in the backlog using a structured technical debt item description enhances the clarity by documenting who, what, where, when, and why. As demonstrated in our examples, a description that allows stakeholders to monitor the debt and take appropriate action has the following parts:

- **Name**. ***What*** is it? This field is a representative name for the technical debt item.
- **Summary**. ***Where*** do you observe the technical debt in the affected development artifacts, and where do you expect it to accumulate?
- **Consequences**. ***Why*** is it important to address this technical debt item? Consequences include immediate benefits and costs as well as those that accumulate later, such as additional rework and testing costs as the issue stays in the system and costs due to reduced productivity, induced defects, or loss of quality or security incurred by building software that depends on using elements that have technical debt.
- **Remediation approach**. Describe the rework needed to eliminate the debt, if any. ***When*** should the remediation occur to reduce or eliminate the consequences?
- **Reporter / assignee**. ***Who*** is responsible for serving the debt? Assign a person or team. In some situations, the debt resolution may need to be assigned to external parties. If remediation is significantly postponed, this field can communicate that decision.

Prioritizing technical debt items for resolution in the backlog is no different than managing the backlog in general. Therefore, a key to a successful technical debt management practice is to start recording the items so that known metrics and techniques can be used, such as the number of technical debt items, their resolution time, and their key detection mechanism. Which technical debt items get prioritized for

resolution depends on the level of detail of the information recorded as technical debt items are identified as well as the priorities of the system. A technical debt item description is most effective when it is concretely linked to the needed improvement to the system. For improved efficiency and effectiveness, consolidate technical debt items or link to related issues in the backlog where possible, considering answers to these questions:

- In what ways are technical debt items related to development of features and cybersecurity capabilities visible to the customer?
- What architectural decisions have an impact on technical debt?
- What bugs and vulnerabilities are consequences of a technical debt item?
- Which of these bugs and vulnerabilities are operational problems?
- Are any technical debt items blocking progress?
- Do any technical debt items need to be refined?

Technical debt should be a concrete consideration at every level of software development planning: *iterations, releases,* and *system increments.*

A plan to remediate smaller code quality issues which signal technical debt items might involve allocating a fixed percentage of resources for an *iteration* to service technical debt. This is analogous to adding a buffer of time within an agile sprint for fixing defects. A fixed percentage gives a software development team the discretion to deal with code quality issues while controlling spending. In cases of extreme debt, you might allocate an iteration or two to work on paying back technical debt.

A plan to repay technical debt that shows problems with design will affect the cost of evolving the system, which in turn will affect the decision to repay some debt or evolve the system. Reasoning about architecture alternatives and using the architecture to guide implementation choices during *release planning* provides opportunities for managing technical debt. Here is an approach for developing a plan to manage technical debt while you maintain and evolve a system:

1. Choose an item in the backlog and plan development tasks as usual (e.g., add a new feature, resolve a defect or vulnerability).
2. Identify the parts of the system that will be affected by the item chosen from the backlog.
3. Determine whether other technical debt items are associated with these parts of the system.
4. Identify the consequences of technical debt on this and possibly other changes.
5. Estimate the cost of the debt repayment and add it to the cost of the change.
6. Estimate the benefit of the debt repayment in enabling the development of this and possibly other changes.

This approach is contingent upon grasping which areas in the system have more technical debt as well as generating a few maintenance and evolution scenarios to compare potential outcomes. Experienced teams consider aspects of evolution as they debate design options, backlog grooming, and technology change.

Conducting these discussions explicitly for the technical debt items will improve a team's understanding of the consequences and help members make decisions based on the benefit gained by fixing the related technical debt items. The team will also become comfortable with understanding tradeoffs concretely and deciding what technical debt items to not resolve, but continue to monitor. Monitoring the status of existing technical debt items in the backlog will include revisiting their status during sprint retrospectives and architecture reviews and monitoring their dependent metrics such as number of new defects and vulnerabilities, velocity, and any specific metrics that relate to the context of the technical debt items, such as complexity.

A technical debt repayment plan for a full *system increment* can be created at planning events for portfolio-level considerations. This event provides an opportunity for stakeholders from multiple development teams and product management to develop plans in support of the system-level needs of the increment and the roadmap in support of program goals.

Software developers and architects bring knowledge of architecture and any related technical debt recorded in the backlog. This includes insight into architecture from code quality metrics and any project-related data regarding code churn, conceptual design integrity, and defect data. The product manager brings knowledge of the roadmap for the next three to six months. This includes what parts of the architecture are involved and the effort required to deliver each roadmap item.

Given these inputs, following agile planning approaches, the stakeholders together create a repayment plan that identifies key features and cybersecurity capabilities not possible without debt repayment and potential technical debt repayment that may be deferred. Other outcomes of the planning event include a shared strategic vision for paying down technical debt and commitment to position the architecture to carry the project into the future.

Manage Technical Debt Continuously Throughout the Acquisition and Development Life Cycle

Technical debt management is most effective when it is interwoven into current software engineering practice regarding features, defects, vulnerabilities, security risk, and process. The benefit of technical debt management is that it is focused on issues that current software engineering practices have not historically tracked and managed systematically and clearly. Explicitly managing technical debt presents an opportunity to represent tradeoffs among architectural design decisions and their changing consequences as a system evolves.

Beginning with a quick sanity check of the business goals against the system architecture, development practices, and organizational context will provide guidance for successfully executing a deeper analysis of the system, determining actionable outcomes, and formulating a strategy for managing technical debt. These key criteria should also be reviewed during important milestones such as the handover of a software project from contractor to sustainment organization. The contractor is expected to identify all technical debt items and work with the government program office and stakeholder community to prioritize

remediations based on program goals and system-level needs prior to final qualification and release of the software.

Monitoring technical debt will provide continuous visibility into the decisions being made by the software development teams and product owners to better determine when corrective action is needed. This information will also help teams identify how technical debt is injected into the system and develop strategies to prevent its further accumulation.

Approach technical debt identification, monitoring, and remediation in an incremental and iterative manner. Include the following:

- *Definition of done criteria* – Identify and document technical debt that becomes a hindrance during development. The kinds of technical debt items uncovered can initially be based on team experience and software engineering practice. These later can be augmented with metrics collected by the project (e.g., from commit history, issue trackers, static analysis tools, and design reviews).
- *Definition of ready* – Check the backlog before starting a new feature or fixing a defect or vulnerability to assess technical debt items that should be considered for remediation during implementation.
- *Planning* – Consider technical debt prevention at every level: iterations, releases, system increments, and product roadmaps. Look to include remediation with the code being updated. Consider the effort to remediate or prevent any new debt when estimating the size of a feature.
- *Retrospectives* – Ask if technical debt has been discovered, not only in the system under development but in the production infrastructure; then create technical debt items. Monitor how debt is accumulating, and look for productivity and quality metrics that indicate when it is time to take action on remediating the debt.

The table below provides a sampling of indicators that facilitate analysis in detecting technical debt. Tools that support code analysis are becoming increasingly sophisticated and often support dependency analysis as well. Lightweight architecture analysis techniques have proven useful and cost-efficient early in the software development life cycle. Given advances in script-driven automation supporting integration and deployment, analyzing the technical debt in production infrastructure shares some indicators with technical debt in code and architecture. It also introduces new challenges in monitoring alignment between production infrastructure and code.

Detect technical debt from …	**Sample indicators**
Code	*Maintainability and evolvability violations against established industry measurement standards:* ISO/IEC 25010 standard for system and software quality, SEI CERT Secure Coding Standards, relevant CWE measures *Code complexity measures*: combination of source lines of code, coupling and cohesion, fan-in/fan-out, dependency propagation associated with the current maintenance costs Architecture insight from code • *Complexity of individual software elements*: lines of code, module size uniformity, cyclomatic complexity

	• *Interfaces of software elements*: dependency profiles identifying hidden, inbound, outbound, and transit modules; state access violation; API function usage • *Interrelationships among the software elements*: coupling, inheritance, cycles • *System-wide properties*: change impact, cumulative dependencies, propagation, stability • *Interrelationships between software elements and stakeholder concerns*: concern scope, concern overlap, concern diffusion over software elements
Symptoms	*Defect and vulnerability trends*: recurring defects and vulnerabilities, increasing number of defects and vulnerabilities in one particular area of the system, or defects that have not been possible to resolve, reducing delivery tempo *Software development trends*: amount of time spent patching, changing velocity, potential effort spent per violation
Architecture	*Risks in design decisions surfaced using lightweight analysis techniques*: thought experiments, reflective questions, checklists, scenario-based analysis, analytics models, prototypes, and simulations *Risks in design decisions surfaced using model-based techniques:* simulations, experiments and measurements against expected concrete response measures such as latency and availability
Development and deployment infrastructure	Similar indicators as above for code and architecture applied to infrastructure Technical debt in production appears in misalignment of the build, test, deployment, and delivery strategies and accompanying tools. Indicators include • *build and integration*: unwanted and/or dead dependencies (links to unused code), zombie targets (no builds for months), unbuildable code, dead flags • *testing*: lots of preprocessing required for test automation, tests that are time intensive or difficult to set up, difficulty automating certain tests, test cases or test harnesses that are hard to modify or extend • *Infrastructure*: manual tasks repeatedly performed (e.g., deployment scripts), lack of observability (monitoring debt) Runtime efficiency and security checking provide additional operational indicators that should be communicated to development.

In Conclusion

We described in this paper that technical debt is a third critical issue category that should be explicitly managed, similarly to defect and vulnerabilities. As our examples illustrated, understanding technical debt and managing it explicitly provide opportunities to manage its architectural tradeoffs more proactively, which reduces a system's cybersecurity risk exposure. Consequently, it is critical to emphasize that technical debt is not simply a project management panacea where all unplanned and to-be-done work can be lumped as technical debt.

A key aspect of any successful technical debt management strategy is to recognize that a *cause* contributes to the occurrence of technical debt in the system, that a *symptom* is an indicator of technical debt, and the cause or symptom is not the technical debt itself. Recognizing what is *not* technical debt, though it may be a cause or symptom, will also assist in assessing how to take advantage of, and augment where possible, existing acquisition practices to manage technical debt. A few guidelines will help an organization get started on managing the technical debt that is expanding both cybersecurity risks and other risks to a system.

- *New features not yet implemented are not technical debt, but misunderstood requirements can cause it*. Aspects of evolution visible to the user in the form of additional functionality are distinguished from the technical debt issues that are visible only to software developers. Requirements shortfall—in the form of not understanding architecturally significant requirements such as security, performance, and availability that crosscut the system—will cause technical debt.
- *Low external quality in the form of defects and vulnerabilities are not technical debt but rather possible symptoms.* Most defects and vulnerabilities have an immediate impact and the need to resolve them is well understood. Fixing them will not necessarily fix the underlying cause, as our examples demonstrated. Technical debt will be felt in the future in the form of additional costs and challenges when upgrading functionality and remediating security vulnerabilities. Technical debt with underlying architecture issues should be considered during release planning. Open defects and vulnerabilities that accumulate other costs and have complex remediation strategies can be symptoms of technical debt.
- *Lack of following software development processes is not technical debt, but is likely to cause technical debt.* Due to resource constraints, not all software life-cycle activities may be completed on time, such as reviewing code and design, running all of the test suites, or documenting the complete architecture. However, improving the processes will not fix the technical debt that has accrued in the system. Effective technical debt reduction involves understanding how undisciplined ways of executing processes influence the system, create unintentional system complexity, and result in technical debt. Process improvement need to be part of a focused strategy to prevent some of the new debt from occurring.

All systems have technical debt. Since technical debt reflects consequences of tradeoffs, not all debt is bad or needs to be resolved immediately. However, unmanaged debt will bring large-scale, long-lived systems to bankruptcy sooner or later.

Adding technical debt management to your existing software architecture and DevSecOps practices will help manage your cybersecurity risk and improve software quality. Incorporating an explicit focus on identification and documentation of technical debt items is an essential first step for organizations who are just starting their journey of managing technical debt proactively. The examples summarized in this paper are intended to guide that process by focusing attention on the distinct categories from which to detect technical debt.

Bibliography

[CISQ 2019] CISQ. List of Weaknesses Included in the CISQ Automated Source Code Quality Measures. June 2019. https://www.it-cisq.org/pdf/cisq-weaknesses-in-ascqm.pdf

[IEEE 2014] IEEE Center for Secure Design. Avoiding the Top 10 Software Security Design Flaws. Be flexible When Considering Future Changes to Objects and Actors. September 2014.

[ISO/IEC 2021] ISO/IEC 5055: Automated Source Code Quality Measures. March 2021. https://www.it-cisq.org/standards/code-quality-standards/

[Izurieta 2019] Clemente Izurieta, Mary Prouty. Leveraging SecDevOps to Tackle the Technical Debt Associated with Cybersecurity Attack Tactics. TechDebt@ICSE 2019: 33-37

[Kruchten 2019] Philippe Kruchten, Robert Nord, Ipek Ozkaya. *Managing Technical Debt: Reducing Friction in Software Development*. Pearson. 2019.

[Nord 2016] Robert L. Nord, Ipek Ozkaya, Edward J. Schwartz, Forrest Shull, Rick Kazman: Can Knowledge of Technical Debt Help Identify Software Vulnerabilities? CSET @ USENIX Security Symposium 2016.

[OMG 2018] Object Management Group. Automated Technical Debt Measure. September 2018. https://www.omg.org/spec/ATDM/1.0/PDF

Contact Us

Software Engineering Institute
4500 Fifth Avenue, Pittsburgh, PA 15213-2612

Phone: 412/268.5800 | 888.201.4479
Web: www.sei.cmu.edu
Email: info@sei.cmu.edu

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM21-0690