Developer Ecosystems for Software Safety

CHRISTOPH KERN

CONTINUOUS ASSURANCE AT SCALE

How to design and implement information systems so that they are safe and secure is a complex topic. Both high-level design principles and implementation guidance for software safety and security are well established and broadly accepted. For example, Jerome Saltzer and Michael Schroeder's seminal overview of principles of secure design was published almost 50 years ago,[11] and various community and governmental bodies have published comprehensive best practices about how to avoid common software weaknesses—for example, CWE (Common Weakness Enumeration) and OWASP (Open Worldwide Application Security Project) Cheat Sheet Series.

Despite these efforts, common types of software defects prevail, and many occupy top ranks of "worst vulnerabilities" lists such as the OWASP Top 10[10] or the CWE Top 25 Most Dangerous Software Weaknesses[4] for years if not decades.

Based on work at Google over the past decade on managing the risk of software defects in its wide-ranging portfolio of applications and services, the members of Google's security engineering team developed a theory about the reason for the prevalence of defects: It's simply too difficult for real-world development and operations teams to comprehensively and consistently apply the available guidance, which results in a problematic rate of new defects. Commonly used approaches to find and fix implementation defects after the fact can help (e.g., code review, testing, scanning, or static and dynamic analysis such as fuzzing), but in practice they find only a fraction of these defects. Design-level defects are difficult or impractical to remediate after the fact. This leaves a problematic residual rate of defects in production systems.

We came to the conclusion that the rate at which common types of defects are introduced during design, development, and deployment is systemic—it arises from the design and structure of the *developer ecosystem*, which means the end-to-end collection of systems, tooling, and processes in which developers design, implement, and deploy software. This includes programming languages, software libraries, application frameworks, source repositories, build and deployment tooling, the production platform and its configuration surfaces, and so forth.

In short, the safety and security posture of a software application or service is substantially an emergent property of the developer ecosystem that produced it. (A safe system mitigates risks of relevant harms and adverse outcomes for its users and stakeholders. A secure system does so even in an adversarial context. Security is about defending against active threats, beyond accidents or mistakes.[17])

It follows that to truly improve the situation, focusing on design and implementation guidance in the context of individual applications comes too late in the process.

Instead, development and operations teams need to shift-left even further and incorporate software safety and security considerations in the design of developer ecosystems. (While this article focuses on safety and security, many of the principles and practices discussed here transfer to reliability engineering, and it is often helpful to consider security and reliability together.[1])

This article argues, based on experience at Google, that focusing on developer ecosystems is both practical and effective, and can achieve a drastic reduction in the rate of common classes of defects across hundreds of applications being developed by thousands of developers.

There are two key aspects to this approach for achieving assurance at scale.

➡ **Preventing bugs through Safe Coding.** First, many common implementation-level security defects, such as injection or memory safety vulnerabilities, are difficult to avoid entirely in large and complex codebases, even for experienced developers who thoroughly understand the nature of the vulnerability in principle. When a codebase has many instances of coding patterns that are *potentially vulnerable*—placing the onus on developers to be careful every single time—defects will happen.

Thus, the only approach that can significantly reduce the rate of defects is for the developer ecosystem to take responsibility for preventing vulnerabilities by presenting a *Safe Coding* environment with respect to the class of defects in question.

In this model, the developer ecosystem is responsible for ensuring that every version of the system satisfies *safety and security invariants*—that is, properties that the system is expected to ensure at all times, even when operating in an adversarial external environment. In many cases, safety invariants can be expressed through language, API and application framework design, or through domain-specific code and configuration conformance checks.

At Google this approach was successfully applied to several classes of previously intractable defects, including XSS (cross-site scripting) and SQL injection, which occupy positions 2 and 3 in the CWE Stubborn Weaknesses ranking.[3] Today, many Google user-facing applications are developed in a Safe Coding ecosystem and exhibit close to zero residual rates of relevant defects.

➡ **Secure design for application archetypes**. Second, a substantial number of applications and services can be grouped into a much smaller set of common archetypes. For example, the high-level architectural shape of many user-facing services can be characterized as “a web app with microservices back ends and a SQL database” or “a client-side mobile app that relies on a web services API.”

It turns out that many aspects of an individual application’s safety and security risk model are common to all applications in the archetype: Every web app must worry about XSS vulnerabilities, and every RPC (remote procedure call) back end must authenticate and authorize its callers.

This observation can be leveraged by designing developer ecosystems tuned to the given archetype, and by structuring ecosystem components (such as libraries, application frameworks, and production platforms) to address common aspects of the archetype’s risk model template. Developing in such an environment reduces effort, cognitive load, and opportunities for mistakes and omissions for individual product teams—and mitigates risks across the entire ecosystem.

In short, the key to safety and assurance at scale is to design developer ecosystems that ensure secure design best practices and prevent relevant classes of vulnerabilities across all applications of an archetype. This also increases development velocity, because application developers don't have to think about vulnerabilities while focused on functionality.

Comprehensive awareness of all relevant assumptions is particularly difficult to achieve when large teams maintain software over long periods of time.

SAFE CODING

Many common classes of security vulnerabilities, such as memory corruption, SQL injection, and XSS, arise when a developer makes an incorrect assumption about the possible behaviors of a large and complex software system (especially when faced with adversarial inputs) while adding or modifying code whose correctness and safety depends on those assumptions. Comprehensive awareness of all relevant assumptions is particularly difficult to achieve when large teams maintain software over long periods of time.

Past attempts to mitigate these types of vulnerabilities focused on developer education along with tools and processes to discover and fix defects later in the development cycle. Neither approach proved effective, and these classes of defects continue to occur in "top vulnerability" rankings and feature prominently in the Stubborn Weaknesses in the CWE Top 25.[3]

First, developer education is insufficient to reduce defect rates in this context. Intuition tells us that to avoid introducing a defect, developers need to practice constant vigilance and awareness of subtle secure-coding guidelines. In many cases, this requires reasoning about complex assumptions and preconditions, often in relation to other, conceptually faraway code in a large, complex codebase. When a program contains hundreds or thousands of coding patterns that could harbor a potential defect, it is difficult to get this right every single time. Even experienced developers who thoroughly understand these classes of defects and their technical underpinnings sometimes make a mistake and accidentally introduce a vulnerability.

Second, approaches to after-the-fact discovery of defects, such as static or dynamic analysis (including fuzzing), are inherently incomplete when applied to large systems with large, complex state spaces. In many cases, these techniques are computationally intensive and too slow to apply after a code change is written but before it is committed. When defect discovery happens post-commit, it cannot reduce the rate at which new defects are introduced into the source repository.

When software safety is framed as an emergent property of how it is developed, the potential of implementation security defects should be viewed as a design flaw of the development environment: The potential for defects is a hazard that arises during development, and it's the responsibility of the development environment to mitigate this hazard. (In the context of information systems, a hazard is the potential for a user or other stakeholder to experience harm, or more generally, some adverse outcome. Here, the adverse outcome is the introduction of a vulnerability, which in turn results in a downstream risk of harm to the eventual software user. This specifically does not mean attacks on the developers themselves, such as via malware embedded in a developer tool.)

At Google this approach is called *Safe Coding*, because it's centered on structuring development environments that are safe with respect to the accidental introduction of security defects during application design and development.

In the following sections, this article illustrates Safe Coding principles by showing how they apply to several classes of common software safety and security defects, then briefly discusses the cost effectiveness of this approach.

Memory safety

Some of the most common and impactful classes of security vulnerabilities arise from memory safety defects, including code that accesses memory outside the bounds of valid, allocated objects, as well as temporal memory safety violations such as accessing memory that was already deallocated ("*use-after-free*").

Memory safety issues rank 1, 4, 7, and 12 in the 2023 CWE Top 25.[4] Several organizations have reported that memory safety issues cause a substantial majority of severe vulnerabilities in large C/C++ codebases, including Chrome (bit.ly/482j6Ms), Android (bit.ly/4a6DYnC), Project Zero (bit.ly/416oXxS), and Microsoft (bit.ly/3RpRMCj).

Guidance for developers in memory-unsafe languages such as C and C++ is, essentially, to be careful: For example, the section on memory management of the SEI CERT C

Coding Standard[12] stipulates rules like, "MEM30-C: Do not access freed memory" (bit.ly/3uSMBSk).

While this guidance is technically correct, it's difficult to apply comprehensively and consistently in large, complex codebases. For example, consider a scenario where a software developer is making a change to a large C++ codebase, maintained by a team of dozens of developers. The change intends to fix a memory leak that occurs because some heap-allocated objects aren't deallocated under certain conditions. The developer adds deallocation statements based on the implicit assumption that the objects will no longer be dereferenced. Unfortunately, this assumption turns out to be incorrect, because there is code in another part of the program that runs later and still dereferences pointers to this object.

This example illustrates why the coding rule can be difficult to adhere to in practice: Attempting to fix a memory leak, the developer changes existing code by adding a statement to free memory that they assume is no longer used. After the change, code *elsewhere* in the program—code that the developer didn't modify, and perhaps wasn't even aware of—now violates the *do not access freed memory* rule, resulting in a memory safety bug.

The part of the program that the developer modified, and the separate part of the program that contains a new bug after the change, are implicitly connected through assumptions about the allocation state of the object in question. These kinds of implicit assumptions about the state of a large and complex program are easy to miss for a developer who is familiar with only parts of the whole, which is common for large programs worked on by teams of many developers.

In some cases, it's possible to design and structure a program to make such assumptions more apparent—for example, through the use of pointer types that explicitly encode an ownership and lifetime discipline (such as `unique_ptr` and `shared_ptr` in C++). These kinds of considerations are not always applied comprehensively and consistently, however, and memory safety vulnerabilities are quite common in C++ as well as C.

When the risk of classes of defects is viewed as emerging from the design of the developer ecosystem, it follows that the prevalence of memory safety defects emerges from the design of the programming language and surrounding tooling. Simply put, when millions of lines of code are written in a programming language that places the onus *on developers* to ensure that every dereference of a pointer is valid and in bounds, there are going to be defects.

In contrast, *memory-safe languages* remove this responsibility from developers and ensure memory safety invariants through the design of the language and its runtime. For example, in garbage-collected languages such as Java and Go, developers do not write explicit statements to deallocate memory; instead the language and its runtime take this responsibility and deallocate memory only when an object is no longer referenced. Alternatively, in Rust, object ownership and lifetime are expressed as native concepts in the language's type system. This allows rigorous verification of memory-safety invariants at compile time, avoiding the runtime overhead of garbage collection.

Injection vulnerabilities arise when strings derived from untrusted inputs are passed to an API that interprets the string as code in some domain-specific language such as SQL or HTML.

Code injection vulnerabilities

Injection vulnerabilities arise when strings derived from untrusted inputs are passed to an API—referred to as an *injection sink* in this context—that interprets the string *as code* in some domain-specific language, such as SQL or HTML. In this setting, it's crucial to ensure a *rigorous separation* between the trusted code and the untrusted data: If untrusted, potentially attacker-provided *data* can be unintentionally interpreted and evaluated as *code,* then the attacker can exploit the trust placed on the execution environment and execute actions with its (elevated) privileges.

For example, XSS vulnerabilities arise when untrusted inputs are incorporated into HTML or passed to certain web browser APIs without context-appropriate escaping, sanitization, or validation. This can allow attackers to cause JavaScript code under their control to execute in the context of a user's web application session, which would then allow the attacker to exfiltrate or modify user data. Similarly, SQL injection vulnerabilities can arise when untrusted input strings are incorporated into a SQL database query; in this case an attacker could alter the intended function of the query, causing it to return or modify data that should not be accessible to the attacker. Both types of vulnerabilities are quite common and impactful, and occupy ranks 2 and 3, respectively, of the 2023 CWE Top 25.[4]

In the past, mitigation of these classes of vulnerabilities focused on teaching developers rather complex rules (see, for example, OWASP's XSS Prevention Cheat Sheet at bit.ly/47I9OAt) for treating potentially untrusted data before it is incorporated into HTML markup or SQL queries. This doesn't work well in practice: The rules are complicated, and it's often difficult to keep track of which rules were applied to a given string. For example, when a string that originated in a system's back-end storage layer is incorporated into HTML markup in a web application front end, it can be hard for the front-end developer to tell whether that string was sanitized at the time it was stored. Large web applications can have hundreds of code sites that pass data to JavaScript injection sinks, and incorrect or omitted sanitization or escaping in a single instance can result in a vulnerability that compromises the entire application.

Again, the prevalence of these classes of defects can be viewed as an emergent property of the design of the developer ecosystem rather than a failure of developers to apply the correct one of a set of obscure rules, a thousand times over. In this framing, the root cause for these classes of defects is in the design of APIs that represent potential injection sinks: Typically, these APIs accept statements and expressions in domain-specific languages such as HTML, JavaScript, CSS, or SQL, represented as values of a general-purpose `String` type. In this API design it is the developer's responsibility to ensure that untrusted values incorporated into these strings are sanitized or escaped according to the domain-specific language's rules. This is brittle and prone to mistakes that can result in injection vulnerabilities.

Based on this view of the problem's root cause, these classes of vulnerabilities can be addressed by introducing higher-level abstractions that *take responsibility* for separating trusted code or markup and untrusted data. For example, there are *strict contextually auto-escaping template systems*[6] for HTML to ensure that untrusted inputs are appropriately sanitized or escaped before being interpolated. Similarly, we provide builder APIs to construct SQL statements from trustworthy statement fragments.

To ensure that *all* statements and expressions that are passed to an injection sink API are constructed using these safe abstractions, the design of sink APIs has been changed to require a *domain-specific vocabulary type* rather than plain strings. The vocabulary type's type contract captures the safety precondition of the injection sink API. In turn, values of these types are produced only by corresponding safe abstractions, which ensures that they adhere to their contract.

For example, the SQL query APIs by which internal Google developers interact with the Spanner database[2] expect values of type `TrustedSqlString` and do not accept queries represented as simple `Strings`. This type represents strings that are safe to use as a query without risk of SQL injection vulnerabilities. Values of type `TrustedSqlString` can be created only by using builder APIs and factory functions that ensure this type contract. These APIs allow queries to be constructed from query snippets of known, trustworthy provenance such as trusted configuration files or literal strings that are part of the program itself. Arbitrary, potentially untrusted strings cannot be incorporated into a SQL statement—the `TrustedSqlString` builder API simply has no `append` method that accepts arbitrary `String`-typed values. This results in a typing discipline that enforces the (otherwise ad-hoc) secure-coding guideline to assemble SQL queries from trusted strings and to supply dynamic parameters via query parameter binding (see, for example, OWASP's guidance at bit.ly/3LUPyrk).

Similarly, XSS risk is addressed by defining a set of vocabulary types to represent strings that are safe for web platform injection sinks. For example, values of type `SafeHtml` can be safely returned as `text/html` web server responses, interpolated into an HTML document in an "HTML tag content" context, or assigned to the `.innerHTML` DOM property in browser-side JavaScript. These types and their associated safe builder APIs have been implemented in Google production languages. (For example, open-source implementations for TypeScript and Go are available at https://github.com/google/safevalues and https://github.com/google/safehtml, respectively.)

In addition, we developed a W3C standards proposal, *Trusted Types*, to integrate corresponding types natively into the web platform.[7]

Similar to the type-constrained SQL query API, we augmented server-side application frameworks, HTML templating systems, server-side response APIs, and browser-side application frameworks and templating systems to constrain API parameters to the corresponding type that, by its contract, is safe to use in the given API's context.[15]

Constraining potentially unsafe usage of injection sinks through vocabulary types and safe abstractions achieves a high degree of confidence that *any program accepted by compile- and runtime type checks* is free of injection vulnerabilities—if it compiles, it's secure! (See references 6 and 14 and reference 1, chapter 12, "Writing Code," for more details on preventing injection vulnerabilities through Safe Coding.)

At Google we found that Safe Coding is the only approach that can substantially reduce the incidence of injection vulnerabilities, especially at the scale of Google's codebase. For example, 10 years ago, we tended to encounter tens of XSS vulnerabilities per year for each large, complex web application like Gmail. Since then, Safe Coding practices and safe-types discipline, including browser-side *Trusted Types* enforcement, have been incorporated into internal web application frameworks that are widely used for new and existing web applications (bit.ly/3uLHFip).

Today, the residual incidence of XSS vulnerabilities across *all* frameworks-based applications is in the low single digits (some residual XSS risk arises from, for example, pre-existing application components that have not yet been refactored to conform to the safe-types discipline and are exempted from enforcement on a "legacy" basis). Some large services, such as the Google Photos web front end, have not had *any* XSS vulnerabilities reported over their entire lifetimes. Similarly, SQL injection is essentially a nonissue in the Google internal codebase. In contrast, XSS and SQL injection occupy spots 2 and 3 in the CWE project's 2023 Stubborn Weaknesses ranking.[3]

Safe Coding prevents injection vulnerabilities at modest initial and ongoing costs:[14]

➡ We rely primarily on judicious API design that takes advantage of language-native type systems, augmented with inexpensive code conformance checks where necessary—for example, the `CompileTimeConstant` check implemented as part of the Error Prone framework (bit.ly/3RsSVIf). This results in minimal additional resource demands at application runtime and on build systems and CI/CD (continuous integration/continuous delivery) infrastructure.

➡ Beyond initial efforts to develop safe APIs and frameworks, ongoing maintenance and support costs are modest. For example, at Google a small team of security engineers maintains Safe Coding libraries, framework components, and code conformance checks for secure web application development, and provides user support for a population of many thousands of web application developers at Google.

SAFE DEPLOYMENT

Production deployments of services, and the underlying infrastructure, can quickly get complicated, even in organizations much smaller than Google.

Consider an SRE (site reliability engineer) who is tasked with setting up a new production environment. The production environment includes devices (routers, firewalls, load balancers, database servers, applications servers, and more) made by several vendors, each with its own configuration UIs and config languages. The engineer has a playbook document that outlines the changes to be made, but setting up the environment is essentially a manual process.

This is error-prone—the engineer might accidentally make a change to the wrong device (perhaps caused by a simple typo in a command-line argument) or make a change that has unintended consequences, because of subtle discrepancies in configuration semantics across different vendor devices.

This could result in a misconfiguration with security impact, such as exposing an internal network service to the public Internet, a missing or overbroad access-control list, or an outage in an unrelated service hosted in the same production environment.

When this happens, it's tempting to say "They should have been more careful" but it's ultimately an unreasonable expectation that any human has a perfectly accurate mental model of a production environment consisting of hundreds of devices with thousands of config settings expressed in several different configuration models.

Instead, just as for common coding mishaps, the potential for deployment mishaps should be treated as a hazard, and it should be the deployment environment's responsibility to protect engineers from encountering these hazards.[8]

Safety from deployment hazards can be incorporated into deployment environments in several ways. Examples of practices found useful at Google include cloud platforms, config-as-code, and Zero Touch Prod and Safe Proxies.

Cloud platforms

When deployment environments are based on "bare-metal" servers and network devices, engineers are exposed to the full complexity and nonuniformity of their configuration surfaces. In contrast, cloud platforms provide a higher-level abstraction and a consistent vocabulary of configuration points, and they expose common functionality (such as databases) as managed services. This reduces cognitive load caused by differences between configuration surfaces of different types of network devices and the need to manage lower-level aspects of servers that host higher-level services such as databases.

Cloud platforms can integrate enforcement of security invariants into their control planes. For example, Google's production environment[13] enforces binary authorization policies (https://cloud.google.com/docs/security/binary-authorization-for-borg) to govern whether a deployment package can run with the privileges of a given role. For sensitive roles, these policies typically require that the binary package is accompanied by a provenance attestation (https://slsa.dev/provenance/v1) that it was built by an authorized, trusted build system from code in a trusted source repository where changes are reviewed under the two-person principle. This mechanism ensures, on an ongoing basis, the invariant that only explicitly authorized code can exercise the privileges of a given production role.

Making changes to production systems directly—through configuration UIs or CLIs—is risky.

Config-as-code

Making changes to production systems directly—through configuration UIs (user interfaces) or CLIs (command-line interfaces)—is risky: Changes are actuated immediately, including any mistakes.

A safer approach is to capture the entire configuration in machine-readable config files stored in a versioned repository, and to automatically actuate changes to the production environment based on this configuration. This pattern is often called config-as-code (or sometimes GitOps), because authoritative configuration is maintained in a source repository, just like an application's source code.

Maintaining configuration in this fashion allows the introduction of safeguards against configuration mistakes:

- The configuration repository can be set up to require two-person review. This gives a second engineer the opportunity to catch mistakes.
- Changes to the configuration can be guarded by conformance checks that execute pre-submit and/or before a configuration change is actuated. Similar to conformance checks on source code discussed earlier, such conformance checks can ensure safety and correctness invariants on the configuration on an ongoing basis. For example, a conformance check can ensure that the authorization policies of back-end RPC services adhere to common guidelines and best practices.
- Common types of changes can be automated through tools that generate sections of configuration. The configuration for a new service instance can be generated automatically based on a template, reducing the opportunity for mistakes caused by typos.

Zero Touch Prod and safe proxies

Zero Touch Prod is a set of principles and tools to ensure that every change to a production environment must be made by trusted automation (not directly by a human), prevalidated by trusted software, or made through an audited break-glass mechanism.[5]

Conformance checks imposed on config-as-code are one way of adhering to this principle. It can be challenging, however, to accommodate all actions in a production environment through config-as-code, especially those needed when responding to an incident.

Safe Proxies are trusted systems that are interposed between human engineers and a production environment (see chapter 3 of reference 1). The safe proxy mediates all interactions with the production environment and can, for example:

- Validate the safety of requested actions.
- Impose security policy such as mandatory auditing and mandatory multiparty authorization.
- Rate-limit potentially destructive actions.

In addition to enhancing safety with respect to human error (such as honest mistakes), these techniques also provide an effective control against insider risk and external compromise of privileged operators. When in place, these measures remove engineers' ambient privileges to unilaterally execute powerful and sensitive actions in the production environment. Instead, changes and actions in production are guarded by two-person review, automated validation, and mandatory auditing.

SCALING SECURE DESIGN ACROSS APPLICATION ARCHETYPES

The previous sections discussed how to achieve substantial leverage over implementation bugs and deployment defects, by treating them not as individual defects but rather as an entire class of defects to be addressed through development and deployment ecosystem design (programming languages, application frameworks, build systems, cloud platforms, and so on).

Similar thinking can even be applied to defects that, in isolation, are true design flaws rather than implementation bugs—that is, defects that arise from a fundamental choice about the shape or architecture of a product or service.

A key observation is that many types of potential architectural and design flaws, and the safety and security considerations and practices to avoid them, apply to *all applications of a software-architectural archetype.* Examples of such archetypes might include:

- A system consisting of an end-user-facing web application front end communicating with back-end microservices through RPCs, which in turn rely on a SQL database for persistence.
- A mobile application backed by a service API; the service API front end in turn makes RPCs to back-end microservices.

While many threats and secure-design considerations are indeed specific to a given application (e.g., a banking app is inherently different from a photo editor) typically, a substantial degree of commonality exists in the threat models across the entire class of applications of a given archetype.

For example, the safety and security design of (almost) any application that falls into either of these archetypes must consider areas such as:

- Protecting the confidentiality and integrity of network and RPC traffic over the public Internet and internal networks.
- Ensuring that all external client-server requests and

internal RPC requests are appropriately authenticated and authorized, governed by an explicit, intentional policy (although the details of the policy itself are usually specific to the application and its features).
- Ensuring that the confidentiality and integrity of user/customer data is appropriately protected in conformance with the service provider's policies (for example, through appropriate cryptographic schemes).
- Ensuring that user data is deleted according to the service provider's policies (such as when requested by a user or when a user leaves the service).
- And many more.

Threat modeling and secure design require expertise that is often not available within product development teams.

Google has hundreds of web and mobile applications and external-facing API endpoints, and thousands of microservice back ends and internal RPC endpoints, but even organizations much smaller than Google usually have at least several. It's undesirable for each team responsible for one of these services to develop a comprehensive threat model from scratch and to design appropriate mitigations for each one. Taking such a decentralized approach results in not only duplication of work, but also inferior outcomes: Threat modeling and secure design require expertise that is often not available in product development teams; while an organization's security experts can help through consulting, their bandwidth is typically limited.

At Google we take advantage of common threat model aspects and secure design considerations by building frameworks and platforms that inform, govern, and constrain important aspects of the architecture and design (both in terms of code and mapping to production resources) of applications and services built on the framework.[9] These applications inherit security (also privacy and reliability) best practices designed and built into the framework, usually through collaboration between domain experts (security engineers, SREs) and framework engineers.

Many design choices (Which secure transport protocol should I use? How should I authenticate and authorize requests? How do I encrypt data at rest?) are incorporated in the design of the framework, and application developers are relieved from making these decisions—and from potentially making a suboptimal choice.

This approach reduces the risk of design-level security defects and gives leverage to scarce expert bandwidth—experts can focus their attention on the design and implementation of frameworks and platforms, while having an impact on a large number of development projects that rely on that framework.

Furthermore, after frameworks are widely adopted as a platform for application development, *future* security improvements and mitigations for novel attacks and defect classes can often be deployed swiftly, scalably, and efficiently, taking advantage of the well-defined structure of frameworks-based applications. For example, security and web frameworks teams at Google routinely roll out new security features and mitigations at scale to existing frameworks-based applications, often without any need for involvement or time investment by the teams that maintain individual applications. (See the blog post *A Recipe for Scaling Security,* at bit.ly/3u6C71R, for more details.)

CONTINUOUS ASSURANCE AT SCALE

At Google we sometimes say, “Software engineering is programming integrated over time,” to recognize the vast difference between one or a few people writing a few-thousand-line program in days or weeks, versus hundreds of teams of hundreds of developers jointly working, over years or even decades, on a codebase of several hundred million lines of code.[16]

This distinction matters when it comes to ensuring security invariants for software products and the degree of confidence in the product adhering to these invariants. For a small, self-contained program written by a small group over a short period of time, developers are less likely to make mistaken assumptions that lead to defects, and it is indeed feasible for an expert to read and understand the entire codebase and perform a high-confidence security assessment.

Once a service’s design, codebase, and production footprint get larger and more complex, this no longer works: The risk of defects caused by mistaken assumptions (or plain mistakes and forgetfulness) increases. It becomes infeasible for an expert, or even a group of experts, to fully and deeply understand the entire artifact, resulting in limits on high-confidence security assessments.

If the experts need to read and understand most of a codebase of many hundreds of thousands of lines of code, it’s likely they will miss something, or make a mistake in their assessment. (Tool support such as static analyzers can sometimes help; however, these typically need to accept some degree of imprecision to scale to large codebases, and hence can also “miss things.”)

Furthermore, security assessments by human experts apply to the specific version under review and are difficult to scale to every release of software that is under active ongoing feature development.

As explained in this article, Google addresses this challenge by designing a developer ecosystem to ensure that all services developed and deployed in this environment have the desired properties. We achieve high levels of assurance by applying the principle of "Design for Understandability" (chapter 6 in reference 1): Key developer ecosystem components are designed to ensure the property for *any arbitrary* application, assuming only that application code is well-typed, passes conformance checks, and satisfies basic assumptions. (Code written and reviewed under the two-person principle is generally assumed not to deliberately subvert security invariants—for example, through use of reflection or unsafe casts.)

This allows us to have confidence that the property holds for *all applications*, based solely on understanding key developer ecosystem components, and without having to consider or understand application-specific code. There is still a residual risk of defects, but it is confined within those key components. These tend to be stable, and domain experts can thoroughly scrutinize them for potential defects.

In light of the framing as *programming over time*, designing developer ecosystems as *Safe Coding* and *Safe Deployment* environments allows us to achieve continuous assurance at scale: It provides confidence that every production release of every application of supported archetypes satisfies desired safety and security invariants.

References

1. Adkins, H., Beyer, B., Blankinship, P., Lewandowski, P., Oprea, A., Stubblefield, A. 2020. *Building Secure and Reliable Systems: Best Practices for Designing, Implementing, and Maintaining Systems.* O'Reilly Media; https://sre.google/books/building-secure-reliable-systems/.
2. Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems* 31(3), 1–22; https://dl.acm.org/doi/10.1145/2491245.
3. CWE. 2023. Stubborn weaknesses in the CWE top 25; https://cwe.mitre.org/top25/archive/2023/2023_stubborn_weaknesses.html.
4. CWE. 2023. Top 25 most dangerous software weaknesses; https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html.
5. Czapiński, M., Wolafka, R. 2019. Zero Touch Prod: Towards safer and more secure production environments. Usenix; https://www.usenix.org/conference/srecon19emea/presentation/czapinski.
6. Kern, C. 2014. Securing the tangled web. *Communications of the ACM* 57(9), 38–47; https://dl.acm.org/doi/10.1145/2643134.
7. Kotowicz, K. 2024. Trusted Types; https://w3c.github.io/trusted-types/dist/spec/.
8. Leveson, N. 2019. A systems approach to safety and cybersecurity. Usenix; https://www.usenix.org/conference/srecon19emea/presentation/leveson.

9. Nokleberg, C., Hawkes, B. 2021. Application frameworks. *Communications of the ACM* 64(7), 42–49; https://dl.acm.org/doi/10.1145/3446796.
10. OWASP. OWASP Top Ten; https://owasp.org/www-project-top-ten/.
11. Saltzer, J. H., Schroeder, M. D. 1975. The protection of information in computer systems. *Proceedings of the IEEE* 63(9), 1278–1308; https://ieeexplore.ieee.org/document/1451869.
12. Seacord, R. C. 2014. *The CERT C Coding Standard: 98 Rules for Developing Safe, Reliable, and Secure Systems*, second edition. Addison-Wesley Professional.
13. Verma, A., Pedrosa, L., Korupolu, M. R. Oppenheimer, D., Tune, E., Wilkes, J. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*; https://dl.acm.org/doi/10.1145/2741948.2741964.
14. Wang, P., Bangert, J., Kern, C. 2021. If it's not secure, it should not compile: preventing DOM-based XSS in large-scale web development with API hardening. In *Proceedings of IEEE/ACM 43rd International Conference on Software Engineering*, 1360–1372. https://dl.acm.org/doi/abs/10.1109/ICSE43902.2021.00123.
15. Wang, P., Gumundsson, B. A., Kotowicz, K. 2021. Adopting Trusted Types in production web frameworks to prevent DOM-based cross-site scripting: a case study. In *IEEE European Symposium on Security and Privacy Workshops*, 60–73; https://research.google/pubs/pub50513/.
16. Winters, T., Manshreck, T., Wright., H. 2020. *Software Engineering at Google: Lessons Learned from*

Programming over Time. O'Reilly Media; https://www.oreilly.com/library/view/software-engineering-at/9781492082781/.

17. Young, W., Leveson, N. G. 2014. An integrated approach to safety and security based on systems theory. *Communications of the ACM* 57(2), 31–35; https://dl.acm.org/doi/10.1145/2556938.

Christoph Kern *is a principal software engineer in Google's Information Security Engineering organization. His primary focus is on developing scalable, principled approaches to software security.*

CONTENTS