Google

Google Security Engineering Whitepaper
March 4, 2024

Secure by Design at Google

Christoph Kern
xtof@google.com

Among software security practitioners and experts, it's self-evident that security must be considered an integral part of the design of a software product, and attempts to "bolt on" security after the fact are usually unsuccessful. An abundance of material, including many books on the topic of "secure software design," is available for software teams interested in learning how to build security into their product design.

Yet, a steady stream of vulnerability notifications shows that software products continue to release with security defects, indicating that there is a significant gap between theory and practice. In recent years, more attention focused on this gap, and new information security regulations and guidance by governmental bodies increasingly highlight the importance of secure-by-design as a concept.[1]

Applying this guidance in practice is often difficult: Especially for practitioners who are not software security experts, it can be challenging to bridge the gap between high level guidance like "apply least privilege principles" and how to incorporate this principle in a concrete software product.

This paper provides an overview of how we incorporate safety and security[2] during software design, implementation, and deployment at Google, and shares some key insights that emerged from our experience applying secure software design at Google scale over the years. Perhaps our most significant observation is that the security posture of software products substantially emerges from the underlying developer ecosystem, including software libraries, application frameworks, the developer tooling, production platforms, and so on. It is essential to design developer ecosystems such that they take responsibility for ensuring key security properties of the resulting software, rather than leaving this responsibility to individual software engineers and development teams.

[1] For example, the Secure by Design initiative at CISA (US Cyber Defense Agency), or the UK NCSC's Secure Design Principles.

[2] In this paper, *safety* primarily refers to the prevention of mishaps and accidents: A safe system mitigates relevant hazards, that is, the potential for harm and adverse outcomes for its users and stakeholders. *Security* is about protecting users and stakeholders in an adversarial context, where the system might be under active attack as opposed to mere random chance of mishaps. For brevity, we will often use "security" instead of "safety and security".

Security as a design priority

Many safety and security hazards can only be mitigated when product developers consider them during design of a product—they must incorporate mitigations into its shape and basic structure.

As a simple example from the realm of physical hazards, consider an electrical extension cord: Functionally, extension cords would work just fine if they had plugs on both ends, just like loudspeaker wires that often come with a banana plug on each end. However, for a cable carrying line voltage, such a design results in a dangerous shock hazard: If a user plugs such a cable into a wall outlet before connecting the other end to an appliance, there'd be exposed metal prongs carrying live voltage. After a cable is designed and manufactured in this fundamentally unsafe shape, there's no meaningful way to mitigate the shock hazard. One might attempt a post-manufacturing "patch", like a cover for the plugs or attaching a tag with a warning label ("always connect to appliance first"); however, this is unlikely to be effective.

Making extension cords safe with respect to shock hazards requires incorporating safety into the design from the start—plug on one end, and socket on the other.

Safety and security by design is a foundational expectation for physical systems—we expect our cars to be designed and engineered with crumple zones, anti-lock brakes, anti-theft systems, and so on.

Software security challenges

At Google, we believe that incorporating security considerations into the design of software products and services is equally essential.

However, when applying secure-by-design principles to software design and development, we are faced with some unique challenges: In contrast to mass-produced physical systems, it's common to modify and improve software products with new features on an ongoing basis, especially for cloud-hosted software services.

For a mass-produced physical product, after the design and

manufacturing process are finalized and tuned, the risk of introducing *new* manufacturing defects over time is quite low (unless there is some change, such as a switch to a new parts or materials supplier, or a new design revision of the product).

In contrast, when software is evolving on a continuing basis, there are ongoing changes to the product design, and ongoing updates in the form of rewriting and modifying code. This results in an ongoing risk of introducing new design *and* implementation defects. In this paper, we discuss key practices and techniques that we developed to manage this risk.

Secure default configurations

For many physical products, customer expectations (and in some cases formal standards and regulations) call for the product to mitigate common hazards. Furthermore, fundamental safety mechanisms should be included in every variant of the product sold, and if the user can turn off the safety mechanism, it should be on by default. For example, the shapes of electrical connectors and cables, insulation materials and so on are governed by electrical codes and standards. All trim variants of a car model will have crumple zones built in, and anti-lock brakes are automatically enabled every time the car starts. Some features, like traction control systems, can be turned off, but automatically re-enable when the car restarts.

At Google, we think that software systems should be offered with a similar mindset, with basic safety and security features included in every version of the product, and enabled by default.

Which default safety and security elements should be provided by a product depends on risk-cost tradeoffs (How much does it cost the vendor to include the feature? What risks are users exposed to if the feature is not included?). These tradeoffs tend to change over time as technologies and customer expectations evolve. For example, a few decades ago, a backup camera was an innovative and fairly expensive premium safety option for a car, and safety features like lane assist and collision avoidance did not even exist. Technology evolved to the point where the necessary components (digital cameras, sensors, control systems, screens, etc) are readily available and inexpensive, and in the US, backup cameras are now mandatory.

Similar considerations should apply to software security features. For example, the Google Chrome browser and productivity services like GMail and Google Docs include many security features by default, such as those to protect users from malware and account theft. At the same time, the Google Cloud Platform provides premium offerings for innovative, advanced security features aimed at enterprise use cases.

Software security at scale

As noted above, software security engineers face the challenge that unlike physical products, software tends to be continuously modified, leading to an elevated risk of repeatedly introducing implementation defects in the form of bugs and security vulnerabilities in software code. This risk is indeed borne out in practice: Most of the items in the SANS TOP 25 Most Dangerous Software Errors and the OWASP Top Ten relate to implementation defects. These classes of bugs (like *Use after free*, *Cross-site Scripting* or *SQL injection*) tend to be well understood and straightforward when viewed in isolation. Yet it turns out to be very difficult to prevent accidentally introducing these common types of bugs when large teams of developers concurrently work on a large, complex codebase.

A similar consideration applies to the deployment of software services: Production environments constantly change; new versions of software are regularly deployed, and data centers and clusters are regularly reconfigured or brought on- and offline, such as for maintenance. Every change to a production environment brings with it the risk of configuration errors resulting in a security vulnerability or an outage.

Over the past decades, security and reliability engineers at Google arrived at the key insight that the risk of such implementation and configuration defects being introduced during development and deployment is an ***emergent property*** of the design of development and deployment environments.

Every stage of the software development lifecycle, from design and implementation, to testing, continuous integration, and deployment has the potential to reduce (or, undesirably, elevate) the risk of defects in the resulting software product. Thus, to achieve software security at scale, the design of the developer ecosystem itself must prioritize both safety against accidental introduction of defects and the emergent security posture of the software produced. We cover this topic in more detail in Safe developer ecosystems.

Principles of Secure Design

How to design and implement information systems so that they are safe and secure is a complex topic. On one hand, this field is well established and well understood, at least in theory—for example, Saltzer and Schroeder's seminal overview of principles of secure design was published almost 50 years ago [6]. At the same time, the proliferation of security weaknesses observed in many present-day systems suggests that these principles are difficult to comprehensively and effectively apply in practice.

With this in mind, in this paper we will not provide another presentation of technical principles of secure design, such as "least privilege"; we refer readers to existing material such as Saltzer & Schroeder [6]; Anderson [2]; as well Part II (Designing Systems) of our Secure and Reliable Systems book [1].

Instead, we will touch on a number of higher level principles—informed by Google's experience delivering secure and reliable applications to billions of users—around the

process of applying secure-by-design thinking to software design, development, and deployment.

User-centric design

Perhaps the most important observation related to security engineering is that most incidents resulting in some adverse outcome ultimately start with an action or choice made by a human. In many cases, this happens because the person taking the action simply does not have the training or expertise necessary to realize that their choice could result in an adverse outcome to themselves or another user. In other cases, they do have the relevant training or expertise but make a mistake in applying their knowledge—which is bound to happen occasionally.

Thus, as designers and implementers of information systems, we need to consider our products in the context of their use, and how user actions and choices could lead to adverse outcomes. It is our responsibility to ***design and implement products and services to keep our users safe and secure***, even when they make erroneous choices when using them.[3]

Developers are users, too

Many incidents start with a software development or site reliability engineering (SRE) error when developing and deploying systems: A software engineer might fail to consider a security threat during the design of a system or might introduce a coding error during development that results in a security vulnerability; similarly, an SRE might make a configuration change to a deployed system that exposes it to attack by external adversaries.

We view these kinds of design, implementation and deployment errors through the lens of the systems, tools, and processes that developers and SREs use to accomplish their daily work (collectively, the *developer ecosystem*). Often, the potential for a mistake leading to an incident is ultimately due to the design or implementation of these tools and processes [5].

As noted above, we find the security posture of a software product or service (and conversely, the potential for defects) is an emergent property of the developer ecosystem in which it was produced.

Consequently, we view it as the responsibility of the designers and implementers of developer ecosystem tools and processes to mitigate the potential for user mistakes that could lead to security defects and incidents.[4]

Thinking in terms of invariants

We find it helpful to ground security design in the statement of ***invariants***, that is, properties of a system that we expect to always hold, no matter what.[5] It's the system's responsibility (and by implication, its designers and developers), to *ensure* that the property holds, even if a user takes a suboptimal action, and even when the system is under active attack.

In our electrical cord example, a desired safety invariant might be stated as "no matter in which order the cable is connected to appliances and wall outlets, there is never an exposed metal part carrying live voltage."

The following are some examples of security invariants that might apply to a software product:

- *(I1)* All network traffic traversing untrusted networks is protected using an approved, secure protocol such as TLS or ALTS.
- *(I2)* Every request to all methods of a web services API is mediated by a well-defined authentication and authorization policy.
- *(I3)* For all call sites of a SQL query API within an application's code base, for all possibly reachable program states, for all possible external API requests/inputs, the SQL query string is solely composed of trustworthy query snippets, and all untrustworthy parameters are supplied to the query as values bound to query parameters.
- *(I4)* For all code locations that dereference a pointer or reference, for all possibly reachable program states, for all possible external requests/inputs, the pointer/reference is valid at the time of dereference (points to a valid, allocated object of the correct type).

Failure of the system to ensure each of these invariants could result in a security (or reliability) incident: Violation of *(I1)* could allow an external network-level attacker to read or change sensitive data; violation of *(I2)* could result in an authentication or authorization bypass vulnerability; violation of *(I3)* could result in a SQL injection vulnerability, and violation of *(I4)* could result in a use after free memory safety vulnerability.

Stating invariants in this fashion focuses our attention on the need to design development and deployment ecosystems to actively ensure that they hold, and clarifies when common practice is insufficient. For example, common practice to mitigate the risk of SQL injection vulnerabilities consists of giving developers guidance like "use prepared statements," and having a security code reviewer or pentester look for instances of the vulnerability. The statement of invariant *(I3)* in terms of "for all [...]" makes it clear that this is insufficient, especially in

[3] In practice, it's often necessary to scope protection to broadly reasonable usage of the product. For example, an electrical cord's insulation should be heat-resistant, but is not expected to withstand excessive force and cuts by sharp objects.

[4] While not the focus of this paper, the principles discussed here can be equally valuable when considering reliability incidents such as outages.

[5] We use the term *invariant* somewhat loosely. Some of the properties discussed here concern sequences of states; in a more formal treatment a property that holds for all possible sequences of states of a system would be referred to as a *safety property*.

large development projects: Developers sometimes neglect to follow guidance, and security reviewers/pentesters miss bugs in large, complex development projects. If we want to be confident that invariant *(I3)* holds, we need to think about ways to automatically ensure it for every call site of the query API.

Design for understandability and assurance

The benefit of stating desired security properties as "must be ensured by the system, no matter what" invariants especially comes to light when we think about assurance: By itself, just stating the invariant does not make the system secure. What we really care about is the ability to convince ourselves that the system does indeed ensure the invariant (even in an adversarial context), and that this will remain the case as the system evolves over time as the software development team adds new features to the code and deploys new versions.

Viewing invariants in this way focuses our attention on the robustness and understandability of the processes and mechanisms intended to uphold the invariant. The design and implementation of this mechanism can very substantially impact how brittle or robust enforcement of the invariant turns out to be, and how strongly we can convince ourselves that the invariant in fact holds.

For example, one might attempt to uphold invariant *I2* through developer documentation and code review ("in each web service API endpoint implementation, make sure you add checks that the request is authenticated and authorized"). This is sufficient in theory, but is brittle in practice—all it takes to break the invariant is a single mistake, such as a forgotten authorization check during the implementation of one of many API endpoints. Furthermore, this design has poor understandability, making it difficult to achieve robust assurance: To verify whether the invariant holds, one must read and understand all implementations of all service methods and all ad-hoc authentication and authorization checks they implement; and this review effort must be repeated for every release of the service.

A more robust design places responsibility for authentication and authorization onto an application framework in which the web service is implemented. The framework can enforce authentication and authorization based on a declaratively specified policy, and ensure that policy is applied to all incoming requests before they are handed off to application-specific logic. This design is much more understandable and provides stronger assurance: To verify that all requests are subject to policy, one only has to inspect the code of the application framework, not every API request handler in the application itself. This work is amortized across all applications built on top of the framework, and ongoing review effort is limited to changes to the framework, which tend to be much smaller in volume than changes to applications themselves. Furthermore, the policy itself is specified in a common, declarative domain-specific language, which is much easier to understand and review than ad-hoc checks dispersed across a large code base.

Similar considerations apply to invariants *I3* and *I4*, which we'll discuss in more detail in section Safe developer ecosystems. For a more detailed discussion of this topic, see chapter 6, Design for Understandability in the Secure and Reliable Systems book [1].

Secure by design: A user-centric view

Software and systems are ultimately designed, built, operated, and used by humans. Mishaps and security incidents generally arise from a user mistake somewhere along the way. This applies to both end users of software products and services, and users of tools, systems, and processes used to develop and deploy those products and services.

Here are some examples of systems, their users, and potential mishaps that can lead to security incidents:

- **Non-technical end user of software products**

  This user, with no particular technology background, receives an email message with an embedded link; they view the message using the GMail app on their phone, or using the GMail web application in their browser.

  There are a myriad of things that could go wrong in this scenario: In one variant, the email message might be legitimate (such as a notification from their bank), but the user might be on a public Wi-Fi network with a compromised router; the (now malicious) router is attempting to view or tamper with the network traffic. In another scenario, the email message might come from a malicious source[6], and the link might point to a phishing site, or a site that attempts to compromise the browser, or trick the user into downloading an executable file that will install malware on their computer. And so forth.

  We can't expect end users in the general public to know under what circumstances clicking a link could pose security risks, or what to do to evaluate whether a given link is "safe". Doing so would require in-depth understanding of threat models applicable to the Web at a very technical level (What happens when your browser fetches a URL? What is TLS and what threats does it address/not address? What are HTTP redirects and how do they affect your ability to tell where a URL leads? How is the anchor text of a link in an HTML document related to the URL where the link will go?), and the large space of possible attacks (What happens during a phishing attack? What can go wrong when browsing with an unpatched browser?

[6] GMail's spam and malware filters are very good, but like any heuristic they cannot be 100% accurate, especially when faced with targeted attacks.

What's a zero-day? What happens when the URL fetches an executable? Should I double click that?).

General "common sense" simply does not equip users to make these kinds of security decisions.

- **Software developer using a C++ coding environment**

  A software developer is making a change to a large C++ codebase, maintained by a team of dozens of developers. C++ is not a memory-safe language: Developers have full responsibility for memory allocation and deallocation, and for ensuring that all memory access in the entire program takes place through valid pointers and references and within the bounds of valid memory regions—that is, for writing code so that it upholds invariant *(I4)*.

  A large program typically contains many thousands of code statements that access memory. And it's common for developers to write or modify code whose correctness depends on assumptions about the validity of memory regions that are allocated and deallocated by code written and maintained by a different developer, perhaps in a different team. This makes it very difficult to consistently avoid mistakes that result in accidentally introducing a memory safety vulnerability.[7]

  When that happens it's tempting to blame the developer for the mistake, such as writing code that accesses memory through a pointer that is no longer valid. However, it's unreasonable to expect any human to have a complete and accurate mental model of all memory allocations and deallocations in a large program.

- **SRE using a production environment's deployment and configuration tools**

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

  This is very error-prone—the engineer might accidentally make a change to the wrong device, or make a change with unintended consequences, due to 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.

  It's ultimately an unreasonable expectation that any human could have a perfectly accurate mental model of a production network with hundreds of devices and thousands of configuration settings expressed in several different configuration models.

All these examples have in common that there's an expectation on humans to understand a complex system and make correct decisions based on this understanding, and they can't effectively meet this expectation.

In some examples, it's simply unreasonable to expect that the person has the necessary background, expertise, or time: An end user should not have to know how HTTP and TLS work, what happens when you click a link, the difference between opening a web page and downloading an executable, and so on.

In other cases, the person in question does have the expertise in principle, but the system is ***too complex*** to fully understand for any single person. It's impossible, even for an experienced C++ programmer, to reason with perfect accuracy about heap memory allocations in a large program. And it's impossible for an SRE to keep the entire production deployment topology in their head.

The problem here is not with the user—it's with the system itself. The system's design is making it too easy for its users to accidentally encounter a hazard that results in an adverse outcome.

Secure by design: Software products

Keeping our non-technical end user safe from attacks while using our cloud services (like GMail) and end-user software and devices (like Chrome, Android, and Chrome OS) is a very multifaceted problem, and the details go well beyond the scope of this paper. Here are just a few examples of how our products and services take responsibility for keeping their users safe, even under the assumption that a user has no particular technical background, and is potentially targeted by adversaries on the internet:

- **Defending the user from malicious servers:** The user might click a link that leads to an attacker-controlled server. The user's client-side software (browser, email app, and device operating system) should be resilient against attacks by this server, such as attempts to install malware, extract authentication credentials, or generally make inappropriate state changes to the device.

  This requires careful design and implementation of the browser to, for instance, segregate contexts between different sites the user is interacting with, such as to isolate their GMail tab from the tab in which they opened the malicious site, or defense-in-depth measures to ensure isolation even if a software component might have a vulnerability [3].

- **Defending the user against network-level adversaries:** Even an attacker with full control over the network (ability

[7] Indeed, memory safety defects occupy some of the top ranks of the Stubborn Weaknesses in the CWE Top 25.

to view, modify, delete or replay packets) should not be able to compromise the confidentiality or integrity of user email. This requires the user's browser and email apps to ensure that all network traffic uses secure protocols such as TLS in a secure configuration, to safely handle attempts to spoof TLS certificates, tamper with TLS traffic, and so forth.

- **Defending the user from attacks through downloaded files:** For example, even if a malicious site tricks the user into downloading and then opening a file, the user's device should not be compromised. This requires image and document viewers that are designed and implemented to be robust when presented with arbitrary adversarial inputs; and it requires the system to prevent downloaded files from executing as code in a context that could allow this code to further compromise the device.

  For example, Chrome OS does not permit downloaded files to execute as code in an unconstrained context, and Chrome's image and document viewers rely on the secure design and defense-in-depths approaches mentioned above. Similarly, by default, Android does not allow installation of applications downloaded from arbitrary sources on the internet.

- **Defending the user from phishing attacks:** GMail spam and abuse filters detect and isolate many phishing and spam emails. Furthermore, passwordless authentication (such as Passkeys) and multi-factor authentication (such as 2-step verification for Google Accounts) can mitigate phishing attacks, especially when used with a second factor that is resistant to phishing, such as cryptographic security keys.

- **Helping the user carry out actions as intended:** For example, Gmail auto-completes email addresses as the user enters them. This helps prevent against accidentally sending an email to an unintended recipient because of a typo.

Designing a product's user experience to protect users from security threats often involves a tradeoff between the desire to prevent users from taking actions that put them at risk, and the desire to avoid excessive restrictions for advanced users.

For example, in the past, browsers tended to respond to TLS server certificate spoofing attacks by displaying a warning dialog to users. However, non-technical users were often not equipped to evaluate the implications of dismissing the warning; if they did so in an actual attack their session with a sensitive application (such as their bank, or the GMail web app) could be compromised. To mitigate this risk, the browser could completely block connections when an invalid TLS certificate is presented. However, there are situations where users have a legitimate need to interact with TLS servers without a valid certificate, for example when web developers interact with a development instance of a server. As a compromise, Chrome "hides" the ability to bypass certificate warnings behind somewhat obscure UI elements. This allows technical users to bypass the warning when needed, but makes it less likely that a non-technical user can be tricked by an adversary into doing so.

For certain user segments this risk tradeoff is different than for the general public. For example, compromise of a corporate user (especially someone with elevated access to sensitive resources) could have a much larger impact than for an average user in the general public. To account for this elevated risk, Chrome and Google Workspace allow enterprise administrators to impose stricter controls on user segments. For example, administrators can require 2-step verification (which is optional by default), prevent users from ignoring Safe Browsing warnings, prevent users from changing CA certificate trust settings, and many more.

Safe developer ecosystems

In the previous section, we discussed how software and services can take responsibility for keeping end users safe and secure—both from mishaps (such as inadvertently sending a sensitive email message to the wrong recipient because of a typo) and from attacks by active adversaries.

Similarly, developer ecosystems can take responsibility for creating a safe software development experience for their users—software developers and systems engineers. In this context, the connection to security is indirect: the immediate concern is to prevent human errors that can lead to defects (preventing mishaps—a safety concern); however some defects can result in exploitable vulnerabilities in the resulting product (a security concern).[8]

At a high level, there are three major phases of an end-to-end software development and deployment life cycle (SDLC) where we can attempt to reduce the rate of defects and vulnerabilities:

- *Phase 1*: While the software is designed and code is written, but before code is committed to the source repository.
- *Phase 2*: After code is committed, but before releasing to end users or deploying into production environments.
- *Phase 3*: After code is in production.

It's a widely accepted principle, often referred to as "shift left", that it's preferable to eliminate defects as early in the

[8]The security of development environments is of course very important as well, such as protecting against compromise of developer workstations, source code repositories, build systems, and so on. This is a complex topic in itself and beyond the scope of this paper; approaches include BeyondCorp to protect corporate IT resources and SLSA for software supply chain security.

SDLC as possible. Shifting left is generally desirable for its cost-effectiveness, but is especially important for security defects: When a vulnerability is only discovered in phase 3, after code is deployed, adversaries might discover and exploit it before code owners develop and deploy a patch, turning the vulnerability into a "zero day" and potentially putting users and customers at risk. In addition, deploying patches can be operationally risky and toilsome for users and customers.

In phase 2, there is unfortunately no practical way to find most, let alone all, defects and vulnerabilities that are present in a codebase before it is released. Expert code reviews and automated tools like static analyzers and fuzzers will usually find *some* of those bugs, but they're inherently limited in their ability to find *all* (or even *most*) bugs in a large and complex codebase.

The "shift-left" principle suggests that we should focus on phase 1, and reduce, as close as possible to zero, the rate at which *new defects* are introduced during development: A bug that's never committed into the repository doesn't need to be discovered later on, and can't make it into a production release.

Established practice for secure software design and development expects individual development teams and software developers to be aware of, and adhere to, wide-ranging collections of secure design and secure coding guidelines.[9] This framing places responsibility squarely on developers: They're expected to know all the applicable guidance, and never, ever make a subtle mistake in applying it—which could result in a security defect or vulnerability.

For many types of defects such as memory safety or injection vulnerabilities, their potential is pervasive—during development of a typical software application, developers encounter many hundreds if not thousands of situations where they could potentially make such a mistake and introduce a vulnerability. For other types of defects, their potential is less common, but avoiding them requires deep domain knowledge and expertise (such as in applied cryptography) that is typically not available in development teams. ***With these odds, never making a mistake is an impossible task for any human.***

In our experience at Google, coding guidelines and developer training, combined with partially-effective post-development vulnerability discovery, are insufficient to substantially reduce the residual rate of common types of security defects like memory safety and injection vulnerabilities. This is reflected in industry vulnerability rankings, where these types of weaknesses appear in top ranks year after year.[10]

[9] For example, the Secure Software Development Framework (NIST SP 800-218) stipulates guidance like "Create Source Code by Adhering to Secure Coding Practices (PW.5)"; "PW.5.1: Follow all secure coding practices that are appropriate to the development languages and environment [. . .]"; "Example 1: Validate all inputs, and validate and properly encode all outputs."

[10] See Stubborn Weaknesses in the CWE Top 25, a ranking of software weaknesses that have consistently appeared in the CWE Top 25 Most Dangerous Software Weaknesses over five years.

With this in mind, we find it instrumental to shift our mindset and view the resulting high likelihood of vulnerabilities as an emergent property of a developer ecosystem that presents far too many opportunities for mistakes. That is, the root cause for an elevated likelihood of vulnerabilities is not in humans making an occasional mistake or omission in applying secure coding guidance, but rather in an insufficiently safe developer ecosystem.

In this new mindset, the responsibility for ensuring security invariants, and for preventing defects that invalidate these invariants, shifts from developers to the design of the end-to-end developer ecosystem: We expect the components of the developer ecosystem—programming languages, software libraries, application frameworks, build and deployment tooling, and the production platform and its configuration surfaces—to shield developers from relevant development and deployment hazards.

Putting this principle into practice and designing a developer ecosystem that is both practical, cost-effective and easy to use, and at the same time sufficiently safe, is not at all straightforward. But when implemented and applied successfully, the impact on product security can be very substantial, and at Google we found it to be the only approach that can prevent common types of design and implementation defects at scale. For example, memory safety bugs (including #1, #4, #7 of the CWE Stubborn Weaknesses) are effectively prevented by memory-safe languages such as Java, Go, and Rust. At Google, we have all but eliminated the risk of Cross-Site Scripting (XSS) and SQL injection (CWE Stubborn Weaknesses #2 and #3) in web applications built on our premier internal application frameworks.

We describe in more detail what we learned about how to design and build safe developer ecosystems in [4]. Here we provide a short overview of the key elements:

- **Safe Coding** is an approach to developing software that is secure by design, based on structuring the developer ecosystem to impose security invariants on software developed in the ecosystem. Security invariants are expressed through the design of programming languages, library and component APIs, and application frameworks, or through domain-specific code and configuration conformance checks. For example, compilers and run-times of memory safe languages including Java, Go, and Rust ensure memory safety invariants such as *(I4)*, which implies absence of memory safety violations and vulnerabilities. As another example, at Google we developed a safe coding practice to prevent XSS and SQL injection vulnerabilities. It is based on data types whose type contracts capture the safety preconditions of APIs at risk of injection vulnerabilities, such as database APIs that accept a SQL query in string form. This approach to safe coding leverages the programming language's type system to ensure invariant *(I3)*.

Safe coding provides a high degree of confidence that any program that compiles and runs upholds security invariants and is free of relevant vulnerabilities. It prevents vulnerabilities from ever being introduced into the codebase, because in this model, *if code isn't secure it won't even compile*!

In our experience at Google, maintaining a Safe Coding discipline is typically highly cost-effective and can even provide a net-benefit, because developers need no longer be concerned with the underlying class of defects. Bringing an existing codebase into conformance with Safe Coding invariants requires an initial investment that can range from modest (such as when migration is substantially automatable) to very significant.

- **Safe Deployment** environments actively ensure that all applications and services deployed into the production environment adhere to security best practices. This includes Cloud Platforms that ensure security invariants in their control plane. Additionally, Safe Deployment environments can incorporate Config-as-Code combined with configuration conformance checks that impose security invariants and best practices on all configurations; and Zero Touch Prod, a set of principles and tools to ensure that every change to a production environment must be either made by trusted automation (not directly by a human), prevalidated by trusted software, or made through an audited break-glass mechanism.

- **Well-lit paths for application archetypes**: Many aspects of threat models and respective design and implementation security best practices, for example related to authentication and authorization, protecting data in transit and at rest, or web application security considerations, are common to all applications of an archetype, such as any web application with microservices backends and a SQL database. These common considerations, such as which vetted implementation of which authentication scheme or transport protocol to use, can be incorporated into opinionated application frameworks that assemble vetted components in vetted configurations ensured by domain-specific conformance checks. Such frameworks provide well-lit paths for application developers, and scalably impose secure design practices on all applications of the archetype(s) supported by the framework.

In combination, safe developer ecosystems can provide a high degree of confidence that *every release* of *every application* of supported archetypes satisfies security invariants and thus provide *continuous assurance at scale.*

Conclusion

In this paper, we discussed a number of high-level principles and patterns used at Google to guide the design, development, and deployment of secure products and services.

It's helpful to state safety and security objectives in terms of ***invariants***—properties that a system should ensure will always hold, no matter what, even when under attack.

Security mishaps are often the result of user mistakes or risky actions. Recurring mishaps tend to result from recurring or pervasive potential for user error. It is often unreasonable to expect users to known how to avoid a hazard, and to never make a mistake. Systems and applications should be designed with a ***user-centric*** view and uphold security invariants even when users take risky actions or make mistakes.

The security posture of software products and services is an ***emergent property*** of the developer ecosystem in which they are designed, implemented and deployed. Software developers and deployers are users of the components of the developer ecosystem—software libraries, developer tools, production platforms. In this context, mistakes and omissions by human users can result in the introduction of defects into a product. Careful design of developer ecosystems can drastically lower the incidence of certain kinds of defects and vulnerabilities, and in some cases practically eliminate them.

Developer ecosystems should be designed to ensure that the resulting products and services uphold security invariants. A developer ecosystem provides ***continuous assurance at scale*** when it does so not only at a point in time for one specific version, but for every code change and for every released version across a class of applications.

References

[1] H. Adkins et al. *Building secure and reliable systems: Best practices for designing, implementing, and maintaining systems*. O'Reilly Media, 2020. URL https://sre.google/books/building-secure-reliable-systems/.

[2] R. Anderson. *Security engineering: A guide to building dependable distributed systems*. John Wiley & Sons, 2020.

[3] A. Barth, C. Jackson, C. Reis, et al. The security architecture of the Chromium browser. In *Technical report*. Stanford University, 2008.

[4] C. Kern. Developer ecosystems for software safety. *ACM Queue*, 22(1), Feb 2024. doi: 10.1145/3648601. URL https://research.google/pubs/pub53103/.

[5] N. Leveson. A systems approach to safety and cybersecurity. USENIX SRECon EMEA, 2019. URL https://www.usenix.org/conference/srecon19emea/presentation/leveson.

[6] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975. URL https://doi.org/10.1109/PROC.1975.9939.

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.