High-Confidence, Scalable Secure Development

Submitter: *Eric Brewer, on behalf of Google, LLC*
Topic: *(2) Software development best practices*
Speaker: *Christoph Kern, Principal Software Engineer, xtof@google.com*

Many common classes of security defects result from simple mistakes: Code is written in languages and on top of APIs and frameworks that are inherently prone to developer mistakes, which in turn lead to security vulnerabilities. For example:

- When writing code in a memory-unsafe language such as C or C++, it is very difficult to avoid mistakes resulting in exploitable memory-corruption bugs.
- Application code that relies on SQL query APIs is prone to SQL injection.
- Web applications are prone to cross-site scripting (XSS).

Aside from developer education, which has limits, the typical approach to mitigating these vulnerabilities has been largely reactive: We apply tools and techniques such as fuzzing, static analysis and penetration testing after the fact, in the hope of discovering vulnerabilities before release. Although these tools and techniques are clearly beneficial, fixing defects later in the development cycle is limited in effectiveness, as evidenced by the consistent presence of these vulnerability classes near the top of lists such as SANS Top 25 and OWASP Top 10.

Secure languages and application frameworks[1] can be used to impose a structure on software that enables high-confidence reasoning about its security, at scale.[2]

Without such structure, this reasoning is difficult and error-prone. For example, the CERT C Coding standard requires, "MEM30-C. Do not access freed memory". But ensuring that this requirement is actually fulfilled for real-world C code is challenging, and often requires difficult reasoning about heap memory structure. Similarly, it is difficult to ensure correct validation and escaping for all data that flows into a web application's HTML markup, since data often passes through several components on its way from inputs to outputs, such as through a storage schema.

In contrast, Rust has emerged as a practical alternative to C and C++ as a systems-development language, embodying a secure-by-construction stance on memory safety. Rust's type system imposes an ownership discipline that ensures, for example, that freed memory cannot be accessed.

As another example, in Google we have developed secure abstractions around the Web platform[3,4] and SQL query APIs,[5] which effectively prevent application developers from writing code that is even potentially at risk of XSS and SQL injection vulnerabilities, respectively. Our application framework ensures through its APIs' type signatures that no improperly escaped data can reach an injection sink. Similar to Rust, clearly labeled security-critical code blocks are exempt from safety checks (analogous to Rust's "unsafe" blocks). In our code base, such sections are used very infrequently, and are subject to mandatory (workflow-enforced) domain-expert review.

[1] Nokleberg, C., & Hawkes, B. (2020). Best Practice: Application Frameworks. ACM Queue, 18(6).
[2] Adkins, H., et al. (2020). Building Secure and Reliable Systems. O'Reilly. Chapter 6, "Design for Understandability".
[3] Kern, C. (2014). Securing the tangled web. Communications of the ACM, 57(9), 38-47.
[4] Kotowicz, K., & West, M. (2021). Trusted Types. W3C Editor's Draft. https://w3c.github.io/webappsec-trusted-types/dist/spec/.
[5] Adkins, H., et al. (2020). Building Secure and Reliable Systems. O'Reilly. Chapter 12, "Writing Code".

We have found this approach to be practical at the scale of our very large code base; effective at reducing the residual incidence of vulnerabilities to near-zero; and highly cost-effective[6]. Recurring staffing costs amount to approximately a dozen full-time security engineers, who maintain core libraries and framework components and who consult on and review security-critical code segments. This effectively prevents introduction of XSS during ongoing development of the entire JavaScript and TypeScript codebase in our main source repository;[7] a team of only a dozen can oversee the work of more than ten thousand developers.

In both examples, the structure imposed by the language/framework ensures that application developers can't accidentally introduce common types of vulnerabilities. Thus, simply knowing that code compiles provides high confidence that application code is not affected by these classes of bugs. As such, this approach is in our experience key to scalable software security.

Verified Core Components. Relying on secure-by-design languages and frameworks can scalably ensure that otherwise common types of vulnerabilities are absent from a codebase. However, this property relies on correctness properties of the underlying language or framework, as well as code inside security-critical code segments. Since correctness at this level is so crucial to the security of everything built on top, it is especially important to focus validation in these areas. Because the cost of validation is amortized across all applications built on these foundations, even major validation efforts can be cost-effective.

Particularly strong assurance can be achieved through verification via machine-checked reasoning in a formal logic. Though traditionally difficult and expensive, this process is quickly becoming cheaper, easier, and more scalable, and is particularly cost-effective when applied specifically to security-critical trusted components such as CPUs, SoCs, operating systems,[8] type systems[9] and crypto primitives.[10]

Beyond formal verification of core components, we also expect benefits from improving the usability of rigorous verification methods to make them more widely accessible to developers, for example by bridging the gap between formal verification and commonly used patterns in software testing.[11]

In summary, we recommend that secure development practices incorporate criteria around platforms, languages and frameworks that are designed to ensure security properties of all applications built thereon. This permits validation and verification efforts to focus on these core components, and at the same time provides scalable software security for the resulting applications.

[6] Wang, P., Bangert, J., & Kern, C. (2021, May). If It's Not Secure, It Should Not Compile: Preventing DOM-Based XSS in Large-Scale Web Development with API Hardening. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (pp. 1360-1372). IEEE.

[7] Potvin, R., & Levenberg, J. (2016). Why Google stores billions of lines of code in a single repository. Communications of the ACM, 59(7), 78-87.

[8] Klein, G. et al. (2014). Comprehensive formal verification of an OS microkernel. ACM Transactions on Computer Systems (TOCS), 32(1), 1-70.

[9] Jung, R., et al. (2017). RustBelt: Securing the foundations of the Rust programming language. Proceedings of the ACM on Programming Languages, 2(POPL), 1-34.

[10] Erbsen, A., et al. (2019, May). Simple high-level code for cryptographic arithmetic-with proofs, without compromises. In 2019 IEEE Symposium on Security and Privacy (SP) (pp. 1202-1219). IEEE.

[11] Reid, A. et al. (2020). Towards making formal methods normal: meeting developers where they are. HATRA 2020: Human Aspects of Types and Reasoning Assistants.