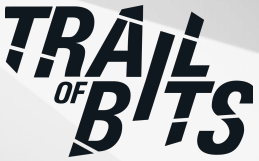

Response to RFI Document ID ONCD-2023-0002-0001

Open-Source Software Security: Areas of Long-Term Focus and Prioritization

November 8, 2023

Prepared by:

The Trail of Bits Assurance, Engineering, and Research groups

opensource@trailofbits.com

Prepared for:

The Office of the National Cyber Director (ONCD), the Cybersecurity Infrastructure Security Agency (CISA), the National Science Foundation (NSF), the Defense Advanced Research Projects Agency (DARPA), and the Office of Management and Budget (OMB)

Introduction

Trail of Bits appreciates the opportunity to respond to the ONCD, CISA, NSF, DARPA, and OMB in their Request for Information (RFI) on areas of long-term focus and prioritization on open source software (OSS) security.

Our response details how we believe these organizations can best support the advancement of OSS security. In particular, we believe the area of focus pertaining to securing OSS foundations should be the highest immediate priority for action. We propose potential policy solutions for each of the four sub-areas of "Securing OSS Foundations" outlined in the RFI. While we consider all of these sub-areas to be important areas of focus for the long term, we believe that fostering the adoption of memory-safe languages and strengthening the software supply chain are the most time-sensitive and should be targeted first.

Our response is followed by an appendix, which contains a short company profile explaining our experience and expertise in OSS and OSS security.

Securing OSS Foundations

The RFI includes five potential areas of long-term focus and prioritization. We find the first potential focus area, securing OSS foundations, and each of its four sub-areas to be the most important. In particular, the fostering of adoption of memory safe programming languages and strengthening the software supply chain are the most time-sensitive sub-areas that should be focused on first. We draw this conclusion based on the number and individual severity of vulnerabilities[1,2] actively being exploited in both of these areas. We present our thoughts and recommendations on each of the four sub-areas in descending order of time sensitivity, starting order memory-safe programming languages.

Fostering the Adoption of Memory-Safe Programming Languages

The public[3] and private sectors have recognized that memory corruption vulnerabilities are a significant, pervasive threat to software security. These vulnerabilities are commonly exploited in high profile attacks, such as the recent FORCEDENTRY iMessage zero-click

[1] https://www.chromium.org/Home/chromium-security/memory-safety/
[2] https://www.crowdstrike.com/cybersecurity-101/cyberattacks/supply-chain-attacks/
[3] https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products

vulnerability[4]. These attacks have made clear the need for solutions beyond runtime mitigations; memory-safe programming languages are the most promising solution currently available.

Memory safe programming languages that are suitable for systems use, such as Rust, are particularly effective due to their relative ease of use, high performance, and interoperability with legacy codebases and C/C++ ABIs. In addition to memory safety, Rust also provides a strong type system that can prevent common logic and concurrency mistakes through its compile-time checks. Rust is not a silver bullet: like all general-purpose programming languages, it cannot prevent all logic errors. Similarly, Rust cannot prevent circumvention of its own safety guarantees within "unsafe" constructs, which are frequently needed to interact with external or native software components. At the same time, Rust marks a significant advancement in memory and type safety for systems software.

Unfortunately, the majority of OSS that is still used, maintained, and extended for critical web and operating system infrastructure predates widespread adoption of memory safe programming languages. Over 65% of websites are served by an open source server written in an unsafe programming language like C or C++[5]. The cost (both monetary and logistically) of migrating all or any significant portion of OSS to memory safe programming languages is substantial. A cost-benefit analysis is therefore necessary.

When performing this analysis, it is important to observe that pre-existing testing should be considered a security property and as important as any core feature: rewriting a piece of software in Rust can actually introduce security problems if, for example, the previous implementation's tests are not included in the rewrite. Thus, these rewrites need to be performed carefully and holistically, with consideration for existing tests and undocumented invariants.

To most effectively identify open source candidates for rewriting in memory safe languages, we recommend targeting libraries that are widely used, relatively small, poorly tested, and susceptible to memory corruption vulnerabilities. A great example of a set of libraries that match these criteria are widely used image format libraries such as libjpeg, libpng, and libwebp, as these libraries are commonly susceptible to memory corruption issues. In fact, during the drafting of this response, a memory corruption vulnerability in

[4] https://citizenlab.ca/2021/09/forcedentry-nso-group-imessage-zero-click-exploit-captured-in-the-wild/
[5] https://w3techs.com/technologies/overview/web_server

libwebp was actively being exploited[6]. Other strong candidates include parsers for common serialization formats, heavily-used browser components (such as JavaScript runtimes and font/layout engines), and operating system kernels.

There are new technologies that show promise in automatically converting code in unsafe languages like C into languages like Rust[7]. However, the resultant Rust code often lacks idiomatic fluency and human intelligibility, and rarely leverages Rust's memory safety features. While these tools represent a positive stride towards a future in which legacy code could be hardened automatically, substantial research investment is necessary to enhance their sophistication and ensure memory safety.

Our Proposed Solution: Perform a systematic analysis to identify the best OSS candidates for rewriting in a memory safe programming language, following our criteria described above. Once these candidates have been identified, fund each's development in an alternative memory safe programming language (preferably Rust) through a Request for Proposal (RFP) process. Long term, fund the development of tools and techniques for automatically translating code from unsafe languages like C to idiomatic Rust.

Strengthening Software Supply Chains

The complexity and opacity of modern software supply chains is a growing risk to software security, as demonstrated by high-profile attacks like the 2020 SolarWinds hack[8]. Like memory safety, we believe that securing the supply chain is critical and requires prompt action.

Supply chain security is a multifaceted and multi-actor problem: each link, from individual software developers, to source hosts, to package indices, to installing clients and downstream users, must be protected. Each link comes with its own unique technical and logistical challenges, as well as potential solutions; we consider a few below.

For the link between individual developers and repository hosts (like GitHub), account integrity is paramount: an attacker who is able to compromise a developer's account on the repository host can surreptitiously introduce source code changes that are automatically re-deployed or consumed by downstream users. Mitigations such as mandatory

[6] https://nvd.nist.gov/vuln/detail/CVE-2023-4863
[7] https://github.com/immunant/c2rust
[8] https://www.cisa.gov/news-events/directives/ed-21-01-mitigate-solarwinds-orion-code-compromise

multi-factor authentication (MFA), minimally-scoped credentials, and secret scanning[9] are already seeing adoption on major hosts; we believe that further adoption and standardization of these techniques is the single highest-value improvement currently possible for this link.

For the link between CI/CD systems (often tied to repository hosts) and packaging indices, user-managed credentials are a recurring weakness and source of compromise: even when package indices support scope-able API tokens, manual scoping is error-prone and subject to fatigue-driven neglect (for example, using a "global" scope because determining the appropriate sub-scope is tedious in a push-driven development cycle). We believe that further adoption of automatic and self-sealing credential schemes like PyPI's Trusted Publishing[10] is the single highest-value improvement currently feasible for this link.

The problem of securing the link between the repository host and the downstream consumer's build process is much more difficult. For instance, many projects have unpinned dependencies, and so the packages that get installed for a particular end user are dependent on their individual deployment environment. This makes the identification and remediation of vulnerabilities significantly more difficult. Systems such as Nix and NixOS that aim to produce reproducible build systems can help with this, but come with significant engineering and ergonomic tradeoffs versus more traditional build-and-deploy flows.

In terms of "umbrella" solutions, Software Bills of Material (SBOMs) have risen in popularity in recent years as a prospective answer to the problem of opacity in software components. Unfortunately, in our estimation SBOMs are currently more of a "buzzword" than a practical technical solution: much of the tooling currently available for generating and consuming SBOMs is of middling quality and poorly aligned with the OSS community's interests (with regards to security fatigue and low-value vulnerability reports). We believe that opportunity for improvement exists, however: SBOM quality can be dramatically improved through direct integration with build systems, wherein the build itself becomes responsible for producing the SBOM from its own interior knowledge of dependencies. This should be more robust than existing post-facto analysis techniques used to generate SBOMs.

Finally, we note that strengthening these links in the supply chain can reveal additional weak links: as the OSS ecosystem realigns around SBOMs and other metadata sharing

[9] https://docs.github.com/en/code-security/secret-scanning/about-secret-scanning
[10] https://blog.pypi.org/posts/2023-04-20-introducing-trusted-publishers/

techniques, the need for tamper resistance in shared metadata will become pressing. We believe that signing schemes like Sigstore[11], along with provenance and attestation standards like SLSA[12] and in-toto[13] will be instrumental in the safe distribution of supply chain metadata.

Our Proposed Solution: CISA should release "strong link" guidelines for individual OSS developers, source repository hosts, OSS package indices, and downstream consumers. These guidelines should include account best practices (e.g., MFA), repository and CI/CD best practices (e.g., secure defaults for GitHub repositories, using technologies like Trusted Publishing for package index authentication), as well as guidance for individual "upstream" and "downstream" developers (proper generation of SBOMs from build processes, guidance for automating SBOM consumption). Where individual links fail to meet these guidelines, CISA should prioritize funding of OSS solutions that enable compliance (such as Trusted Publishing support in additional packaging ecosystems, or improved SBOM fidelity through build system integrations).

Reducing Entire Classes of Vulnerabilities at Scale

In recent years, there have been a few different effective methods that have emerged for reducing vulnerabilities at scale. Automated tools for package auditing and vulnerability tracking like cargo-audit[14] for Rust and pip-audit[15] for Python are highly effective tools for notifying developers to update their vulnerable dependencies. However, these tools' efficacy hinges on the quality and upkeep of the vulnerability databases they leverage. Thus, effectively managing these databases becomes a crucial effort for the community. This is a delicate balance because overreporting of less severe vulnerabilities can lead to security fatigue, where developers get overwhelmed by too many alerts that they end up ignoring all of them, including the actually severe issues.

Also, code scanning tools like weggli[16], semgrep, CodeQL, and dylint[17] play an essential role in automatically performing variant analysis to detect similar issues at scale. Trail of Bits regularly uses these tools to find issues in common libraries; for instance, we've disclosed multiple vulnerabilities in open source libraries by using CodeQL to target OpenSSL misuse.

[11] https://www.sigstore.dev/
[12] https://slsa.dev/
[13] https://in-toto.io/
[14] https://crates.io/crates/cargo-audit
[15] https://pypi.org/project/pip-audit/
[16] https://github.com/weggli-rs/weggli
[17] https://github.com/trailofbits/dylint

The more time that is invested into the development of novel analyses using these tools, and the more codebases that the tools are run against, the more effective they become. Dedicated funding to incentivize further development and testing of these tools could identify a massive number of vulnerabilities at scale.

In general, basic and advanced testing are under-incentivized in the OSS ecosystem. Efforts such as OSS-Fuzz[18], which has found thousands of bugs, have proven to be highly effective at discovering and fixing vulnerabilities at scale. Providing dedicated funding for these efforts through research grants could have a substantial effect on OSS security.

In 2023, eleven of the top twenty-five most dangerous software weaknesses were related to input validation or parsing[19]. Representing approximately 45% of the CVEs in CISA's Known Exploited Vulnerabilities catalog[20], these vulnerabilities make evident the need for enhanced security measures. Over recent years, there have been significant advances in the field of Language Theoretic Security, culminating in tools and techniques for developing safer parsers and safer file formats. The DARPA SafeDocs program has demonstrated the potential of these tools and techniques in securing DoD systems, improving file formats such as PDF, and hardening parsers. However, further refinement of these technologies is crucial towards attaining a higher readiness level suitable for independent application to open source software.

Taking the myriad open source HTTP parser implementations as an example, the likes of Apache, Nginx, and Node.js each boast their own custom parser, as does almost every other web server and proxy. Subtle differences in their implementations can lead to HTTP smuggling attacks[21]. It can also complicate matters for proxy servers engaged in path normalization, as parsing outcomes are contingent on the specific parser utilized. The prevalence of disparate HTTP implementations stems in part from the absence of a reliable, consistent, unambiguous, open source, cross-platform HTTP parsing library with bindings in all popular programming languages. The existence of such a library would alone have the potential to eliminate entire software weakness categories (CWEs).

Our Proposed Solution: Sponsor the development and maintenance of high-value vulnerability tracking tools like cargo-audit and pip-audit. As part of this maintenance, ensure that vulnerability databases are carefully curated and effectively managed,

[18] https://github.com/google/oss-fuzz
[19] https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html#top25list
[20] https://www.cisa.gov/known-exploited-vulnerabilities-catalog
[21] https://cwe.mitre.org/data/definitions/444.html

counteracting misaligned incentives around bogus vulnerability reports. In addition, provide research grants to support the development of advanced testing techniques applied at scale, using projects like OSS-Fuzz as a model. These grants could support projects using code scanning tools, new fuzzing techniques, artificial intelligence, and other novel ideas. In addition, we recommend sponsoring the research, development, and maintenance of tools and techniques for creating safer parsers, file formats, and automatically detecting input validation bugs. In particular, sponsoring the engineering of a robust (i.e., formally verified) open source parser for HTTP requests has the potential to completely eliminate CWE 444.

Developer Education

Most security-focused education for developers has focused on well-known classes of vulnerabilities. Taxonomies such as the OWASP Top 10 and the CWE Top 25 often provide ready-made curriculum outlines. Although such approaches are highly effective at familiarizing developers with common security bugs, they have left gaps in two critical areas that are reflected in security outcomes.

Minimizing attack surface and taming feature accretion

A common problem in popular, critical OSS projects is that of feature accretion: open source contributors add novel features that satisfy their own requirements, often without concern for long-term maintenance or preserving a fixed project scope. Feature accretion necessarily leads to unexpected interactions between components, and therefore a larger attack surface.

At least two of the most prominent and widely reported OSS-related vulnerabilities have stemmed from unexpected interactions in accreted features. The Log4Shell vulnerability family[22, 23] is exploited by invoking a Java Naming and Directory Index (JNDI) lookup via the LDAP protocol, a component interaction that was almost certainly not intended by either feature's author. Similarly, the Apache Struts remote code execution vulnerability[24] leverages a probably-unintended interaction between Object Graph Navigation Library (OGNL) lookups and HTTP request header values. In each case, it is reasonable to presume that few developers using Log4j and Struts relied upon these features, and that most would have taken steps to disable JNDI and OGNL lookups had they investigated the respective software packages and understood this aspect of their operation.

[22] https://nvd.nist.gov/vuln/detail/CVE-2021-44228
[23] https://nvd.nist.gov/vuln/detail/CVE-2021-45046
[24] https://nvd.nist.gov/vuln/detail/CVE-2017-5638

OSS projects are often designed to support a wide variety of use cases, with feature accretion occurring as a natural consequence. Adoption of OSS components often entails adopting more software capabilities than actually required for a particular use case, with an ensuing increase in risk due to sprawling attack surface.

Tackling this problem requires both human and technical solutions: human processes must be realigned to evaluate the full capabilities of OSS components before integrating them for singular purposes, while technical solutions like capability-based security and sandboxing can offer practical guarantees that an integrated component does not exceed its intended behavioral scope. Runtime-level language capabilities like the Node.js permissions model[25] demonstrate promise in the latter area, while other languages like Rust, Python, and Go have considered capability management as a solution to arbitrary code execution during package construction and installation.

Security testing and "shifting left"

In Trail of Bits' consulting work, a high percentage of reported security findings are discovered through the use of security testing techniques that are accessible to development teams throughout the SDLC. Static analysis tools such as Semgrep and CodeQL help our auditors identify coding antipatterns and data flow paths that indicate vulnerabilities, while fuzz testing uses strategic randomization algorithms to generate inputs that invoke a wide variety of execution paths, thereby maximizing coverage. Projects can "shift left" by implementing these tools in earlier phases of their SDLCs, enabling earlier detection of security issues, reduced risk exposure, and better software development practices.

Some security testing tools are documented in a manner that is complex and theoretical, making it difficult for developers to get the most benefit out of them. To this end, the Trail of Bits Testing Handbook[26] is an ongoing project from Trail of Bits that provides practical, actionable guidance on deploying and configuring security testing tools for maximal value.

Software testing is a frequent topic of developer education. Unit testing and methodologies such as test-driven development are often taught as indispensable components of the SDLC. Security-focused testing, on the other hand, is often taught solely as a specialization for offensive security practitioners (i.e., penetration testers and other security consultants). Instead, it should be taught as a tool that can enable development teams to produce better

[25] https://nodejs.org/api/permissions.html#permissions
[26] https://appsec.guide/

software more efficiently, particularly when implemented early in a project's lifecycle and as a part of the CI/CD pipeline.

Our Proposed Solution: CISA should treat attack surface reduction, software capability enumeration and management, and early security testing as indispensable components of OSS security. As such, we recommend that CISA produce guidance and training documentation to emphasize the importance of disabling unneeded features, removing unneeded components, and how to incorporate more advanced testing techniques. Additionally, we recommend that CISA fund both research and engineering initiatives towards automatic attack surface enumeration, as well as runtime capability management for languages like Rust, Go, and Python.

Conclusion

OSS security is a core value within Trail of Bits, and we appreciate the opportunity to provide our response to the US Government's RFI.

Memory corruption vulnerabilities and software supply chain attacks are arguably the most impactful classes of vulnerabilities currently affecting OSS security. Therefore, we believe fostering memory safe programming languages and strengthening software supply chains are the highest priority sub-areas to focus on. After these two, reducing entire classes of vulnerabilities at scale and strategically improving developer education have, in our opinion, the highest potential for positive impact on OSS security in the long-term. For each of these sub-areas, we believe the U.S. Government can maximize impact through a combination of three strategies: provisioning of comprehensive guidance, allocation of funding through agencies like DARPA and ONR, and fostering collaboration with OSS foundations like the OSTIF, OTF, and OpenSSF. This combined approach will enable the sponsorship and monetary support necessary to drive the research and engineering tasks outlined in our proposed solutions.

We pride ourselves on contributing to the improvement of OSS security through our various engineering and assurance projects. Improving OSS security is an incredibly difficult task that will not be solved overnight, and we hope that the recommendations outlined in our response will help move the needle.

We invite you to direct your attention to the Trail of Bits blog[27], where we regularly post updates on our work in OSS, software hardening, and supply-chain security.

[27] https://blog.trailofbits.com/

Appendix

Our Background and Experience

Founded in 2012 and headquartered in New York, Trail of Bits provides security assessment and engineering services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 130+ employees around the globe, we've helped secure critical open source software elements that support billions of end users.

Trail of Bits has extensive experience in open source ecosystems, ranging from security audits and security engineering consulting on some of the world's most critical codebases to our own *pro bono* open source projects and security tooling.

In addition to our ongoing research and development projects funded by agencies like DARPA and ONR, we regularly work with groups like the Open Technology Fund (OTF), Open Source Technology Improvement Fund (OSTIF), Sovereign Tech Fund (STF), and Open Source Security Foundation (OpenSSF) to obtain funding for our own auditing and engineering work, as well as with language- and ecosystem-specific foundations like the OpenJS Foundation and Python Software Foundation.

On the software assurance side, we have performed audits of critical projects and ecosystems like cURL[28], Kubernetes[29], and the Linux Kernel[30]. On the engineering side, we have contributed critical safety features and improvements to projects like PyPI and PyCA Cryptography, as well as novel security tooling (dylint, pip-audit, and sigstore-python).

[28] https://github.com/trailofbits/publications/blob/master/reviews/2022-12-curl-securityreview.pdf
[29] https://github.com/trailofbits/audit-kubernetes/tree/master
[30] https://github.com/trailofbits/publications/blob/master/reviews/LinuxKernelReleaseSigning.pdf