Shortridge Sensemaking LLC

November 2, 2023

Kemba E. Walden, Acting Director
Office of the National Cyber Director
The White House
1600 Pennsylvania Ave NW
Washington, DC 20500

RE: Doc. No. ONCD-2023-0002; Request for Information on Open-Source Software Security: Areas of Long-Term Focus and Prioritization

Dear Acting Director Walden,

As leaders in cybersecurity and software engineering, we appreciate the opportunity to submit this statement in response to the requests for comment by 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) – the "requesting agencies" – concerning the long-term focus and prioritization on open-source software security.

As organizations increasingly adopt software to fulfill their missions, the impact of software failures on their operations increases. If those failures correlate or cascade across organizations and industries, it could unleash systemic harm on our nation. Sustaining resilience in our software systems is no longer aspirational, but imperative if we wish to weather such storms.

What we need is for our systems to adapt to the unexpected and unintended. We envision a future in which continual adaptation to adversity is a natural part of how we maintain systems. All systems are in transition; they are dynamic, ever-changing. Now is a poignant moment to incentivize resilience as part of their ever-changing nature.

Yet, we may just as easily poison resilience through misguided regulation and recommendations. Resilience is not about preventing failure. To sustain resilience is to minimize the impact of failure and ensure we can change systems – their behaviors, designs, practices, and so on – to keep up with the challenges presented by their external reality.

The reality of complex systems is that it is impossible to prevent failure. We cannot prevent software vulnerabilities, nor disk failure, nor network outages from ever occurring. Preventing humans from making mistakes is an even more quixotic ambition. But we can, as an ecosystem, minimize harm when those failures occur. We can ensure that a problem in one part of the ecosystem does not cascade to the rest. We can prepare for the inevitable so that when something goes wrong, we can recover from it swiftly and safely.

Resilience also means adapting to harness opportunities, not solely survive failures. Open-source software (OSS) is a growth engine for the United States; the creativity bursting from its ecosystem nurtures national innovation. The requesting agencies cannot stall this engine through corrosive regulatory interference lest they catalyze more harm than good.

It is not just economic growth, but our geopolitical position that is at stake. The United States is competitive internationally in part due to its technological ingenuity and execution. Existing regulatory requirements already stifle innovation and hinder software velocity in the name of nebulous benefits. Knee-jerk regulatory responses that inevitably ossify may satisfy action bias, but not our noble goal of a resilient software ecosystem.

The software industry added $1.9 trillion to U.S. GDP in 2020, $933 billion of which was directly[1]. While estimating the cost of cyber incidents is fraught[2], the FBI Internet Crime Complaint center estimated $4.1 billion in cybercrime losses in 2020 – many of which are due to phishing and other scams rather than software exploitation. The Verizon Data Breach Investigations Report (DBIR) found that vulnerability exploitation is present in only 5% of breaches (stolen credentials and phishing are the overwhelming attacker modalities in non-error, non-misuse breaches)[3]. NotPetya, arguably the worst cyberattack (due to vulnerability exploitation) in terms of systemic impact, resulted in $10 billion in total damages across the world[4]. There would need to be two NotPetya-level incidents per year within the United States for losses to reach even 1% of the economic benefits stimulated by software.

The worst outcome we perceive from this request for information (RFI) is regulatory intervention that barely makes a dent in already-low losses while generating yet more compliance checklists and "busywork" for organizations. We do not want the requesting agencies' efforts, no matter how well intentioned, to become ripe for regulatory capture.

This motivates us to clarify the core goal outcome of this project to help the requesting agencies navigate away from such troublesome waters. Based on the "critical questions" posed and the rest of the RFI's text, we believe we can synthesize and clarify the core goal outcome into the following statement: **we do not want unintended behavior in code to generate systemic damages.** We believe that the requesting agencies – and likely all stakeholders in the software ecosystem – specifically want software providers to contain the impact of failures in their code**.**

We do not want failure in the software world to cause tangible socioeconomic impact on the national scale. Of course, organizations are motivated to minimize the impact of failure for their localized, individual concerns. But we believe the implicit concern from the requesting agencies is that failure in some software entity – whether an

[1] https://software.org/reports/software-supporting-us-through-covid-2021/
[2] https://www.cisa.gov/sites/default/files/publications/CISA-OCE_Cost_of_Cyber_Incidents_Study-FINAL_508.pdf
[3] https://www.verizon.com/business/en-gb/resources/2023-data-breach-investigations-report-dbir.pdf
[4] https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world/

application, library, framework, language, or tool – will “escape” the digital sphere and cascade into economic or social failures.

Our recommendations throughout the document focus on this core outcome. We believe this is useful to optimize and guide what changes we want to incentivize – to avoid boiling the proverbial ocean – as well as to uncover where we need caution, even if those solutions feel more convenient or obvious.

Our response begins by describing multiple Gordian Knots we believe will offer the requesting agencies alternative perspectives on the problem at hand. The rest of the response is structured with recommendations in the areas and subareas where our expertise is relevant, in the same order as presented in the RFI. Additionally, we identify and recommend multiple new subareas of focus for prioritization, including isolation, modular design, automation (CI/CD), resilience stress testing, and others; many of these are suffused with the spirit of Gordian Knots.

We are aware this is a lengthier response than is typical, but we sought to be exhaustive in offering our expertise across problems and areas of focus. This moment in spacetime is a critical juncture in software – not just OSS – and we feel privileged to submit our recommendations for the requesting agencies to consider as they traverse these challenges.

1. Gordian Knots

1.1 Is OSS critical infrastructure?

There is an implicit decision slinking in this discourse, one with both philosophical and practical implications: should open-source software be considered critical infrastructure?

If we treat OSS like critical infrastructure, then we need the related trappings of it – similar to what we require of physical critical infrastructure, like bridges, or financial critical infrastructure, like the stock market. It is not an understatement that this would upheave the software sector in the United States; we believe the requesting agencies must treat it as the grave proposition it represents and not wave such speculation around casually.

If we care about systemic impacts – about software wreaking widespread socio-economic harm – then we must equally consider the socioeconomic harm that deeming OSS critical infrastructure would inflict. There is a tradeoff between international competitiveness in technology innovation and regulating OSS. There also exists a tradeoff between national economic growth – growth that spans industries – and regulating OSS as critical infrastructure.

Through the lens of socioeconomic concerns, the benefits of OSS dominate its downsides at present; the OSS ecosystem is, overall, working well for us as a nation. Is it worth sacrificing these socioeconomic benefits because, on occasion, there are negative impacts?[5]

We believe a concrete and recent example can illustrate what we mean when we assert that the OSS ecosystem is working well overall. In a recent post about an attack campaign leveraging malicious Python packages, researchers found that the attackers created 272 malicious packages available for download[6]. This reflects approximately 0.06% of the roughly 469,000 total packages in the Python package ecosystem (known as "PyPI" for the Python Package Index).

When we instead look at *downloads* of these malicious packages, which totaled 75,000 over six months[7] (or around 417 times per day), the impact is even smaller. The top 20 libraries in PyPI are downloaded approximately 200 million times per day[8]. This means the malicious PyPI package downloads are 0.0002% of downloads across the top 20 packages. Even if we assume there are concurrent attack campaigns of this nature, it means downloading Python packages is as safe or safer than riding trains[9] and a developer is at least 4,527x less likely to download a malicious Python package than to die in a car crash[10]. This is especially impressive given both the rail and automobile industries face significant safety regulation to achieve these statistics, while the Python ecosystem's efforts are entirely organic.

We believe this demonstrates how well the OSS ecosystem is working in terms of safety, despite what cybersecurity vendors and attention-grabbing headlines would like us all to believe. As we discuss in **Section 2.3**, open-source package ecosystems are *positive* examples of governance. We encourage the requesting agencies to better understand how those ecosystems work and to rely on evidence rather than succumb to sensationalism.

In sum, we do not believe OSS should be regulated as critical infrastructure – especially if we do not wish to sacrifice the benefits the OSS ecosystem gifts us. Yet, there are still opportunities for improvement to reduce socioeconomic impact when something goes wrong in software. To balance these concerns, we again return to the restated goal

[5] This sets aside the net benefits to the global community, which we will treat as a separate concern given the requesting agencies are all located within the United States.
[6] https://gist.github.com/masteryoda101/65b55a117fe2ea33735f05024abc92c2
[7] https://checkmarx.com/blog/the-evolutionary-tale-of-a-persistent-python-threat/
[8] https://pypistats.org/top
[9] https://www.thedailybeast.com/will-i-die-on-a-train
[10] It is safe to assume that the total number of daily downloads across PyPi greatly exceeds that of the top 20 packages.

above that we seek containment of impact from software failures – and not just in OSS, but software of all kinds, as we will turn to next.

1.2 Expanding the scope beyond OSS

One of the cybersecurity industry's latest obsessions is around the "software supply chain," and we can understand why: open source's viral adoption capability means it now underpins nearly all software products and infrastructure on the market, and the ease at which one can examine open-source software suggests to many that it is straightforward to discover and exploit vulnerabilities.

OSS also exhibits a much wider variety of development practices than a single software vendor would. This dynamic results in simplistic, surface-level arguments that open-source software delivery pipelines are insecure and require more oversight.

That may be true, but, if so, it is equally true for *proprietary* software delivery pipelines. Many companies have poor practices around code review, access auditing, version control, and build reproducibility – not to mention vulnerabilities in the source code itself. Proprietary software (sometimes called "closed source") is not exempt from abysmal, insecure development practices just because of its licensing. From the perspective of systemic impact, proprietary software is often implicated in the most damaging attack campaigns, including the NotPetya and SolarWinds incidents.

Some of the best managed software projects are open source, where multiple organizations contribute according to norms that the community informally or formally agrees upon. It is perhaps not a coincidence that we struggled to find reference examples of exploits in OSS vulnerabilities that generated significant systemic damages.

The RFI references the Log4j vulnerability, which discharged far less damage than feared. The 2023 Verizon DBIR revealed that Log4Shell was only present in 0.4% of incidents in their data set[11], and security vendors witnessed Log4Shell attacks wane considerably as 2022 progressed[12]. Heartbleed, an OpenSSL vulnerability disclosed in 2014, was touted as "the worst vulnerability found"[13]; but outside of a few cases of personal identifiable information (PII) theft, the impact was low (so low, in fact, no one seems to have bothered quantifying it).

Accordingly, we recommend the requesting agencies expand their scope to cover all software, not just open-source software, for a few reasons:

1) Proprietary solutions are not inherently more secure. If we assess how attackers gain access to organizations by exploiting software, it is predominately through commercial "bolt-on" solutions, like in the SolarWinds incident or, in the case of

[11] https://www.verizon.com/business/en-gb/resources/2023-data-breach-investigations-report-dbir.pdf
[12] https://news.sophos.com/en-us/2022/01/24/log4shell-no-mass-abuse-but-no-respite-what-happened/
[13] https://www.forbes.com/sites/josephsteinberg/2014/04/10/massive-internet-security-vulnerability-you-are-at-risk-what-you-need-to-do/

ransomware campaigns, vulnerabilities in virtual private networks (VPNs)[14]. Network appliances are especially notorious for their subpar practices[15], both for code development as well as configuring the environment where the code runs[16][17]. These proprietary appliances typically lack software security "basics" like ASLR, stack canaries, and allocator hardening.[18][19] While nation state actors may be able to bypass these "basics," most attackers lack the resources and sufficient return on investment (ROI) to defeat such mechanisms; ensuring those basics are in place raises the cost of attack across all attacker types.

2) Applying onerous requirements *only* on OSS will engender perverse incentives. Such requirements would serve as an intangible subsidy to proprietary vendors (who again, we must stress, do not inherently produce safer or higher quality code). The effect would likely be less use of OSS in systems delivered to the federal government. Another potential outcome is that vendors would produce proprietary, unmaintained variants of OSS projects for use in their products to evade OSS requirements – and we believe this outcome does not beget safety.

3) The federal government already has more visibility into open-source software as compared with proprietary software. Why is more insight into OSS the primary point of concern when government agencies already have much more visibility into OSS as compared to proprietary software? In **Section 2.3.2** we offer recommendations to ameliorate this problem.

4) Potential systemic impact depends on a software system's ubiquity of adoption, criticality, and degree of access in customer systems; its licensing model is one of the least important factors determining impact. Big shocks come from widely deployed, vulnerable components that are trivially accessible from the internet or via connectivity to a vendor's centralized control plane. Thus, we believe the requesting agencies should focus on the general problem of software components with deep access in customers' systems – regardless of licensing – especially those with widespread adoption that are "too big to fail."

The SolarWinds incident was an example of this final point; it is not open source, but the damage resulting from a malicious software update in 2021 spanned industries – and, of course, compromised federal agencies, too. It is worth noting that a core value proposition driving SolarWinds' adoption is compliance requirements. We believe, from a correlated risk perspective, that incumbent software solutions that address federal compliance requirements (including PCI, HIPAA, FedRAMP, and others) are especially

[14] https://attack.mitre.org/techniques/T1133/
[15] https://www.cisa.gov/news-events/alerts/2016/09/06/increasing-threat-network-infrastructure-devices-and-recommended
[16] https://iopscience.iop.org/article/10.1088/1742-6596/1714/1/012045/pdf
[17] https://www.darkreading.com/perimeter/attackers-heavily-targeting-vpn-vulnerabilities-/d/d-id/1340770
[18] https://browse.arxiv.org/pdf/2007.02307.pdf
[19] https://www.usenix.org/system/files/login/articles/login_summer17_14_wetzels.pdf

valuable targets for attackers. If a vendor is “sticky” in a customers’ technology stack because they “must” buy it to meet non-negotiable regulatory requirements, then there is less incentive for the vendor to innovate or prioritize security.

Worse, many of these proprietary solutions require significant levels of access to and control over customer systems to fulfill compliance requirements. For example, if a “zero trust” solution offers a central control plane to manage access policies across a customer’s systems, then it becomes a valuable target for attackers, who can exploit it to manage access as they please. Even more dangerous are solutions that can push code to other machines, like some endpoint detection and response (EDR) tools which run with privileged access on production systems. A heuristic to determine potential systemic impact might be to look for commercial software solutions that have high privilege, widespread deployment or access, low oversight, and few vendors (i.e., a few incumbents have most of the market share).

1.3 Systems thinking

We also must caution the requesting agencies about a narrow focus on *code* rather than on *systems*. The philosophy underlying the RFI appears to be that if we secure individual software components, the overall system will be secure. This can be characterized as a bottoms-up approach. Unfortunately, it defies the nature of complex systems – and software is inevitably complex (involving many interacting, interconnected components).

Vulnerable code itself is harmless until it runs on infrastructure when it interacts with users, whether humans or machines. Individual code components may be secure in themselves but not when interacting with other parts of the software – not unlike how a coffee maker may be “safe” in isolation but lead to catastrophic failure when placed near electrical equipment on an airplane[20].

This interactivity can even be *helpful* for security. Why was Log4Shell’s impact so negligible, despite what many experts forecasted? Possibly it is due to swift collaboration across the industry to protect against the attack and patch systems as quickly as possible. But a key contributing factor is that attackers must exploit Log4Shell based on the application’s context[21]; if attackers must configure or customize the attack for it to work on a target system, then mass exploitation is infeasible.

This reveals an important insight: OSS, like most software, is rarely a standalone component. Organizations integrate OSS with other software components as part of their applications, services, and systems – and those systems are usually highly customized and configured for the organization’s context. Few implementations are exactly the same

[20] https://www.jasoncollins.blog/posts/perrows-normal-accidents-living-with-high-risk-technologies
[21] https://news.sophos.com/en-us/2022/01/24/log4shell-no-mass-abuse-but-no-respite-what-happened/

across organizations; where they are the same, their exposure is typically limited (e.g., few organizations stick MySQL on the public internet, and it generally requires authentication).

We believe mass exploitation is the most likely driver behind potential systemic catastrophe due to OSS, but we also suspect mass exploitation of any one OSS component is challenging. In contemplating other correlated, tail-end impact attacks, we believe NotPetya to be the exemplar. NotPetya leveraged the leaked EternalBlue exploit, which exploited a vulnerability in Microsoft's implementation of the Server Message Block (SMB) protocol in the Windows operating system[22] – an implementation which was relatively homogenous across Windows systems.

If correlated socioeconomic impact is the requesting agencies' concern, the focus should not be on critical software *components* but on critical software *systems* – how we can ensure that unintended interactivity between elements in a system does not instigate harm. We encourage the requesting agencies to adopt a systems perspective when prioritizing their efforts, rather than a component-focused, bottoms-up approach.

2. Secure Open-Source Software Foundations

Our recommendations throughout this section are grounded in what we believe would help reduce correlated socioeconomic impact from attacks in the future, whether in OSS or commercial software (per the Gordian Knot in Section 1.2). The section itself is structured to follow the sub-areas listed in the RFI under the primary area of "Secure Open-Source Software Foundations" and then proposes new sub-areas of focus.

2.1. Fostering the adoption of memory safe programming languages

The RFI suggests a core aim is to "reduce the proliferation of memory unsafe programming languages," which we interpret to mean slowing the growth of unsafe codebases (i.e., developers writing more memory unsafe code). Instead, we believe the goal should be to reduce the **impact** of memory unsafe code, which involves reducing the growth of new unsafe code, slowing the furthered use of existing unsafe code, refactoring existing unsafe code, and minimizing the impact of all unsafe code (whether new or old). Our recommendations throughout this section reflect these four endeavors.

We agree that, when possible, software providers should refactor their C or C++ code into memory safe languages. To "refactor" a system is to change its underlying materials, methods, structure, or organization while maintaining the functionality it provides. There are different degrees of refactoring: changes can be localized or widespread; they can wholly replace a specific component or apply a common change to numerous components; refactors can "land" all at once, or in stages. All approaches are suitable for combatting the hydra of memory unsafety.

The adoption of memory safe programming languages is occurring organically in the private sector – and largely not for security reasons. Engineering teams are selecting

[22] https://www.cisa.gov/news-events/alerts/2017/07/01/petya-ransomware

memory safe languages because they support reliability and developer productivity better than unsafe languages. Few developers enjoy chasing stability problems provoked by the hazards of unsafe languages, as that interrupts their coveted "flow."[23]

Nevertheless, we believe the federal government can accelerate this adoption. In particular, we believe the requesting agencies can and should incentivize the adoption of memory safe languages among its contractor and vendor base.

One potential incentive mechanism is setting a timeline for federal contractors to only build memory safe systems; that is, by a certain date, contractors must write and implement any new software in memory safe languages if they are receiving financial assistance from the federal government. This also implies that contractors must select memory safe OSS components for the software they deliver to federal agencies, which may indirectly influence the open-source market. As a steppingstone incentive towards this paradigm, federal agencies could prefer contracts from vendors who only build memory safe systems.

Grants are another financial incentive we believe the requesting agencies should consider. The requesting agencies could require grant recipients – including non-profit entities like national labs and universities – to implement software in memory safe languages. Similarly, the requesting agencies could provide grants or other financial benefits to universities who incorporate memory-safe languages into their engineering and computer science curriculum. Many students today are future open-source maintainers, so we should make it easy for them to learn memory safe languages.

These incentives must extend to the lowest levels of software, too, where memory safety is less common, such as BIOS and firmware (including hardware roots of trust). The requesting agencies should prioritize components that allow extensive control or access in the system and / or must process data from a variety of sources, some of which may be exogenous and therefore not trusted.

This incentive scheme could take the form of a nearer-term deadline to deliver such foundational components in memory-safe languages, or immediately paying a premium during procurement for hardware and software written in memory-safe languages. In the spirit of Gordian Knots, we suggest that advanced techniques for applying memory safety to otherwise memory-unsafe languages, such as Checked C[24], CHERI[25] and Apple's memory-safe iBoot[26], should also be eligible.

With all this said, we recognize that not every organization can immediately begin refactoring their code into memory safe languages. A requirement such as "rewrite everything in Rust" is infeasible – especially if the requesting agencies' scope is beyond their

[23] https://queue.acm.org/detail.cfm?id=3454124
[24] https://www.microsoft.com/en-us/research/publication/checkedc-making-c-safe-by-extension/
[25] https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/
[26] https://support.apple.com/guide/security/memory-safe-iboot-implementation-sec30d8d9ec1/web

contractor base. In the private sector, for-profit organizations, by definition, will prioritize projects that support revenue and profit goals; it is tricky to tie memory safety to those goals except through the counterfactual of avoiding potential security and performance incidents. As a result, new systems are often written in memory safe languages, but existing systems continue using whatever language they were originally developed in, even as they receive substantial expansions and deepen their criticality. Our recommendations throughout the rest of Section 2.1 – and indeed all of Section 2 – reflect this reality of needing variegated solutions that supplement refactoring.

The biggest challenges organizations face with refactoring their software into a memory safe language are often not technical, but social. Production pressures make it difficult to allocate finite resources to efforts that bear little direct business payoff. C or C++ may be the only language known by all members of a team (who were originally hired to work on a C or C++ system). Technical decision makers may experience ownership and other biases towards design and implementation choices they made years ago[27]. Parts of the system may require direct interaction with hardware or hard real-time performance, which may be incompatible with some memory safe languages. Support contracts may require vendors to maintain legacy outdated versions and the vendor may be insufficiently staffed to support developing two systems in parallel. Government agencies could lead by reforming their support contracts to prefer continual updates to new versions rather than backporting of security updates to old versions.

However, we do not believe in the commonly espoused notion that not all systems can be memory safe. All *systems* can be primarily memory safe, but not all system components necessarily must be. Again, we believe memory safety should be the default. It is a lack of vision that has us still designing new critical systems without memory safety in 2023. Numerous memory safe languages offer escape hatches to directly address hardware in cases that require it. In memory safe languages, the escape hatch isn't open by default. Not all memory safe languages are suitable for all systems, but we now have memory safe languages for even the most stringent of system requirements, thanks to the availability of a qualified version of Rust[28]. Again, it's better, but it's not a panacea, and we need to be careful about the incentives – especially when recommendations calcify and can't keep up to date with innovation.

2.1.1 Design-based mitigations

Given it is infeasible to magically rewrite the entire C and C++ ecosystem into Rust, we believe the government should consider additional design-based mitigations, including isolation and modularity (which enables iterative migration of components from C/C++ into memory safe versions); we discuss each in **Section 2.4** and **2.5** respectively.

[27] https://thedecisionlab.com/biases/endowment-effect
[28] https://ferrous-systems.com/blog/qualifying-rust-without-forking/

There are other steps organizations can take short-term with their unsafe code that will still improve security outcomes, which we encourage the requesting agencies to consider as they identify and prioritize focus areas.

To address interdependencies during refactoring, we believe the requesting agencies should focus on integration testing and modularity. Integration testing is the process of applying automated testing to a fully integrated system, verifying the system behaves as expected from a set of well-defined inputs. Unlike physical systems, software systems can be easily coordinated, tested, monitored, validated and reset to a default state. Constructing and performing automated full system integration tests is critical to building confidences in structural changes to a system[29]. As one example, Jepsen is a system for testing distributed systems that instantiates the system anew each time, generates transactions, injects faults, and searches for inconsistencies[30]. FoundationDB similarly tests their database by simulating entire clusters of machines including their networks in a single deterministic process[31].

2.1.2 Unique considerations with unsafe code

When organizations write and maintain their software in C or C++, they face unique concerns:

1. Organizations with C or C++ code should assume the presence of vulnerabilities by default. It is impossible to build C or C++ software at scale without having some memory corruption vulnerabilities, so maintainers must turn on hardening features, use program analysis tools, and generally be cautious to compensate.

2. Hardening features present a poor developer experience (DX). Many are not enabled by default on popular platforms. They are inconvenient to programmers, who may be prone to turn them off, especially as some impose a performance penalty and make debugging more cumbersome. Worse, common embedded, real-time, and obscure proprietary operating systems have poor support for hardening features compared to their mainstream peers. We should consider environments without basic hardening features enabled unfit for use in critical infrastructure, but, at present, there is no incentive for vendors to change this practice. Opaque vendor-supplied appliances often present such environments.

3. Program analysis tools such as static analyzers and dynamic sanitizers/checkers are separate steps that require dedicated effort to configure, operate, and respond to. Most of their findings are benign or appear benign, but it can be difficult to tell benign findings from critical vulnerabilities without an educated assessment[32]. Many

[29] https://queue.acm.org/detail.cfm?id=2889274
[30] https://github.com/jepsen-io/jepsen/blob/main/README.md
[31] https://apple.github.io/foundationdb/testing.html
[32] https://arxiv.org/pdf/1805.09040.pdf

vulnerabilities are completely opaque to static analysis tools[33]. These assessments distract from productive development activity.

In the spirit of federal agencies offering specific recommendations on hazardous physical materials, like lead or asbestos, we offer specific recommendations for hazardous software materials like C/C++ code. While we again stress that we must incentivize organizations to refactor their code into memory safe languages, below are suggestions organizations should consider as “the basics” to improve the quality and safety of their C/C++ code, including:

- Turn on hardening-related compiler options such as stack protection, ASLR, relocation read-only, and trivial variable initialization
- Disable executable stacks and heaps, as many platforms do by default
- Use parser-generators when parsing inputs
- Separate your software into separate services that operate with limited permissions for the purpose they serve
- Consider which compiler warnings are right for your security requirements
- Write comprehensive automated tests and run them against the system compiled with address sanitizer and undefined-behavior sanitizer enabled
- Set up fuzz testing for security sensitive components, such as those that process untrusted data
- Disable variable-length arrays and alloca(), preferring dynamic allocation instead
- Opt into _FORTIFY_SOURCE and other platform- or library-specific hardening
- Use a security-hardened default allocator for malloc/free
- Carefully validate custom memory arena sizes and abort when limits are exceeded
- Consider enabling -fwrapv (or your compiler's equivalent) for consistent signed arithmetic in less performance critical systems or modules
- Avoid pre-fork style architectures, which render ASLR ineffective

Some organizations will perceive this as an onerous list. But, if we continue with the analogy of hazardous physical materials like lead to hazardous software materials like C/C++ code, the point is clear: some materials are so dangerous that the guidelines for dealing with it *must* be different – and more stringent – because the hazards they present significantly exceed the baseline dangers presented by safer materials (like unleaded fuel or memory safe languages).

Like lead, we envision a future where the hazardous material sees continued use by industry, but in a reduced capacity and with appropriate caution. Most software contains much more memory unsafe code than is necessary and, like efforts to reduce lead exposure, reducing the amount of it will take concerted efforts over decades. It will require

[33] https://mediatum.ub.tum.de/doc/1659728/1659728.pdf

standing up to an industry (software) that is poisoning both the community at large and its own workers – and additionally to an industry (cybersecurity) that profits off the continued existence of unsafe code.

2.1.3 Safer change practices

Memory safety is a pressing problem, as the NSA stressed last year[34]. However, to be resilient to any type of failure – including attacks – is to adapt and evolve with speed and grace. Whether to transmogrify their codebase into a memory safe language or to better compete in their market, organizations must pursue safer change practices. Such practices, as we describe below, explicitly embrace speed and encourage development velocity. We caution regulators to avoid equating security with slowness or friction, as heavy change processes are often what impede an organization's ability to sustain cyber resilience[35].

Specifically, we recommend an iterative change model rather than the traditional "big bang" release model:

- **Iterative change model**: Software maintainers can pursue an iterative change model: selecting one part of the program (a "component") and changing its underlying materials – language, libraries, frameworks, tools – to achieve the desired goal characteristics (often reliability, quality, or safety). After this refactored component is released, they could proceed to refactoring the next component, refactoring and releasing over time until the entire program is changed. This is typically considered "best practice" in software engineering and is referred to as the "Strangler Fig" pattern (as it imitates the biological process of the Strangler Fig tree)[36].
- **"Big bang" release model:** There is also the "big bang" refactor, where the entire program is changed as part of one release rather than iterative releases. This is considered an anti-pattern in modern software engineering, as a large volume of changes pursued at once is harder to test and debug, resulting in higher rates of failure in production and slower times to restore service health when failure occurs[37].

We encourage the requesting agencies to incentivize an iterative change model, encouraging organizations to prioritize refactoring system components that pose the greatest impact if exploited.

For many organizations, the system components that pose the greatest impact are those that sustain business operation: for a retail company, the ability for their customers to purchase goods; for a mining company, their ability to operate their fleet; for a

[34] https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues/
[35] https://cloud.google.com/architecture/devops/devops-process-streamlining-change-approval
[36] https://martinfowler.com/bliki/StranglerFigApplication.html
[37] https://core.ac.uk/download/pdf/326836096.pdf

manufacturing company, their ability to run machinery; for an energy company, their ability to supply power to customers. There are complimentary services that, while still valued by the organization, are not as critical for business operation and therefore are not as high priority, including: analytics, business intelligence, reporting, customer relationship management, marketing, A/B testing, and more.

From an outcomes perspective, we want organizations to improve the resilience of their systems to cyberattacks, which means ensuring the system can still provide its critical functions under adversity. Refactoring 90% of system code, but not the 10% that constitutes this critical functionality – like the path processing business-critical transactions – will be closer to “security theater” than cyber resilience.

2.1.4 Tooling

We do not believe the government can develop tools to “automate and accelerate the refactoring” of software into memory safe languages[38]. Tooling is not the problem; our tooling to decompose software and refactor is sufficient. The problem is incentives and funding. We *can* rewrite our entire ecosystem into memory safe languages. We have the tools to do it and enough people to do it, but the missing piece is paying them for those efforts.

2.1.5 Presence of C in the Lowest Layers of the Software Stack

C and C++ is deeply embedded in the lowest layers of the modern software stack, with operating system kernels, standard libraries, memory allocators, cryptographic libraries, compressors, image decoders and more implemented in it. These components process untrustworthy data on behalf of higher-level components written in safe languages, and thus even predominantly memory safe systems can be subject to memory safety vulnerabilities. As it is rare for software vendors to differentiate themselves in the market through these layers, they get less maintenance than other layers. Federal agencies could benefit the entire ecosystem by incentivizing active maintenance of these layers, including funding of memory safe substitutable replacements or rewrites. Crucially, these must be production-ready replacements and not languish as research projects.

2.2 Reducing entire classes of vulnerabilities at scale

We believe this focus area deserves reframing. Instead of “reducing entire classes of vulnerabilities at scale,” we believe it is more helpful to pursue a goal of “reducing impact at scale”. We believe the requesting agencies should be open to reframing their areas of concern, including this one, so as to not limit the opportunities at their disposal – and to ensure any resulting recommendations are outcomes-driven.

[38] We especially caution the requesting agencies from believing AI tools can help, such as with automated translation; the quality of automated translation is universally abysmal, and we strongly believe AI would *reduce* the quality. Neither regular nor AI-flavored automated translation is a worthwhile endeavor.

Related to reducing impact scale, we can reframe reducing entire classes of vulnerabilities to a subgoal of “reducing entire classes of attacker *action*.” Ultimately, a vulnerability does not matter much unless it is exploited *and* leads to tangible impact. For example, isolation (discussed in Section 2.4) makes remote code execution (RCE) – the ability for an attacker to run whatever code they want without having physical access to the machine – less meaningful for an attacker. The attacker can gain RCE in an isolated component, but they cannot expand their access to other components or resources without additional difficulty. Unless that component offers the precise access the attacker needs to achieve their goals – such as a filesystem containing sensitive intellectual property – it will not provide value.

Capabilities-based security models are similar in this regard. If a runtime only allows specific operations by a workload on specific resources, then it will cut off options for attackers, who thrive off the flexibility to run whatever code they want and access whatever resources they want.

2.2.1 Design Patterns

Software systems often follow common archetypes, like "RESTful API with data stored in a relational database management system (RDBMS)" or "web frontend with user-specific data stored in database.” Instead of letting vendors struggle to create secure designs themselves, the federal government could provide a set of base requirements, design recommendations, and reference architectures for each archetype. We would suggest that an open RFI process would be appropriate here, with the government describing archetypes of interest and participants submitting secure reference designs.

Examples of base requirements could include requiring single sign on (SSO) or two-factor authentication (2FA); use of certified middleware for authenticating users; and requiring TLS for communicating with databases. As new software practices and attack strategies emerge, the federal government should update the guidelines to meet evolving conditions. Credit card issuers apply a similar strategy to great effect with the PCI-DSS suite of standards[39]: to interoperate with their transaction network, a party must follow their requirements and, in some cases, certify. Federal agencies should not accept shoddy workmanship by their vendors, and it is reasonable to set clear standards and guidelines.

2.3 Strengthening the software supply chain

We suspect that the requesting agencies are unaware that many of the concerns listed in this sub-area are addressed by common development practices in the private sector. Developers weave their workflow through code and design review, automated testing, build systems and CI/CD (discussed in Section 2.6.1) to achieve confidence in the code they ship, whether it's written by themselves, a peer, or someone they've never met. Key to these development practices are the role of package managers and the open package ecosystem. Package ecosystems are built on trust, and this is largely sufficient for

[39] https://www.pcisecuritystandards.org/standards/

the open source and private sector development community to date. There are occasional incidents, which end up in the headlines and drive fear, but the evidence suggests the resulting impact is low. Even if the impact is 10x of public visibility, it is still minimal.

As one example of how these concerns are addressed by package managers, consider the bullet in the RFI on "incorporating automated tracking and updates of complex code dependencies." This is a mostly solved problem in the private sector by package managers coordinating a software package's complete dependency graph via lock files[40]. Package managers resolve the dependencies a developer has declared the software requires into a full dependency graph representing specific versions of the software's direct and transitive dependencies. This information including exact signatures of each dependency is recorded into a lock file, where it can be used later to produce an exact replica of the build or analyze the set of dependencies in use. Lock files are named so because they "lock in" the version selections the package manager resolved. They first gained traction in the Ruby ecosystem as a mechanism to ensure reproducible builds in the presence of a dynamically evolving package ecosystem, and all major language package managers have since adopted this approach.

One major exception, of course, is C/C++, which lacks a standard language package ecosystem. In fact, we believe the lack of a standard package ecosystem is a key contributing factor to many of the C/C++ ecosystem's security troubles (alongside memory safety, which we discussed in **Section 2.1**).

We believe the requesting agencies can reinforce the importance of lock files as part of software delivery "best practices," along with automation like CI/CD. We do not recommend that the requesting agencies reinvent the proverbial wheel on this front and instead adopt this standardized solution.

In a similar vein, we are confused by the suggestion of, "Incorporating zero trust architecture into the open-source software ecosystem" and assume this is to satisfy a mandate to mention zero trust at some point in the RFI.

2.3.1 Reducing "free riding" among contractors

Another solution we believe the requesting agencies should consider is correcting the pervading free rider problem[41] engendered by federal contractors. As of now, the government purchases software from vendors while those vendors gain free benefits from OSS without contributing anything back; that is, the OSS allows them to more efficiently develop software that they then sell to the federal government.

To correct this market distortion, the requesting agencies could require federal contractors to fund the parts of the OSS ecosystem they are using in the software they sell to the government. That is, if a federal agency buys a software application *LavaLamp* from a

[40] https://semgrep.dev/blog/2022/the-best-free-open-source-supply-chain-tool-the-lockfile/
[41] https://www.britannica.com/topic/free-riding

contractor, and that contractor implements the open-source library *LavaLib* as part of *LavaLamp*, then the contractor must make a monetary contribution to the *LavaLib* project. In the private sector, GitHub makes this a relatively straightforward exercise for organizations (and individuals) by listing the developers who maintain their dependencies (based on the code in their own GitHub repositories)[42].

As an alternative, the requesting agencies could adopt a similar model to the Women-Owned Small Business (WOSB) federal contract program, but instead sourcing contracts from vendors who employ some percentage or absolute number of OSS contributors and maintainers. We suggest ensuring these contributions are in "notable" OSS repositories to avoid the perverse outcome of vendors creating open-source projects only used by themselves just to claim eligibility. The criteria for what make an OSS project "notable" should be restrictive enough to avoid gaming the system but permissive enough to cover the vast, variegated menagerie of projects that uphold the software ecosystem as we know it. We believe a wider standard is beneficial since use cases that emerge from OSS components can surprise us; for instance, the Rust library btleplug's[43] humble beginnings belied its eventual adoption in a variety of commercial products.

2.3.2 In-house development

A bold solution we believe the requesting agencies should consider is moving more software development in-house. In the 1980s, the executive branch decided that federal agencies should outsource more work, including software development, to private contractors.[44] The ramifications of this decision today are that nearly 70% of the intelligence budget is spent on contractors[45] and a full 40% of all federal discretionary spending goes to contractors, too[46]. We believe that this decision should be revised given the federal government's national security interest in software.

To be blunt: there will always be misaligned incentives between organizations that seek to optimize for national security and those with profit motives. Much of the government's concern with the software ecosystem, including open source, would be greatly relieved if they had more liberty to write their own software in-house – even down to refactoring "fundamental" software components like OpenSSL as they prefer. By writing more software in-house, federal agencies could apply their preferred practices, policies, tooling – like fuzz testing – to reach their desired safety goals. Relying on for-profit contractors who inherently have fewer resources and a different *raison dêtre* makes it nearly impossible to achieve the extremely high degree of confidence the federal government requires.

[42] https://github.com/sponsors/explore
[43] https://nonpolynomial.com/2023/10/30/how-to-beg-borrow-steal-your-way-to-a-cross-platform-bluetooth-le-library/
[44] https://www.nytimes.com/1985/03/11/us/us-pressing-plan-to-contract-work.html
[45] https://about.bgov.com/news/intelligence-contractors-vying-for-slimmer-spy-budget-in-fy-2021/
[46] https://www.gao.gov/assets/gao-17-244sp.pdf

At a minimum, we believe the government should have the right to see the source code of everything they deploy. The scale of the federal government is such as they could negotiate this, much like Medicare with prescription drug prices. It is not uncommon for software vendors in the private sector to provide their source code in a form of escrow to large customers who require review of it (or who want to mitigate the potential for abandonware). Even Microsoft offers source access to its largest customers via the Shared Source Initiative[47]. Usually, this "shared source" or "source available" offering comes with a fee, but given the government's scale, it feels reasonable to include "shared source" as part of the contract amount.

Right now, the government does not know how much of a contractor's product is open source – which they could therefore inspect – rather than original contributions by the contractor itself. An escrow process could reveal this. This is, of course, at direct odds with the "privacy-preserving" quality described in the RFI, but we believe that the government's unique national security concerns likely allow them to require review of contractor's source code before deployment.

Whether developing more code in-house or requiring source code review before deployment, we believe a downstream recommendation is to pay in-house software engineering talent more competitively. Within our own network in the software ecosystem, the only engineers who feel they can "afford" to work at a federal government agency are those who have already cultivated significant wealth in the private sector, usually from the largest Silicon Valley technology companies.

In fact, we believe – in Gordian Knot fashion – that the ability for the requesting agencies to attract and retain the necessary talent is a barrier to achieving many of the goals outlined in the RFI and elsewhere by the requesting agencies. "Can the government evaluate the security of their contractors?" No, because the government cannot recruit and retain talent that is capable of such a task. "Can the government develop its own software if it cannot rely on the security of their contractors?" No, because the government cannot recruit and retain talent that is capable of that task.

This refrain continues until the final question in this logical chain of, "What is the strategy that the federal government can execute with budget but limited staff and expertise?" We believe any answer to that question is unsatisfactory, both for the federal government and for the private sector – the latter because it makes the likelihood of misguided regulation higher in the attempt to "do something" about the national security problems the federal government faces.

Compare the salaries of a senior software engineer in the federal government to one in the private sector. A software engineer based in the Washington D.C. region with 7 years of experience is likely Grade 11 and would therefore be paid between $78k and

[47] https://www.microsoft.com/en-us/sharedsource/

$102k[48] in total compensation annually. Compare this with a software engineer with 7+ years of experience (a “senior” engineer) in the private sector, who could make $145k - 280k (median of $171k)[49] in the Washington D.C. area – with total compensation figures well above $500k at the largest technology companies for that region. In essence, to work for the federal government, this experienced engineer would need to take a pay cut of 46% at the bottom percentile to 63% or more (and as much as 80% relative to what they could earn at large technology companies).

We strongly believe that the return on investment (ROI) of making software engineering salaries more competitive, in line with median market rates, would be **significantly higher** than the status quo of outsourcing software development to federal contractors and reduce the amount of wasted resources in the process. The options, in our view are either:

1) Pay talent more money to attract (and retain) more engineers (and higher quality ones), gaining the ability to execute on the unique requirements of federal agencies; also opens the potential for the federal government to actively contribute to maintaining critical OSS projects
2) Or pay even more money to contractors who are incentivized to maximize how much revenue they earn from the federal government; create the conditions for regulatory capture and “grift”; create new regulatory hurdles that hurt private sector innovation to force alignment between private vendor incentives and the goals of federal agencies (tension that would not exist if developing software in-house)

We appreciate the significant difficulties of making headway in option one from a political perspective, but moving software development in-house clearly solves multiple woes in one fell swoop; as such, we urge the requesting agencies – and stakeholders beyond – to consider it. It is a chance to reduce financial waste while improving national security outcomes and avoiding undue interference in free market dynamics, outcomes that we hope would be considered wins regardless of one’s politics.

2.3.3 Avoiding package ecosystem balkanization

If the federal government were to demand and gain oversight of the software supply chain, it would usher in a troubling future of package ecosystem balkanization. We believe it is important to preempt and discourage this eventual conclusion, specifically to deter the creation and requirement of a government hosted package ecosystem that suppliers must use for all parts of every software system sold to the federal government. This "GovComponents" ecosystem could even require that every asset bear provenance traceable back to a U.S. citizen who has attested to its quality – which would deepen the

[48] https://www.opm.gov/policy-data-oversight/pay-leave/salaries-wages/salary-tables/23Tables/html/DCB.aspx
[49] https://www.levels.fyi/t/software-engineer/levels/senior/locations/northern-virginia-washington-dc

mistake. Such a future would further divide the software ecosystem into the general software community and the cabal of vendors selling software to the federal government.

As the center of gravity for development would remain outside this walled garden, the effect would be that government-certified components would be out of date and limited in selection. It would also encourage other governments to create their own package ecosystems under their control, which further fragments the software ecosystem and works against the spirit of cooperation in open source. One might assume that this would permit the federal government to perform more in-depth analysis and verification on the software they consume, but this is no less effective than requiring contractual access to the source code (as proposed in Section 2.3.2).

2.4 New focus area: Isolation

The current framing of the "Secure Open-Source Software Foundations" area implies that the goal outcome is to avoid the problem of exploitable vulnerabilities from existing. This is an impossible goal, as explained in our introduction. We believe there is a higher ROI from ensuring the impact of vulnerability exploitation is negligible than from attempting to eliminate vulnerabilities or prevent them from ever existing, which – we must stress – is impossible. Consequently, we believe **Isolation** is an invaluable tactic in this reframing of the goal as: how do we minimize the impact of vulnerabilities when exploited?

Our suggestion that isolation should serve as a new focus area relates to our earlier caution that memory safety is not a silver bullet; it will help software security significantly, but not solve all challenges. Vulnerabilities still exist outside of memory safety, and it is inevitable that attackers will exploit some of them. Microsoft indicated that 70% of exploited vulnerabilities in their software were related to memory safety – again highlighting the acute nature of the memory safety problem[50]. Yet, 30% remains.

We must minimize the access attackers gain should they succeed in exploiting a vulnerability. Isolation is a powerful mechanism to achieve this and is aligned with adjacent goals of Secure by Design and by Default[51]. Isolation is the practice of placing enforced limits on the interactivity between a component and the rest of the system. Successfully isolating a component requires first understanding exactly how it interacts with the rest of the system. With that understanding it can be deployed in a compartment that enforces access to only those required capabilities.

Developers can implement isolation at different layers and may employ multiple techniques to account for potential vulnerabilities in the isolation mechanisms themselves. Software isolation mechanisms include access control, process isolation, filesystem permissions, hardware virtual machines, memory protection units, containerization and namespacing, software-defined networks, library sandboxing, language virtual machines,

[50] https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/
[51] https://www.cisa.gov/resources-tools/resources/secure-by-design-and-default

service accounts, IAM roles, per-domain isolation, cache partitioning, capabilities, and more[52].

The Mozilla Foundation, in partnership with the University of California at San Diego (U.C. San Diego), described how using a novel sandboxing tool means they no longer must worry about zero-day vulnerabilities in the parts of the system it is applied to. This tool, called RLBox[53], uses WebAssembly to isolate subcomponents along functional boundaries. Faults in vulnerable components become constrained to functional boundaries rather than leading to an exposure of the host system.

RLBox also supports Mozilla's efforts to refactor their codebase into Rust[54]. Because refactoring code takes time and distracts from feature development, even for organizations with advanced software delivery practices like Mozilla, RLBox minimizes the impact of the remaining unsafe code. We believe other organizations would benefit from such an approach: introducing isolation and sandboxing either first or concurrently with iterative refactoring of their codebase into memory safe languages.

By using RLBox, it means, in Mozilla's own words, "we can treat these modules as untrusted code, and — assuming we did it right — even a zero-day vulnerability in any of them should pose no threat to Firefox."[55] We like to imagine a world in which organizations in the public and private sector alike no longer must fret about zero-day vulnerabilities posing threats to their systems. Integrating RLBox requires a strong engineering culture and commitment to security that not all organizations possess. As the technology matures – as well as other tools of a similar nature – we expect it to require less effort and specialized expertise to adopt successfully. We strongly believe the requesting agencies should prioritize isolation as a focus area and as a foundational best practice in software development and delivery to help the software achieve this vision.

In summary, we believe isolation addresses the more meta goal outcome: we do not want unintended behavior in code to generate systemic damages**.** It is worth noting that "attacks" or "attackers" is not included in that statement, because we believe it is irrelevant whether such systemic impacts arise due to attacker maleficence or performance failures. As our socioeconomic stability increasingly depends on resilient software, we do not want that stability disrupted or eroded regardless of whether it is an accident or attack.

2.4.1 Limited Resources and the Quest for Safety

An inherent tension in our recommendation is between improving isolation and improving memory safety. Software organizations have limited resources and, for those with a codebase filled with legacy memory-unsafe code, it can be difficult to know which

[52] https://dl.acm.org/doi/fullHtml/10.1145/3365199
[53] https://rlbox.dev/
[54] https://wiki.mozilla.org/Oxidation
[55] https://hacks.mozilla.org/2021/12/webassembly-and-back-again-fine-grained-sandboxing-in-firefox-95/

investments will pay the highest security dividends. Some organizations find it difficult to address both memory safety and isolation in a sustained fashion.

In some sense, we can consider the prevalence of memory unsafe code as an industry-wide crisis. Memory safety flaws are responsible for approximately 60-70% of critical vulnerabilities in popular operating systems[56]. Dealing with this crisis will require rewrites of systems over decades, given the extent of unsafe code across the entire software ecosystem. It will require mitigations, redesigns, and rewrites across an untold number of systems.

Any rewrite presents the opportunity for software maintainers to correct other flaws present in the original design and build to more modern, modular standards. This is where employing isolation becomes especially feasible: as parts of the system are peeled off and rewritten in memory safe languages, their precise access requirements become clear to maintainers, and it is much easier to adopt isolation. The newly added modularity in the system is another factor in making it easier to apply isolation boundaries. Similarly, where modular boundaries already exist in software, maintainers can adopt isolation to limit the damage of flaws in memory unsafe code more cheaply than rewriting it.

Our aim is pragmatism rather than purity; we realize that the path to designing safe systems and redesigning unsafe ones is inherently system- and context-dependent. As such, our recommendation to the requesting agencies is to treat both isolation and memory safety as tools to improve the safety of systems that will bear differing degrees of feasibility depending on each organization's context.

We stress the importance of giving designers and maintainers of systems the space to make appropriate and informed choices on how to best improve the security of the systems in their care. Regulators should be wary to mandate one architectural pattern over another and instead should aim to create that space. Incentivizing rushed rewrites and redesigns leads to worse security outcomes.

2.4.2 Exogenous Inputs, Memory Safety, and Isolation

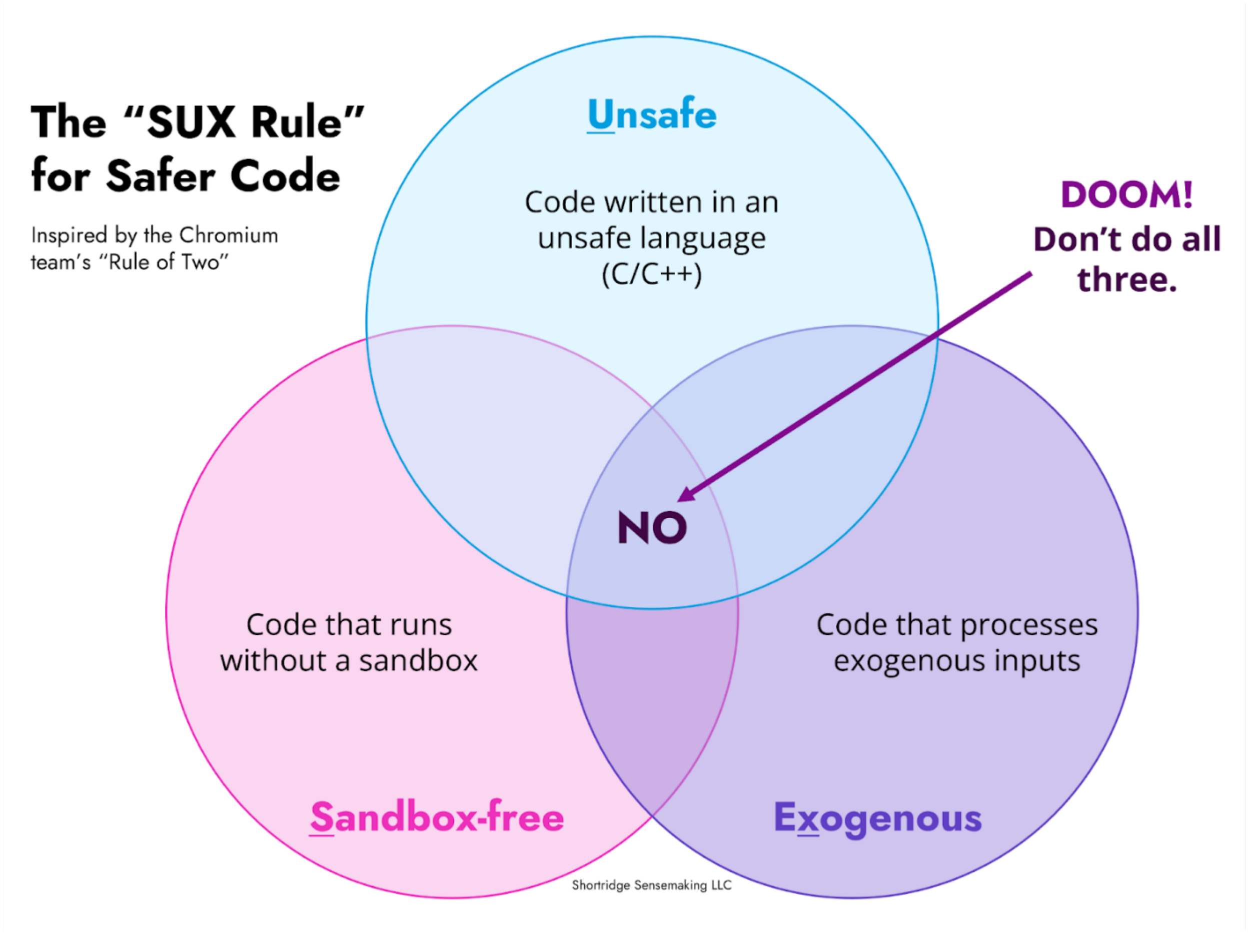
Practitioners should focus their attention on parts of the system that process untrustworthy inputs. Untrustworthy inputs are those that are exogenous to the system, such as user input. Any such code should either be written in a safe language or run inside a tightly scoped sandbox with limited access to only the data and peer components it requires.

We propose the "SUX Rule"[57] which states that high privilege components should always be *sandboxed* if they are written in an *unsafe* implementation language and process *exogenous* data. This is inspired by the guidelines the Chromium project has on how to

[56] https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/
[57] https://kellyshortridge.com/blog/posts/the-sux-rule-for-safer-code/

build new high privilege components suitable for their browser.[58] We recommend designers follow this rule for new and substantially rewritten components.

Figure 1: The "SUX Rule"


The "SUX Rule" for Safer Code
Inspired by the Chromium team's "Rule of Two"
Unsafe
Code written in an unsafe language (C/C++)
DOOM! Don't do all three.
NO
Code that runs without a sandbox
Code that processes exogenous inputs
Sandbox-free
Exogenous
Shortridge Sensemaking LLC

2.5 New focus area: Design-based software security

We believe it would be a critical mistake and miscalculation for the requesting agencies to focus on "bolt-on" solutions – which is the preference implied in the RFI – rather than design-based solutions and practices. This section enumerates design-based solutions that can support more secure software foundations.

2.5.1 Modular architectures

Returning to our north star of "reducing the systemic impact of code failures," we encourage the requesting agencies to consider the safety of architectural patterns, too.

The traditional "monolith" pattern treats an entire application as a single, tightly coupled unit; a monolithic application will unify components into a single program (deployed as a single component). Monolithic software architectures are fragile, resist

[58] https://chromium.googlesource.com/chromium/src/+/master/docs/security/rule-of-2.md

change, and are difficult for engineers to reason about; failure in one part of the system avalanches to the rest. Yet, monolithic architectures are arguably more common at "conservative" organizations – like highly regulated industries – despite their tendency to enable failure propagation.

An alternate architectural pattern is modularity – independent software components that communicate and coordinate to achieve a shared purpose (the application's functionality). More generally, modularity is a system property that allows distinct parts of the system to retain autonomy during periods of stress and allows for easier recovery from loss[59]. When failure occurs in a component within a highly modular system, it does not "infect" the other components with failure; the failure does not propagate across the system but instead stays confined to the afflicted component.

Modularity is deeply aligned with resilience because it keeps systems flexible enough to adapt in response to changes in their external environment (operating conditions). As a simple example, the human body is quite modular; if you sprain your right wrist, your other arm and legs usually still retain their typical function. From the perspective of reducing the systemic impact of code failures, we want software components to have a similar outcome: an attack on or failure in one component (like the wrist) should not disrupt or corrupt the entire system (like the human body).

Modular systems are easier to change by design. This means vulnerabilities in modular systems are easier to patch because we worry less about the side effects the patch might have on other parts of the system. It also means modular systems are easier to refactor, which supports the goal of fostering the adoption of memory safety as described in Section 2.1.

When we discover a vulnerability in one part of the system, it is easier to replace or change in a modular architecture; in a monolithic architecture, the associated feature or function must be untangled from the "big ball of mud" – the single, tightly coupled unit where all the system's concerns are combined together. Splitting a system into modules also carves a local boundary across which developers can introduce isolation (discussed in Section 2.4).

We believe the requesting agencies should encourage software engineers to adopt modular architectural patterns. To be clear, this does not mean adopting a *microservices* architectural pattern or a specific design; we encourage software leaders to make choices that are appropriate for their systems. Organizations can divide or segment a system into loosely coupled modules with well-defined boundaries without writing and deploying them as individual services; modules can be libraries, plugins, namespaces, or other units that end up in a single application.

[59] https://www.nps.gov/subjects/culturallandscapes/resilientsystems_modularity.htm

2.5.2 Greater interoperability

We believe ecosystem diversity will reduce the systemic impacts of unintended failure in software. The requesting agencies could promote ecosystem diversity by encouraging the adoption of standards with independent and interchangeable software implementations.

A larger diversity of software that implements interoperable standards at all levels of the stack would provide additional flexibility when choosing, designing, redesigning, and deploying systems. This is an extension of modular architectures, where components not only are designed to be independent of their peers, but actively swapped out for like equivalents.

Policymakers are in a unique position to require cooperation and coordination among parties that would otherwise be competitive, and this would result in interoperability and the ability for operators to substitute implementations in markets where this would not emerge naturally. This avoids dominant supplier effects, encourages new market entrants, and drives prices down for all consumers. It also grants researchers and auditors the ability to apply differential testing between independent implementations, with associated security and reliability benefits[60].

2.5.3 Elimination of the perimeter security model

To reduce systemic damages, we must eliminate the wishful thinking that is the perimeter security model (what we could also refer to as the "VPN security model"). Systems should be designed under the assumption that the internal network is compromised. Organizations with more modern security programs already design systems under this assumption in the private sector, but it is hardly a widespread philosophy. The requesting agencies can incentivize this mindset shift.

VPNs are an insecure extension of the perimeter security model that has long been proven unsafe – especially in recent years as a common vector for ransomware[61]. Organizations that want to protect internal services must assume the VPN can be compromised at any point and design accordingly. Many organizations deploy VPNs only to satisfy legacy security compliance requirements – that services should only be accessible on the internal network – since VPNs extend the definition of "internal."

In addition to the questionable model of VPNs, there is the even more questionable implementation and security practices of many VPN providers. Software and hardware VPN vendors have long been a font of vulnerabilities, including remote code execution, improper access control, and faulty cryptographic implementation[62]. We encourage the requesting agencies to recommend reducing reliance on VPNs and, where required, that

[60] https://www.cs.swarthmore.edu/~bylvisa1/cs97/f13/Papers/DifferentialTestingForSoftware.pdf
[61] https://www.bleepingcomputer.com/news/security/cisco-warns-of-vpn-zero-day-exploited-by-ransomware-gangs/
[62] https://iopscience.iop.org/article/10.1088/1742-6596/1714/1/012045/pdf

organizations closely monitor VPN traffic, maintain rigorous patching schedules, and remain poised to substitute vendors on-demand.

It is worth noting that the community-driven, high-performance software VPN Wireguard[63] cannot be FIPS certified because its cryptographic primitives are too new to meet regulatory compliance, while legacy vendor systems with storied histories of vulnerabilities and outdated cryptography are certified. This puts the federal government at a disadvantage.

2.6 New focus area: Resilient practices

To reify a resilient future, the federal government is poised to adjust its approach to preferring outcomes over processes, and to wield its mighty influence to accelerate some of the most lagging practices – the ones that moor the software community to an insecure past.

As we stressed in Section 2.1, we believe organizations should adopt memory safe languages. But as we stressed elsewhere, refactoring codebases into memory safe languages involves time, effort, money, and expertise that not all organizations possess today. When considering the realistic constraints most organizations face, rewriting legacy software into a memory safe language like Rust is much harder than adopting practices that improve the system's resilience to attacks.

This section enumerates some of these practices that can complement the adoption of memory safe languages to strengthen the security of our software foundations.

2.6.1 Continuous Integration/Continuous Deployment

Resilient practices like continuous integration / continuous delivery (CI/CD) can accelerate security changes – like patches – and reduce failure impact. In fact, practices like CI/CD are more accessible to organizations of all sizes and innovation levels than rewriting into memory safe languages. Refactoring to Rust is more of a "fancy Silicon Valley company" tactic than CI/CD, despite common protestations we have heard from some representatives of the requesting agencies.

The practice of CI/CD accelerates the delivery of software changes without compromising on system reliability or quality. A CI/CD pipeline consists of a series of automated tasks that deliver a new software release. It generally involves compiling the application (the "build" step), testing the code (the "test" step), deploying the application to a repository or staging environment, and delivering the application to production (the "delivery" or "deployment" step). Automation ensures these activities occur at regular intervals with minimal interference required by humans. The end result is that software releases are straightforward, predictable, and frequent when compared to legacy development models.

[63] https://github.com/WireGuard

CI/CD is not limited to hyper-scale tech companies and careless startups; it is compatible with and is used by some of the most highly regulated private sector companies, who automate the compliance steps through required approval steps and value the predictability automation gets them[64]. In fact, CI/CD leans into the natural human desire to reduce one's workload. If a developer is tired of tedious, manual, repetitive work, they automate it. A CI/CD pipeline means developers don't have to "babysit" deployments; delivering software becomes a process that can even be performed while the developer is on vacation.

CI/CD also gives teams more options during an emergency — the CI/CD pipeline can safely and quickly rollback the system to a safe, functioning version on demand, and it's easy to deploy additional infrastructure to account for increased demand or respond to a denial of service (DoS) attack. This allows operators and maintainers to be more aggressive at applying software patches, validating hypotheses, distributing work amongst lesser experienced team members, and responding to attacks. Gone are the days where releases need to be approved and coordinated with senior engineering staff. Even a junior engineer can perform a software release safely.

The increased predictability and release velocity of CI/CD proffers substantial resilience and security benefits. Engineers can automate "toil" work such as dependency updates, automated vulnerability checking, and record keeping. Instead of performing this toil work, the engineer now grants approvals or exceptions in a "human in the loop" model. In this model, there is always a version of the source code ready for the CI/CD pipeline to deploy; if a developer needs to make a change, they can push it to the system, which validates and applies it immediately. Frequent and predictable deployments mean engineers can deploy emergency patches or changes independent of other changes. It also means engineers can easily roll back a change should it cause outages or regressions.

Auditors are also well served by CI/CD pipelines. Every operation is recorded in CI/CD systems, including the full audit record of who performed what action when. Automation even permits annotating software builds and deployments with provenance records[65].

Simply put, if organizations can deploy software on demand, they can deploy security fixes and changes whenever they need to.

2.6.2 Automated patch cycles at all levels of the stack

We believe the requesting agencies should encourage organizations to automate and accelerate their patch cycles at all levels of their software stacks. Organizations should automate software delivery to end users as well as the stages of integration leading up to

[64] https://martinfowler.com/articles/devops-compliance.html
[65] https://grepory.substack.com/p/der-softwareherkunft-software-provenance

deployment and delivery. The requesting agencies could even insist their contractors adopt CI/CD (described in Section 2.6.1) to enable quicker patch cycles[66].

Patching *agility* should be the goal, not "vulnerability management"[67]. Organizations should implement automation to ensure that when there is an important update, relevant systems pick it up without a human having to access those machines and perform manual work.

Vendors should integrate software fixes for their dependencies by continually updating to the latest versions and should work with their suppliers to reduce integration times. It is reasonable for agencies to set delivery deadlines for fixes to publicly available vulnerabilities, with financial compensation as penalty; we will discuss this more in **Section 4.2**.

Long patch cycles lead to dangerous gaps: the software community will understand the vulnerability; attackers will exploit it in the wild; but active systems in use by the federal government still await a fix deployed by contractors. A similar dynamic plays out in private sector systems with their vendors, although an organization's own heavy change processes can also delay their ability to patch. To determine how out of date a system is, organizations should use the date a flaw was discovered rather than the announced patch dates by their suppliers. We believe this would encourage vendors to integrate fixes from vulnerable dependencies more quickly.

For complex dependency trees, integrators should reserve the right and exercise the technical capability to substitute transitive software dependencies that their suppliers have provided. Many software ecosystems operate on this model now (see our description of lock files in **Section 2.3**). This avoids the scenario where vulnerable components deep into a supply chain take a long time to be integrated as each vendor in the chain must perform their own quality assurance and certification.

There is a particular anti-pattern the requesting agencies should discourage: "shading."[68] If a library vendor embeds or "shades" the library's dependencies before distribution to a software integrator, the integrator is now unable to substitute those dependencies with updated versions. To fix security vulnerabilities in transitive dependencies, software integrators must therefore wait for the vendors of included libraries to provide a fix. We recommend against this pattern and believe it represents poor dependency hygiene, even if it is simpler for integrators in the short-term (since it means contracts close more quickly).

Instead, we believe library vendors should declare which dependency versions the library supports and let integrators select appropriate versions. Many package

[66] https://www.hashicorp.com/resources/redeploying-stateless-systems-in-lieu-of-patching-petco-packer-terraform
[67] https://mumble.org.uk/blog/2022/02/14/vulnz-being-up-to-date-is-not-the-goal/
[68] https://jlbp.dev/JLBP-18

management ecosystems provide mechanisms to support this automatically for both closed and open-source software. In fact, it is OSS – and the packaging ecosystem around it – that moved software away from the norm of shading. With this pattern, integrators of the end-use software system can patch vulnerable libraries deep into their system's dependency tree.

Operators of critical systems may even choose to take on the integration capability themselves, offering them the greatest ability to respond to flaws. This model is not without its downsides since it requires a level of care and attention beyond deploying a vendor-provided amalgamated software bundle. By taking this approach, operators are free to apply their preferred security practices – such as generation of SBOMs – without forcing their practices on the rest of the ecosystem.

As an example, consider Log4Shell from the perspective of a federal agency. Some of the systems contractors build for them contain components written in Java. Some of those Java components are licensed from suppliers, rather than written by the contractor themselves. Many of those Java components will use the Log4j logging library, and some may use libraries that depend on Log4j. The result is that a vulnerable Log4j library may lurk in numerous software systems at multiple levels.

2.6.3 The D.I.E. triad

The D.I.E. triad – distributed, immutable, and ephemeral – reflects system properties that offer security by design and are powerful techniques to facilitate adaptation:

- Distributed infrastructure involves physically disparate components that coordinate to perform work. For example, content delivery networks (CDNs) operate points of presence (POPs) around the world, and these distributed networks are consequently more resilient to DDoS attacks.
- Immutable infrastructure means that the software does not change after it’s deployed; when an operator or developer wants to deploy an upgrade, they will instantiate new infrastructure running the upgraded version rather than modify the existing infrastructure. Immutable infrastructure means operators can vastly simplify the operations allowed on their systems, blocking activities like shell access from developers and attackers alike, but it also facilitates automated change processes that result in fewer mistakes, such as misconfigurations.
- Ephemeral infrastructure has a shortened lifespan, living only for the duration of a task. Such infrastructure is easier to “kill” and restart by design; frequent changes are baked into the assumptions of using it. From an attack perspective, ephemeral infrastructure makes it difficult for attackers to persist on a system without re-compromising it each time or “escaping” it for deeper access.

Combined, these system properties can help software systems stay flexible and ready to adapt to evolving conditions. While these properties require upfront investment, they make it much easier to operate complex systems that are resilient to change – with the additional benefit of achieving higher performance. They also make it harder for attackers to conduct automated attacks against the system, or weaponize exploits at scale. We believe the requesting agencies could encourage D.I.E. as a design pattern.

2.6.4 Resilience stress testing

If contractors make claims about security or reliability properties of their product, federal agencies could require evidence generated by resilience stress tests to verify these claims. Resilience stress tests can elucidate where an organization's mental model of the system deviates from its reality. Federal agencies could give contractors a list of failure scenarios to conduct in their software or require those experiments to run continuously as ongoing assurance. For example, if contractors supply software that includes a login page, federal customers could require the contractor verify that all pages including sensitive data issue appropriate login challenges when accessed directly by unauthenticated traffic.

The evidence generated by these experiments should include a description of end-to-end system behavior. If the resilience stress test injects a malicious build into the contractor's software delivery pipeline, is it blocked by any tests or checks? Do developers approve the request? If a tool generates an alert about it, do human operators notice the alert? If so, is there enough context for them to take action? Such details could give federal agencies greater confidence in the security properties of the software supplied by vendors.

This is not unlike the resilience stress tests the Federal Reserve conducts to evaluate the systemic resilience of the financial system. We believe this is an area well worth the requesting agencies' focus to uncover its potential for minimizing systemic impact.

2.6.5 Vendor-managed deployments

We believe the requesting agencies should usher a migration away from client-managed deployments towards vendor-managed deployments. Customers who manage many systems are prone to mismanaging deployments of vendor software. When customers misunderstand how the software works and cannot devote sufficient attention to it, they deploy misconfigured or stale versions that foment conditions for failure.

This guidance may seem at odds with our recommendation in Section 2.6.2, but there is a time and place for each strategy. Vendor managed deployments are safer when the software is standard and deployed to many customers; customer-managed deployments are safer when the customer is receiving bespoke software or has unique operational concerns.

Some vendors try to encourage their customers to patch more frequently by providing a regular schedule and clear guidance on the impact patches may have on the system. However, a more effective pattern is for the vendor to manage the deployment and maintain responsibility for keeping it patched, either through SaaS or a vendor-managed on-premises deployment.

Vendors can amortize the cost of patching across their entire customer base, can deploy patches much more safely due to their intimate understanding of the system, and can even patch some flaws before public knowledge of them is disseminated. This leads to more reliable and secure systems with lower maintenance cost.

2.7.6 Rate limiting

We believe the requesting agencies should encourage organizations to enforce rate, location, and other limits on human-operated user accounts. Incident evidence shows that most compromises involve attackers hijacking accounts[69] – either with leaked credentials or by phishing those credentials – to use the human's access to perform aberrant operations.

To sharply curtail damage from compromised accounts, organizations can attach limits on the volume or throughput of operations a human-owned account can perform over a specified time period; doing so only inconveniences one person when their account is locked. This can stop many attacks in their tracks, including those which gained access by exploiting a vulnerability. For instance, the 2022 Uber breach involved an attacker "spamming" employees with multi-factor authentication (MFA) requests; rate limiting would restrict the number of MFA requests a single user account could make, eliminating this potential path in for attackers. If rate limiting becomes a standard requirement as part of system or product design, similar attack paths may crumble.

Similarly, organizations can bind access tokens or login cookies to the network address that requested them[70] so attackers cannot use exfiltrated or intercepted login cookies. This tactic causes only mild inconvenience to users who must log into their accounts again whenever they switch networks.

2.7 Disincentivize known-unsafe architectures and patterns

We believe the requesting agencies should incentivize the reduction of known-unsafe architectures and patterns that can contribute to systemic damages from software exploitation. By way of analogy, older dams are often burdened by lower quality designs, which make them more prone to catastrophic failures[71][72]. But newer dams are higher quality and safer in large part because they leverage better design principles (and a better understanding of them)[73]. While dam failures represent a lethal force, unlike software, the drastic improvement in dam safety over the past few decades emphasizes the importance of disincentivizing known-unsafe design patterns.

Many software patterns are well-known to be actively unsafe, yet there is little incentive for engineers to disrupt the status quo when creating new software systems.

[69] https://www.verizon.com/business/en-gb/resources/2023-data-breach-investigations-report-dbir.pdf
[70] https://learn.microsoft.com/en-us/power-platform-release-plan/2022wave2/data-platform/stop-cookie-replay-attacks-ip-binding
[71] https://phys.org/news/2017-09-experts-bad-dangers-tallest.html
[72] https://www.npr.org/2022/05/05/1096940224/dams-poor-condition-hazardous-dangerous-infrastructure
[73] https://www.fema.gov/sites/default/files/2020-08/fema_dam-safety_P-93.pdf

These unsafe patterns are usually more convenient, and their downsides are often paid by someone else – or not at all, if engineers are lucky. We suspect some federal contractors find it convenient to use these patterns as a way to “cut corners” (and therefore maximize profits), while others may simply be unaware that they are unsafe due to being largely divorced from software innovation.

Unsafe patterns include, but are not limited to: memory unsafe languages, monolithic architectures, stored/encrypted passwords, manual testing, custom tooling, coarse-grained authorization, custom authentication, bolt-on security controls, object graph pickling, infinite-duration access tokens, handwritten format parsers, and custom communication protocols.

2.8 Areas to deprioritize

Knowing what to deprioritize can be as valuable as knowing what to prioritize. This section reflects our recommendations for what should *not* be included in the requesting agencies’ focus.

2.8.1 SBOMs

We do not believe software bills of materials (SBOMs) will support the requesting agencies’ stated goals. They may remunerate consultants and vendors who can help organizations navigate this new requirement, but they are not actionable and are divorced from tangible security outcomes. We strongly suggest the requesting agencies reduce their focus and evangelism of SBOMs in favor of considering more actionable and impactful measures (as described throughout this document).

SBOMs, at their best, generate large numbers of JSON blobs that enumerate all the components within an application. While it seeks to answer, “What are the security properties of my software?” it cannot, because it relies on software composition analysis of what is “inside” the software. If we receive a long list of components in an airplane, do we feel safe enough to fly in it? No. Like any complex systems, the resilience and security of software systems depends on how components *interact*.

SBOMs would not have helped SolarWinds at all, nor Colonial Pipeline, nor the Microsoft Exchange Server compromises. It is debatable if it would even help with Log4Shell. An SBOM does not reveal whether the parts listed within the long JSON blob are reachable by the internet, are configured in the precise way attackers need to exploit it, nor other context necessary to determine the security implications of a software issue. The profuse information it produces is unactionable.

SBOMs fit the danger we described in the introduction: regulatory requirements may hurt the benefits software begets the United States far more than they reduce the already low impact of software exploitation. As a thought experiment, would we have had the iPhone if we had SBOMs for the past 30 years? Similarly, would we have the cloud if we required every programmer be licensed by a certification board? Are we willing to forgo future innovations like these – and their associated benefits to the nation's economy and global standing – for a hypothetical "Cybergeddon" resulting from vulnerability exploitation?

This is not to say we do nothing; the numerous recommendations throughout our response reflect many things we could do to improve the resilience of the software ecosystem. But we believe it means requesting agencies must very carefully consider the second order effects of their proposals. Just as the requesting agencies lament the unintended consequences of developers' code, they must scrutinize their own proposals and recommendations for unintended consequences.

We believe SBOMs – and the fervor for it emanating from the federal government – is a palpable case of myopic thinking that should be forsaken if the federal government seeks to maintain credibility on software security.

3. Sustaining Open-Source Software Communities and Governance

3.1 Abandonware

We believe the requesting agencies should include abandonware as a focus area. Actively maintained projects – both open and closed source – can update code to fix vulnerabilities and other issues. Abandoned projects cannot do so. We encourage the requesting agencies to carefully consider the special case where software is abandoned by its maintainers or is nominally maintained.

Most open-source software gets abandoned[74]. Whether from life's inevitable vagaries and vicissitudes or changes in corporate strategy, maintainers abandon their OSS projects. But this abandonment may not percolate into an organization's awareness.

Software composition analysis tools will inform organizations that they are on the latest version – but the latest version may be a decade old. And such tools may indicate that there are no vulnerabilities in this component – but that is because the vulnerability database is no longer maintained. The abandoned component is festering in the system and the system is maintained by people who don't know how that component works; their job is to use the component towards some end.

We would also suggest expanding the definition of abandonware to include software that has a maintainer, but that maintainer cannot sustain the level of investment necessary to keep a library secure. Projects like these will often merge external contributions or have a trickle of commits but will seldom make releases and security issues will go unaddressed. These projects could be described as rotting.

As stated elsewhere, we believe providing financial assistance to open-source maintainers is the best means to solve this problem.

[74] https://thenewstack.io/what-happens-when-developers-leave-their-open-source-projects/

3.2 OSS Governance

We believe OSS governance is not the problem and adding yet another stakeholder into the mix will not help. OSS communities self-assemble in whatever structures fit them best, usually a mix of passion-motivated individuals and engineers employed by organizations with commercial interests in the project.

4. Behavioral and Economic Incentives to Secure the Open-Source Software ecosystem

While we wove behavioral and economic incentives into our recommendations throughout our response, this section covers a few other incentives and considerations we believe are relevant. In general, we encourage the requesting agencies to focus on strategies that work with human behavior, rather than against it (such as making the secure way the easy way).

4.1 Frameworks and models for software developer compensation that incentivize secure software development practices

Software engineers do not go out of their way to be insecure. That is never their goal (except in extremely rare cases of espionage). It is a matter of priority and resourcing: do they have the time and effort to expend on security? Suggesting that engineers should spend more time making their software secure or reducing the number of bugs is an obvious one but does not change the reality. They are already doing it to the extent they can within their current incentives and constraint structure.

We believe the government could decide that certain software projects are of strategic value and hire maintainers on their payroll. This could involve hiring people who already maintain a specific OSS project or others who possess the skills and interest to do so. Google and Red Hat already use a similar strategy in the private sector to achieve their reliability and security goals – as well as to shape the open source ecosystem to suit their needs.

4.2 Software liability

We believe software liability is perhaps the swiftest way to kill the OSS ecosystem in the United States. We do not think the government needs to invoke this show of force to meet its goals. The nexus of OSS could and would flee the United States if the federal government made OSS maintainers liable for bugs in their projects.

However, we believe it is more reasonable for the federal government to penalize contractors and other providers of critical infrastructure who do not sufficiently investigate, test, and mitigate the OSS components they use. In effect, the federal government could enforce the mantra, “you own your dependencies.” This penalty may induce a second-order effect of limiting the population of federal contractors, but we feel it will incentivize contractors to avoid incorporating OSS components without understanding them first (including the security implications of those components).

The requesting agencies could also apply this approach to patching standards. Similar to service-level agreements (SLAs) in the private sector, the federal government could insist that contractors patch *any* critical vulnerability present in their software within some period of time (like 15 days), requiring them to write their own patches for OSS components if necessary. Contractors would become the effective maintainers of these projects on behalf of the federal government.

A tight timeline of 15 days, or perhaps sooner[75], also incentivizes contractors to adopt safer software delivery practices, like automation (as described in Section 2.6). To meet that timeline, contractors must understand all their dependencies; monitor the public feeds for vulnerabilities in them; know when there is a vulnerability they must patch; safely incorporate that patch into the larger system and validate it; construct a release for that system; and deploy it to the federal government.

We believe this is a reasonable expectation in return for receiving federal funds. It is unacceptable for providers of these systems to blame the open-source community while simultaneously extracting value from their donated efforts.

We expect this approach to flatten the dependency graph over time. Imagine a custom system built by a federal contractor in their language of choice. This system will include their custom code (perhaps written in Java), some open-source libraries it depends on (potentially Apache Commons), some commercially licensed components (each with their own open source library dependencies, perhaps Log4J), and a base operating system for the program to run on (perhaps Red Hat Enterprise Linux). All of this is packaged and distributed to the federal government to be operationalized. Patching could be required in any part of the assembled system. Meeting the timeline for patching becomes more difficult the more parties you have involved. It becomes the contractor’s responsibility to clean up this supply chain rather than the federal government's.

Making it the contractor's responsibility will encourage their agility to patch, leading to either closer relationships with their suppliers or new operating models with respect to dependency management. It is a common practice for commercial component vendors to bundle transitive dependencies, which makes updating them more difficult. This harmful practice would be discouraged by short patch timelines and would incentivize them to behave more like the OSS package ecosystem.

To the extent the software security problem is open source, it is in the commercial repackaging of open-source software. Ultimately, liability should lie with the providers of a service; it would be unreasonable to expose the OSS community to liability on transactions

[75] We suspect the requesting agencies have data on how long it takes for an attacker to write a successful exploit after a vulnerability is discovered (whether through private research or public disclosure). We would love if the requesting agencies shared that data publicly but understand why they may not. Regardless, this data can inform the appropriate deadline as described in our recommendation.

to which they are not party – and would likely stifle technology innovation the United States.

4.3 Regulatory Incentives

Regulations compound the software security challenge, especially in conjunction with outsourcing by the federal government. Many regulations set vague quality requirements, often without minimums, that the private sector tries to meet in the quickest and cheapest possible way. This thriftiness often takes the form of assembling OSS in a haphazard fashion and building a management portal on top of it. Regulations therefore do not reward vendors with the highest quality software, but instead those who can pirouette through all the required hoops. Once those vendors are implemented in customers' stacks – and therefore difficult to replace – there is minimal incentive for them to improve quality or innovate.

We believe the federal government is a contributing factor to this problematic dynamic. It often promotes older standards over more modern approaches because the modern approaches have not been pushed through the appropriate approval processes – or do not have lobbyists advocating for them. A network security tooling vendor will lobby for "zero trust" being a requirement, as will an application security vendor lobby on behalf of software composition analysis. An architectural pattern like modularity, while enormously beneficial for software resilience and security by design, has no such deep-pocketed propagandist in its corner.

In the private sector, it is well known that you must often "fight" your auditor after you adopt more secure designs or safer practices, as they often no longer fit neatly into the regulatory checklist. For example, disabling SSH access to production servers is often considered a best practice for both security and performance reasons; many attackers rely on SSH access to conduct their operations, so disabling it cuts off this common attack path. Yet, auditors will insist that the organization must still allow SSH access because a compliance requirement defined decades ago requires the ability to SSH into the machine. But, from the perspective of Secure by Design and Default, no one should be able to access the machine in this manner.

The important question for organizations becomes: should our cybersecurity strategy align to the compliance checkboxes or aim to achieve more secure outcomes, even if it defies what auditors expect? Too often, organizations must, by necessity, craft their strategies for compliance and sacrifice their ability to harness security innovation. And even if organizations are 100% compliant, they are still insecure[76].

As the requesting agencies identify and prioritize focus areas, we encourage them to think beyond compliance checklists that inevitably calcify and impede improved security.

[76] https://josiahdykstra.com/wp-content/uploads/2020/02/NDSS2020_Compliance_Cautions.pdf

Our goal is to minimize the systemic impact of unintended behavior in software, not remunerate the for-profit tools that help organizations check compliance boxes.

4.3.1 Federal Information Processing Standards and Modernity

Encryption standards can be especially stale. Federal Information Processing Standard (FIPS) Publication 140-certified encryption libraries use ciphers that are inferior to those used by uncertified software. The encryption community generally reaches consensus around best practices and standards well ahead of the ratification of federal information processing standards.

From the perspective of macro-level socio-economic harm, it is dangerous to implement systems that enter the market with their cryptographic components already outdated and unlikely to be updated until much later – only once updated FIPS regulation prohibits that encryption standard.

Modular architectures, where customers can choose, deploy, operate, and later substitute the cryptographic components independently of the vendor would make it easier for vendors to support cryptography in their products and would make it easier for customers to switch to stronger cryptography when it becomes available.

5. R&D/Innovation

5.1. AI and Machine Learning

Regarding the potential for AI and machine learning techniques to solve secure software development and delivery, we believe matrix multiplication at a grand scale will not help.

5.2 Other innovation

We believe an important, but overlooked, area for innovation is in making it easier to substitute and isolate software components. We believe the government should fund more research into “swappability” and isolation – and encourage the adoption of each.

Applying standard isolation techniques to existing complex software systems, where it is most useful, is largely a difficult and cumbersome affair. It is much easier to apply isolation as a system is being built, but often there is less pressure to do so as new, yet-to-be-deployed systems by definition have no active users. There has been only mild extrinsic incentive for anyone to make isolation easier and simpler to apply to complex systems and breaking them down into individual isolatable units requires deep system's context. We welcome study into ways to improve the developer experience of isolating existing systems and new mechanisms to do so — indeed, our area of research includes this.

Substituting alternative libraries and components is more cumbersome and difficult than we believe it has to be. The software community has had relative success standardizing low level data formats and protocols so that multiple systems can interoperate on the same data. It has had less success making the software components themselves swappable. Even when two libraries perform the exact same function, they usually interact with the rest of the system in ways that are mechanically different. Engineers refer to this as libraries' application programming interfaces (APIs) being incompatible. Commercial software vendors have a disincentive to produce interoperable APIs except in rare circumstances. We welcome research into incentives for increasing swappability of APIs, mechanisms for abstracting over functionally equivalent APIs, and other techniques for automated swappability of components.

Conclusion

For the reasons cited herein, we encourage ONCD, CISA, NSF, DARPA, and OMB to incorporate our recommendations as they identify and prioritize focus areas for improving open-source security. We believe these recommendations can nourish a future where all stakeholders are confident in the resilience of our critical systems to software failures – that we will not crumple from catastrophe when unintended behavior unfolds.

We urge the requesting agencies to respect the power of OSS as an innovation engine that propels our national economy – indeed, that touches nearly all sectors today. Obstructing and stalling that engine through ill-informed regulatory requirements would sabotage this socioeconomic velocity and trammel the nation’s business ecosystem.

Sincerely,

Kelly Shortridge

Kelly Shortridge
Founder
Shortridge Sensemaking LLC

Appendix

About the Responders

We, Kelly Shortridge and Ryan Petrich, are recognized experts in cybersecurity and software engineering as well as frequent collaborators on open-source projects, including Deciduous[77] and Patrolaroid[78]. We have included our biographies below to highlight our expertise in the areas covered above. Again, the views expressed herein are not necessarily the views of our employers or any of their affiliates.

Kelly Shortridge is a Senior Principal in the Office of the CTO at Fastly, a cloud computing company. Shortridge is lead author of *Security Chaos Engineering: Sustaining Resilience in Software and Systems* (O'Reilly Media) and is best known as an expert on resilience in complex software systems, the application of behavioral economics to cybersecurity, and modern cybersecurity strategy. Shortridge frequently advises Fortune 500s, investors, startups, and federal agencies and has spoken at major technology conferences internationally, including Black Hat, RSA Conference, and SREcon. Shortridge's research has been featured in scholarly publications such as *ACM*, *IEEE*, and *USENIX* as well as top media outlets including *BBC News*, *CNN,* and T*he Wall Street Journal*. Shortridge also serves on the editorial board of *ACM Queue,* a bimonthly computer magazine founded and published by the Association for Computing Machinery (ACM), the world's largest learned society for computing.

Ryan Petrich is a Senior Vice President at Two Sigma Investments with over two decades of involvement in the open-source software, software security, and software quality communities. Previously, he served as Chief Technology Officer at Capsule8, a cybersecurity provider of enterprise detection and response software for Linux after leading engineering teams in advertising technology. Petrich is also known for his contributions to open-source projects as well as maintaining foundational libraries at the core of the jailbreaking ecosystem. As part of his leadership in the jailbreaking community, he provided aftermarket patches for iOS to fix security vulnerabilities for users before the vendor was able. Petrich's research extends into software security via Callander[79], a sandboxing system used to apply tightly scoped policies to software automatically. His work is published in *ACM Queue* and *Communications of the ACM*. He regularly speaks at software reliability and security conferences, including previously at All Day DevOps, Cloud Native Wasm Day, and JailbreakCon.

[77] https://www.deciduous.app/

[78] https://github.com/rpetrich/patrolaroid

[79] https://github.com/rpetrich/callander