SHORTRIDGE SENSEMAKING LLC

February 20, 2024

Jen Easterly, Director
Cybersecurity & Infrastructure Security Agency
110 N. Glebe Road
Arlington VA 20598-0630

RE: Doc. No. CISA-2023-0027; Request for Information on "Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Secure by Design Software"

Dear Director Easterly,

As leaders in cybersecurity and software engineering, we appreciate the opportunity to submit this statement in response to the requests for comment by the Cybersecurity Infrastructure Security Agency (CISA) concerning their white paper "Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Secure by Design Software."

Our adversaries possess a critical advantage in cyberspace: the ability to adapt with speed. If we are to defend against them, we must capture this advantage for ourselves. We believe the Secure by Design approach, done well, grants us this capability.

Yet, if what we require of software manufacturers slows them down, then we impede their capability to adapt to evolving conditions (including attacks). We fear many of the recommendations in the whitepaper create such friction. Instead, we believe Secure by Design can align with business priorities like software velocity, developer productivity, and reliability in production. Our response enumerates principles and practices towards this noble goal.

Like CISA, we also dream of "a future where technology is safe, secure, and resilient by design and default" (page 5). But we worry the whitepaper misleads the software community towards a future in which we are slower than ever than attackers.

Our goal in this response is to describe the priorities, investments, and decisions that stand a chance of achieving that future where software is safe, secure, and resilient by design and default. We hope our recommendations shepherd and champion the software community in this direction.

Our response begins with overarching commentary on the whitepaper, followed by addressing the sections within the request for information (RFI) where our expertise is relevant.

The views expressed herein are not necessarily the views of our employers or any of their affiliates. The information contained herein is not intended to provide, and should not be relied upon for, investment advice.

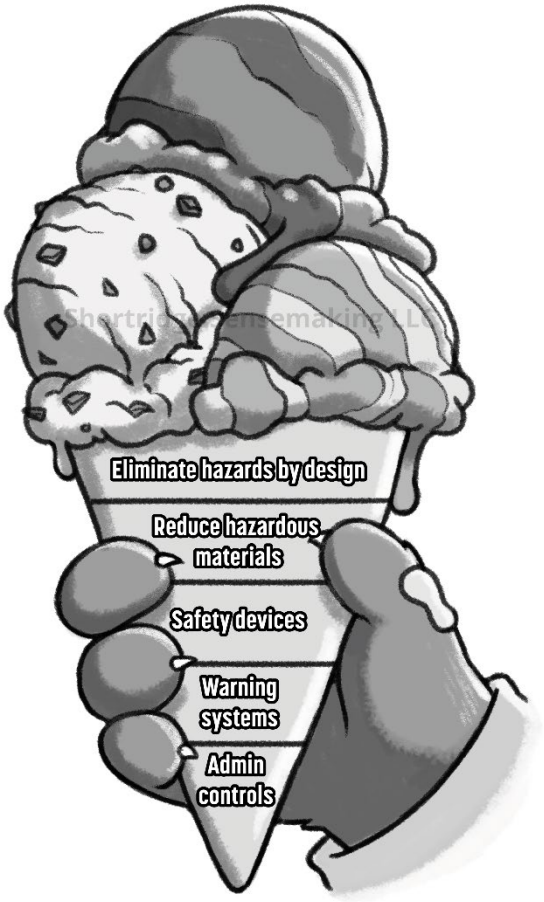
1. Commentary on the Whitepaper

1.1 What does Secure by Design mean?

We disagree with how the requesting agencies define "secure by design," as: "Products that are secure by design are those where the security of the customers is a core business goal, not a technical feature."

Whether Secure by Design is a core business goal is far less relevant than whether the software manufacturer achieves the goal outcome. Indeed, there is a troubling undercurrent of resentment throughout the whitepaper that security does not matter more to software providers. Our goal should not be to force them to care, but instead to ensure that the software they develop and deliver minimizes the impact of attacks by design.

Figure 1: Shortridge's Ice Cream Cone Hierarchy of Security Solutions



We believe Secure by Design means software that does not require human intervention to sustain security (or resilience, more generally). This definition draws on safety wisdom from other domains[1], as does the Ice Cream Cone Hierarchy of Security Solutions created by Shortridge (Figure 1). We specifically believe Secure by Design reflects the first two elements: system design or redesign to eliminate hazards or substitute less hazardous methods and materials.

The Ice Cream Cone Hierarchy of Security Solutions[2] visualizes how we should prioritize resilience and security mitigations. The top two categories of solutions count as "Secure by Design." The rest overly rely on human behavior to succeed. A handy heuristic is that the less the solution relies on human behavior to succeed, the better it is; if it is entirely

---

[1] https://aeasseincludes.assp.org/professionalsafety/pastissues/050/05/030505as.pdf

[2] From *Security Chaos Engineering: Sustaining Resilience in Software and Systems* by Kelly Shortridge, and illustrated by Savannah Glitschka

dependent on human behavior to succeed (like training or policies), it is our least preferred option.

Measures that rely on human behavior – from warning systems down to control measures, like training – are inferior to solutions that eliminate hazards. Once we get closer to the bottom of the cone, we cannot scoop as much resilience ice cream into it. In between these ends are solutions that somewhat rely on human behavior, like any safety devices that can be forgotten or bypassed; these are among the "bolt-ons" referred to in the whitepaper (although we believe warning systems also count as "bolt-ons"). Relying on human behavior makes security unreliable for a few reasons: our cognitive resources are finite, we face competing pressures, we can be tired, stressed, distracted. We also believe that most humans have better things to be doing with their time and energy than expending effort on security.

What does it mean to either eliminate or reduce hazards – and what are hazardous methods and materials? Hazards are the potential for harm. Hazards include the characteristics of technology (things) and the actions or inactions of people (activities) that can produce harm[3].

1.1.1 Hazardous methods

Hazardous methods relate to the common folk wisdom of, "Don't roll your own crypto(graphy)." Generalizing this advice is what we mean by substituting less hazardous methods: companies should not roll their own database, logging pipeline, observability, nor other middleware. Hazardous methods manifest as injection from an attack perspective; SQL injection (SQLi), for instance, can be characterized as the result of rolling your own database query builder.

When organizations do not need to differentiate at a software level, they should standardize and "choose boring." If a software manufacturer's competitive advantage is in a particular facet of infrastructure or software, then they should invest more effort in those areas and potentially "roll their own," ideally considering design-based mitigations to potential hazards.

For example, if an ad tech company's differentiator is operating at scale, then it is worth the resource investment for engineers to figure out the multi-region, distributed nature of their systems. Or if the business is constructing user profiles and refining them into usable segments, it is worth the resource investment to build and maintain a reliable machine learning system that can perform that function successfully.

As part of substituting less hazardous methods, we should develop "paved roads" for easier changes, whether faster deployment of resilience and security-related configurations or faster patching.

---

[3] https://aeasseincludes.assp.org/professionalsafety/pastissues/050/05/030505as.pdf

1.1.2 Hazardous materials

We characterize “raw materials” in software as languages, libraries, and tooling (this applies to firmware and other raw materials that go into computer hardware, like CPUs and GPUs, too). These raw materials are elements woven into the software that need to be resilient and safe for system operation.

When building software services, all organizations must be purposeful with what languages, libraries, frameworks, services, and data sources they choose since the service will inherit some of the properties of these raw materials. Many of these materials may have hazardous properties that are unsuitable for building a system as per your requirements. Or the hazard might be expected and, since there isn’t a better alternative for your problem domain, we must learn to live with it or think of other ways to reduce hazards by design. Generally, choosing more than one raw material in any category means we receive the downsides of both.

Hazardous materials include unsafe programming languages. Substituting C or C++ code for a memory safe language (of which there are many) reduces memory management hazards. That is not to say other languages are immune to safety problems that haunt us in other ways; but given 70% of vulnerabilities relate to memory safety, adopting memory safe languages is one of the most impactful substitutions of hazardous materials organizations can make.

In general, any technology – whether libraries, code snippets, frameworks, and so on – that is harder to understand is more hazardous. “Boring” technology that is easier to mentally model, is better documented, and offers a supporting community is a less hazardous substitute.

Sensitive data can also be thought of as a hazardous raw material – at least when there’s a breach of customer’s payment data. Rather than requiring a billion hours of security training, we can propose breaking apart an application into smaller services with isolated access to data.

The billing service will have access to payment data, as it must, but now the rest of the application—all the other services that make up its functionality—will not have access to that data. Or, we could outsource payment handling to a third party.

When we hear something failed due to “user error” or a problem persists because “humans keep doing X wrong,” we can recharacterize it through the Ice Cream Cone Hierarchy of Security Solutions. If the “error” is due to human perception, it is a red flag that a “poorly-designed system, product, or environment”[4] is adulterating human interactions in our system. It reflects a call to action to brainstorm better solutions and prioritize those at the top of the cone.

---

[4] https://www.visualexpert.com/why.html

1.1.3 Eliminating and reducing hazards by design

Design-based solutions that eliminate hazards feature two key traits:

- They do not depend on human behavior.
- They provide complete separation of the user from the hazard.

Both features engender more reliable success outcomes, without forcing senior leadership to magically care about security or product teams having to invest most of their effort on "radical transparency."

As one example of system design to eliminate hazards, consider a software manufacturer that is reeling from a breach of customers' payment data. Someone proposes another 20 hours of "secure development" training so software engineers "stop writing exploitable bugs." Of course, such training will never work, because mistakes are inevitable – and also, if we believe that "by design" beats reactive manual efforts, then training should not be treated as a serious option.

Instead, the software manufacturer could break apart their application's monolith into smaller services with isolated access to data. The billing service will have access to payment data, as it must, but now the rest of the application – all the other services that make up its functionality – will not have access to that data. If, for example, their order volume is proprietary, then this also gives them the benefit of being able to partition and slice up data to keep it private (as a form of classification or compartmentalization).

An alternative the software manufacturer could pursue is outsourcing payment processing to a third party; that way, they eliminate the hazard by design by not even storing or handling payment data in their systems. Many Software Engineering teams already do this for tricky engineering problems, like content delivery (to handle scale, regionality, and caching) or transaction processing, which will let your database handle the hazard of concurrent operations on the same data instead.

Another approach is to avoid collecting hazardous data that is not required to satisfy the system's function. Subscription billing need not require storing credit data for the duration of the subscription. One could set up a recurring payment authorization with the issuer and store this information instead. Subscriptions can still be renewed and canceled with this information, but attackers would be unable to issue new transactions.

Similarly, marketing-technology and advertising-technology companies will avoid storing email addresses and phone numbers, instead preferring to work with opaque cookie or consumer identifiers. They may choose this to eliminate the hazard entirely throughout the system or translate to and from opaque identifiers at the edges of the system to avoid the hazard throughout most of it. We hope CISA pays particular attention to this example, as it illustrates how companies can achieve secure by design outcomes *because* of how little they care – or want to care – about security.

Another example of eliminating hazards by design is isolation, as it usually does not rely on human behavior and can completely separate the user from the hazard (including machine users, like other services interacting with another service). Table 1, developed by Shortridge, lists opportunities for either eliminating hazards by design or substituting less hazardous methods or materials by design.

Table 1 - Potential design-based solutions to sustain resilience

Goal outcome	Potential mechanisms
Can absorb the impact of surprises with spare capacity	Buffers, failover, autoscaling, queueing, backpressure, job systems, background/batch processing
Independent, or flexible dependencies	Isolation, declarative dependencies (like Infrastructure as Code)
Easier to untangle interactions after incidents	Isolation, standardization, design documentation
Easier to debug and troubleshoot at runtime	Standardization, iterative design, break-glass debugging mechanism with audit
Cost-effective over the long term	Standardization
Issues arising from interactions are more visible	Failover, isolation
Supports reusability	Standardization, design documentation, libraries, component model, specifications
New changes can be implemented independently	Isolation, iterative design, design documentation
Adding new components increases system resilience	Failover, buffers
Processing delays are tolerable	Isolation, failover, autoscaling, queues, job systems, background/batch processing, asynchronous processing
Order of sequences can be changed	Message passing, queues, background/batch processing
Alternate methods available to achieve the goal	Design documentation, standardization, specifications, common protocols
Slack in resources possible	Autoscaling, spare capacity/failover
Buffers and redundancies are natural; emergent from the design	Message buses, queueing, log-oriented designs, resource pools
Substitutions are natural, available, and extensive; emergent from the design	Failover, standardization, specifications, common protocols
Logical isolation	Sandboxing/virtualization, physical isolation
Dedicated resources	Virtualization, resource limits/allocation, physical isolation, functional diversity
Isolated subsystems	Sandboxing/virtualization
Easy substitutions	Fast, easy change paths; standardization; machine-verifiable schemas and protocol definitions, functional diversity
Few surprising causal chains	Standardization, choose "boring" technology
Single-purpose, isolated controls	Virtualization; modularity, functional diversity
Direct, documented information	Design documentation
Extensive understanding	Standardization, choose "boring" technology, common "raw" software materials (languages, libraries, tooling, protocols, data formats), design documentation

1.1.4 Defaults: the principle of least resistance

We believe CISA should make their guidance more concrete on what defaults are and why software manufacturers should implement them. Inspired by behavioral science research, we view defaults as one of the more powerful tools in the "nudge" arsenal and define it as placing the ideal behavior on the path of least resistance[5]. A default means that users must opt out of that option or path, which substantially reduces friction caused by the user needing to opt in.

A classic example of the power of defaults is making 401k contributions opt-in rather than opt-out. As shown by experimental evidence, automatic 401k enrollment results in 85% participation rates, a dramatic increase from the 26% to 43% participation before automatic enrollment kicked in[6]. In safety outside of security, newer cars lock by default when you walk away with the keys, which means human behavior no longer determines safety.

We agree with CISA that the use of defaults as a tactic for encouraging more secure behavior is not widespread in software engineering, nor is it in traditional cybersecurity programs.

Yet, the benefits of defaults apply not just to end users, where CISA focused in the whitepaper, but to the entire software lifecycle. We believe defaults can promote less hazardous methods or materials as well as lower choice overhead early in the development of software. For instance, we consider the curation of vetted, secure libraries for engineers designing systems and the automating provisioning of vetted configurations in CI/CD pipelines as forms of default.

As a general principle, platform engineering teams – whether at software manufacturers or at organizations who consume software – should strive to provide teams with preferred choices of frameworks, middleware, orchestrators, authorization / authentication patterns, and IaC tools with templates. By blessing these options by default, platform teams (or security teams) endorse standardization and reduce choice overload for product teams. They sow a more resilient and higher-quality default for their organization, even if some teams opt out and select other options. If they receive feedback that users are

---

[5] Van Gestel, L. C., M. A. Adriaanse, and D. T. D. De Ridder. "Do nudges make use of automatic processing? Unraveling the effects of a default nudge under type 1 and type 2 processing." *Comprehensive Results in Social Psychology 5*, no. 1-3 (2021): 4-24. https://www.tandfonline.com/doi/full/10.1080/23743603.2020.1808456

[6] Choi, James J., David Laibson, Brigitte C. Madrian, and Andrew Metrick. "For better or for worse: Default effects and 401 (k) savings behavior." In *Perspectives on the Economics of Aging*, pp. 81-126. University of Chicago Press, 2004. https://www.nber.org/system/files/chapters/c10341/c10341.pdf

upset about security policies getting in the way of their work, this indicates opportunities for further user research.

1.1.5 Cognitive load

We encourage CISA to embrace the reality of how humans behave. Humans, like computers, possess finite levels of computational resources. When those resources are burdened by heavy overhead, errors or disruption are more likely to occur in both software and wetware (the human brain).

Cognitive load represents the level of resource overhead occurring in wetware, typically considered through the lens of humans learning or solving problems[7]. Because of the brain's processing constraints, software manufacturers must assess cognitive load when designing systems for use by humans – that "working memory architecture and its limitations should be a major consideration."[8] Yet, CISA, as well as other federal agencies, must assess cognitive load when designing recommendations and guidelines for use by humans, too.

Just like the variety we encounter in computer systems, there are some brains capable of high levels of performance and some possessing less performant processing capabilities[9]. Each brain is optimally efficient at different levels of working memory, described as "a limited amount of information that can be temporarily maintained in an accessible state" in support of cognitive processing.

Just as we expect code can successfully run on any relevant computers within a system, we must ensure that any tools, policies, programs, procedures, and other system components that we design – again, including recommendations and guidelines we hand to human software engineers – can successfully run on the relevant brains within the overall system in question.

Recognizing cognitive load when spelunking through user problems means we appreciate that human attention is a finite and precious resource. UX is more than just

---

[7] Kirschner, Paul A., John Sweller, Femke Kirschner, and Jimmy Zambrano R. "From cognitive load theory to collaborative cognitive load theory." *International journal of computer-supported collaborative learning* 13 (2018): 213-233. https://link.springer.com/article/10.1007/S11412-018-9277-Y

[8] Paas, Fred, Juhani E. Tuovinen, Huib Tabbers, and Pascal WM Van Gerven. "Cognitive load measurement as a means to advance cognitive load theory." In *Cognitive Load Theory*, pp. 63-71. Routledge, 2016. https://www.researchgate.net/publication/252083119_Cognitive_Load_Measurement_as_a_Means_to_Advance_Cognitive_Load_Theory

[9] Jaeggi, Susanne M., Martin Buschkuehl, Alex Etienne, Christoph Ozdoba, Walter J. Perrig, and Arto C. Nirkko. "On how high performers keep cool brains in situations of cognitive overload." *Cognitive, Affective, & Behavioral Neuroscience* 7, no. 2 (2007): 75-89. https://link.springer.com/content/pdf/10.3758/CABN.7.2.75.pdf

figuring out the right button placement to drive clicks; it explores how information should be presented and how to help practitioners better perform their work.

Software manufacturers and federal agencies designing guidelines alike should conduct user research, starting in question-asking mode. What sorts of events attract user attention? How can we draw their attention toward potential security concerns instead? While it may be tempting to brainstorm answers on our own, we cannot answer these questions without user research; otherwise, it simply amounts to guesswork and naive wish making.

In sum, we strongly encourage CISA to reject the belief – common among security gatekeepers – that security should be the top priority in any situation. It is unrealistic that security will always be top priority when so many other things are competing for limited cognitive bandwidth.

1.2 Achieving Secure by Design in practice

Our recommendations throughout this section are grounded in what we believe would help achieve a future of safe, secure, and resilient software.

We strongly believe that achieving Secure by Design should not be at the expense of business goals like software velocity, developer productivity, or product differentiation. Further, we disagree with CISA's stance that software manufacturers must dedicate special resources towards Secure by Design (page 8), because this reflects a "bolt-on" process of the kind explicitly dissuaded elsewhere in the whitepaper.

The designers and builders of software systems must employ practices and tools that are likely to lead to secure systems; the last thing software security needs is further segregation.

1.2.1 Opportunities for software manufacturers

In the spirit of aligning business goals with security goals, we compiled the following list of practices software manufacturers can adopt that both nourish business goals and sustain security. In many cases, software manufacturers may already employ these practices but remain unaware of their potential Secure by Design benefits.

We believe the following opportunities – many of which are not covered in the whitepaper – reflect the most valuable investments software manufacturers can make in Secure by Design. We anticipate that software manufacturers – or indeed any organization – can select which of these opportunities make the most sense given their context, and few organizations will achieve all of them:

1. **Make updates easy to deploy for services and automatically applied for end-users**

Keeping systems up to date is critical for systems reliability, as zero days and other security vulnerabilities can erode system resilience. Software manufacturers should simplify the system update process by automating their own build, test, and deployment pipelines and by automatically releasing updates to end users.

For cases where customers maintain their own deployments of software, manufacturers should make updating to the latest version require few steps that are consistent from version to version and should avoid releasing changes that require human intervention to apply. Since many customers will not upgrade through every version, manufacturers should notify users of critical updates that are required to preserve safety.

These practices also increase release reliability and reduce change failure rate.

2. Decompose software into modular components and isolate the security critical ones into their own fault/privilege domains

Decomposing software into modules is a design practice that makes it possible for software manufacturers to scale software development by allowing engineers to make changes to one part of the system without disrupting other parts. All but the most tangled systems have some level of modularity to them, yet software manufacturers that design their systems with a modular architecture do not always isolate those modules when deploying them.

Isolating modular components into their own fault or privilege domains will limit the damage of faults and failures – not only minimizing the impact of attacks but also facilitating faster incident response.

Some forms of isolation, such as message brokers, queues, and batch processing even allow temporarily suspending non-critical parts of the system to preserve the system's critical functionality. This can be a critical capability during emergencies, such as when a system is under high load or active attack.

3. Adopt external systems to manage and rotate keys or tokens

When software manufacturers manage their own security keys and tokens within their software, they must develop practices to handle this hazardous material safely. Instead, software manufacturers should adopt external systems designed to manage secrets and keys.

Outsourcing secrets management eliminates this hazard within their software by design and frees up time for product teams to invest in differentiating features.

4. Use memory safe languages where possible, reducing the use of "hazardous" materials

Memory unsafe languages, such as C and C++, are a hazardous material used in the production of software. They expose direct control over a program's memory layout with the responsibility that engineers must diligently manage all of the system's memory

manually. Any mistake in memory management results in catastrophic security failure within the isolation domain where the software runs. Even the most diligent software engineers with substantial training are incapable of managing memory safely without flaw on the scale of real systems.

We agree with CISA's recognition that memory unsafe systems are a danger to the security of software systems. New systems should be built primarily in memory safe languages. Where possible, software providers should refactor their existing C and C++ code into memory safe languages. Where not possible, software providers should aim to isolate the impact of memory unsafe components via isolation technologies.

Teams that migrate away from memory unsafe languages see substantial benefits to developer productivity; manual memory management is cumbersome and uncivilized.

5. Employ continuous integration on all software components

Continuous integration (CI) is the process of merging code changes into a central repository regularly, upon which automated systems build and test every commit. Software engineers gain early detection of regressions and can ensure the system is always in a releasable state. With the system always in a releasable state, it is much easier for software teams to respond to faults, failures, and vulnerabilities by shipping code changes and emergency releases of the system on-demand.

Teams that adopt continuous integration see improvements to developer productivity, reductions in change failure rate, and reduced time to restore service after incidents.

6. Deliver software assets via an automated build process

By delivering releases from a continuous integration system, software manufacturers gain consistency and provenance that is not possible when developers release software manually. Humans do not excel at performing the same steps repeatedly and releases are among the most tedious of tasks for engineers. Having machines perform the tasks of turning code into released software means the release will be consistent every time and it will be much more difficult to tamper with the process. Secure by design should extend to all stages of the software delivery lifecycle, and the whitepaper should draw more attention to the flaws and uncertainty that manual releases introduce.

Teams that adopt automated builds see reductions in change failure rate and improvements to developer productivity.

7. Validate security, privacy, and other important properties of the system through automated integration testing

Software manufacturers should validate important properties of the system through automated integration testing. Testing the system for security, privacy, and correctness before it is deployed provides greater confidence that the system can be released safely.

This reduces the change failure rate and increases release frequency. This applies both to regular, everyday releases and to emergency releases, such as in response to an incident or newly published vulnerabilities in a dependency. Advanced teams may even decide to enable continuous deployment when their investment in automated testing provides sufficient confidence.

Teams that adopt automated testing see reductions in change failure rate, reduced time to restore service after an incident, increased deployment frequency, and a reduced lead time for changes.

8. Automate library and other dependency updates to stay recent and intervene when automated upgrades fail

Software teams can integrate automated dependency updating into their continuous integration systems. An automated job can periodically attempt to upgrade dependencies to the latest versions and run the system's tests against the proposed change. Engineers can approve successful upgrades in a single step and, if the upgrade fails, determine (and make) the necessary changes to the software to support the updated dependency's change in behavior.

Automated updates make patching less disruptive and time consuming by reducing the amount of manual effort engineers must spend on routine updates and by reporting failures related to complex updates earlier.

Teams that adopt automated dependency updating see increased deployment frequency and higher developer productivity.

9. Make use of vetted, standardized components where possible

Crucially, software manufacturers should use vetted, standardized components in security-sensitive areas of the system such as those employing cryptography or authenticating access.

Choosing vetted, standardized components for the organization allows developers to build new software without succumbing to decision fatigue when selecting specific technologies. It also improves the implementation similarity of codebases with similar behaviors, making it easier for engineers to switch between teams, review the code of other teams, and share code between systems.

Developers have lower cognitive overhead when understanding and reviewing changes that use technologies they're familiar with. This is especially important for security sensitive areas of the system such as those employing cryptography or authenticating access — most engineers are not experts in these domains and shouldn't need to become experts. Where in doubt, choose boring technology that is well understood and supported by the community.

Organizations that adopt standardization experience higher developer productivity and reduced change failure rates.

10. Employ common patterns and style throughout the system, reducing mental burden on engineers and making mistakes easy to spot

Defining and using standard patterns at all scales of software development reduces the mental burden of understanding the system and mistakes easier to detect. When writing new code or making changes to existing code, engineers won't need to figure out which approach is best when they have the standard pattern to follow.

Organizations that adopt common patterns experience higher developer productivity and reduced change failure rates.

11. A culture that requires peer review for all source code contributions and rewards diligence

Collaboration and peer review at all stages of development reduces the impact of biases and human mistakes. Healthy software organizations encourage diligent review of peers' work in design documents, code review, and other ceremonies.

Teams that adopt code review experience reduced change failure rates.

12. Make the default configuration secure and straightforward enough to be used safely without significant modification by operators

Software organizations should aim to make the default configuration of their systems both straightforward to operate safely and reasonably resilient to the types of threats it is likely to encounter. Curators of default configurations understand both the types of threats the system may encounter and the depth of knowledge of the system its operators will have, balancing the two conflicting requirements appropriately.

Teams that adopt standardized configurations experience higher developer productivity.

13. Hide and discourage insecure configuration options, such as the ability to disable certificate validation

Options known to be insecure, such as the ability to disable certificate validation, should be discouraged, hidden or removed from the software. Often these options are kept for ease of local development or to provide opt-in compatibility with legacy systems and should not be used in new production systems. Legacy options are often overlooked during development and represent a risk to stability of customer systems. As a stark example, Knight Capital's failure to understand the presence of a legacy configuration option in its system resulted in losses of over 440 million[10].

---

[10] https://www.henricodolfing.com/2019/06/project-failure-case-study-knight-capital.html

Teams that retire outdated configuration options experience higher developer productivity and lower change failure rates.

14. Design the system with its operation in mind

Software manufacturers should monitor the system so operators will know when to respond. The system should offer sufficient logging for responders to be able to understand the system when it falters or nears the edge of safe operation.

Systems should be designed to be operated by humans. Operators should be confident that the system has appropriate monitoring in place so the system can report when it is unhealthy, appropriate alerting in place so they can be confident they'll be alerted to its unhealthiness, and appropriate logging in place so they'll be able to understand the system when alerted that it is unhealthy. Furthermore, responding to incidents will often require making changes to the software and redeploying it. Software systems that have cumbersome release processes will necessarily have more severe incidents.

Teams that design systems for operation experience lowered time to restore service.

15. Monitor the system to verify that it continues to perform its function

Monitoring the system to ensure it continues to perform its function is a critical requirement of operating a production service. Monitoring systems should ensure not only that the system is available, responding to health checks, and has available free resources, but that it is actually performing its designed function. This may involve keeping counts of transactions and reporting on the queue depth of work to perform.

Teams that monitor systems experience lowered time to restore service.

16. Employ object-relational mappers (ORMs) to eliminate the hazard of potential SQL injection

Object-relational mappers (ORMs) and other database abstractions exist to make querying and modifying data in databases easier for developers. In addition to ORMs' productivity benefits for many types of systems, they have a substantial security benefit as well. Using an ORM makes it much trickier for an engineer to write a SQL injection vulnerability — they would have to go out of their way to fight the ORM's defaults to write a SQL injection. Such efforts will usually be straightforward to spot during code review and may even be reported by linters and other code analysis tools.

Teams that implement ORMs experience increased developer productivity.

17. Use infrastructure as code (IaC) tools to reduce the hazard of stateful infrastructure

Infrastructure as code (IaC) is the ability to provision and revise your computing infrastructure declaratively in code instead of via manual processes and settings. This eliminates the hazard of production infrastructure drifting from the expectation of designers, since all infrastructure is declared in code that is updated as the software is. IaC tooling will make sure that each environment is consistent with the software as software is deployed into it, removing the burden of applying infrastructure changes from engineers and avoiding the possibility that they might forget to apply them or apply them incorrectly.

Teams that implement infrastructure as code experience lower change failure rates, increased deployment frequency, and quicker time to restore service.

18. **Outsource sensitive functions like authentication or payment processes to remove "hazardous" methods or materials (see Section 1.1.1 and Section 1.1.2)**

Features involving sensitive actions or data that aren't core to the business value the systems deliver are best left to be outsourced to other parties that make that sensitivity their core business value. By outsourcing features like payment processing or identity verification to third parties, these features can receive dedicated security attention.

In particular, securely authenticating users by storing and validating credentials is a feature that is common to many systems but requires ample caution during design and implementation. Outsourcing these behaviors to a common authentication service or third party service allows engineers to focus on building the parts of their system that deliver value.

Teams that outsource sensitive functions like authentication experience higher developer productivity.

19. **Create patterns for common cross-cutting concerns such as logging, authentication, database access, audit, secret management, health checks, error propagation, retries (i.e. "paved roads")**

Software leaders should choose standard patterns for cross-cutting concerns that most teams in their organizations will encounter. Choosing defaults lowers cognitive overhead for individual teams, since they won't have to make these choices themselves, and makes it easier for teams to understand their peers' systems, who are likely to choose the default options. Critically, these patterns should not be in the form of mandates — depriving designers of agency reduces their ownership over the systems they should be nurturing and the default options may not be right for every system.

Organizations that define standard patterns – also known as "paved roads" – experience increased developer productivity, lower lead time for changes, and lower change failure rate.

20. **Create vetted templates with reasonable defaults for common codebase types, such as service, web frontend, CLI, etc.**

Software leaders should build vetted project templates for the common codebase types that teams within their organization will build. As examples, these can include templates for services, web frontends, command line tools, event processors, and software libraries.

Templates are an ideal vessel to provide examples of common patterns, encourage use of continuous integration, and to default to common choices since most teams will use a template when starting a new project. Teams will not only feel lower cognitive overhead when using a template, but will also be empowered to continuously integrate their software and add automated tests.

Critically, the template should not be seen as a mandate to use any of its patterns or selections, only an endorsement of them — depriving designers of agency reduces the ownership over the software systems they should be nurturing and the default choices may not be right for every system.

Organizations that build standard templates experience increased developer productivity and lower change failure rate.

21. Use type systems as a mechanism to mitigate engineers' programming mistakes

Type systems are machine-enforced requirements for how and what kind of data is allowed to flow through a system. By making effective use of type systems, engineers can encode rules about what data is valid in various parts of the system and have these rules be enforced ahead of time, before the system is deployed instead of at runtime, where the mistake will be exposed to users.

Type systems lower the cognitive overhead for engineers to understand which shapes of data are valid at which points in the program and protects the system from related mistakes. Choosing to forego static typing can be mitigated through additional automated testing.

Teams that make effective use of type systems enjoy increased developer productivity and lowered change failure rate.

22. Create guidelines detailing safety requirements that should be satisfied before a system is initially deployed to users

Build an organization-specific set of guidelines that systems should follow before they are deployed to users and review new systems before they go live. Software manufacturers should use the guidelines to direct the product team's attention to hazards they commonly overlook during the initial implementation of systems. These guidelines should be advisory only. Once the product team launches the system to users, stakeholders should expect the maintainers of the system to assume responsibility for its safe operation. Guidelines commonly include recommendations on monitoring, logging, alerting, authentication, configuration management, data retention, and privacy.

Organizations that define safety guidelines experience lowered change failure rate, both in the initial launch of systems and on an ongoing basis.

23. Define clear ownership of services, components, and systems to encourage pride in work and a depth of knowledge

Software systems are sociotechnical in nature and evolve over time. Clear ownership and focus allows engineers to better understand the system's context, design, and implementation, leading to greater ability to evolve the system towards quality and resilience.

Teams that have clear ownership over their work experience increased developer productivity and reduced change failure rate.

1.2.1.1 Patch cycles

We do not see a way out of the cycle of creating and applying fixes. Reducing the velocity of software delivery and stifling experimentation makes firms less competitive and innovative in their markets. Additionally, slowing software delivery is empirically correlated with higher change fail rates[11]. Change is healthy. One of attackers' core benefits is that they can adapt more quickly than their targets; we must encourage similar nimbleness, whether in software manufacturers or customers.

Instead of avoiding patch cycles, software deploying organizations should aim to make the patching quicker and more frequent. Automated test suites and dependency update bots make testing new versions straightforward and easy.

In the common case, dependencies will be compatible, and a human operator can approve the deployment of the updated software. In the rare case where updated dependencies cause breakage, the test suite will detect exactly what part of the system is incompatible with the update, providing helpful context to the person that investigates.

In either case, patches are deployed much more quickly than with manual patching. This entirely avoids the need for temporary mitigations and workarounds to long patching cycles that become tomorrow's technical debt.

1.2.1.2 Integration testing

We believe CISA should actively encourage the adoption of integration tests. Integration tests observe how different components in the system work together, usually with the goal of verifying that they interact as expected. These tests are valuable in exposing issues attackers enjoy exploiting, like error-handling bugs, misconfigurations, and lack of permissioning.

---

[11] https://services.google.com/fh/files/misc/state-of-devops-2021.pdf

How does an integration test look in practice? Consider a basic example of a web application connected to a database. An integration test could and should cover the case of "disconnect and reconnect the database to make sure our database abstraction layer recovers the connection" in most database client libraries.

The AttachMe vulnerability – a cloud isolation vulnerability in Oracle Cloud Infrastructure (OCI) – exemplifies what software manufacturers should uncover with an integration test. It also serves as an example of how hazardous it is to focus only on "happy paths" when testing and developing in general.

The AttachMe bug allowed users to attach disk volumes for which they lack permissions – assuming they could name the volume by volume ID – onto virtual machines they control to access another tenant's data. If an attacker tried this, they could initiate a compute instance, attach the target volume to the compute instance under their control, and gain read/write privileges over the volume (which could allow them to steal secrets, expand access, or potentially even gain control over the target environment).

Aside from the attack scenario, however, this is the sort of interaction software manufacturers do not want in multi-tenant environments for reliability reasons too. The software manufacturer could develop multiple integration tests describing a variety of activities in a multi-tenant environment, whether attaching a disk to a VM in another account, multiple tenants performing the same action simultaneously to a shared database, or spikes in resource consumption in one tenant.

As a general principle, software manufacturers should conduct integration tests that allow them to observe system interactions across space-time. We believe CISA should encourage software manufacturers to follow this principle and not solely rely on testing individual properties of individual components (like unit tests). One input in one component is insufficient for reproducing catastrophic failures – nor exposing vulnerabilities at the system-level – in tests. Multiple inputs are needed, but this need not overwhelm software manufacturers. A 2014 study found that three or fewer nodes are sufficient to reproduce most failures – but multiple inputs are required and failures only occur on long-running systems[12], highlighting both the deficiency of unit testing and the necessity of practices like integration testing or resilience stress tests.

1.2.1.3 Modularity and isolation

We do not believe the framing of "damage control" is constructive (page 4). Damage control involves customers reacting to an intrusion so as to limit, or fix, its damage.

---

[12] Yuan, Ding, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed {Data-Intensive} Systems." In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 249-265. 2014. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-yuan.pdf

Suggesting that customers should never have to do this is a lofty goal that we believe is unattainable; it requires perfect vigilance on the part of software vendors, integrators, and customers.

Instead, we encourage CISA to recommend that software manufacturers should limit damages by breaking software into modules that have isolated fault domains. This is similar to bulkheads seen in ships, airplanes, and automobiles – that is, systems resilience through modularity. As far back as the 12th century, humans built ships in compartmental sections so that when one compartment was damaged, the rest could still function without the ship losing buoyancy and sinking. Yet, bulkheads offer more than structural safety and protection against water, fire, or electrical damage; they also divide the ship into functional areas with designated purposes.

Faults or intrusions should not lead to a catastrophic loss of function or integrity as they do in monolithic software designs.

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 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[13]. 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 organizations worry less about the side effects the patch might have on other parts of the system. It also means modular systems

---

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

are easier to refactor, which supports the goal of fostering the adoption of memory safe languages.

When a software manufacturer discovers 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.

We believe CISA should encourage software manufacturers (as well as customers – really, it applies to anyone using software) 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.

1.2.2 Opportunities for CISA

To nurture Secure by Design in practice across the software community, we believe CISA must consider the following calls to action:

1. **Rewrite commonly used software in memory safe languages**

Given that memory unsafe languages are one of the more dangerous raw materials used in software, CISA could finally propel the industry towards safety by taking an active role in rewriting commonly used software in memory safe languages. The industry is moving towards this end in fits and starts, but numerous core software libraries and end user packages are written in memory-unsafe C or C++. CISA could work with the maintainers of these critical software packages to plan and implement incremental migrations to safe languages.

2. **Invest in the usability of software isolation technologies**

Isolation is the core of numerous computing advancements, but isolation's benefits are difficult for everyday developers to realize in their own software. We believe this is due to the cumbersome usability of software isolation technologies. Just as the capabilities and technologies behind containers were available for decades leading up to the container revolution, so too are the technologies for fine-grained isolation. Fine-grained isolation would limit the damage of any security vulnerability to the module that it is present in. This would drastically curtail the impact of most security vulnerabilities and allow teams to confidently declare that they don't need to patch most vulnerabilities.

3. **Work with maintainers of popular languages and libraries to institute secure defaults and deprecate hazardous APIs and features**

CISA is in a unique position to push maintainers of popular libraries and programming languages to institute secure defaults for their ecosystems and take steps to deprecate hazardous APIs and features. One can look to the PHP ecosystem as a beacon of success in its efforts to deprecate unsafe APIs and features. PHP was well known for having safety flaws endemic in its language design and standard library. Through decades of effort, safer replacements have been built with the unsafe facilities deprecated and much of the ecosystem migrated. It takes leadership to challenge the status quo and build a safer future.

4. Fund the redesign of computer science/engineering curricula — towards memory safety

Universities over-index on teaching C and C++, when other languages are better for exploring and studying the behavior of computing machines. CISA could encourage universities to teach Python, Pascal, Java, Scheme, Logo, JavaScript, Ada, Go and others instead and assist in rewriting curricula. When teaching the low level mechanics of a specific machine, its flavor of assembly is a much better choice than C or C++. The greater the population of new graduates that have a memory safe language as their best-known language, the more memory safe software will be written.

5. Set standards/requirements for software consumed by the federal government

To discourage market segmentation via the availability of security features, the federal government could set requirements that any software it purchases must offer the same access to specific security features in every one of its editions. Given the federal government's strong purchasing power, this would shift the incentives on availability of these features. Such requirements would have to be written carefully to ensure even malicious compliance meets CISA's goals.

6. Encourage the adoption of interoperable standards and the ability to freely switch between vendors/providers

To encourage the ability to switch between vendors, CISA and the respective agencies could encourage the adoption of interoperable standards by requiring strict compliance to specific standards for certain types of systems.

The presence of multiple compatible implementations in the market would let customers switch implementations in time of emergency or when a vendor is not meeting their needs. It would also encourage the disaggregation of software and reduce the ability of market dominating software vendors to influence neighboring software markets. Such requirements would have to be written carefully to ensure even malicious compliance meets CISA's goals.

7. Contribute to open source components

With the widespread use of open source components in software systems, CISA could directly contribute to open source components in wide use to instill secure by design principles and make a direct impact on any downstream systems. This could be in the form of grants, collaboration, design review, and even directly contributing code to address flaws and unsafe defaults. As an incentive, CISA could add cooperating software projects to a curated list of secure by design components.

For all the fuss about the software supply chain and its security, few in the cybersecurity community are directly involved in improving the software itself. We encourage CISA to leverage its software engineering expertise to actively contribute to improving popular projects in the software supply chain rather than pursue flamboyant, but ultimately fruitless pursuits like SBOMs.

1.3 The software market

The software market is enormous and diverse. Our concern is that the whitepaper largely treats all software manufacturers as if they are similar in type and resources. We believe CISA should clarify who they mean when they refer to both customers and software manufacturers. When referring to "customers," is CISA referring to non-technical consumers purchasing routers for their home? Hospitals with medical equipment that runs software? Fortune 500 corporations using databases? And for "software manufacturers," are they referring to publicly-traded corporations with billions in revenue? Or is CISA referring to a few friends who maintain an open source project beloved by software engineers?

We understand the need to generalize recommendations, but we feel that the whitepaper's current form does so at the expense of clarity. For example, the whitepaper states, in the context of reducing burdens on small to medium organizations (SMOs): "Conversely, security investments by the **relative few** manufacturers will scale" (page 13, emphasis ours). We are puzzled why CISA believes there are "relatively few" software manufacturers. There are approximately 6 million SMOs with paid employees in the United States[14]. While there are approximately 557,000 software and IT services companies in the United States[15], there are at least four million software developers in the United States. When considering the open source ecosystem, GitHub reported that developers started 52 million new OSS projects in 2022 alone[16]. If CISA believes that "software manufacturers" only applies to large businesses, then they should state so. As is, they simultaneously lament the onerous burdens SMOs face but ignore that many software manufacturers *are* SMOs themselves – or even less resourced in the case of OSS maintainers. Software is an

---

[14] https://advocacy.sba.gov/wp-content/uploads/2021/12/Small-Business-FAQ-Revised-December-2021.pdf

[15] https://www.trade.gov/selectusa-software-and-information-technology-industry

[16] https://github.blog/2022-11-17-octoverse-2022-10-years-of-tracking-open-source/

exceptionally diverse industry; there is not an overlord class of negligent elites that provide all the software.

We remain skeptical to what extent software issues like vulnerability exploitation result in tangible harm to SMOs; the oft-repeated myth that "60% of small businesses go out of business within 6 months of a data breach" is a trivially disproven statistic[17]. Nevertheless, we strongly believe CISA should encourage SMOs to adopt software-as-a-service (SaaS) tooling. SaaS tooling would alleviate the overhead of deploying and maintaining the software SMOs use; SaaS manufacturers handle upgrades on the customer's behalf. We will discuss SaaS more in the context of accountability in Section 1.6.

1.4 Customer value

We disagree that security should be the focal point of product design and development processes; the focal point should be delivering value for customers. Assuming our economy will continue to embrace market competition, CISA's objective of forcing companies to prioritize something neither customers (nor shareholders) value over the things they do value – as demonstrated by their willingness to pay – will be not only farcical but also alienating to private industry.

Security will never ever be a core business goal except for companies that are selling security. Software manufacturers should absolutely add product features that allow them to innovate in their markets. If a software manufacturer obeys this request (page 8), then they will let their competitors sail ahead of them (the damage to "brand reputation" CISA reaches for as justification in the whitepaper is not backed by evidence[18]). It also creates a disincentive for customers to prefer secure products; customers will still buy products that solve the problems they care about most, and, again, barring a departure from our free market system, then the federal government cannot force them to sacrifice prosperity in the name of security.

However, we believe the whitepaper offers a false dichotomy between security and innovation; software manufacturers can build new features in a way that preserves system security – but the whitepaper's recommendations do not enable them to do so. Security cannot and will not stifle all other priorities. Our recommended Secure by Design practices in Section 1.2.1 reflect this philosophy.

1.5 Components vs. systems

Software may be straightforward for developers to understand at the component level but understanding a component's place in and impact on the system requires a much greater level of comprehension. Many of the most subtle and worst software failures occur at boundaries between components or are artifacts of the system's interactions as a whole.

---

[17] https://www.usenix.org/conference/enigma2023/presentation/sanabria
[18] https://kellyshortridge.com/blog/posts/markets-dgaf-about-cybersecurity/

Consider the recent privacy incident with Wyze's smart security cameras[19]. Wyze integrated a new asset caching system in a way that mistakenly delivered previously-cached videos from one user to other users who should not have had access. As a home security company, they would have known that granting users access to other users' videos should never be permitted and was a violation of user trust – and yet, their system "let" it happen and it took customers reporting the issue for Wyze to notice.

So, how did this happen? We suspect that there was component-level testing to ensure that users could access only their own recordings; it would be negligent not to have such tests. Similarly, we expect the cache was well tested to ensure it performed its function, which was to reduce load on other parts of the system by caching assets. But the system, with the caching layer integrated, was not tested to ensure users could access only their own recordings. Each of the components may have been secure in isolation, but the overall system was not. It is essential for software manufacturers to test at the system-level to ensure it is performing its function, and that it upholds the properties it is expected to.

We encourage CISA to remember that the security of a system is as it is deployed, not as it is imagined. Alas, little in the whitepaper addresses this distinction between code components and the system once it is deployed into production.

1.6 Accountability

The whitepaper paints a strange picture of accountability for software security incidents. In particular, we are left wondering where we must draw the line for customer accountability. Is a customer never culpable for software mistakes? Is it always the responsibility of the software manufacturer to ensure the customer is incapable of shooting themselves in the virtual foot? Should software manufacturers, in effect, surveil their customers for their purported own good (page 22)?

Imagine a hypothetical customer back before the days of software who experienced a damaging loss because they could not find important documents within the sea of unsorted papers across their filing cabinets. Perhaps they would exclaim, "We can't find what we need in our filing cabinets, our filing cabinet vendor should've imposed a structure on us so we couldn't get confused!" Would we find this to be a reasonable complaint?

Given the elaborate use of the seatbelt analogy throughout the whitepaper, when does the customer become accountable for ignoring the seatbelt warning? Or, what if they drive drunk? What if they are texting? Should car manufacturers angrily beep at them whenever their hands leave the steering wheel? Should we blame car manufacturers for when customers speed? After all, manufacturers "allow" speeding and even market their cars based on how quickly the consumer can accelerate to a glamorous top speed.

---

[19] https://www.cbsnews.com/news/wyze-camera-breach-let-13000-customers-peek-into-others-homes/

The requesting agencies should strongly consider whether and when such paternalism – and the infantilization of customers – is healthy.

Beyond this, there is no equivalent in software to a seatbelt. Seatbelts rarely get in the way of legitimate, necessary activity; can be installed in any form of automobile; and meet the needs of most human bodies. The whitepaper seems to espouse that MFA is the closest equivalent to seatbelts. However, what happens when MFA impedes a nurse trying to save a patient's life? Is it then the software manufacturer's fault for requiring users to authenticate so frequently?

This gets to the heart of a core counterargument to the whitepaper's philosophy: should the federal government require *everyone* to use available safety features? While not stated explicitly in the whitepaper, that is the next logical step if they require software manufacturers to offer those features for free, to all customers. If only, say, 2% of customers adopt those now-free features – like the real 2FA adoption rate of 2.3% among Twitter users[20] – should we consider it a Secure by Design success?

Indeed, how many "misconfigurations" are legitimate features that customers only realize they do not want through the lens of hindsight post-incident?

Furthermore, when the whitepaper suggests that software manufacturers must "take ownership of customer security outcomes" (page 10), is this ownership in a product design sense or in a legal sense? We suspect no one in the private sector wants this in a legal sense.

SaaS obviates many of these concerns because the software vendor performs ongoing maintenance rather than the customer. But when organizations refuse to adopt SaaS out of "security" concerns, then fail to resource the ongoing maintenance of the infrastructure they chose to own – does culpability still reside with the software manufacturer? When organizations refuse to invest in rectifying or redesigning their architecture so it does not require less secure and less arcane configurations, is it the software manufacturer's fault for still offering those "complex" configurations per the customer's request?

The whitepaper seems to imply that customers are not accountable for their fragile, frangible, festering legacy systems that they avoid modernizing by demanding software manufacturers contort on their behalf; it is instead seemingly the software manufacturer at fault for creating the "complexity of security configurations" despite it being the explicit desire of these customers (no software manufacturer wants to create such a UX mess for fun). SaaS providers are lucky in this regard; they can deploy software and attest that they meet some minimum standard. They need not worry about a customer's legacy environment not accepting modern cryptographic protocols or having a broken corporate

---

[20] https://www.bitdefender.com/blog/hotforsecurity/despite-all-the-advice-97-7-of-twitter-users-have-still-not-enabled-two-factor-authentication/

certificate authority or any of the countless other “quirks” that could require customization.

We strongly believe CISA should be far more cautious when soliciting input from experts representing the customer population so they do not allow leaders at these companies to absolve themselves of accountability for their choices.

1.7 “Radical Transparency”

We do not believe software manufacturers are obliged to offer “guided tours” of their software delivery practices (page 21), as they have important work to do (like sustaining software quality).

With that said, practitioners can and do share methods to help the broader software community; we suspect that federal agencies are not often exposed to the spaces where these learnings are shared. These spaces include software engineering conferences, in-person meetups, industry groups, blogs, social media, corporate offsites, teams’ bimonthly sprint retrospectives, and local “third places” (where software engineers often complain about the absurd hoops their security team asked them to jump through). Furthermore, we argue that maintainers of open source software perform development out in the open, for anyone to observe.

We believe it does not make sense for software teams to perform “secure SDLC self-attestations” (page 23). None of the "secure software development lifecycle" frameworks align with the workings of modern software teams that deliver software continuously. All are too prescriptive in their required artifacts and ceremony in a manner that disallows for continued iteration on process and workflow. They all drastically understate the value of automation in the modern development workflow and require a separation of roles that reduces ownership of product outcomes.

We believe CISA should only recommend secure software development methodologies that incorporate recent research on developer productivity and outcomes, such as *Accelerate: The Science of Lean Software and DevOps* and the SPACE developer productivity framework[21].

1.7.1 SBOMs

The construction and assembly of software is complex in a way that a software bill of materials (SBOM) can neither capture nor communicate. An SBOM instead communicates a high-level description of some of the components used to assemble a software asset, but communicates nothing of the structure of the software nor anything about the most critical custom components.

---

[21] https://queue.acm.org/detail.cfm?id=3454124

One learns very little about the security properties of a piece of software by examining its SBOM. Practitioners would yield better security results by focusing on pinning their software dependencies to ensure build reproducibility, setting up systems to automate library updates, and ensuring new versions of software can receive an automated validation of their behavior. Each of these steps leads to tangible results in the form of swifter patching and more confident software delivery.

Yet again, we request CISA abandon its quest to force SBOMs on the commercial sector; if we are to achieve a future where software is safe, secure, and resilient, then organizations must invest in practices that proffer a tangible, meaningful ROI rather than fling their fortunes into shallow security theater.

2. Incorporating security into the SDLC

We covered how organizations should incorporate security into their software delivery activities at length in Section 1.2.1.

2.1.1 Effective tactics

In our experience, the most effective tactics (which echo our recommendations in Section 1.2.1) include:

- Automation
- Templates
- Patterns
- Defaults
- Clear ownership
- Checklists
- Integration testing
- Paved roads
- Adoption of type systems
- Isolation / sandboxing
- Message queues
- Logs
- Monitoring

Inline security reviews, a common tactic among cybersecurity teams, are not effective at achieving more secure outcomes and serve only to slow software delivery – and that rigidity erodes resilience. Periodic source code audits can be effective but are usually expensive.

2.1.3 & 2.1.4 Smaller software companies and best practices for them

We do not believe smaller software manufacturers are at a disadvantage in terms of implementing the secure by design principles and practices we enumerated in Section 1. But they will be at a disadvantage if the practices espoused in the whitepaper become

requirements. Just as is true today, small software vendors are at a disadvantage when it comes to applying expensive bolt-ons and meeting cumbersome compliance requirements.

Smaller software manufacturers are typically not implementing tools for security purposes; they choose tools and practices to boost productivity and deliver value to customers more quickly. The closest dynamic in the private sector is for smaller companies to implement practices for a compliance benefit, rather than a security benefit (since compliance and security are distinct aims, but the former prohibits selling to certain customers).

With that said, there are tactics feasible for smaller, resource-constrained companies to implement, including:

- Automated version upgrading to reduce the disruption of patch cycles
- Object-relational mappers (ORMs) to eliminate the hazard of potential SQL injection
- Infrastructure as code (IaC) to reduce the hazard of stateful infrastructure
- Outsourcing authentication to avoid the danger of mishandling credentials
- Outsourcing sensitive functions, like payment processing, to remove “hazardous” materials (see Section 1.1.2)
- Automated testing to improve release frequency
- Writing new code in memory safe languages to remove "hazardous" materials

Ideally, smaller companies would make choices that mean security concerns are less intrusive to their delivery and operations.

2.1.6 Continuous security education

Before digging into details on continuous security education, we feel it is worth asking whether such programs result in more secure outcomes – or are they a waste of time that distracts from improving software quality? Will it uplift the least capable developers more than it annoys and sidetracks the most productive? The most effective engineers who invest in honing their skills are already teaching themselves more than is covered in the vast majority of commercial security training.

Some hazards cannot be mitigated through developer training. At great cost, a software manufacturer could pause any software project implemented in C or C++ for six months to train its developers on how to avoid memory management flaws. When they return, they would still make memory management mistakes – many of which wouldn't be discovered until after the software was deployed. They might make fewer mistakes, yet the hazard would not be completely mitigated.

The only effective remedy is to migrate to memory safe languages over the long term, and isolate software components to constrain potential damage in the meantime.

There are a great many hazards to which this applies, as illustrated in Section 1.1 with the Ice Cream Cone Hierarchy.

In the real world, companies who build paved roads to make the secure way the easy way for developers – so developers need not be troubled by security concerns because they are handled for them as part of the tooling they use – achieve more secure outcomes than those who rely on training and finite developer attention.

We cannot expect engineers to remember every recommendation given to them during security training at every moment as they are writing software. Their work entails understanding the relevant parts of the system and making transformations to their behavior. Security training is often not as relevant as delivering value to stakeholders.

We are skeptical that there is a market for continuous security education that is not created by compliance requirements. That is, we cannot find evidence that proves continuous security education improves security outcomes. Considering the considerable time and attention required to complete such education programs, it creates an opportunity cost away from investing in activities that actually improve the software's quality.

3. Education

3.1 Demand signals to universities

Companies that hire security researchers evaluate them on security merits. Software engineers are generally not evaluated during the hiring stage nor reskilled after being hired. Security is not a priority in employee's growth and is an afterthought during most development ceremonies.

3.2 Security knowledge in computer science curricula

Universities include foundational security knowledge in their computing science curricula; operating systems courses are practically all about how to implement a privileged agent that abstracts the underlying machine from unsafe and potentially malicious user programs. Similarly, any good language course will explore the nature of compilation, the importance of correct transformations, and the differences between the abstract machine and its implementation.

Such topics are important for those few who work on operating systems or languages, but practical, "everyday" security knowledge is rare in universities. Software engineers in private industry generally assume that code written by academic researchers is subpar, both from security and software quality perspectives. Universities over-index on teaching C and C++, when other languages are better for exploring and studying the behavior of computing machines. CISA could encourage universities to teach Python, Pascal, Java, Scheme, Logo, JavaScript, Ada, Go and others instead. When teaching the low level mechanics of a specific machine, its flavor of assembly is a much better choice than C or C++.

3.3 Online programs

Online computer science and coding education programs accurately determined that cybersecurity reflects a distinct market of engineers. Thus, they offer separate course material and programs for cybersecurity professionals. There may be room for universities to reposition the small amount of cybersecurity education that their software engineering programs include, but not substantially expand it.

One aspect of computing science education that could change and would, incidentally, offer a security benefit would be to adjust which programming language is used in courses where language isn't the specific focus. Many topics can just as easily be taught in Python, Java, Rust, Scheme, or OCaml as they could in C or C++. We believe CISA, and other interested federal agencies, could encourage universities to deplatform memory unsafe languages.

4. Hardening / loosening guides

Software manufacturers (and their customers) should consider hardening guides a tacit admission that the default configuration is insecure. Every hardening guide recommendation is a missed opportunity for a safer default. The best practice for hardening guides is to write software that does not require them. We cannot expect users to be experts nor go out of their way to receive a secure configuration of it. Software vendors should design a secure, default configuration that is accessible to all customers.

Thus, while we agree with the whitepaper that relying on hardening guides does not scale (page 13), we encourage CISA to explicitly denounce hardening guides as an anti-pattern.

We disagree with some of CISA's guidance as to what actions software manufacturers should take when settings deviate from their recommendations (page 16). Surveilling and nagging users does not meaningfully contribute to security. Customers are not enabling legacy features or options out of a desire to make the system less secure; they do so out of a necessary requirement to make the software function in their environment. To be clear, customers are turning on insecure features for the value those features offer, not because of anything to do with security.

CISA should place more responsibility on the consumers of software to deploy it to the manufacturer's default specifications. Customers deploying security products to mitigate their own lack of configuration and deployment hygiene is the height of software dysfunction. The blame should not lie with software manufacturers who have no power to clean up customer environments and at best could refuse the customer’s business. We hope that CISA recognizing this market dynamic and appropriately assigning accountability would encourage more companies to improve their internal architecture so they no longer demand their software vendors contort to meet their (often less secure) requirements.

4.1 What are some best practices for hardening guides?

We believe the best practice for hardening guides is to not require them. Again, we believe software manufacturers should treat every hardening guide as a missed opportunity for a safer default.

4.2 How do software manufacturers decide on their products' default configurations?

Software manufacturers choose their default configurations primarily based on ease of installation and momentum. They typically preserve the previous version's defaults in the new version; the new version often inherits these defaults from the tools and frameworks software manufacturers use to build the software.

From the software manufacturer's perspective, they must invest in substantial education campaigns to migrate away from poorly chosen defaults without disrupting activity. Microsoft's campaign to migrate its customers off the insecure NTLM password hash started in 2010[22] and continues to this day.

SaaS providers face an easier burden; the provider can swap out or reconfigure implementations without customers knowing, and the expense of doing so can be amortized over many customers.

4.3 Loosening guides

We believe a "loosening guide" is a losing proposition. Consider Microsoft's progress in retiring old password hashing and authentication standards; at no point would they write a "loosening guide" as if that were a noble goal. Instead, they provide guidance to customers on a case by case basis on how to re-enable certain old features to provide compatibility with the systems that require it.

4.4 Staffing for hardening guides

Hardening guides are sometimes written by well-meaning security teams who possess insufficient clout to address deficiencies in defaults. Else, they may be written by sales engineering teams hoping to position an otherwise incompatible product into high-regulation or security-sensitive environments. In fact, guidelines on enabling *insecure* features are often written or requested by customer success teams as they aim to keep customers with legacy configurations or infrastructure running up to date versions of the product.

4.5 Automated hardening mechanisms

Many consumer and small business software products strongly encourage users to configure multi-factor authentication (MFA), strong passwords, and biometric authentication during setup, but offer users the ability to decline. Many enterprise software

---

[22] http://msdn.microsoft.com/en-us/library/cc236715(v=PROT.10).aspx

products strongly encourage SSO integrations and use of vendor provided templates for on-premises deployments. In general, product teams face a strong incentive to constrain the installation options wherever possible to avoid customer confusion or misconfiguration.

4.6 Customer experiences with multiple hardening guides

Customers often consider a single hardening guide to be a nuisance, let alone multiple hardening guides. Any configuration straying from the vendor's defaults increases the likelihood of failure or requiring costly maintenance. Customers therefore prefer vendors that secure their products by default – but, it is worth stressing, not at the expense of the software's primary value proposition.

5. Economics of implementing secure by design practices

5.1 Types of costs incurred by software manufacturers

Humans should be able to design, build, and operate reasonably secure systems without having to think about security too much. The end goal should be that secure by design programs within companies will not be necessary – and we remain unconvinced that they are necessary now.

Software is not a unique domain that requires security to be a primary design concern. It will never be a primary design concern for all but the most paranoid. The best way to encourage adoption of secure by design principles is not to convince people that security is important, but instead to shift prevailing practices towards secure defaults (and others we enumerated in Section 1.2.1).

5.2 How costs are absorbed or passed along

Commercial software vendors consider developer training, security analysis tools, technology modernization efforts, and the design of new features to be part of research and development; the customer is not billed. Consulting businesses operate on a different model and may see the client's desire to redesign or improve the security of a system as an opportunity to bill a new project.

5.3 Which secure by design practices are the most effective?

We covered which secure by design practices are most effective at considerable length in Section 1.2.1.

6. Economics of software vulnerabilities

6.1 Impact of vulnerabilities on software manufacturers

6.1.1 How do software manufacturers measure their costs for each vulnerability?

It is unclear to us which, if any, manufacturers attempt to accurately measure costs of vulnerabilities in their software. One might assume that the strongest public signal of the

cost of a vulnerability may lie in the payouts of bug bounty programs, but we believe there are more factors than the knowledge benefit alone. There is much debate as to the costs of a flaw discovered at various stages in the software development lifecycle, and security vulnerabilities are one facet of this discussion.

6.1.4 How do software manufacturers determine how to remediate vulnerabilities?

We do not believe there is a generalized process for how software manufacturers determine what vulnerabilities to remediate and how best to address them; product teams within larger software manufacturers even differ in how they handle this.

The “MAMAA” companies – the tech giants – typically take every potential vulnerability seriously and will patch nearly every security vulnerability without affecting product functionality, to the extent they can. In response to vulnerabilities, product teams in these companies will often build tools to find similar vulnerabilities in their codebases and prevent new instances. One such example is the new dead code analysis passes that were added to clang after the "goto fail" incident. Sometimes, product teams in the tech giants need to change defaults for security reasons and their userbases inevitably protest.

Beyond MAMAA, the remaining software manufacturers are far more variegated – and often haphazard – in their vulnerability remediation approaches.

The next best performers pin their dependency versions and implement automation to bump library versions whenever new ones are released. They generally keep their dependencies up to date, and incorporating security fixes is a straightforward, low-friction process. For security issues in their own code, remediation is less clear. Often developers will not understand vulnerabilities that are reported to them, and it can take a long time for vulnerabilities to be fixed. Sometimes this involves a dedicated security team, but security teams are even less likely to understand the vulnerability than the engineers are. Most often, the remediation process is an engineer googling "SQL injection" and stumbling through the instructions. Most code is written in-house, though the company may not dedicate a maintainer to old systems.

The tier after the next-best software manufacturers tends to invest some effort to stay up to date and uses tools that tell engineers that software is out of date – but no one makes it a priority to update and because they are already so far behind their roadmap and have slow, high-friction software delivery processes that make updating a boggy slog. The smarter players in this space know they cannot keep up and therefore constrain their dependency stack, often relying on Red Hat or Microsoft to provide “blessed” old stacks that will receive security backports. When projects get too old for the platform vendor to support, the manufacturer either rewrites or mothballs them. Some remain alive, and these are the most dangerous. Their ability to patch and understand their own vulnerabilities is perhaps a little worse than the previous group, but not by much. Many systems are maintained by contractors and will have no internal owner.

The last tier includes software manufacturers who are barely keeping their systems functioning and lack any real capability to manage security issues. They will invest the bare minimum to maintain compliance and nothing more, often bolting-on vendor tooling.

6.1.5 Where are tradeoffs made based on this financial data?

To our knowledge, at no point is someone directly comparing the cost of fixing a vulnerability (or a class of vulnerabilities) to financial data. Organizational culture will suggest how much time product teams can reasonably allot to addressing technical debt. Security vulnerabilities often fit in this category as “emergency work”; it is debt that the team must address shortly after it is uncovered. Some teams proactively search for vulnerabilities or set up automated processes to avoid vulnerabilities or discover them earlier, while other teams prefer to maintain a reactive stance.

In many cases, software organizations assess that it is not worth the effort to continue maintaining a product, even if it exposes users to hazards. IoT and network appliances are among the worst offenders here; manufacturers often abandon their devices shortly after they launch the replacement model. Fixing these problems will require addressing the underlying incentives.

6.4 Impact of vulnerabilities on customers

6.4.1 Do software manufacturers calculate costs for consumers?

Software manufacturers do not calculate the average cost for customers to deploy software updates – but they do know the additional overhead results in fewer updates applied and fewer renewals. This continues the trend towards SaaS, where everyone is always up to date (because the vendor hosts the software), and supporting older versions is not necessary. When vendors charge separately for support contracts there is a disincentive to design software that is easy to patch, as a mechanism to encourage customers to pay for support.

6.4.2 How do software manufacturers determine the aggregate cost across all customers for patching?

In general, software manufacturers do not care enough to calculate the aggregate cost of patching. We suspect the major OS vendors have thought about this most, given they automatically schedule operating system and application updates when the customer is least likely to be using the device.

7. Economics of customer demand

7.1 In what ways do customers ask software manufacturers to make products more secure?

Customers mostly do not ask software manufacturers to make products more secure. The most they ask for is for their software products to meet some compliance standard, usually one that they themselves must meet to do business in their industries. PCI-DSS may have done more for software security than any other effort so far.

The other dimension bearing an indirect incentive to secure software is contractual stipulations that indicate money will be returned if some function is not performed. The providing party will perform some level of diligence to ensure they meet the requirement, which can include cybersecurity basics. More often, it will include things like DoS protection and failover (that is, availability concerns).

To the extent that open source software is a market, developers pick and choose which technologies they use. This leads to the furthered adoption of convenient, productive, and secure languages and tools. Dangerous and cumbersome tools eventually lose out. Infrequently insecure components are redesigned and reimplemented, but more often better alternatives displace the older component as engineers build new generations of systems.

7.2 In what ways do customers ask for specific security features rather than asking for products that are secure by design?

Customers do not and cannot ask for a product to have a secure design in any meaningful fashion, except perhaps to prohibit certain practices or attributes known to be insecure. Customers will more often ask for specific features that meet compliance requirements, and the vendor decides how to implement those features.

If a software generates too many CVEs or earns a bad reputation, customers may replace it with an alternative – but they are much more likely to switch due to non-security reasons, such as if it is burdensome to switch or the alternative is cumbersome to operate (as may be the case if it requires frequent, difficult to apply patching or complicated integration)

7.3 How can customers measure the security of a product? Can they take that measurement and translate it into long-term costs to decision makers in a business?

In our experience, it is laborious – if not theoretically impossible – for software designers to accurately measure the security of their software, let alone for customers to do so. Further, we cannot overstate the difficulty of measuring the likelihood of long-tail effects; should CISA determine how to accurately forecast long-tail effects, then the cyber insurance industry will rejoice.

7.4 What are the inhibitors to customers creating a strong demand signal that software should be secure by design?

Apathy is the greatest inhibitor to customers creating a strong demand signal that software should be secure by design. Customers largely do not care because they do not have to care. Breaches offer relatively little financial penalty or inconvenience, except in extreme cases. Indeed, the most common penalty is the burden of more stringent security measures imposed post-breach that impedes business velocity.

8. Field studies

We discourage CISA from using the term "field studies," which is an academic term that is out of touch with how the private sector interacts with customers. The term of art in software product management is "user research."

8.1 Do software manufacturers carry out such field studies?

Formal field studies are usually outside the realm of commercial software development. Some of the largest firms perform such studies, though rarely on security specifically. Product usability, sentiment, and brand awareness are dimensions of which firms are most likely to study interactions with customers as they measure the customer's likelihood to recommend the product or service and purchase again.

The closest practice to field studies in software manufacturing is user research – understanding how customers interact with a product or feature. However, this user research is typically conducted on prototypes to understand whether the draft implementation meets customer requirements, rather than performed after a product or feature has been adopted for a while.

The typical channels for product teams to discover potential improvements to existing features are through account teams (the sales professionals who interact with customers) or customer support requests. For instance, if there are a large volume of support tickets related to password resets, then this might inform a change to login design. Or an account representative will relay a request from the customer to the product team, like wanting more granular RBAC or the ability to download data in CSV format.

8.2 What are some best practices for conducting field studies and incorporating the results into the SDLC?

Progress in user experience and design is hard-won. The usability aspect of software security is infrequently studied by designers and user experience professionals and remains a sorely neglected topic in the cybersecurity industry itself. UX roles are more often aligned to study measurements that are of importance to the business such as conversion rates, retention, and comprehension.

The federal government could encourage academic research into the usability of security features, with findings made open access so all software manufacturers could benefit.

9. Recurring vulnerabilities

9.1 What are the barriers to eliminating recurring classes of vulnerability?

The continued teaching and use of unsafe practices and technologies.

9.2 How can potential customers determine which software manufacturers have been diligent in removing classes of vulnerability rather than patching individual instances of that class of vulnerability?

It is expensive and difficult for customers to assess the security properties of the software they are considering purchasing or have purchased. Some of the most security concerned and well-resourced potential customers with in-house capability may perform a security audit of the software at their own cost, but this is exceptionally rare and applies to only the most sensitive systems.

A more common approach is for customers to examine a company's track record. Customers should consider companies that are open about their vulnerabilities – registering and reporting vulnerabilities they discovered in their own software – among the most diligent. Similarly, customers could treat investments in tooling to discover and fix vulnerabilities as a signal that the company is serious about minimizing the harm of security vulnerabilities. It often takes a public security failure wreaking significant harm for even large organizations to invest in tooling. For example, Microsoft invested heavily after Windows XP and Apple invested in compiler tooling after the "goto fail" TLS bug.

9.3 What changes to the Common Vulnerabilities and Exposures (CVE) and Common Weakness Enumeration (CWE) programs might lead to more companies identifying recurring vulnerability types and investing to eliminate them?

Instead of focusing on educating software engineers, these programs should take a more active role in improving the most common libraries and frameworks in use. The sheer size of the software engineering community eclipses the outreach these programs could reasonably perform, even with substantial increases in funding. These programs should instead be purposeful with their outreach, rather than a “spray and pray” approach, prioritizing groups that themselves seed ideas into their respective subcommunities (like languages and frameworks).

In this vein, we analyzed the top 10 CWEs to determine the best means of addressing the vulnerability classes they represent:

- 1, 2, 3 (use after free, heap-based overflow, out-of-bounds write): Addressed through use of memory safe languages, which provide temporal and spatial memory safety and prevent buffer overflows and use after frees.

- 4 (improper input validation): Partially addressed through increased unit testing, increased reliance on type systems, and by following the principle of "parse, don't validate." We believe this set of weaknesses to be among the most difficult to address categorically, since the space of input validation is vast and domain specific.
- 5 (OS command injection): Could be partially addressed by working with language maintainers and library authors to deprecate the system function and equivalents. Additionally, this is addressed by improving the usability of sandboxing mechanisms; many services/programs should not spawn subprocesses or should be limited to a list of allowed subprocesses.
- 6 (deserialization of untrusted data): Work with language maintainers and library authors to deprecate unsafe object deserialization frameworks as was done with python's pickle module; default to requiring an allow-list describing permitted data types in unsafe serialization/deserialization frameworks.
- 7 (SSRF): Could be addressed by working with language maintainers and library authors to change defaults of URL fetching libraries to disallow localhost, file://, and local network requests, or require callers specify a list of allowed endpoints to contact.
- 8 (type confusion): Addressed through use of memory safe languages and languages that require explicit type conversion. Memory unsafe languages make it all too easy to read data after it is no longer valid or beyond the correct bounds, and thus interpret unrelated data of one type as another. Languages that perform implicit type conversion on data make on data make it all too easy to mistakenly interpret data of one type as if it were another.
- 9 (path traversal): Addressed through use of type-safe languages that treat paths separately from strings; to a lesser extent with opt-in path types and linting
- 10 (missing authentication for critical function): Addressed with better language, library, framework and system defaults; also addressed by standardized authentication middleware.

10. Customer upgrade reluctance

Given our combined experience in software engineering and product management, we found ourselves flummoxed by the implication that software vendors want to maintain backwards compatibility (page 13). We, and our peers at other software manufacturers, would proverbially (in some cases perhaps literally) jump for joy if we could retire all the legacy software we must keep around because an important – usually high revenue – customer wants it.

Every software manufacturer that experiences market success faces the problem of upgrading users (page 19). When a product is SaaS or available for free, then customers must cope with their displeasure in being forced to upgrade. Even still, SaaS vendors will keep features alive long past their shelf life to avoid losing revenue. GitHub only disabled Subversion support this year even though it has been basically dead for half a decade. They

even published a blog post justifying this decision by indicating that only 0.02% of traffic was via Subversion in response to customer outrage.

To be clear, most software manufacturers would love if all users updated to the latest version, disabled all the weird legacy options, and properly secured their accounts. Alas, customers often do not wish to do those things; at most, software manufacturers can nudge them to with end of support / end of life policies. What does CISA believe software manufacturers should do beyond this – refuse the customer's business? This defies the dynamics of our current economic system and is a topic we feel is outside our scope and that of a cybersecurity agency.

To reiterate: we have never, in all of our combined experience, seen a software manufacturer maintain backwards compatibility for anything but an important customers' insistence.

10.1 What are the primary barriers to customers investing in upgrades that should reduce their risk?

Customers care about security so long as it does not cost them anything. Software vendors find it difficult enough to compel customers to upgrade to newer versions even when the upgrade can be dropped in without making changes. Asking users to make changes to update versions often results in them being stuck on an old version.

We admit this is a thorny problem. Security usability is an underexplored area of study. For instance, do you improve overall outcomes by keeping an outdated design (say one that runs under a single privilege domain, at worst as root), but upgrading more users to versions that don't have mundane vulnerabilities? Or is the better outcome from performing a vast redesign to substantially reduce the hazard of future vulnerabilities, at the cost of leaving many users behind?

Support contracts are a way to incentivize this, but they reflect an approach that is more "stick" than carrot. We believe the best way to pull users along is to provide tangible non-security reasons to upgrade and make it easy to do so via automation.

SaaS providers face fewer obstacles than providers that deliver software running in their customers' own environments. An anti-pattern is for software manufacturers to ship software as root to anticipate what permissions it might need in the future. The app store and browser extension model works well here: users receive automated upgrades, and if a new version of the software asset requires more permissions, the system prompts the user for approval.

10.2 What are some examples of security improvements where customer adoption was swift despite those barriers?

We found it difficult to find public examples of security improvements where adoption was swift despite barriers to upgrading.

11. Threat modeling

Threat modeling is an exercise unfamiliar to most engineering teams – and even some cybersecurity teams. We want to stress that developers do not want to write insecure software, and so this lack of threat modeling is not due to negligence. In our experience, the real reason why developers do not threat model is that they do not know where to begin and cybersecurity lingo is foreign to them – a byproduct of a cybersecurity industry pockmarked by jargon and festooned with folk wisdom.

Engineers are more familiar with architecture diagrams and infrastructure terms. For better and sometimes worse, engineers are designing systems with an optimistic view of their behaviors. More experienced designers will be familiar with faults and failures in production systems and anticipate these "threats" to availability when designing software. Some experienced software designers will be somewhat familiar with cybersecurity attacks in theory, but few will have personal experience with them. Meanwhile, in cybersecurity land, security engineers are predominantly focused on modeling threats without regard for other possible disruptors to the system's function.

Software is less secure because these two groups are not speaking the same language and are performing duplicate work. Security engineers are often frustrated that the most potent mitigations are inaccessible to them. Software engineers wish security teams would learn how software is actually developed and delivered to find opportunities to leverage existing practices towards sustaining resilience (as we have attempted in Section 1.2.1).

We believe the unifying principle behind both lines of thinking lies in resilience literature[23], which aims to identify a system's behavior in response to all sorts of stressors. Engineers should excavate possible stresses to a system's purpose, explore how the system responds to the stressor, and design mitigations that improve the behavior of the system when under stress. The final stage is to implement the mitigation and retest the systems against a simulated stressor. This process is more widely known in other disciplines as "resilience stress testing," but which has taken on the name "chaos engineering" within software (we do not like the name, either).

Bringing security threats under the same framework as more mundane threats, such as hardware failure, allows developers to assess likelihood and impact evenhandedly. It also makes it more obvious which mitigations offer the most benefit. Rather than try to identify every possible failure upfront when designing software, engineers should consider the actions attackers will likely take based on their system's context and evolve their models alongside the system. This leads to continual improvement, even as the context and the system's purpose shifts.

---

[23] https://kellyshortridge.com/blog/posts/security-chaos-engineering-sustaining-software-systems-resilience-cliff-notes/

While we don't want to be prescriptive about what precise methodology is best, the specific methodology we suggest is decision trees[24], a tactic from behavioral game theory which we have both spoken about[25] and developed open-source software[26] around.

We recommend software manufacturers adopt decision trees into their git workflows to maintain developer velocity. Software engineering teams should publish their decision trees alongside their architecture diagrams for consumption by peers. In cases where customers install software on their own infrastructure, software manufacturers may opt to distribute decision trees to their customers to better communicate which stressors the software product is resilient to.

12. Charging for security features

12.1 What are some examples of security improvements where customer adoption was swift despite those barriers?

Most software manufacturers employ product management or product marketing professionals who decide pricing. The Pragmatic Marketing Institute is a popular product management training program that offers a module on pricing; the gist is that products and features should be priced based on the value they provide to customers. With that said, not all product managers or organizations follow this wisdom and instead price based on cost incurred (that is, how much it costs to provide the product or feature to customers) or what competitors charge for the product or feature.

Under the "value provided" pricing strategy, if security features were provided for free as part of the product offering, it suggests that they offer no value to customers (and we think the requesting agency agrees with us that such features do offer some sort of customer value). Under the "cost incurred" pricing strategy, then security features will always bear a price because – as we have experienced multiple times in our careers – integrating with SSO or other authentication services is typically a non-trivial endeavor. Under the "match the competition" pricing strategy, if the incumbents or other primary competitors in the market charge for security features, then the company will, too.

Stated succinctly, no matter the pricing strategy, the dominant incentive software vendors face is to charge for security features.

A tiered structure for pricing is common for SaaS vendors to adopt. There are typically three tiers, based on the longstanding "good-better-best" approach to pricing:

---

[24] https://www.usenix.org/conference/srecon23americas/presentation/shortridge
[25] https://www.youtube.com/watch?v=CqwzWoJdbTc
[26] https://www.deciduous.app/

1. A basic tier that is the least expensive[27]. It usually features a limited deployment (whether a constrained number of hosts, users, or data volumes) and feature set. Security features like audit logs – which require the vendor to store them – are often not included in this tier (especially if this tier is free).

2. A medium or "premium" tier with additional features – often the ability to customize or configure elements of the product – that reflects the "just right" point in the Goldilocks principle. This tier will sometimes include password protection, role-based access control (RBAC), or audit logs (with limited retention) – but not always. This is usually when the vendor begins offering administrative controls, albeit limited in scope.

3. An "enterprise" tier reflecting the greater demands by larger enterprises or higher-scale companies, with the highest price to match. The luxury tier of software. Sometimes there is not even a standard per month or per year price associated, but instead a call-to-action of "Call Sales" to negotiate. This tier always includes SSO / SCIM (if available), compliance-related features (like detailed auditing with longer retention periods), and advanced admin controls (whether authentication, allowlisting, rate limiting, and so on) – along with other valuable non-security features, like reporting or uptime service level agreements (SLAs).

The three-tier approach is not solely due to the GTM efficacy of the "good-better-best" principle, either. As anyone with experience in product management at a software vendor knows, pricing can derange into a nightmarish process befitting a Kafka novella. Maintaining each security feature as a separate line item may work for a small software startup but becomes a bottleneck at scale.

Our impression is that the federal government believes that if software vendors begin breaking out such security features diner-menu style for customers to purchase, regardless of tier, then many more companies will purchase SSO, audit trails, and so forth. We believe the federal government overestimates demand among corporations for those features. Furthermore, if the federal government begins requiring software vendors to offer these security features as separate line items, we should expect longer sales cycles (possibly slowing growth sector-wide), and, potentially, the incentive for software vendors to charge their customers even more for security. For instance, if a software vendor offers the "enterprise" tier that includes SSO for $5,000 per month, they are unlikely to reduce the price after breaking out SSO as its own line item; instead, the customer may pay another $500 per month just for SSO, then another $300 per month for rate limiting, another $200 per month for audit trails, $100 for allowlisting, and so on. Through that lens, there is an

---

[27] Sometimes the basic tier is free. But for many companies, their free tier is not included as part of "good-better-best"; that is, they offer three paid tiers following good-better-best, with the free tier meant for hobbyists. For example, Atlassian offers a free tier for JIRA that allows up to 10 users and 2GB of storage; their next tier, "Standard," allows up to 35,000 users and 250GB of storage – a considerable jump.

even greater “tax” for security features than before – although they are now more accessible to more buyers.

The next logical move is for the federal government to require software vendors to offer SSO integration for free – but we disagree with that move, too. Aside from benefiting incumbent, for-profit SSO vendors – who have the resources to offer support services to software vendors who now must integrate with at least *some* SSO provider – this move would also create a market moat for better-resourced software vendors. Upstart or smaller software vendors must ruthlessly prioritize what features they implement; product teams often face a tradeoff between implementing “table stakes” features like SSO or granular audit trails vs. differentiating features that would help them gain market share against incumbent vendors. Requiring software vendors to offer SSO or audit logs for free would curtail smaller vendors’ ability to enter and compete in their market. Realistically, it means fewer software vendors would invest in these features in the first place.

There is also the possibility that the market “votes” with their dollars and does not buy these security features, even when available as separate line items. Or, that software vendors will “offer” SSO integration – but in the form of requiring customers to manually integrate it by themselves – or “offer” audit logs – but with extremely limited retention windows that require customers to spend even more money hoovering the logs out for longer-term storage. Does the federal government expect to validate these implementations at each software vendor? We feel that would be a profligate waste of time and energy.

12.2 What considerations do they factor into their decision?

In addition to the considerations described in Section 12.1, software companies also consider market segmentation. Software manufacturers will use market segmentation to offer a single product to a wide market and this involves explicitly withholding features from some customers purely to entice them to upgrade to a more expensive version or plan – it costs software vendors little or nothing to offer a feature like SSO once they build it. Offering a single lower price would mean they'd lose out on capturing value from customers that could pay more. Offering a single higher price would mean much of the market would either forego purchasing the product or choose a cheaper competitor.

If an enterprise is likely to require a feature that a mid-market or prosumer customer doesn't, that feature is strongly likely to be used for market segmentation. Very often the feature is already built and costs the vendor nothing or very little but is left out of the less expensive plans so that enterprises must upgrade. Requiring security features in enterprise compliance requirements without similar requirements for smaller firms or individuals may result in the features becoming less accessible overall.

Conclusion

For the reasons cited herein, we encourage CISA to incorporate our recommendations to nurture a future in which software is safe, secure, and resilient. We hope these recommendations prove that security and business goals are not at odds; indeed, there are ample opportunities to nourish software velocity while watering the roots of resilience, too.

We urge the CISA to avoid busywork and theater; action bias is tempting but will not help us against our adversaries. Instead, all stakeholders in the software security problem space must work together to make the secure way the fast, easy – and yes, business-compatible – way.

Sincerely,

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[28] and Patrolaroid[29]. 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[30], 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.

---

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

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

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