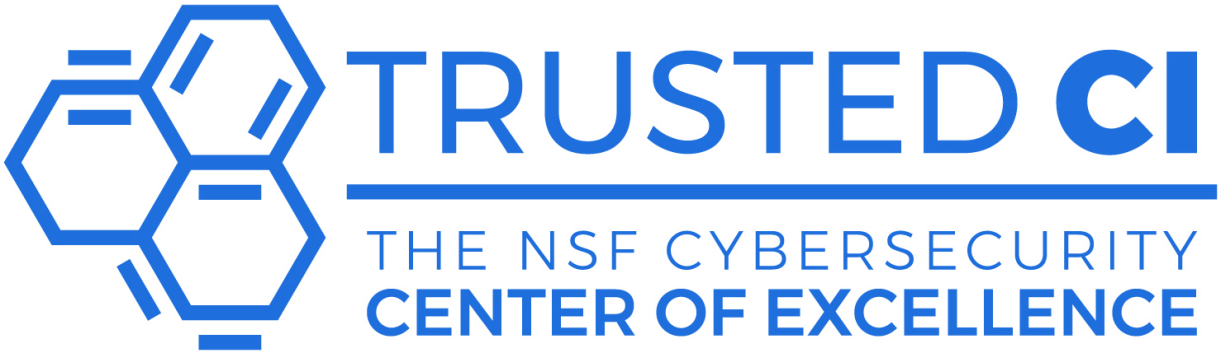

The State of the Scientific Software World: Findings of the 2021 Trusted CI Software Assurance Annual Challenge Interviews

September 29, 2021

Status: Final Report v1

Distribution: Public

Andrew Adams, Kay Avila, Elisa Heymann, Mark Krenz, Jason R. Lee,

Barton Miller, and Sean Peisert

About the 2021 Trusted CI Annual Challenge Team

The 2021 Annual Challenge team is a collaborative effort of Trusted CI members from Indiana University, Lawrence Berkeley National Laboratory, the National Center for Supercomputing Applications, the Pittsburgh Supercomputing Center, University of Wisconsin, Madison.

About Trusted CI

The mission of Trusted CI is to provide the NSF community with a coherent understanding of cybersecurity, its importance to computational science, and what is needed to achieve and maintain an appropriate cybersecurity program.

Acknowledgments

In support of this effort, Trusted CI gratefully acknowledges the input from the following software development teams who contributed to this effort:

- FABRIC: Charles Carpenter, Yongwook Song, Michael Stealey, Maruicio Tavarres, Komal Thareja
- The Galaxy Project: Enis Afghan, Dannon Baker, Nate Coraor, Marius van den Beek
- High Performance SSH/SCP (HPN-SSH) by the Pittsburgh Supercomputing Center (PSC): Chris Rapier
- Open OnDemand by the Ohio Supercomputer Center: Alan Chalker, Eric Franz, Jeff Ohrstrom
- Rolling Deck to Repository (R2R) by Columbia University
- Vera C. Rubin Observatory: Yusra AlSayyad, Frossie Economou, Wil O'Mullane

This document is a product of Trusted CI. Trusted CI is supported by the National Science Foundation under Grant #1920430. For more information about Trusted CI, please visit: http://trustedci.org/. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Using & Citing this Work

Cite this work using the following information:

Andrew Adams, Kay Avila, Elisa Heymann, Mark Krenz, Jason R. Lee, Barton Miller, and Sean Peisert. "The State of the Scientific Software World: Findings of the 2021 Trusted CI Software Assurance Annual Challenge Interviews," September 2021.
https://hdl.handle.net/2022/26799

1 Executive Summary

In 2021, Trusted CI is conducting our focused "annual challenge" on the security (sometimes called "assurance") of software used by scientific computing and cyberinfrastructure. The goal of this year-long project, involving seven Trusted CI members, is to broadly improve the robustness of software used in scientific computing with respect to security. It is our hope that this report will help scientific software projects better understand some of the most important gaps in the security of scientific software, and also

to help policymakers understand those gaps so they can better understand the need for committing resources to improving the state of scientific software security. Ultimately, we hope that the report will support scientific discovery itself by shedding light on the risks incurred in creating and using scientific software.

The study team spent the first half of the 2021 calendar year engaging with developers of scientific software to understand the range of software development practices used and identifying opportunities to improve practices and code implementation to minimize the risk of vulnerabilities. In this document, intended for consumption by software project leads and for scientific policy developers both inside and outside NSF, the study team presents a summary of its findings from the project engagements.

In general, the team found that very few of the groups we interviewed have the support to implement complete software security programs, as defined by traditional industry and government practices, and all could benefit from additional guidance and funding resources specifically supporting secure software development. This guidance and funding can help groups better understand the scope of a software security program, and enable secure development processes from the outset, rather than as an add-on, only once a project seems to be large enough to warrant it. At a high level, the team identified challenges that developers face with robust policy and process documentation; difficulties in identifying and staffing security leads, and ensuring strong lines of security responsibilities among developers; difficulties in effective use of code analysis tools; confusion about when, where, and how to find effective security training; and challenges with controlling source code and external libraries to ensure strong supply chain security.

2 Background

The target audience for this report is policy makers and software project leads who want to understand more about the state-of-the-practice in software security in the cyberinfrastructure research community. The software projects that we interviewed for this guide generally are ones that have a higher level of exposure (for example, they provide a user-facing front end that is exposed to the Internet) and therefore are subject to the common threads of vulnerabilities and attacks facing such software. We examined several scientific software projects, looking for commonalities among them related to security. Much of our focus was on both procedures and practical application of security measures and tools. Robust software security takes explicit focus — a focus that isn't always forefront on the mind of developers of software used in scientific computing. However, we found that there were important gaps in software security that could be ameliorated through careful attention to and restructuring of process and organization, the appropriate use of tools and

systems used in secure software development, and the greater availability and use of training appropriate to scientific software developers.

This study was conducted in the context of scientific software development where the main pressure is, of course, to produce tangible scientific progress. These projects often start from small efforts of a single graduate student or staff member and grow organically. As such, there is likely no formal design or clear roadmap for the evolution and distribution of the software, let alone security as a consideration in the conceptualization and design of the software. These science projects, even large ones, are generally based on grant budgets that include little of any support for security. Project leaders often view the budget as a zero-sum issue: if more money is allocated to security issues, then less money will be available for science. The urgency to produce scientific results can overwhelm any concerns about security threats, even though this perspective can cost the project more time and dollars in the long run than if security were incorporated from the beginning.

This report was written by team members of Trusted CI, the NSF Cybersecurity Center of Excellence. The team included security experts from various parts of the discipline including operational security, secure software development, and security research.

3 Process

When looking for projects to interview, we focused on projects that had worked with Trusted CI in the past and thus we already had established contacts with. Additionally, we looked for those projects meeting our criteria of developing widely used software that has exposed attack surface, and aimed to have representation across a variety of NSF directorates.

After reaching out to nine different projects, we received a positive response from and interviewed six of them. These projects are funded by the BIO, CISE, GEO, and MPS directorates of NSF.

3.1 Caveats

Although four of the NSF directorates were represented, we were unable to interview projects from the ENG, SBE, or EHR directorates. It's possible though unlikely that projects in these directorates would have given us very different responses than our sample projects.

We chose projects that already had some familiarity with Trusted CI in order to increase the odds of people being willing to work with us. However, this may also have inadvertently biased our results since we spoke with projects that were already interested enough in security to be interacting with Trusted CI.

Finally, we left it up to our initial contacts to guide us toward the right people working in software development within each project. We realize that some projects have disparate software efforts and the views expressed were necessarily limited to the experiences and perspectives of the few individuals we were able to talk to.

3.2 Interview Process

We began our interview process with each project by sending our interviewees a link to a Google Drive document with pre-interview questions for them to answer before we spoke with them. Some of these questions were tailored toward individuals regarding their professional role, such as asking for their background and role in the project, as well as which programming language most heavily influenced them. The majority asked about the software developed by the project, including whether the project uses a style guide, how many developers work on the project, who the expected audience is for the code, and what other software the code depends upon. We also asked for either a code sample or a link to code repository, as well as for an estimate of the number of lines of overall code and list of languages used.[1]

Our team then reviewed these answers prior to the interview, and as many Trusted CI members as possible joined each interview. We also recorded the interviews so that any members unable to attend could later review the videos.

The number of individuals from the project and the pace of communication influenced the length of the interviews, but on average they tended to last around an hour and a half. We grouped our questions into different categories, and based on how the participants answered, would also follow-up questions to clarify or further explore a subject.

These categories included questions about the project's purpose, training received, software practices, processes and procedures, testing and scanning, code reviews, code release and tracking, dependencies and downstream projects, and general open questions about what practices are working well and what current security concerns or open questions the individuals have.

4 Findings

[1] Projects were asked to provide this by running the "cloc" script if possible, available at https://github.com/AlDanial/cloc.

4.1 Positive Findings

We start by highlighting good practices present in the projects we interviewed. It is our goal to relate success stories as well as practices that we found that were more concerning in the current state of secure scientific software development.

One success story we found is that all projects incorporated code repositories and some form of version control on their software. It's possible that the presence of the latter could be an artifact of the former, i.e., GitHub strongly encourages versioning in order to track branches/releases. Regardless of the reason, version control is a key component in conveying/managing security updates, and thus, necessary. Moreover, all interviewees used some manner of bug tracking software, e.g., Jira, RT and GitHub. Recording and tracking bugs/vulnerabilities is crucial for a strong security-minded development environment.

Most of the projects we reviewed also used modern languages (e.g., Python) which avoid many of the problems and security issues associated with older languages like C and C++ (e.g., buffer overflows). Projects also tended to use tools such as dependency tracking in creating their code, which avoided some common security issues.

Finally the fact that all of the software projects we interviewed used standard libraries for cryptography instead of attempting to create their own ciphers, routines, or protocols is a positive, security-focused development decision. Cryptography is hard, but more importantly, it takes an extensive peer-review before it can be trusted. There are many examples in the real world of software developers not abiding by this axiom with disastrous results.[2]

4.2 Management Process

During the analysis of the management processes of the various software projects we reviewed, we found some commonalities among them.

Threat Modeling: Threat modeling in software development refers to how security threats are conceptualized and considered in the software being developed. We looked to see if threat modeling was being performed, and in particular was being performed in a way that outlined the types of threats for which specific mitigations must be designed, implemented, and maintained. Traditionally, at one end of the spectrum, threat modeling is a formal process where tabletop exercises are considered, examining what level of privilege might be necessary to attack a system at each node, and what the consequences of that privilege

[2] https://threatpost.com/weak-homegrown-crypto-dooms-open-smart-grid-protocol/112680/

might be. At the other end of the spectrum, threat modeling can be less formal where there is a simple discussion of the different risks associated with such aspects as particular users, servers, libraries, and binaries considered at a higher level of risk. In most of the scientific software projects that we evaluated, we did not observe threat modeling being performed, either formally or informally.

Documentation: The second point that we noticed across the projects was with the lack of documentation and where the documentation was out-of-date for the version of code that was in production. Documentation can include:

- Process and policy documentation, such as onboarding and offboarding of developers.
- Policies regarding who is allowed to submit and approve commits, and what process must be gone through before approval is granted.
- Code and development standards, and design documents.

We observed situations where even when documentation existed, sometimes developers chose not to use the project's own developer guides, and the use of those guides was not enforced.

We also observed "documentation rot," where documentation went out of date and was not updated in a way that made it less useful over time, and was gradually ignored and did not serve its original purpose.

Language Choice: Another observation was finding code that is not written in modern, type-safe languages. Examples of this include code, particularly older code that is either written in languages that are not type safe (e.g., C, C++), inherently insecure (e.g., PHP), or simply out of date (e.g., Python 2).

Maintenance: There were several significant observations revolving around the maintenance of code for the project. In the projects that we examined, the code was almost never reviewed internally or audited by an external entity. One of the best and easiest ways to secure code is to have it reviewed. Many times the coder who creates the section of code is focused on the section they are writing and do not understand how it interacts with other sections of the project or what other programmers may expect from their code. These errors are usually around things like sanitized input or unexpected return values. In some cases, there were audits, but the audits were light in nature, and in turn gave a false sense of security. This issue is closely related to what is commonly referred to as "code rot", where sections of code are never updated, but left in place unused and untested until an unexpected error occurs which then exercises that code.

Vulnerability Tracking: Related to code maintenance is communicating with users about vulnerabilities, and patching those vulnerabilities. We found that projects did not have prioritized communication channels in place to notify of security issues. Similarly, we also found that security releases were generally not separated from feature releases. The latter can often cause issues where end users may not wish to install a new feature version, perhaps due to concern about new bugs introduced with those features which may cause security or stability problems, but do wish to install urgent security patches. Another issue that was observed was that of adding security patches only to the version control HEAD, not also to branches, thus failing to capture the patches across all versions that might eventually be deployed.

Priorities: Finally, one observation, which is a classic challenge in software development but is nonetheless a critical problem to overcome, is that of developers and their leadership prioritizing feature progress over *secure* development process.

4.3 Organization/Mission

A key component of a strong security program is the existence of a single point of contact for security issues, e.g., a chief information security officer (CISO). Software development is no different, yet nearly all the projects we interviewed did not have anyone in a role responsible for managing and-or coordinating security matters, which resulted in security tasks being addressed ad-hoc. Similarly, although not as indisputable as requiring a role for security, project management was a role that few projects possessed, but most stated they wished they had a PM. And in fact, this role could easily have addressed a third issue that most of the projects illuminated, that of being forced "to do too much". Multiple developers expressed the concern that insufficient resources had been allocated to the project, or that their particular part of the project was requiring more attention then they could provide.

A different organizational issue that cropped up with some of the projects was that of using multiple silos or repositories. This decentralization was found to lead to challenges with coordination and global visibility within the project. However, it was also conveyed by others that having multiple team members acting as managers of one central repository was difficult, and in many instances caused the repository to handle merges incorrectly.

Although having many admins can be problematic, one project that displayed strong security (even though they lacked a chief information security officer), stated that they benefited from this approach by leveraging the different points of view that accompanied the larger staff working on the project. This observation seems to go against empirical evidence, and we can only postulate that perhaps someone who was security-savvy was indeed acting as a manager, but unaware their behavior could have been classified as a

security officer. That said, a more diverse team certainly can improve code reviews, and perhaps that project's strong posture was directly attributed to their ability to spot abnormalities within their code.

Also in the vein of organization, it was revealed that one of the projects had their institution's centralized IT perform a security audit of their work. We applaud them for this initiative. Oddly, though, the institutional audit reported nothing amiss. Again, we can only postulate that the institution's audit was redacted, censored, or insufficient, as it is very rare indeed to have absolutely nothing flagged during an audit. Similarly, other projects mentioned trying to leverage centralized IT, e.g., vulnerability scanning, but reported various levels of success.

Another mission-centric approach that appeared to improve the security of a project was deliberately limiting what users are allowed to do with the software/interface. Although not stated, "conservatism" in user permissions (Saltzer and Schroeder's Least Privilege principle) is a fundamental component of security, and thus, makes sense that those developers intentionally limiting the actions of users improve their security posture.

Finally, while in most cases we observed a desire for more assistance with security, in a subset of cases we observed that organizations believed they had all the tools/help needed to produce secure code. That is, they choose to pursue a "going it alone" approach. We believe that the basis for this approach is that since nothing bad had happened in the past, the future would be the same. The project believed that they already knew how to develop good code. Indeed, if a project has a strong security posture, or at least a software security engineer, then the project is already a step ahead of most. That said, how should projects make this self-assessment, since success in the past may well have been due more to luck (i.e., falling under the radar of malicious attackers) than to a sound security program? The culture of "going it alone" could prove disastrous if not well founded.

4.4 Tools

Program analysis tools are a critical resource for any software project. These tools can be static, applied to the program's code or repository, or dynamic, where they monitor the program's execution. Dependency analysis tools identify the names and version numbers of the packages, modules, and libraries on which a program depends, then reports those dependencies that have known vulnerabilities.

Many organizations use some form of static or dependency tool. The key questions that they struggle with are which tools to use and how to run these tools effectively. In addition, some organizations are looking to run tools directly on a container, but most tools are not

designed for that mode of operation. Some organizations use the tools that are integrated in GitHub. While the convenience of these integrated tools is attractive, often these integrated tools do not represent the best functionality available.

4.4.1 Static Analysis Tools

Static analysis tools are commonly employed in continuous integration (CI) workflows and help detect security flaws, poor coding style, and potential errors in the project. A primary attribute of a static analysis tool is the set of language-specific rules and patterns it uses to search for style, error-prone, and security issues in a project.

An issue with static analysis tools is that they report a high number of false positives. This is apart from the fact that there are some security issues that they are unable to detect, such as incorrect design, code that incorrectly implements the design, configuration issues, since they are not represented in the code, and complex weaknesses involving multiple software components.

Code formatting affects the number of results found by some static analysis tools. Code formatting organizes expressions into different lines without changing its functionality. For example, "minifying" code can reduce the effectiveness of some tools. (Minifying means removing extra spaces and comments, reducing variable names to a minimum, and putting the program onto a single line.) Without more common formatting of the code, some static analysis tools report a lower number of security issues.

We observed a lack of available resources to guide programming teams in the selection and use of these tools.

4.4.2 Dependency Tools

Dependency checking tools are commonly employed to ensure the continued security of a project's dependencies as new vulnerabilities are found over time. Due to their widespread use, most common languages are supported by at least one dependency tool. Examples of

dependency checking tools are Dependabot,[3] Snyk,[4] Depfu,[5] and OWASP Dependency-Check.[6]

A primary attribute of a dependency analysis tool is the sources of information it uses for vulnerabilities. As a dependency tool relies on associating a certain version of a piece of software with a vulnerability, it is essential that the tool has access to as complete a list of known security issues as possible. One of the most common sources is the National Vulnerability Database (NVD).[7] The NVD records Common Vulnerabilities and Exposures (CVEs) that are assigned to a specific vulnerability that is found in some range of versions of a piece of software. However, not all vulnerabilities are assigned a CVE, so many tools rely on other sources of information as well, like the NPM Security Advisory[8] database, which tracks vulnerabilities found in NPM packages. Some of the tools augment the NVD with other sources of vulnerability data, providing more comprehensive results.

To identify the versions of software dependencies used in a project, dependency tools scan the frameworks and libraries, including transitive dependencies, used throughout the codebase, and then compare the versions used to the versions known to have vulnerabilities. The build systems used by projects may rely on manifest files that list the dependencies required by the project and allow the build system to resolve the required versions and fetch them from a package repository automatically, rather than require a developer to do so manually.

We observed that projects often had difficulties with the subtleties of how to run a tool and selection of their options. For example, some tools can be run in multiple modes, such as directly against a code repository or locally on the user's machine. Due to differences in what dependencies and configuration information is available, these two modes will provide different results. Tool options can also have a significant effect on results. For example, the choice to include "development" dependencies can add many reported errors that are unlikely to be of value.

---

[3] Dependabot: https://dependabot.com/

[4] Snyk: https://snyk.io/

[5] Depfu: https://depfu.com/

[6] OWASP Dependency-Check: https://owasp.org/www-project-dependency-check/

[7] National Vulnerability Database (NVD): https://nvd.nist.gov/

[8] NPM Security Advisory: https://www.npmjs.com/advisories

4.5 Testing

Testing of software before its release is a critical activity. However several organizations reported testing was done in an ad hoc way. The main findings that we noted in the testing area are:

- Several organizations perform unit testing.
- Most organizations do either manual testing or a limited amount of automated testing.
- The majority of the organizations test for functionality, while only a minority went on to address the question of testing for security.
- Testing is done by the same developers that write the code, not by different personnel or organizations. This can allow the biases, misapprehensions, and blindspots of the developers to persist across testing.
- One organization reported using a form of fuzz (random) testing. The rest of the organizations did not use fuzz testing.
- Two organizations reported one-time penetration testing carried out by an external organization.

4.6 Training

Training of software developers in doing software development securely is key to ensuring that a team is capable of producing a secure system. However, several organizations reported limited, if any, use of outside training resources, and limited internal resources. Most of the groups said that their developers never received any formal security-related training, and that they learned about security in a rather informal way.

This broadly observed lack of security training was caused by a wide variety of concerns:

- Not finding training useful at all, or unable to find appropriate training.
- Security training is expensive for organizations for two reasons:
    - It is too costly to have their developers offline while they attend several-day long training sessions.
    - Companies offering training in security charge several thousand US dollars for a few days of training.
- Given that security is a wide field, encompassing areas such as software security, network security, and cryptography, it is difficult for the projects to find training focused on the areas where they have a need.
- The importance of training is not fully understood. The consequences of having their software exploited are overshadowed by the pressure to produce the next software

release. Methodologies for agile software development do not help as they emphasize short term gains and frequent releases over comprehensive planning.

It is clear that we need to find a way to identify proper topics, develop resources in these topic areas, make science teams aware of these resources, and provide wide support in their use. We know that many resources are already available, so this is an area in which progress could quickly be made.

4.7 Code Storage

Of the projects we interviewed, all of them use Git-based revision control to manage their code, although their practice in managing their code repositories varied. Each of the projects we interviewed released most of their code as open source. One of the projects couldn't release all their code due to 3rd party license requirements. Most projects used branching for their development workflows, however one did not due previous difficulties with the technique. Only half of the projects formally use security branches when there is a security issue, a practice that is beneficial to producing security patches for multiple versions of the software. None of the projects that we interviewed were running their own repository server.

Proper code storage can help with collaboration and allow others to contribute code to a project; however without proper review of submissions, vulnerabilities can be easily inserted by accident or with intent. We found that while most of the projects were reviewing code submissions or pull requests, some projects relied on ad hoc procedures or routines for these types of reviews. One of the projects mentioned that all pull requests receive public review first before being committed to main. These pull requests allow for public comment followed by a vote by approved committers using the Apache Voting Process[9] to determine if the pull request will be accepted.

A major issue we encountered was that of dealing with massive amounts of dependencies, such as libraries and 3rd-party code that the project software depends upon. Both by admission of the problem and our observation, “dependency hell” presents a significant burden to developers as well as a significant risk if the developers are not using a dependency tool to vet the libraries they choose to use, and ensure that the libraries are being actively maintained.

[9] https://www.apache.org/foundation/voting.html

Dependency tools can assist in notifying developers about emerging vulnerabilities in source code, but external libraries should also be vetted manually on a regular basis to ensure libraries are being actively maintained.

All the projects that we interviewed had to deal with 3rd-party code in some way, whether it was through libraries that were included to provide functionality, tools used for the build process or testing, or the parent software that a patch was dependent on changing. With this relationship comes additional security risk for the project. One of the questions we asked each project was about how they determine if they can trust a library and how they stay informed on that code's security. All the projects expressed some awareness of the risks involved with including 3rd-party code, but also touted the benefits. For example, a library that is designed to check user input before submitting data to a database can help to reduce the security risk of a project; on the other hand, a plugin for handling website stylesheets that also has a vulnerability allowing an attacker to view the filesystem.

As one interesting point of note, one of the projects we spoke with avoided code branches entirely. They had used them in the past, but found that developers would check out code and not merge their branches for a long time, leading to merging issues that were difficult to reconcile. This led to their practice of siloing development so that each developer had their own set of code they worked on and in small sections.

4.8 Reticence to Use Newer Cybersecurity Techniques

During our discussion with one project, they expressed the need for being able to securely store passwords and credentials for accessing systems, databases, websites and so on among a group of developers. Some developers expressed concern over how password managers worked and were worried about the risk of the password management provider seeing their passwords or that all the passwords were kept in one place that could be compromised.

Although all projects used 2FA where it was required such as on GitHub and where their institution required 2FA for authentication, there was weak adoption of it in places where it can help, such as access to a special VPN. One project went so far as to say that they would rather not use 2FA and that it was common to avoid 2FA where possible within their scientific field.

One of the questions we asked all teams was how they are receiving notifications about security vulnerabilities to third-party software that they are using. About half the projects expressed that they are receiving notifications from some process, whether it was via mailing lists, requires.io, or Dependabot for checking library dependencies; the other half

said that they did not have an official plan for receiving vulnerability notifications. One project said that they don't receive vulnerability notifications for their library dependencies because there are too many of them.

5 Conclusions

In conducting this study and drafting this report, it was our hope to learn what the most important gaps are in the security of scientific software to serve as the basis for understanding what the most important solutions appropriate to scientific software development might be. Given the insights that the study has revealed so far, Trusted CI will be developing a guide consisting of software security solutions targeted toward anyone who is either planning or has an ongoing scientific software project that needs a security plan in place. This guide will be intended to present a "best practices" approach to the software lifecycle and will be available at https://trustedci.org. Based on our findings, science projects struggle to find appropriate training, and we will work on addressing that gap both through newly written prose as well as pointers to existing guidance not specifically tailored to the scientific community but useful for it nonetheless.

It is our hope that both this report and the solutions guide will help scientific software projects better understand some of the most important gaps in the security of scientific software, and also to help policymakers understand those gaps so they can better understand the need for committing resources to improving the state of scientific software security. Appropriately tailored to a project, security need not be viewed as a zero-sum effort that takes resources away from scientific advancement. In reality, a laser focus producing scientific results at the expense of appropriate attention to security can ultimately cost projects more time and dollars than if security were incorporated from the beginning. We believe that education of software developers greatly reduces the cost to invest in these practices, whether those developers are dedicated project staff or temporary graduate students. Knowledge is power, and appropriately educated individuals can advocate for the benefits of secure software approaches by justifying how initial investment can save resources in the end.

Ultimately, this work by Trusted CI shares the same goal that drives projects to write software in the first place: advancing science. We hope that both documents will support scientific discovery itself by providing guidance around how to minimize possible risks incurred in creating and using scientific software.