Report from the Steering Committee Meeting on Open Source Software Security

July 2022

ORGANIZERS

NATIONAL SCIENCE FOUNDATION

Nina Amla

Robert Beverly

Jeremy Epstein

Sol Greenspan

James Joshi

Juliana Nazaré

Daniela Oliveira

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

Jon Boyens

Cherilyn Pascoe

Matthew Scholl

Kevin Stine

STEERING COMMITTEE MEMBERS

Abhishek Arya

Principal Engineer and Head of Google Open Source Security Team

Abhishek Arya is a Principal Engineer and head of the Google Open Source Security Team. His team has been a key contributor to various security engineering efforts inside the Open Source Security Foundation (OpenSSF). This includes the Fuzzing Tools (Fuzz-Introspector), Supply Chain Security Framework (SLSA, Sigstore), Security Risk Measurement Platform (Scorecards, AllStar), Vulnerability Management Solution (OSV) and Package Analysis project. Prior to this, he was a founding member of the Google Chrome Security Team and built OSS-Fuzz, a highly scaled and automated fuzzing infrastructure that fuzzes all of Google and Open Source. His team also maintains FuzzBench, a free fuzzer benchmarking service that helps the community rigorously evaluate fuzzing research and make it easier to adopt.

David Brumley

CEO and Co-Founder of ForAllSecure and Full Professor at Carnegie Mellon University

Dr. David Brumley is CEO and co-founder of ForAllSecure and a full professor at Carnegie Mellon University. His accomplishments include winning the DARPA Cyber Grand Challenge, a United States Presidential Early Career Award for Scientists and Engineers (PECASE) from President Obama, a Sloan Foundation award, a Carnegie Science Award, several patents, numerous academic papers, a DEFCON black badge, and mentoring one of the most competitive hacking teams in the world.

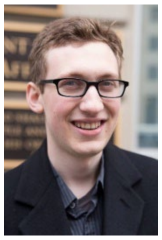
Deirdre Connolly

Cryptographic Engineer at the Zcash Foundation

Deirdre Connolly is a cryptographic engineer at the Zcash Foundation. She works on secure implementations of cryptographic software with an eye on privacy applications, misuse-resistance, and an eye on quantum adversaries. She obtained her BS from MIT in 2009.

Alex Gaynor

Deputy Chief Technologist for Security at the Federal Trade Commission



Alex currently serves as Deputy Chief Technologist for Security at the Federal Trade Commission. Prior to that he was at the United States Digital Service. He has previously worked at Alloy, Mozilla, and another stint at the United States Digital Service. Alex has a long history of involvement in the open source community. He is a core developer of the Python Cryptographic Authority and previously has served as a member of the board of directors of both the Python and Django Software Foundations. Alex lives in Washington, DC and likes delis and bagels.

Royal Hansen

Vice President of Privacy, Safety, and Security at Google

Royal Hansen is Vice President of Privacy, Safety & Security at Google, where he is responsible for driving strategy and implementation in these areas across the company's technical infrastructure and product lines. There, he was responsible for solutions protecting the security and integrity of the company's technology systems and the customer, business, and employee information they processed. Before American Express, Royal served as both the Managing Director, Technology Risk and the Global Head of Application Security, Data Risk and Business Continuity Planning at Goldman Sachs. Royal was also previously at Morgan Stanley and Fidelity Investments, where he managed Enterprise IT Risk, Application Security and Disaster Recovery. Royal began his career as a software developer for Sapient before building a cyber-security practice in the financial services industry at @stake, which was acquired by Symantec. Royal holds a BA in Computer Science from Yale University. He was awarded a Fulbright Fellowship in information sciences and Arabic language study, which he completed at the United Arab Emirates University.

Sumana Harihareswara

Project Manager, Programmer, and Trainer at the Python Software Foundation's Packaging Working Group and Founder of Changeset Consulting

Sumana Harihareswara is a project manager, programmer, and trainer who leads a consultancy working with open source software projects and maintainers. She led the rollout of the next-generation PyPI.org and pip resolver, and has worked on HTTPS Everywhere, Autoconf, Mailman, MediaWiki, and several other open source projects across industry, academia, nonprofits, and volunteer settings. She works with the Secure Systems Lab at New York University on securing the software supply chain in Python and is a member of the Python Software Foundation's Packaging Working Group. She is writing a book on rejuvenating and managing legacy open source projects and teaches workshops in maintainership skills. She earned an Open Source Citizen Award in 2011 and a Google Open Source Peer Bonus in 2018. She lives in New York City and founded Changeset Consulting in 2015.

Angelos Keromytis

Professor, John H. Weitnauer Technology Transition Endowed Chair, and Georgia Research Alliance (GRA) Eminent Scholar at the Georgia Institute of Technology

Dr. Angelos Keromytis is Professor, John H. Weitnauer Technology Transition Endowed Chair, and Georgia Research Alliance (GRA) Eminent Scholar at the Georgia Institute of Technology. He is an ACM and IEEE Fellow, and President of Voreas Laboratories Inc and Aether Argus Inc, two Georgia Tech technology spinoffs. He has served as Program Director with the National Science Foundation and Program Manager at DARPA. His field of research is systems and network security, and applied cryptography.

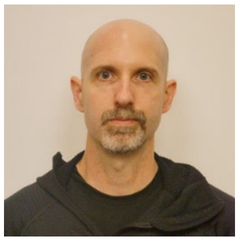
Mathias Payer

Associate Professor, École Polytechnique Fédérale de Lausanne (EPFL)

Mathias Payer is a security researcher and associate professor at EPFL, leading the HexHive group. His research focuses on protecting applications in the presence of vulnerabilities, with a focus on memory corruption and type violations. He is interested in software security, system security, binary exploitation, effective mitigations, fault isolation/privilege separation, strong sanitization, and software testing (fuzzing) using a combination of binary analysis and compiler-based techniques.

Eric Rescorla

Chief Technology Officer, Firefox at Mozilla



Eric Rescorla is Chief Technology Officer, Firefox at Mozilla, where he is responsible for setting the overall technical strategy for the Firefox browser. He has contributed extensively to many of the core security protocols used in the Internet, including TLS, DTLS, WebRTC, ACME, and QUIC. He was editor of TLS 1.3, which secures over 50% of web sites. In order to remove barriers to encryption on the web, he co-founded Let’s Encrypt, a free and automated certificate authority that now issues more than a million certificates a day, and helped HTTPS grow from around 30% of the web to over 80%. Previously, he served on the California Secretary of State's Top To Bottom Review where he was part of a team that found severe vulnerabilities in multiple electronic voting devices.

Nikhil Swamy

Senior Principal Researcher at Microsoft Research

Nikhil is a Senior Principal Researcher at Microsoft Research (MSR) at its headquarters in Redmond, USA, where he has worked since 2008. His expertise is in programming language design and semantics, formal verification, and software security. He is perhaps best known for his work on F*, a proof-oriented programming language. Verified cryptographic algorithms, communication protocols, blockchain components, and network virtualization software produced in F* are deployed in the Linux kernel, in Windows, in the Microsoft Azure cloud, in the Firefox web browser, and several other industrial software components, improving computer security and reliability for billions of users every day.

David A. Wheeler

Director of Open Source Supply Chain Security at The Linux Foundation

Dr. David A. Wheeler is an expert on open source software (OSS) and on developing secure software. His works on developing secure software include "Secure Programming HOWTO", the Open Source Security Foundation (OpenSSF) Secure Software Development Fundamentals Courses, and "Fully Countering Trusting Trust through Diverse Double-Compiling (DDC)". He also helped develop the 2009 U.S. Department of Defense (DoD) policy on OSS. David A. Wheeler is the Director of Open Source Supply Chain Security at the Linux Foundation and teaches a graduate course in developing secure software at George Mason University (GMU). Dr. Wheeler has a PhD in Information Technology, a Master's in Computer Science, a certificate in Information Security, a certificate in Software Engineering, and a B.S. in Electronics Engineering, all from George Mason University (GMU). He is a Certified Information Systems Security Professional (CISSP) and Senior Member of the Institute of Electrical and Electronics Engineers (IEEE). He lives in Northern Virginia.

Laurie Williams

Distinguished University Professor in the Computer Science Department at North Carolina State University

Laurie Williams is a Distinguished University Professor in the Computer Science Department at North Carolina State University (NCSU). Laurie is a co-director of the NCSU Secure Computing Institute, the NCSU Science of Security Lablet, and the North Carolina Partnership for Cybersecurity Excellence (NC-PaCE). Laurie's research focuses on software security; agile software development practices and processes, particularly continuous deployment; and software reliability, software testing and analysis. Laurie is an IEEE Fellow and an ACM Fellow. Laurie received her Ph.D. in Computer Science from the University of Utah, her MBA from Duke University Fuqua School of Business, and her BS in Industrial Engineering from Lehigh University. She worked for IBM Corporation for nine years in Raleigh, NC and Research Triangle Park, NC before returning to academia.

CONTENTS

EXECUTIVE SUMMARY

The Steering Committee members met on May 20, 2022, to discuss four main themes related to the security of open source software: (1) trust and safety, (2) memory-safe programming languages, (3) dependency management, and (4) behavioral and economic incentives to secure the open source software (OSS) ecosystem. These themes were uncovered by NSF and NIST staff based on position statements submitted by each Committee member.

For each theme, the structured discussion focused on answering the following questions:

1. What is the problem? (i.e., define the problem)
2. What about the problem do we not yet understand?
3. Where are the boundaries of the problem? Are there any constants that cannot be changed?
4. Who are the key stakeholders (e.g., specific sectors or people) to get involved?

TRUST AND SAFETY

While coding mistakes leading to vulnerabilities are the most common type of weakness in OSS projects, socio-technical vulnerabilities present real and challenging threats. This type of vulnerabilities can be decomposed into the following categories:

1. Cyber social-engineering attacks in OSS code repositories, e.g., code contributions that attempt to insert vulnerabilities by pretending to offer bug fixes or new features.
2. Attacks against the code repositories themselves, seeking to modify code or packages in surreptitious ways (e.g., bypassing code review).
3. OSS developers/maintainers who started as or became malicious (e.g., bribed to insert a vulnerability).
4. Malicious developer/maintainer ascendency leading to OSS project takeover (e.g., original developer paid to relinquish control).

Socio-technical vulnerabilities in OSS projects is a challenging and under-studied topic. The Committee recommends that the following challenges be tackled. First, there are no good metrics on the prevalence and impact of such type of attacks and it is hard to distinguish among: (1) unintentional vulnerabilities, (2) vulnerabilities inserted by an external attacker subverting the project, and (3) vulnerabilities inserted by a developer/maintainer. The likelihood/prevalence decreases from (1) to (3), but severity and difficulty of detection increases. Second, there are no models for detection of malicious actors in the context of OSS project development and maintenance and no best practices for what to do once a bad actor is detected. For example, should a project keep the bad (developers/maintainers/software) out to begin with, or is it acceptable to rely on detection and recovery/remediation of the process? It is likely that such socio-technical issues are more pronounced in OSS ecosystems. For example, many critical OSS projects rely on a single maintainer, leading to a single point of failure of the OSS project, e.g., the developer might go rogue or fall for a social engineering attack. Finally, the community still does not know how to incentivize developers (especially in smaller projects) to adopt software security practices.

MEMORY-SAFE PROGRAMMING LANGUAGES

The software vulnerability landscape has not significantly changed in the last decade despite various innovations in exploit mitigations (e.g., Address Space Layout Randomization – ASLR- and Control Flow Integrity -CFI-) and software testing techniques (e.g., fuzzing and sanitization). Several recent reports[1] indicate that roughly 70% of all software vulnerabilities continue to be memory safety issues arising from the use of memory unsafe programming languages such as C and C++. How can we (gradually) transition software developers to use memory-safe languages?

The Committee discussed several problems that have impeded this transition. Getting developers enthusiastic to learn and write in a new programming language while meeting their performance expectations is hard. Also, there is a large amount of legacy code and converting it to memory-safe programming languages will likely be a manual, cumbersome process that might take years.

To make significant progress on this front, we need a better understanding of clear migration path for projects, tactical community interactions, and sustainable funding mechanisms and automation. While progress on transitioning software to memory-safe programming languages will mitigate memory corruption vulnerabilities, there is still the continued need for securing coding practices to prevent other classes of errors, such as design flaws, information disclosure, or Denial of Service -DoS- attacks.

The Committee believes that the community needs to change the fundamental economics of software development. We need to incentivize developers to write secure code and make security a part of their training and course curriculums. The transition to memory-safe programming languages is several years away, but we can speed this process up by increasing investments in usability, interoperability, and automation.

DEPENDENCY MANAGEMENT

Software reuse is on the rise. Instead of writing code for a needed functionality themselves, developers are increasingly relying on external software libraries to fill that gap. This practice enables modularity, increases code reuse, and reduces software development costs. Similarly, if a software dependency is well developed and well maintained, then the overall security of the software will benefit. One example is

[1] Catalin Cimpanu. Microsoft: 70 percent of all security bugs are memory safety issues. February 11, 2019. ZDNet (https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/).

widely used cryptographic libraries, which frees developers from having to implement their own cryptographic code. The downside of this practice of code reuse is that software dependencies become liabilities that need to be carefully managed. Any third-party code that is included in a software project becomes part of the trusted computing base of the developed software yet remains under external control.

Key challenges of leveraging external software dependencies are: (1) tracking all dependencies, (2) keeping the dependencies up to date, and (3) detecting if any used feature has changed or requires an update. While bug fixes should be applied immediately, a developer may hold off on updates with feature changes as they will require changes to their code.

Software dependency management is evolving, and different programming language and runtime environments have slightly different concerns and challenges. We must better understand the scope of the problem, such as how deep dependency chains are or how fast software changes. Alongside, we must define metrics on the security of software dependencies and develop technological solutions that empower developers to keep track of their externally included code.

The Committee agrees that both research into solutions and metrics, along with development of better tooling will be necessary to address these challenges. Dependency management is an important and challenging problem that needs to be solved by integrating the solutions gradually into the developer workflow, enabling them to react to challenges whenever dependencies are updated.

BEHAVIORAL AND ECONOMIC INCENTIVES TO SECURE THE OPEN SOURCE SOFTWARE ECOSYSTEM

How do we incentivize cyber-resiliency in the teams of the volunteers that build open-source software (OSS)? Resiliency includes both creating more secure software and creating robust systems for maintaining that security throughout the software lifecycle.

The Linux Foundation reports[2] that a key challenge is that OSS contributors are volunteers, and that efforts focused on increasing the time contributors spend on security are unlikely to be welcome. Some studies[3] also show that most OSS developers have primarily non-monetary motivations, thus purely economic incentives (e.g., paying for better maintenance) are likely insufficient.

Despite indications that monetary compensation alone does not necessarily drives developers towards security practices, we do not know what incentive structures would work. Thus, the Committee's

[2] The Linux Foundation & The Laboratory for Innovation Science at Harvard. Report on the 2020 FOSS Contributor Survey, 2020.

[3] Dr. Oliver Alexy. Free Revealing: How Firms Can Profit From Being Open (Innovation und Entrepreneurship). Gabler Verlag, 2009.

consensus is that the U.S. needs to fund and conduct more research into the behavioral and economics aspects of OSS software development security in at least three areas:

1. **Software development lifecycle**. What incentive structures would encourage volunteer OSS developers to incorporate better software security practices? And how do we provide this information to developers? During the software lifecycle, what incentive models encourage active and speedy remediations when security issues are discovered? What if there is only one primary developer, and they leave the project? Are incentives purely contributor-based, or are there other incentive models, such as matching security experts to OSS projects?

2. **Using paid professional development/assistance wisely**. How can we leverage the limited amounts of government, industry, and professional assistance to maximize our overall security posture? How do we assess the security of OSS dependencies beyond the vendor-independent fashion? And how do we ensure that we encourage security practices in small startups rather than predominantly in large existing businesses? Can we identify the weakest links, shore up security, and then rinse and repeat with the next highest priority? For example, companies often professionally support OSS projects (e.g., through grants, computing resources, or paid on-going development), but cannot support every single project they depend upon. How does the U.S. government, as well as individual companies, get the most bang for their buck out of the limited professional funding and development we can put behind OSS?

3. **Informed choice**. Today's OSS consumers (commercial users, governments, and even other OSS projects) do not heavily weigh security when making choices. How do we make security a priority feature in software development processes?

These challenges also require that we create metrics to estimate whether security is improving or not. We do have some data, such as the total number of vulnerabilities and the total number of new OSS projects created each year. However, we do not have a code index for security to measure whether we are trending to more secure or less secure software on average over time.

THEME 1: TRUST AND SAFETY

DISCUSSION LEAD: LAURIE WILLIAMS

SCRIBE: ANGELOS KEROMYTIS

WHAT IS THE PROBLEM?

The Committee focused on the issue of socio-technical vulnerabilities of open source software (OSS) projects, decomposed to several different threat vectors/weaknesses:

1. Cyber social-engineering attacks in OSS code repositories (e.g., code contributions that attempt to insert vulnerabilities by pretending to offer bug fixes or new features).
2. Attacks against the code repositories themselves, seeking to modify code or packages in surreptitious ways (e.g., bypassing code review).
3. OSS developers/maintainers who started as or became malicious (e.g., bribed to insert a vulnerability).
4. Malicious developer/maintainer ascendency in a collaborative project or OSS project takeover (e.g., original developer paid to relinquish control).

The group discussed the full scope of such socio-technical vulnerabilities in OSS, trying to bound the problem. In particular, there was lively discussion on the issue of coding mistakes that lead to vulnerabilities, which is viewed as by far the most common case. While there is a danger of over-expanding the problem scope, there was general agreement that finding ways to incentivize developers (especially in smaller projects) to adopt software security/quality tools and workflows is an open problem.

While many of the same issues can manifest in commercial software development, vendors typically (but not always) do some type of background check of employees. Differently from commercial software development, OSS is driven by reputation, desire to participate, and quality of contributions, and these socio-technical issues manifest more acutely in such environment.

Furthermore, many critical/widely used OSS projects/packages rely on a single maintainer, which increases the likelihood of attacks exploiting the weaknesses described above. For example, the scenario of an OSS contributor going rogue and eschewing any type of code peer review for components they are responsible for is not far-fetched. Thus, in such resource-limited projects, how do we incentivize good processes? Furthermore, many projects are critical but tiny (e.g., many Node Package Manager -NPM- projects have zero or one functions), which makes it hard to motivate and justify significant investment of resources, including time and effort, by the maintainers.

WHAT ABOUT THE PROBLEM DO WE NOT YET UNDERSTAND?

The Committee agreed that we do not have models for detection of malicious actors in the context of OSS project development and maintenance. Research and products on insider threats are generic and do not address the specifics of software development, much less OSS development.

The Committee is in consensus that it is hard to distinguish among: (1) unintentional vulnerabilities, (2) vulnerabilities inserted by an external attacker subverting the project, and (3) vulnerabilities inserted by a developer/maintainer. The likelihood/prevalence decreases from (1) to (3), but probable severity and difficulty of detection increases. What are the different classes of mechanisms that are applicable in each case, and how can they be made practical to wide adoption and use? One challenge to be confronted is that there are no good metrics on the prevalence and impact of these types of attacks.

It is also hard to understand the "good" vs "bad" software packages, or even compare functionalities. Generally, developers do not understand the security/reliability guarantees they get from shared packages. This issue ties into Theme 4 - Behavioral and Economic Incentives to Secure the Open Source Software Ecosystem.

Abstractly, establishing and managing trust seems key to addressing some of these issues. What does trust mean here, and to what does trust apply? Rather than applying trust only to the concrete outcome, perhaps we should build trust in the development process.

The Committee posed the following question on resilience: *Do we keep the bad (developers/maintainers/software) out to begin with, or is it acceptable to rely on detection and recovery/remediation of the process?* We do not understand the tradeoffs involved, especially when downstream users, who may be software vendors/developers, are concerned.

If we assume that there will be compromise and that we need to be in a better position to react, what can we do to harden OSS projects/systems and their dependencies?

What processes and metrics can there be to understand the trust chain of a package and its development/maintenance process? We cannot secure everything, so it is necessary to prioritize projects. If a developers' reputation and/or credibility is an input to these metrics, how can it be measured and formalized? Can developers' reputation be made robust against the types of adversarial behavior identified above (e.g., patient malicious contributors)?

What does the implicit web of trust on sharing ownership and maintenance of OSS projects look like? Is there value in making this web of trust explicit? What would be the best way to provide guidance on how an individual OSS maintainer can do this in a responsible way (see Theme 4 discussion)? "Artificial Intelligence (AI) for code" is starting to be adopted, currently assisting code development, but eventually perhaps leading to completely automatically generated code. How does "trust" translate in such scenario, not just for training data (inputs to the AI model), but on the output (i.e., produced code)?

WHERE ARE THE BOUNDARIES OF THE PROBLEM? ARE THERE ANY CONSTANTS THAT CANNOT BE CHANGED?

Generally, it was unclear to the Committee how far down the software supply chain one could or should go in addressing socio-technical vulnerabilities. For example, OSS projects are widely used in closed source but very popular ecosystems (e.g., app stores and associated platforms). What is the interface between the open source and closed source software project worlds? While it would be great to have high-quality and high-assurance open source ecosystems, it is probably unrealistic to expect that end users' (e.g., a cellphone consumer) preferences could be influenced significantly by such concerns.

A big constraint is the large volume of software written in legacy code, which is unlikely to be re-written (and its dependencies migrated) anytime soon (see Theme 2 - Memory-safe Programming Languages for details) without some technological breakthrough, such as high-quality software translation.

Often, there is close integration between package managers and programming language runtimes (e.g., NPM and JS/NodeJS[4]). While there are efforts to create universal package management tools, wherein various conceivable solutions could be integrated (e.g., Software Bill of Materials - SBOM - propagation/handling), in at least the near- and medium-term, any solution would have to deal with heterogeneous environments, as well as different rates of development across dependent packages.

Ultimately, OSS developers get to choose the programming language and runtime environment to use, thus "memory safety" may not be something that they can be induced or incentivized to do. Are there solutions that can work in the presence of security-obstinate or disinterested (but not malicious) developers?

When it comes to certain trust-building techniques, identity would appear to play a key role. How does that interact with privacy of these same developers? Furthermore, OSS development is global in nature; and yet, we live in an increasingly balkanized world. Is it possible to reconcile security-motivated restrictions (e.g., countries with foreign assets prohibition) with openness?

It is also important to encourage the use of multi-factor authentication to limit the scope of exploitation of social engineering attacks, at least for the low-level (or most common) threats, especially against weakness number 2 listed above: attacks against the code repositories themselves, seeking to modify code or packages in surreptitious ways.

[4] An open-source JavaScript runtime environment that executes JavaScript code outside a web browser.

THEME 2: MEMORY-SAFE PROGRAMMING LANGUAGES

DISCUSSION LEAD: ALEX GAYNOR

SCRIBE: ABHISHEK ARYA

WHAT IS THE PROBLEM?

C and C++ are widely used programming languages and are a critical part of Internet users' common day-to-day activities, from the underlying operating systems (e.g., Windows, Linux) to web browsers, databases, and cloud applications. On one hand, these languages have high performance benefits and allow communication with lower layers of computer systems abstraction. On the other hand, these languages are the leading cause of memory safety vulnerabilities that are explored in many cyberattacks. As per several recent reports[1], roughly 70% of all software vulnerabilities are memory corruption issues coming from the use of these memory unsafe programming languages. While exploit mitigations and automated testing have helped reduce bugs and made code exploitation harder, memory corruption errors are still widespread across software developed in memory unsafe programming languages.

Over the last decade, we have seen the rise of several memory-safe programming languages, such as Rust and Go. They provide end-to-end memory management, hence mitigating several classes of memory safety bugs, such as buffer overflows, use-after-frees, and programming language type confusions. We would have expected the software development community to have transitioned to these safer programming languages by now, but that has not been the case. One of the unsolved problems is how to incentivize new developers and their organizations to write code in these memory-safe languages, while maintaining certain performance and interoperability expectations. Another challenge is the large amount of legacy code, which makes rewriting it in memory safe languages either infeasible or extremely time-consuming (probably years to conclude).

Another issue is that, despite the security guarantees of memory-safe programming languages, there is always the risk that code written in such languages bypasses available safety guarantees via, for example, the use of unsafe modes or calls into unsafe third-party code.

WHAT ABOUT THE PROBLEM DO WE NOT YET UNDERSTAND?

In general, we do not understand the methodology and feasibility of transitioning projects, especially large, complex projects like operating systems, web browsers, or language environments to memory-safe programming languages. We lack an understanding of the community and ecosystem interactions needed to navigate through such a massive change. Also, it is unclear how we can expect critical projects to undertake these long-term initiatives without any sustainable funding incentives.

Several of the recent initiatives of adding support for memory-safe programming languages in critical places, such as the Linux kernel and web browsers, have been slow and painful due to a lack of understanding on where to start and whether any parts of this process could be automated (e.g., through shim[5] generation). Another set of challenges are in the embedded device space, where there is limited availability of memory-safe languages (C is the most widely used) and no clear solution on how to update these devices to another language.

While memory-safe programming languages will fix most classes of memory corruption vulnerabilities, they should not be considered a panacea. There are several classes of bugs that impact memory safe languages, such as remote code execution (e.g., the log4j vulnerability), information disclosure, and denial of service (DoS) attacks. As we encourage developers to transition to memory safe languages, it will be important to educate them on secure coding techniques to avoid introducing these other types of vulnerabilities.

WHERE ARE THE BOUNDARIES OF THE PROBLEM? ARE THERE ANY CONSTANTS THAT CANNOT BE CHANGED?

To make progress on this front, we need some fundamental changes in the economics of software development. Currently, a developer makes a programming language choice for their project based on their own interests and knowledge, without considering security risks. A large percentage of software is developed by early-career developers whose focus is on writing more code, rather than secure code.

There are several constraints that the Committee finds hard to change. One of them is the large amount of C/C++ legacy code that will take several years (if at all) to transition to a memory-safe variant. Currently, there are no requirements to transition away from them within critical systems, such as national defense, energy, and healthcare sectors. The lack of such requirements potentially creates a lack of incentive to transition. Another constraint is the need for scalable funding models to help projects invest in these long-term initiatives and ways to deal with the migration challenges in regular software release cycles (e.g., Application Programming Interface -API- breakages, community acceptance, or user upgrades). We need to assume that the community transition to memory-safe programming languages is several years away, but we can speed up this process by increasing investments in usability, interoperability, and automation.

[5] From www.quora.com: "*A shim is a small piece of software that fits between two layers of software that communicate with each other and is typically used to adapt one interface to another*".

THEME 3: DEPENDENCY MANAGEMENT

DISCUSSION LEAD: ERIC RESCORLA

SCRIBE: MATHIAS PAYER

WHAT IS THE PROBLEM?

The Committee is in consensus that the problem of software dependencies boils down to external software libraries becoming security liabilities. Software increasingly relies on third-party libraries that are included into the trusted computing base of the software. Due to lack of compartmentalization, any library included in a software project gets full privileges of all the software. The more libraries are included, the higher the risk of a developer either overlooking a vulnerability in a library or a library becoming a liability. Keeping track of software dependencies and their versions is extremely challenging. Thus far, there are no best practices for software dependency management and tools have no defined feature lists. Each software project is required to develop their own best practices. Software development strategies are changing and the practice of reusing existing functionality (instead of writing them from scratch) is increasingly encouraged. This leads to developers freely including dependencies for small functionality without considering the long-term effects of future engineering burdens and the security risk of externally controlled code. For example, Kubernetes, a container system for automating software deployment, now has more than 1000 dependencies. While some platforms like Rust, Python, or Node Package Manager (NPM) provide (inconsistent) dependency management systems, software written in C/C++ have no dependency management support at all. The only weak version of a dependency management system for C/C++ offering some consistency is the package manager of the underlying operating system, which provides a “bare bones” version of packages.

One of the key challenges is that software library dependencies cannot simply be updated to their most recent version because their features and functionalities may change, thus potentially causing inconsistencies or failures in the software using them. While bug fixes should be applied immediately, a developer may hold off on updates with feature changes, as such changes will require modifications to their code and may also break existing tests and functionality. Developers value stability and, thus far, there is no way to distinguish security patches from feature updates, especially when going deeper in the dependency chain. Interoperability between different dependency management tools (if they even exist) is challenging. Software projects cannot easily support different kinds of dependency managers because of software legacy ties and the complexity of these managers. We will require powerful and flexible tooling to mitigate the risk of libraries becoming security liabilities.

WHAT ABOUT THE PROBLEM DO WE NOT YET UNDERSTAND?

As software dependencies are evolving, we must better understand the scope. So far, we neither understand the breadth nor the depth of the problem. Dependency management has only recently started to evolve, together with the change in developers' best practices of aggressively reusing components and quickly releasing new software library versions. Different programming languages have different challenges. We will have to infer the fanout depth for each language to evaluate to what extent different programming languages or platforms are prone to dependency violations. For C/C++, for instance, we need to figure out if some form of dependency management system is even feasible as so far, we rely on the operating system to provide this information. For example, the Debian package management system uses maintainers of software packages to keep track with upstream and to backport patches to their old and stable versions. This requires substantial work by maintainers to keep track with upstream and to provide patches for downstream. Other distributions have moved to a more aggressive model where they simply more closely follow the latest version of software libraries, likely at a more frequent exposure to security vulnerabilities.

WHERE ARE THE BOUNDARIES OF THE PROBLEM? ARE THERE ANY CONSTANTS THAT CANNOT BE CHANGED?

The focus of this theme's discussion was primarily technical. While there will be human-based solutions and human-based training for some of the other themes, this theme will be primarily driven by advances in technology and tools. The boundaries are therefore the build infrastructure and the surrounding tooling. Similarly, secure distribution of signed binaries to customers, potentially along with pushing for reproducible builds (i.e., ensuring that the binary code has been compiled from untampered source code) will further increase trust into the software and its dependencies.

The consensus of the discussion was that technical solutions will open many opportunities for making progress on this front. For example, software may only need a small part of the library functionality. Tracking this small subset of needed code would (1) reduce the risk of attacks (by reducing the attack surface), (2) allow tracking of code provenance (i.e., if the used features change), and (3) potentially enable automatic updating as the ultimate goal. Dependency management tools need to be built into native language tooling and package management. While we can change the tools, we cannot teach all developers to switch to completely new tools. The move will be gradual and incremental.

THEME 4: BEHAVIORAL AND ECONOMIC INCENTIVES TO SECURE OPEN SOURCE SOFTWARE

DISCUSSION LEAD: SUMANA HARIHARESWARA

SCRIBE: DAVID BRUMLEY

WHAT IS THE PROBLEM?

First, we use the term *professional developer* to mean a developer that is managed and paid for their work, e.g., as a company employee or independent consultant. *Non-professional developer* meant in our discussion someone not managed and paid for by a company, including those not paid at all for their work. We note that the word *volunteer* has two different meanings in the community. In the open source software (OSS) community (e.g., Linux Foundation), a volunteer means anyone doing work on OSS that is not expressly required to by their employer. Nevertheless, many people in the OSS community associate volunteer with unpaid work. An OSS contributor is anyone who is contributing (e.g., with code, test cases, or documentation) to an OSS project.

During discussion, the most precise problem statement definition was "*How do we incentivize human resilience in OSS teams*?" This definition included the idea that OSS development could be fragile, e.g., a project may depend upon a sole contributor. It also included the idea of resilience, meaning that overall, the project should be in a secure state, even if is temporarily insecure at some points in time (eventual security). For example, the question the Committee focused on was rather how one would build an ecosystem that detects a malicious change and revert it, rather than how to prevent a malicious change altogether.

The Committee's focus was on building incentives and behaviors that specifically target security rather than just volume of contributors. The Committee agreed that current data shows that OSS contributors are not motivated by security, citing the Linux Foundation report[2] that states that "*efforts focused on dramatically increasing the time current contributors spend on security are unlikely to be welcome*".

Most of the discussion focused on narrowing down the problem to behavioral incentives. For example, recognition by peers (e.g., a "secure badge" associated with a developer on a project repository) is a behavioral but non-economic motivation. The Committee also envisions the problem as one of group rather than individual incentives. For instance, one suggestion discussed was to pair a security researcher (who is intrinsically motivated by security) with a project lacking security experts.

Another aspect of the discussion was that the OSS community seems especially inefficient at translating money to security gains. For example, an additional problem identified was how to use the limited professional support (e.g., industry grants, government support) to have the most overall security impact.

Even large companies like Google depend upon far more open-source projects than they could support. The industry also has not systematized ways to train, hire, and deploy process-focused contributors (such as release managers and project managers) to work with maintainers to aid existing projects.

WHAT ABOUT THE PROBLEM DO WE NOT YET UNDERSTAND?

In general, there is a lack of understanding on how to build an effective ecosystem that creates secure software. One aspect that the Committee does not know is how to include security training that is appropriate for everyone. Another aspect is that the community does not know how to support the wide variety of project sizes, especially the very small contributor team on a very important project.

The Committee agreed that there is no consensus on what indicators of success would be. Moreover, there was a lack of new ideas compared to other themes in the meeting. The Committee posed questions, like "*what are the dependent variables?*" and "*how do we measure if we're getting better or worse over time?*" Also, while each Committee member had an opinion on what would be a good motivator for adding security to OSS projects, the Committee were unsure about why people decide to adopt or not a secure approach, and how to measure that quantitatively.

The Committee also noted that today the consumer of OSS often has little ability to judge its level of security or exposure. For example, it is not clear what metrics would have identified log4J as a top weakness and top exposure compared to other OSS software. The Committee conjectures that the ability to assess security would likely drive more secure development behavior because it was then measurable and obvious to others. In this context, the idea of a badging system was given again as an example where getting a security badge might incentivize some contributors. However, we do not have data to support or refute this conjecture.

Another point brought up during discussion is that the Committee does not understand the balance between detecting vs. preventing a problem. There are economic factors such as cash investment. The Committee left open the possibility that detecting vulnerabilities after development may end up being a more scalable model.

WHERE ARE THE BOUNDARIES OF THE PROBLEM? ARE THERE ANY CONSTANTS THAT CANNOT BE CHANGED?

The Committee does not have good research on the constants and boundaries of this problem. The members overall had difficulty identifying specific known constraints or constraints for creating the economics that drive change. One aspect all agree on was that OSS contributors are not likely to be driven by monetary rewards. The Committee also agreed that there is a need to consider different levels of developers, from hobbyists just starting out to developers working on larger projects. Many projects start small, and then blow up.

There were also several conjectures raised during discussion. Some Committee members conjectured that changing infrastructure may be easier than changing behavior, but it was unclear where to go with that. Others conjectured that economic incentives will improve if we have tools that prevent vulnerabilities from being injected in the projects.

Thus, the Committee's consensus is that the U.S. needs to fund and conduct more research into the behavioral and economics aspects of OSS software development security in at least three areas:

1. **Software development lifecycle**. What incentive structures would encourage volunteer OSS developers to incorporate better software security practices? And how do we provide this information to developers? During the software lifecycle, what incentive models encourage active and speedy remediations when security issues are discovered? What if there is only one primary developer, and they leave the project? Are incentives purely contributor-based, or are there other incentive models, such as matching security experts to OSS projects?

2. **Using paid professional development/assistance wisely**. How can we leverage the limited amounts of government, industry, and professional assistance to maximize our overall security posture? How do we assess the security of OSS dependencies beyond the vendor-independent fashion? And how do we ensure that we encourage security practices in small startups rather than predominantly in large existing businesses? Can we identify the weakest links, shore up security, and then rinse and repeat with the next highest priority? For example, companies often professionally support OSS projects (e.g., through grants, computing resources, or paid on-going development), but cannot support every single project they depend upon. How does the U.S. government, as well as individual companies, get the most bang for their buck out of the limited professional funding and development we can put behind OSS?

3. **Informed choice**. Today's OSS consumers (commercial users, governments, and even other OSS projects) do not heavily weigh security when making choices. How do we make security a priority feature in software development processes?

CONCLUSIONS

The Committee focused on defining the problem, the boundaries and the unknows in four themes related to the security of open source software ecosystems: (1) trust and safety, (2) memory-safe programming languages, (3) dependency management, and (4) behavioral and economic incentives to secure the open source software ecosystem.

Socio-technical vulnerabilities in the OSS ecosystem are understudied. The following aspects are not well understood by the community:

- How to incentivize developers to adopt security practices? Is it even reasonable to request developers to handle security and functionality requirements concurrently, given the high cognitive demands of both tasks?
- How to detect bad actors, such as malicious or compromised developers or vulnerable team dynamics, such as suspicious developers' ascendancy in projects?
- How to distinguish and measure intentional vs. unintentional attacks to OSS projects?

Although there is consensus on the crucial role **memory-safe programming languages** can play in securing the OSS ecosystem, the likelihood of ubiquitous use of such languages in software development is still unclear for the following reasons:

- Prevalence of C/C++ in critical applications (operating systems, browsers, and databases) and in the IoT ecosystem.
- Lack of economic incentives to transition legacy code to safer programming languages, a dauting task that can take years to complete without automatic tools.

Software development has been increasingly relying on third-party libraries (**dependencies**), which become liabilities and are hard to manage for the following reasons:

- There are no best practices on dependency management.
- Developers need assurances for updating dependencies because new versions of dependencies can break existing code.
- Dependency management systems need to handle individual programming languages idiosyncrasies and currently have poor interoperability.
- The community does not know how to scope dependency management in terms of breadth and depth.

Related to socio-technical aspects of securing the OSS ecosystem are the underlying **behavioral and economic incentives**: how to incentivize developers to adopt security practices? This problem is understudied and challenging for the following reasons:

- Initial research gives evidence that OSS contributors are not motivated by security.
- There is a high diversity of roles in the OSS ecosystem: paid vs. unpaid and employed vs. unemployed developers.
- OSS teams can be fragile, for instance, a single or a few maintainers responsible for a crucial piece of code.
- Group vs. individual incentives are not necessarily aligned.
- Monetary compensation for developers does not necessarily translate into security gains.

A subsequent workshop with a larger and more diverse set of stakeholders will develop recommendations informed by the take-aways of this report with the ultimate goal of achieving tangible progress in securing OSS development processes.