RESEARCH FEATURE

Third-Party Software's Trust Quagmire

Jeffrey Voas, National Institute of Standards and Technology
George Hurlburt, STEMCorp

Integrating software developed by third-party organizations into a larger system raises concerns about the software's quality, origin, functionality, security, and interoperability. Addressing these concerns requires rethinking the roles of software's principal supply-chain actors—vendor, assessor, and evaluator.

Software is ubiquitous and ethereal. As lines of code that can be reproduced (even printed) it is physical, but as runtime behavior it is not. Partly art and partly science,[1] software is difficult to measure and assess, yet unchecked it can induce harmful consequences, particularly in safety-critical operations.

This potential for harm makes software assessment a nontrivial concern that comes with a range of pressing considerations. Unfortunately, many assessment issues remain unaddressed, largely because the process is clouded by questions, such as what do we measure and how trustworthy are the measurement criteria? These murky waters are further muddied by the ascent of third-party software.

By its nature, software defies physical analysis. Much like well-written literature, well-composed software is aesthetic. Also as in literature, syntax and context make a difference, not only as contributors to an elegant package but also in the dynamics of their execution, whether being read or run. Software as an art speaks to the creative side of its most innovative purveyors: a coding artist structures exquisite mathematical algorithms as a painter might fashion a masterpiece. But software is also an engineering feat, the building of an architecture whose pieces can be decomposed and rapidly reassembled to construct useful functions on demand.

Unfortunately, the coexistence of art and engineering viewpoints—both equally valid—blurs the lines of the software assessment picture, adding ambiguity to the questions that drive the process, such as

- What is good software?
- What is good enough software?
- What enables software comparisons that make characterizations like "better product" plausible?
- Who creates the criteria for answering the first three questions?

Plodding through this morass of unknowns to solid ground—namely, being able to integrate third-party software with confidence in its function and security—will require calling out the principal actors in the third-party software supply chain and better understanding their roles as well as basing assurance on quantifications of behavioral aspects such as reliability, security, and performance.

0018-9162/15/$31.00 © 2015 IEEE

PULL QUOTE HERE

FIRST AND SECOND PARTIES

Putting third-party actors in perspective requires identifying the first and second parties and their roles. The first party is the end user or consumer, who likely has no knowledge of the component's development and production. The second party is the system integrator—the entity that composes a product from parts.

The first and second parties do not always communicate. If the product is a mass-market offering, interaction is unlikely, but if the product is based on a specification that the first party supplies, the two will likely communicate throughout development and testing.

Communication between users and system integrators is foundational to the software's trustworthiness. On the basis of an exchange with users, system integrators might decide to integrate one product instead of another. The resulting system will better fit the users' needs, which could be more valuable to those users than any assessor feedback.

THIRD PARTIES

Most views of third-party software focus on the vendor, which is certainly one important role. However, acquired software parts need some type of a quality check before being integrated, which suggests that "third party" should have other roles as well—namely, the assessor and evaluator. This idea is not novel, but no one has been taking it seriously since the 1990s when software became modular blocks. The sidebar "Pieces and Parts Theory" describes the implications for assurance.

Vendor

The component vendor is responsible for producing the software components and descriptions of their functionality.

Assessor

The assessor amasses the necessary quantitative and qualitative information about the vendor's product. The initial assumption is that components are not trustworthy; assessment determines if there is a reason to dispute that assumption. The assessor should provide a value proposition that assessment is worth the cost because the resulting knowledge about the software' functions and quality makes composition easier and integration quicker, and reduces the integrator's liability from any defects in the vendor's software and hence reduces the integrator's risk. The assessor should also emphasize the benefits of having access to a person or an organization that independently publicizes the limits of vendor claims.

Given that data about the software's quality and performance is the primary product, the assessor's first task is deciding what qualities to collect, how to collect them, and the best way to package the proposed assessment process in a "why use us?" sales pitch to the integrator. Data must be carefully tailored to the integrator's needs. If integrators view an assessor's results as irrelevant component measures or other meaningless-to-me discoveries, assessment becomes a lose-lose scenario that benefits no one.

The assessor is much like the traditional independent software tester, as the vendor and assessor cannot be in collusion. However, having only a vendor and assessor places too much authority in the assessor's hands, which motivates the need for the evaluator role.

Evaluator

The evaluator determines the quality and accuracy of the assessor's information—reviewing the assessor's work, not the vendor's product. The evaluator is, in essence, the judge of the assessor and fulfills the role of independent evaluator in traditional test and evaluation, basically skimming through results to spot any serious anomalies. The evaluator rarely redoes the assessor's work unless it is suspicious or useless.

REDUCING LIABILITY

Because software is not physical, direct quality measurements are harder to take and believe than measurements based on physical attributes, which leaves indirect quality measurements to fill the void. Consequently, the assessor is in a far more precarious position than either the vendor or evaluator in terms of inappropriate certifications made and the associated liability.

Unlike hardware quality, which rests on physical measurement tools like the periodic table, software quality has no quantifiable assessment basis. Given this unpleasant truth, is it reasonable to expect reduced liability, indemnification, and risks from third-party code and a defined, timely, repeatable, and fair process from the assessor?

The degree of reasonableness depends on the answers to two other questions: how much liability do assessors actually incur from their assessments, and how do they indemnify themselves? On one hand, incorrect assessments can be

PIECES AND PARTS THEORY

In the 1990s, the promise of using the same maintenance paradigm in software development that was the basis for successfully swapping decommissioned parts for new hardware components generated much excitement. If the paradigm could truly transfer to the software world, its best practices could alter earlier life-cycle phases such as architecture, design, and implementation.

There was talk of treating smaller software chunks as Lego units, or logic commodities. From a commerce standpoint, this strategy would open marketplaces of component catalogs and eliminate the need for bespoke systems. Adopting such a paradigm would propel software engineering practice to the utopian goal of faster, better, and cheaper software development[1] and foster competition.

The popularity of concepts like component-based software engineering (CBSE), software of unknown pedigree (SOUP), and commercial off-the-shelf (COTS) software[2] peaked in the late 1990s and early 2000s. CBSE is concerned with the rapid assembly and maintenance of component-based systems, in which components and platforms have certified properties, and these certified properties provide the basis for predicting properties of systems built from components.[3] SOUP is "software for which full and complete access to the source code, documentation, and/or development history is unavailable."[4]

SWAPPING PERSPECTIVES

However, all those benefits come at a cost. Transferring a hardware perspective to software requires understanding what makes the hardware component viable in its own marketplace and shifting that appreciation to the software realm. In this case, it is access to information about component tolerances and quality and possibly how components were tested. Information is also needed about a component's functionality, interfaces and interoperability, and expectations from all other subsystems it will interact with (user interfaces, software, and hardware).

These expectations evolve as information begins to morph into an understanding of potential emergent behaviors forged after composition—some good, some bad, some malicious. A simple example is a mobile app that controls a camera but has no permission to connect to the Internet. By colluding with another app that does have Internet access, the camera app now has it as well.

BETTER VERSUS FASTER AND CHEAPER

Thus, from the commerce standpoint, faster, better, and cheaper is achieved after component integration. However, there is still the time and resources to produce and tag components with component-specific information, and this

so obvious that they point to assessor error, which is all on the assessor; on the other hand, they can be subtle enough to look fine on the surface but later harm the vendor's or integrator's reputation. In the second case, liability might be harder to determine.

Both these scenarios imply that decreased liability is a function of what is being assessed and the results' accuracy and timeliness—two important assessment goals. The sidebar "Assessor's Checklist" gives some others.

Having an evaluator judge the assessor's work is a model that the Defense Test and Evaluation structure tends to mirror in both developmental and operational testing.[2] In the commercial sector, the model stimulates a competitive market of assessors, which provides the opportunity for promoting the quality and accuracy of the services they offer. An evaluator can deem an assessor competent to check *only* for certain component properties.

The National Institute of Standards and Technology's (NIST's) Cryptographic Module Validation Program (CMVP; http://nist.gov/cmvp) follows this practice. To test their cryptographic modules, vendors use independent, accredited cryptographic and security testing laboratories. The laboratories, in turn, use derived test requirements, implementation best practices, and relevant CMVP programmatic guidance to evaluate the modules against applicable standards. NIST's Computer Security Division and Canada's Communications Security Establishment jointly serve as CMVP's authorities for validating test results and issuing certificates.

effort might sacrifice faster and cheaper in favor of better.

In 1998, the software assurance community got a wake-up call on this subject in a report by the US National Academy of Sciences:

> *A consumer [patient] may not be able to assess accurately whether a particular drug is safe, but [they] can be reasonably confident that drugs obtained from approved sources have the endorsement of the U.S. Food and Drug Administration (FDA) which confers important safety information. Computer system trustworthiness has nothing comparable to the FDA. The problem is both the absence of standard metrics and a generally accepted organization that could conduct such assessments. There is no Consumer Reports for trustworthiness.*[5]

Have things changed? Unfortunately, they have not. Software is reused; repurposed, often used in operational contexts for which it was never intended; becomes bloated with new functionality over time when no one understands its original logic; and is exploited for its vast malleability benefits—the list goes on. Except for issues related to supply chains, why should anyone care about the software's pedigree or origins? To most, it is simply a third-party product, with time to market and near-instant availability often being the key factors in the decision to acquire it. However, that view is quickly becoming too risky. With the rapid evolution of software parts from untested sources, it is dangerous to keep accepting that the piece-and-parts mindset comes with no harmful side effects.

References

1. T. Menzies et al., "Can We Build Software Faster and Better and Cheaper?," *Proc. 5th Int'l ACM Conf. Predictor Models in Software Eng.* (PROMISE 09), 2009; http://menzies.us/pdf/09bfc.pdf.
2. W.J. Perry, "Secretary of Defense Memo: Specifications and Standards—A New Way of Doing Business," 1994; http://sw-eng.falls-church.va.us/perry94.html.
3. F. Bachmann et al., *Volume II: Technical Concepts of Component-Based Software Engineering*, 2nd ed., tech. report CMU/SEI-2000-TR-008/ESC-TR-2000-007, Software Eng. Institute, Carnegie Mellon Univ., May 2000; https://resources.sei.cmu.edu/asset_files/TechnicalReport/2000_005_001_13715.pdf.
4. G. Rowlands, "Software Safety and SOUP," *Standards in Defense News*, vol. 187, 2003, p. 14.
5. F.B. Schneider, ed., *Trust in Cyberspace*, Nat'l Academy Press, 1999.

AT THE CENTER OF THE SWAMP

Even with competitive assessors and evaluator checks, multiple risks from faulty assessments are still possible. Would adding yet another role to oversee the evaluator reduce these risks? If so, where does it end—a fifth role to oversee the fourth, and so on? In other words, how many eyes are enough?

Clearly, piling on more overseers is not a practical solution to reducing liability, which means looking at the assessment method for possible answers. Some developers of proprietary systems use formal test and evaluation methods; others—developers of both proprietary and open source systems—opt to bypass formal methods in favor of streamlining the evaluation process through rapid crowdsourcing, getting as many eyes on the code as possible to quickly improve it. The moderator of an open source utility decides what gets checked in and what free modifications to dismiss, thus acting as an evaluator but in a different context. Formal and crowdsourcing methods have their respective security advantages, as well as risks.

In addition to determining the test and evaluation method, assessment must address what issuing a certificate ultimately means. At the highest abstraction level, there are three claims, but not all are applicable to the same degree:

- the software was developed and tested according to prescribed life-cycle rules or possibly best practices,

ASSESSOR'S CHECKLIST

To decrease the potential liability from an assessment, assessors have these goals:

- *Present results on time.* If your results for version 1.1 come when version 1.2 is already on the market, no one is likely to care much about them.
- *Aim for repeatable results.* Documentation enables repeatability, so be sure to record when tools were applied, on which platform, with which use cases, and so on. Documenting these details will allow you to get the same result at a later date if required.
- *Present results in an intuitive format.* One approach is to use a color-coded system similar to the former Department of Homeland Security's threat warning model; another is to use simple comments, such as "We could not get this product to do 40 percent of what the manufacturer claims it does."
- *Reveal the process.* Create some level of transparency about what occurs in your assessments; treating assessment as a black box only arouses suspicion that the assessor has little to offer.
- *Investigate assessment tools.* Do not blindly trust automated tools on the basis of their vendors' claims, but rather exercise due diligence on them before making them key process assets. Having a variety of tools is better than relying on a select few.
- *Treat similar products fairly.* If at all possible, apply the same tools to all products under assessment, so you cannot be accused of applying more rigorous tools to product A than to product B because of a bias against A.
- *Limit data collection.* Avoid results that look like heavy, amorphous, unreadable baggage. Limits will also help meet the timeliness goal. Engineering precision should be your goal.

These goals might sound complicated and lofty, requiring somewhat duplicative work, but the outcome can affect whether or not the software gets integrated. There is precedent in the automotive repair and safety-critical regulated industries, in which a regulatory agency often judges detailed evidence from one or more assessors before authorizing a product to go to market. The assessor has a similarly serious responsibility to conduct a fair and thorough software evaluation.

- the software complies with its functional requirements or specification, and
- the software is fit for its purpose.

Thus, at the quagmire's core, we have three third-party roles, no direct measurement method, a nonphysical entity to evaluate, and the need for certification to convey three key messages. Providing access to certified software parts has not raised us from the quality deficit of the 1990s. Clearly something else is wrong.

ROADS TO FIRMER GROUND

As software becomes increasingly ubiquitous, with application developers appearing and disappearing at an alarming rate, the assurance picture is changing from the evaluation of modular legacy systems to a scramble to provide some assurance of trustworthy consumer applications. Indeed, marketplace reputation seems to be driving systems assurance. Software independent validation and verification (IV&V) takes time, and the cost is high. It was great for old-school software engineering but is ill suited to application development, which requires a lean-and-mean approach.

An example of de facto lean-and-mean assessment can be found in the app store, a large repository of third-party applications. Hopefully, app stores validate a vendor's product before releasing it for public consumption and then license out the vendor's offerings. They might regard the quality-assessment data they collect to be for internal use only, but the quality of products in their repositories can promote or tarnish their reputation. For example, users of the apps in AppStore X might send out "buyer beware" messages because AppStore X did nothing to check for malware, which would not benefit AppStore X's reputation.

The software development community must recognize that selling software directly to consumers is raising security and privacy concerns to the point that users are starting to indirectly demand trusted apps. In this dynamic development world, in which just about anyone can write and distribute code, where will trustworthiness come from?

Elements of system assurance

As Figure 1 shows, system assurance is based on three elements that affect the system, which are expressible as three sets with unique members. **A** and **B** comprise the *system*: **A**

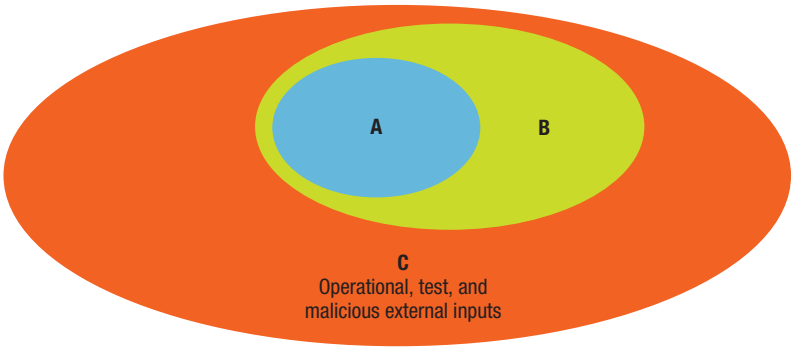

FIGURE 1. How assurance elements interact in a systems model. The system consists of **A**, software with one or more component versions, and **B**, hardware with one or more configurations that a component version will run on. **C** consists of one or more input environments and malicious inputs from external sources. System assurance is a function of all three elements.

represents one or more versions $\{v_1, v_2, v_3, \ldots\}$ of a component, and **B** represents one or more hardware configurations $\{p_1, p_2, p_3, \ldots\}$ that some v_i will execute on, where hardware is the computing platform and **B** affects software quality issues such as performance and security. **C** represents the *implementation universe*, consisting of one or more input environments $\{e_1, e_2, e_3, \ldots\}$ that v_i will execute in as well as a special category of malicious internal computations that create corrupted internal data states or malicious external inputs $\{m_1, m_2, m_3, \ldots\}$.

The environment includes operational input signals as well as their probability density function. Operational input signals are received from users, other executing software functions that communicate with a particular component version (v_i) at runtime, the hardware (**B**), and other physical entities, such as sensors. Malicious internal computations are caused by malware that forces v_i into insecure internal states at runtime. Corrupted internal data states are a function of the input environment selected.

Selecting one member from **A,** one member from **B**, and one or more members from **C** creates one instantiation. For example, $\{v_1, p_6, e_2, m_{10}, m_{44}\}$ represents implementation version 1 running on platform 6 executing input profile 2 while facing security threats 10 and 44. From instantiations, assurance arguments are built.

Redefining assurance

Any assurance model such as that in Figure 1 must be based on certain assumptions about what is meant by assurance. In our view, assurance and trust are interchangeable terms because their distinctions are subtle, not to mention political. In the cybercommunity, trust stems from the early computer security days of developing and evaluating trusted operating systems and platforms, and we still see topics like "roots of trust" in security research. Our use of "assurance" is more aligned with discussions in the software assurance community (https://buildsecurityin.us-cert.gov/swa).

An assurance model must also consider policy because policy defines what is demanded of a system's functionality and quality, or that which is technically feasible. For example, the realization in the 1990s that a 10^{-9} probability-of-failure estimate as a reliability measure was not feasible prompted the blacklisting of some budding safety-critical projects until such fine-grained levels of fidelity were deemed believable. The lesson here might be to determine what is feasible before writing a policy for what is impossible.

Assurance qualities

In traditional views, software is either static or dynamic, and assessors might use both in their evaluation. A static view can provide insights into integration and what is needed from the interfaces during composition, while dynamic views can shed light on how executing software will interact with a universe. Repeated execution with a particular universe produces evidence, which essentially exposes the software's behavior.

However, invisible behaviors such as security, reliability, and performance ultimately determine confidence in claims such as "the software complies with requirements and is fit for its intended purpose." The software quality community refers to these behavioral qualities as "ilities" regardless of their word ending because they are equally essential to system assurance.

Reliability. Reliability is the probability that software will work properly in a specified environment and for a given time. It is based on the probability of failure, which is determined by testing a sample of all available input states. Probability of failure is the number of failing cases divided by the total number of cases under consideration.

Security. Security is the use of software, hardware, and procedural methods to protect applications from external threats.[3] Security is not apt to be quantified because threat spaces are typically unknown. Therefore, security is more likely

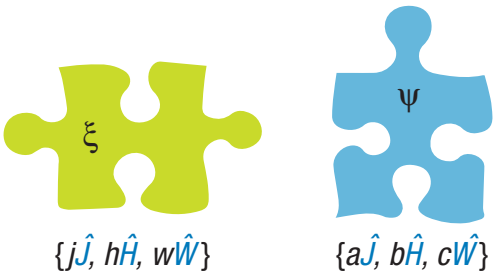

FIGURE 2. Integrating two components. Integration also involves composing the stand-alone behaviors of each component (j,h,w and a,b,c) and ensuring that composite reliabilities are compatible. $\hat{J}$ represents a reliability estimate, $\hat{H}$ represents security evidence, and $\hat{W}$ represents a performance estimate.

qualified, possibly relying on documented evidence concerning which security risks were mitigated using generally accepted development standards, tools, or methodologies.

Performance. Performance is the software system's quantitative behavior. Performance is based on the comprehensive analysis of the software's structure and behavior from design to code.[4]

Quantifying qualities

When directly measured (quantified), these ilities can be significant assessment evidence.[5,6] Reliability and performance are quantifiable using repeated trials with selected data from a specific universe. Because these behaviors are assessed at specific times, the results are time-stamped.

We caution that, although quantifiable measures are evidentiary, they are not a panacea. Different reliability models will give different estimates, particularly when distinct models input different parameters or weigh them differently. For example, in quantifying a basket of apples, do you weigh or count them? If you count them, is it by variety or color? Reliability metrics are no different, which will be problematic for assessors.

Even so, a model that accounts for ilities will serve assessment better than one based simply on the components, hardware configuration, and environment. Equation 1 states that *software* assurance is a function of reliability, security, and performance assessments at time t_0:

$$Assurance\ (v_i \in \mathbf{A})_{t0} = f\,(p_k \in \mathbf{B}_{t0}, e_l \in \mathbf{C}_{t0}, m_n \in \mathbf{C}_{t0}, \hat{J}_{t0}, \hat{H}_{t0}, \hat{W}_{t0}), \quad (1)$$

where $\hat{J}$ represents a reliability estimate, $\hat{H}$ represents security evidence, and $\hat{W}$ represents a performance estimate.

Equation 2 illustrates that *system* assurance is also more than a function of **A**, **B**, and **C**:

$$Assurance\ (System)_{t0} = f\,(v_i \in \mathbf{A}_{t0}, p_k \in \mathbf{B}_{t0}, e_l \in \mathbf{C}_{t0}, m_n \in \mathbf{C}_{t0}, \hat{J}_{t0}, \hat{H}_{t0}, \hat{W}_{t0}) \quad (2)$$

Applying these measures can be impractical for large systems, however. Consider the plight of an integrator asked to sequentially integrate two components—the ξ and ψ in Figure 2. Not only must the integrator be confident that the composed components will yield the expected emergent functionality, but also that $\xi_{\hat{J}ty}$ composes satisfactorily with $\psi_{\hat{J}tz}$, where *ty* and *tz* are particular times. That is, the integrator must be confident that the composite reliabilities are compatible; so, for example, two highly reliable components do not have an emergent behavior of low reliability.

Composite compatibility is also needed for security and performance, which is more complicated to provide when composing heterogeneous attributes—for example, $\xi_{\hat{W}ty}$ composed with $\psi_{\hat{H}tz}$. Performance built into ξ could negatively affect the security built into ψ, and vice versa.

Figure 2 involves just two components and three ilities—imagine the same process for hundreds of components' and additional ilities for each. The interactions between ilities from stand-alone software parts at runtime create yet another influence on assurance claims, in much the same way that drug interaction can invalidate missing safety claims that were previously unknown.

Given that the third-party model appears to be gaining strength for business and legal reasons, to say nothing of improved assurance, some actions are necessary. Vendors should continue to innovate with sophisticated algorithms that will challenge assessors to get ahead of the technology they must assess. Otherwise, they will be relegated to the sidelines and unable to make valid contributions to the industry and consumers, much less assure levels of security.

Evaluators must stay a step ahead of assessors to effectively pass judgment on relative assessment acumen. In essence, it

becomes everyone's responsibility to be on top of their game. Despite current perceptions, fueled by a burgeoning entrepreneurial movement, innovation is not the sole province of the vendor—much is left for the other third-party roles.

DISCLAIMER

Any mention of commercial products or organizations is for informational purposes only; it is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the products identified are necessarily the best available for the purpose.

REFERENCES

1. S. McConnell, "The Art, Science, and Engineering of Software Development, *IEEE Software*, vol. 15, no. 1, 1998, pp. 118–120.
2. US Department of Defense, Test and Evaluation Management Guide, Dec. 2012; www.dau.mil/publications/publicationsDocs/Test%20and%20Evaluation%20Management%20Guide,%20December%202012,%206th%20Edition%20-v1.pdf.
3. K.R.M. Rao and D. Pant, "A Threat Risk Modeling Framework for Geospatial Weather Information System: A DREAD based Study," *Int'l J. Advanced Computer Science and Applications*, vol. 1, no. 3, 2010, pp. 20–28.
4. V. Cortellessa, A. Di Marco, and P. Inverardi, *Model-Based Software Performance Analysis*, Springer, 2011.
5. J. Voas, "Software's Secret Sauce: the `ilities," *IEEE Software*, vol. 21, no. 6, 2004, pp. 2–3.
6. J. Voas, "Software Quality Unpeeled," *Crosstalk*, June 2008, pp. 27-30.

ABOUT THE AUTHORS

JEFFREY VOAS is a computer scientist at the US National Institute of Standards and Technology. His research interests include the Internet of Things and fundamental computer science shortcomings. Voas received a PhD in computer science from the College of William and Mary. He is a contributing editor for Computer's Security column and a Fellow of IEEE and the American Association for the Advancement of Science (AAAS). Contact him at j.voas@ieee.org.

GEORGE HURLBURT is chief scientist at STEMCorp, a nonprofit that works to further economic development via adoption of network science and to advance autonomous technologies as useful tools for human use. His research interests include network science and complexity theory. Hurlburt received a BS in psychology from the University of Houston. He is a member of IEEE. Contact him at ghurlburt@change-index.com.