DOI:10.1145/3199601

Fred B. Schneider

Privacy and Security
Putting Trust in Security Engineering

Proposing a stronger foundation for an engineering discipline to support the design of secure systems.

WHEN WE MUST depend on a system, not only should we want it to resist attacks but we should have reason to believe that it will resist attacks. So security is a blend of two ingredients: mechanism and assurance. In developing a secure system, it is tempting to focus first on mechanism—the familiar "build then test" paradigm from software development. This column discusses some benefits of resisting that temptation. Instead, I advocate that designers focus first on aspects of assurance, because they can then explore—in a principled way—connections between a system design and its resistance to attack. Formalizing such connections as mathematical laws could enable an engineering discipline for secure systems.

Trust and Assumptions

To computer security practitioners, the term "trust" has a specific technical meaning, different from its use in everyday language. To *trust* a component *C* is to assert a belief that *C* will behave as expected, despite attacks or failures. When I say that "we should trust *C*" then either I am asking you to ignore the possibility that *C* is compromised or I am asserting the availability of evidence that convinced me certain (often left implicit) aspects of *C*'s behavior cannot be subverted. Availability of evidence is required in the second case, because what convinces me might not convince you, so psychological questions that underpin trust claims now can be explored separately in discussions about the evidence.

Trust is often contingent on assumptions. These assumptions must be sound, or our trust will be misplaced. We often make explicit our assumptions about the environment, talking about anticipated failures or the capabilities of attackers. But we also make implicit assumptions. For example, when expectations about behavior are couched in terms of operations and interfaces, we are making an implicit assumption: that attackers have access to only certain avenues for controlling the system or for learning information about its state. We also are making an implicit assumption when we ignore delays associated with memory access, since attackers might be able to make inferences by measuring those delays.

Assumptions are potential vulnerabilities. If a component will behave as expected only if some assumption holds, then an attacker can succeed simply by falsifying that assumption.

IMAGE BY ANDRIJ BORYS ASSOCIATES/SHUTTERSTOCK

Most attacks can be deconstructed using this lens. For example, buffer-overflow attacks exploit an assumption about the lengths of values that will be stored in a buffer. An attack stores a value that is too long into the buffer, which overwrites values in adjacent memory locations, too. The recent Spectre[1] attack illustrates just how subtle things can get. With access to an interface for measuring execution times, an attacker can determine what memory locations are stored in a cache. Speculative execution causes a processor to access memory locations, transferring and leaving information in the cache. So, an attacker can learn the value of a secret by causing speculative execution of an instruction that accesses different memory depending on that secret's value. The implicit assumption: programs could not learn anything about speculative executions that are attempted but reversed.

One aspect of a security engineering discipline, then, would be to identify assumptions on which our trust depends and to assess whether these assumptions can be falsified by attackers. Whether such assumptions can be falsified will depend, in part, on an attacker's capabilities. The Defense Science Board[3] groups attackers into three broad classes, according to attacker capabilities:

- Those who only can execute existing attacks against known vulnerabilities;
- Those who can analyze a system to find new vulnerabilities and then develop exploits; and
- Those who can create new vulnerabilities (e.g., by compromising the supply chain).

Or we might characterize attackers in terms of what kind of access they have to a system:

- Physical access to the hardware;
- Access to the software or data; or
- Access to the people who use or run the system.

Cryptographers characterize attackers by bounding available computation; they see execution of PPT (probabilistic polynomial-time) algorithms as defining the limit of feasible attacks.

To build a system we are prepared to trust, we eliminate assumptions that constitute vulnerabilities we believe could be exploited by attackers.

- Analysis of a system or its components could allow weaker assumptions to replace stronger assumptions, because we then know more about possible and/or impossible behaviors. To embrace the results of such an analysis, however, we must be prepared to trust the analyzer.
- Incorporating security mechanisms in a system or its components provides a means by which an assumption about possible and/or impossible behaviors can be made, because the security mechanism prevents certain behaviors. So we can weaken assumptions about system behaviors if we are prepared to trust a security mechanism.

In both cases, we replace trust in some assumption by asserting our trust in something else. So assumptions and trust have become the driving force in the design of a system.

An example will illustrate this role for assumptions and trust. To justify an assumption that service *S* executes in a benign environment, we might execute *S* in its own process. Process isolation ensures the required benign environment, but we now must trust an operating system *OS* to enforce isolation. To help discharge that assumption, we might run only the one process for *S* in *OS* but also execute *OS* in its own (isolated) virtual machine. A hypervisor *VMM* that implements virtual machines would then allow us to assume *OS* is isolated, thereby requiring a reduced level of trust in *OS*, because *OS* now executes in a more benign environment. Since a hypervisor can be smaller than an operating system, *VMM* should be easier to understand than *OS* and, therefore, easier to trust. Any isolation, however, is relative to a set of interfaces. The designers of *VMM* and *OS* both will have made assumptions about what interfaces to include. And, for example, if the interface to a memory cache was not included in the set of isolated interfaces then attacks like Spectre become feasible.

To build a system we are prepared to trust, we eliminate assumptions that constitute vulnerabilities that could be exploited by attackers.

Bases for Trust

The approach advocated in this column depends critically on having methods to justify trust in components and in systems built from those components. There seem to be three classes of methods: axiomatic, analytic, and synthesized. They are often used in combination.

Axiomatic Basis for Trust. This form of trust comes from beliefs that we accept on faith. We might trust some hardware or software, for example, because it is built or sold by a given company. We are putting our faith in the company's reputation. Notice, this basis for our trust has nothing to do with the artifact we are trusting. The tenuous connection to the actual component makes axiomatic bases a weak form of evidence for justifying trust. Moreover, an axiomatic basis for trust can be difficult for one person to convey to another, since the underlying evidence is, by definition, subjective.

Analytic Basis for Trust. Here we use testing and/or reasoning to justify conclusions about what a component or system will and/or will not do. Trust in an artifact is justified by trust in some method of analysis. The suitability of an analysis method likely will depend on what is being analyzed and on the property to be established.

The feasibility of creating an analytic basis for trust depends on the amount of work involved in performing the analysis and on the soundness of any assumptions underlying that analysis.

- *Testing.* In theory, we might check every input to every interface and conclude that some properties about behaviors are always satisfied. But enumeration and checking of all possible inputs is likely to be infeasible, even for simple components. So only a subset of the inputs to certain interfaces would be checked. An assumption is thus being introduced—that the right set of inputs is being checked.

- *Formal Verification.* Software is amenable to logical analysis, either manual or automated. Today's state of the art for automated analysis allows certain simple properties to be checked automatically for large components and allows rich classes of properties to be verified by hand for small components. Research in formal verification has made steady progress on widening the class of properties that can be checked automatically, as well as on increasing the size and complexity that can be handled.

An analytic basis for trust can be conveyed to some consumer by sharing the method and the results the method produced. When testing is employed, the artifact, set of test cases, and expected outputs are shared. For undertaking other forms of automated analysis, we would employ an analyzer that not only outputs a conclusion ("program type checked") but also generates and provides a transcript of the inferences that led to this conclusion—in effect, the analyzer produces a proof of the conclusion for the given artifact. Proof checking is, by definition, a linear-time process in the size of the proof, and proof checkers are far simpler programs than proof generators (that is, analyzers). So, without duplicating work, a consumer can check the soundness of a manually or automatically produced proof.

Synthesized Basis for Trust. Trust in the whole here derives from the way its components are combined—a form of divide and conquer, perhaps involving trust in certain of the components or in the glue used to combine them. Most of the mechanisms studied in a computer security class are intended for supporting a synthesized basis of trust. *OS* kernels and hypervisors enforce isolation, reference monitors and firewalls restrict the set of requests a component will receive, ignorance of a secret can impose unreasonable costs on an outsider attempting to perform certain actions.

With synthesized bases for trust, we place trust in some security mechanisms. These mechanisms ensure some component executes in a more-benign setting, so the component can be designed to operate in an environment characterized by stronger assumptions than we are prepared to make about the environment in which the synthesis (mechanism plus component) is deployed.

An engineering discipline should provide means to analyze and construct artifacts that will satisfy properties of interest.

Assumptions about independence are sometimes involved in establishing a synthesized basis for trust. With *defense in depth*, we aspire for a combination of defenses to be more secure than any of its elements. Two-factor authentication for withdrawing cash at an ATM is an example; a bankcard (something you have) and a PIN (something you know) both must be presented, so stealing a wallet containing the card alone does not benefit the attacker.[a] Defense in depth improves security to the extent that its elements do not share vulnerabilities. So an *independence* assumption is involved—we make an assumption that success in attacking one element does not increase the chances of success in attacking another.

Independence does not hold in replicated systems if each replica runs on the same hardware and executes the same software; the replicas all will have the same vulnerabilities and thus be subject to the same attacks. However, we can create some measure of independence across replicas by using address space layout randomization, which causes different replicas of the software to employ different memory layouts, so an attack that succeeds at one replica is not guaranteed to succeed at another. Other randomly selected per-replica semantics-preserving transformation would work, too.

Program rewriters are another means for creating synthesized bases. The rewriter takes a software component C as its input, adds checks or performs analysis, and outputs a version C' that is robust against some class of attacks, because C' is incapable of certain behaviors. If we trust the rewriter, then we have a basis for enhanced trust in C'. But even if we do not trust the rewriter, we could still have a basis for enhanced trust in that rewriter's output by employing a variant of *proof-carrying code*.[2] With proof-carrying code, the rewriter also outputs a proof that C' is a correctly modified version of C; certifiers for such proofs can be simple and small programs, independent of how large and complicated the rewriter is.

Conclusion

An engineering discipline should provide means to analyze and construct artifacts that will satisfy properties of interest. This column proposed a foundation for an engineering discipline to support the design of secure systems. It suggests that system design be driven by desires to change the assumptions that underlie trust. Security mechanisms change where we must place our trust; analysis allows us to weaken assumptions. C

[a] Kidnapping the person usually gets the wallet, too. So the two mechanisms here have a vulnerability in common.

References

1. Kocher, et al. Spectre attacks: Exploiting speculative execution; https://spectreattack.com/spectre.pdf
2. Necula, G.C. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming* (Paris, France), 1997, 106–119.
3. *Resilient Military Systems and the Advanced Cyber Threat.* Defense Science Board Task Force Report (Oct. 2013).

Fred B. Schneider (fbs@cs.cornell.edu) is Samuel B. Eckert Professor of Computer Science and chair of the at Cornell University computer science department, Cornell University, USA.

Supported in part by AFOSR grant F9550-16-0250 and NSF grant 1642120. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. government.

Schneider is grateful to Susanne Hambrusch, John King, Carl Landwehr, Steve Lipner, and Greg Morrisett for their very helpful feedback on early drafts of this column.