acmqueue

Who Must You Trust?

You must have some trust if you want to get anything done

Thomas Wadlow

In his novel *The Diamond Age*,⁷ author Neal Stephenson describes a constructed society (called a *phyle*) based on extreme trust in one's fellow members. Part of the membership requirements is that, from time to time, each member is called upon to undertake certain tasks to reinforce that trust. For example, a *phyle* member might be told to go to a particular location at the top of a cliff at a specific time, where he will find bungee cords with ankle harnesses attached. The other ends of the cords trail off into the bushes. At the appointed time he is to fasten the harnesses to his ankles and jump off the cliff. He has to trust that the unseen fellow *phyle* member who was assigned the job of securing the other end of the bungee to a stout tree actually did his job; otherwise, he will plummet to his death. A third member secretly watches to make sure the first two don't communicate in any way, relying only on trust to keep tragedy at bay.

Whom you trust, what you trust them with, and how much you trust them are at the center of the Internet today, as well as every other aspect of your technological life.

TRUST IN HARDWARE

During the race to the moon in the 1960s, the Apollo program was faced with the unprecedented problem of guiding two manned spacecraft to a rendezvous in lunar orbit.⁴ Because of the speed-of-light delay in radio transmissions to and from the moon, guidance from ground-based computers would have an unacceptable delay from anything close to real-time, endangering the mission and the lives of the astronauts. A better answer was to have on-board computation with minimal lag time to help the pilots determine how to rendezvous the two spacecraft.

At that time, computers filled rooms and weighed tons. In order to build computer systems that were small and lightweight enough to fly with the Apollo Command and Lunar Modules, NASA engineers wanted to turn to the newly developed integrated circuit chips. The problem: these early chips were not particularly reliable even for ground uses, let alone for mission-critical spacecraft flight hardware.

In the pre-microprocessor 1960s, computer systems were assembled from individual components. At first these were individual transistors, but with chips the components could become gates and flip-flops, registers and arithmetic logic units. But all of these chips were unreliable—especially, but not exclusively, under spaceflight conditions, where extremes of pressure, temperature, vibration, radiation, and high and low gravity added to the potential problems.

Chip reliability could be developed, but perfecting the reliability of even a single chip required enormous R&D, as well as high-volume purchases. Developing the whole suite of reliable chips needed to build a flight-capable computer system appeared to be beyond even NASA's resources.

NASA's solution was simple, brilliant, and exploited the subtleties of computer logic. Rather than trying to perfect dozens of chip types, NASA selected a single chip design that would contain two

3-input NOR gates per chip. Boolean logic teaches us that all logic functions can be built up from NOR gates. By playing to the strengths of Boolean logic, the flight computer was designed using just this single type of chip. A great deal of specialized ground equipment was designed using the same chip as well, so that lesser-than-flight-quality chips could still be used productively. This practice ensured large-volume purchases of the one type of chip, which made it worthwhile for both NASA and the chipmakers to devote huge resources to perfecting the reliability of that chip. With one trustworthy chip as the basis for the flight computers, no Apollo mission suffered even a single computer component failure.

The Space Shuttle went one step further. As one of the first fly-by-wire avionics systems, when such systems were not well trusted, it used five computers that voted on flight-control actions. Four computers ran identical software; the fifth ran software written by a different vendor, but with identical specifications, to guard against a bug in the primary software package. If one of the primary computers disagreed with the other three and the fifth, it could be isolated and ignored. The system was layered and redundant in both hardware and software.

Hardware can fail for any number of reasons, including inadequate testing, design faults, component faults, power-supply issues, ground loops, interface voltage mismatches, externally induced power surges, and human error in maintenance. It can also be untrustworthy because of HTs (hardware Trojans),⁹ in which a device is designed or modified to behave in a manner benefitting the Trojan designer, not the purchaser or operator of the hardware. Detecting a hardware Trojan can be fiendishly difficult, and reverse engineering the intentions of an HT if found is even more so.

TRUSTING SOFTWARE

Technologists tend to prefer technological solutions. If there's a problem, then perhaps there's a box, or a piece of software, that can be bought or built to solve it. This can often work well for problems of physics, or logic, or regular organization. File servers hold files; backup servers back them up. Atomic clocks measure time; routers move packets between networks.

That same regularity is not true of security. It's critically important to remember that security is about people, not technologies. Many people act honorably, but some make mistakes, and some lie, cheat, and steal—sometimes for their own profit or advantage; other times to demonstrate a system's vulnerability. People cause their technologies to act for them in a like manner.

Ken Thompson, co-inventor of the C programming language and the Unix operating system, demonstrated this issue in a memorable way¹⁰ in 1984. He created a C program fragment that would introduce Trojan Horse code into a program compiled by the C compiler. For example, when compiling the program that accepts passwords for login, you could add code that would cause the program to accept legitimate passwords or a special backdoor password known to the creator of the Trojan. This is a common strategy even today and is often detectable through source-code analysis.

Thompson went one step further. Since the C compiler is written in the C programming language, he used a similar technique to apply a Trojan to the C compiler source itself. When the C compiler is compiled, the resulting binary program could be used to compile other programs just as before; but when the program that accepts passwords for login is compiled with the new compiler from clean, uncompromised source code, the backdoor-password Trojan code is inserted into the binary, even though the original source code used was completely clean. Source-code analysis would not reveal the Trojan because it was lower in the tool chain than the login program.

He then went on to point out that this technique could be applied even lower still, at the assembler, loader, or hardware microcode level.

Thompson's moral was: "You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me)."

TRUSTING THE NETWORK

In the late 1700s, a Frenchman named Claude Chappe³ invented a visual telegraph that used semaphore towers spaced 10 to 15 kilometers apart. With a telescope, each station could see neighboring towers and relay their semaphore positions to the destination.

In a world where long-distance communication was via horse-carried letters, Chappe's telegraph was a wonder. Signals from Paris were received in Lille, where the first stations were built (about 218 kilometers or 136 miles apart), after only a few minutes. The network soon spread to Brest, Toulon, Strasbourg, and later even across the English Channel.

Semaphore positions were defined in a codebook allowing a variety of messages to be sent. The well-crafted codebook provided easily recognizable semaphore patterns for common symbols, as well as error-correction codes that allowed a proper message to emerge despite errors in transmission or reception.

The Chappe network was in continuous use for more than 60 years. An electrical telegraph network supplanted it once the technology of long-distance insulated wires made that invention practical.

For our purposes, though, the most interesting aspect of the Chappe network came about in 1836 with the discovery that stock market information was being relayed across the network, buried in other messages, by means of what we now might call steganography. The semaphore operators were paid to introduce certain errors in the messages of various customers. Because of the error-correction codes the messages arrived intact, but if someone was privy to the raw symbols transmitted, the introduced errors contained a message that provided an advantage in buying or selling stocks.

A network can seem trustworthy but still be used for purposes other than its intended one.

Even the time of day can be exploited. In 2013 a network attack known as NTP Amplification used Network Time Protocol servers across the Internet in a distributed denial-of-service attack. By spoofing the IP address of a requester, an ever-larger stream of packets could be aimed at a target, swamping the target's ability to respond to TCP/IP requests.

EXPLORING TRUST

Here's an experiment to try. Take walks in various mixed-use neighborhoods, each with a variety of residences and businesses, such as restaurants, cafes, hardware stores, and hairdressers. Walk in the daytime, before and after lunch. Walk in the nighttime, at the height of the evening activities. Walk late at night, after most things have shut down. With each outing, put yourself in a security mindset. Which is to say: look with the eyes of a thief and notice what you see.

During the day, for example, at busy sidewalk cafes, do people reserve outdoor tables by placing their possessions on the table and then going inside to order? Do they use their grocery bags for this? Their car and house keys? Their wallets?

Late at night, are those same tables and chairs stacked outside or inside? Are they chained together? Are the chains lightweight or substantial? Do the homes in the neighborhood have porch furniture or lawn tools visible from the street? Are they locked up?

Do you see bars on the windows of the residences? On all the windows, or just the back-alley windows? Are there possessions stored on the lawns, in the carports, or on the porches outside at night? Are the family cars parked outside? Are the car windows open? Do they have steering-wheel locks?

Do you see loading docks, such as those in the back of a grocery store, with open doors but nobody visible?

In a district with nightclubs, do you see people who are visibly drunk on the street late in the evening?

Do kids walk to school unchaperoned?

Do postal workers or delivery services leave packages unattended by the front doors of houses? Are bundles of newspapers and magazines left in front of newsstands before they open?

You can learn a lot about neighborhoods this way. It's an especially interesting exercise if you are shopping for real estate. These observations, and many more, are flags for the implicit levels of trust that people have in their neighbors and neighborhoods. The people themselves may not even think of these things. They may leave things on their porches, perhaps accidentally, and nothing bad happens, so they don't worry if it happens again. After a while, it becomes something they don't even notice that they do.

TRUST ON THE INTERNET

Walking around a physical neighborhood, you can gather a lot of information if you are open to it. On the Internet, it can be a very different story. Imagine an office with a single PC, used in the morning by Albert and in the afternoon by Betty. Each has a different account and logs out at the end of the shift. Albert and Betty work from the same physical place, and from the same IP address, but they may have very different experiences on the Internet, depending on what they do.

Extend that to a data center hosting many companies. Each company's servers may be separated by only a few feet, but how they experience the Internet, and how the Internet experiences them, can vary widely. The physical distance between the servers is irrelevant. What matters is the hardware they are composed of, the operating systems that run on them, the application server software, and the configuration data for each of those things, plus all of the utility and ancillary software needed to support and maintain them. The quality and quantity of users, as well as administrators, also matter a great deal.

HEARTBLEED

In spring 2014, a bug in the open-source package OpenSSL became widely known. The bug, now known as Heartbleed (<http://heartbleed.com>), had been present for some time, and may have been known by some, but the full disclosure of the problem in the OpenSSL package came to the public's attention only recently. OpenSSL had been reviewed by many experts and had been a well-used and trusted part of the Internet ecosystem until that point. As of this writing, there is no evidence suggesting any cause other than a programming error on the part of an OpenSSL contributor.

On the morning before the Heartbleed bug was made public, few people were familiar with OpenSSL and they hardly gave the functions it provided a second thought. Those who knew of it often had a strong level of trust in it. By the end of the day, that had all changed. Systems administrators and companies of all sizes were scrambling to contain the problem. Within just a

few days, this obscure piece of specialized software was at the top of the news cycle, and strangers—perhaps sitting in outdoor cafes at tables they had reserved with their house and car keys—were discussing it in the same tones with which they might have discussed other catastrophes.

SYSTEMS ADMINISTRATORS

At the heart of everything that works on the Internet are systems administrators. Sometimes they are skilled experts, sometimes low paid and poorly trained, sometimes volunteers of known or unknown provenance. Often they work long, unappreciated hours fixing problems behind the scenes or ones that are all too visible. They have access to systems that goes beyond that of regular users.

One such systems administrator worked for the NSA (National Security Agency). His name is Edward Snowden. You probably know more about him now than you ever expected to know about any sysadmin, even if you are one yourself.

Another less familiar name is Terry Childs,^{5,11} a network administrator for the city of San Francisco, who was arrested in 2008 for refusing to divulge the administrative passwords for the city's FiberWAN network. This network formed the core of many city services. According to reports, Childs, a highly qualified and certified network engineer who designed and implemented much of the city's network himself, was very possessive of it—perhaps too possessive, as he became the sole administrator of the network, claiming not to trust his colleagues' abilities. He allowed himself to be on-call 24/7, year-round, rather than delegate access to those he considered less qualified.

After an argument with a new boss who wanted to audit the network against Childs's wishes, the city's CIO demanded that Childs provide the administrative credentials to the FiberWAN. Childs refused, which led to his arrest. Even after his arrest, Childs would not provide administrative access to the network. Finally he relented and gave the mayor of San Francisco the access credentials, ending the standoff.

His supervisors claimed he was crazy and wanted to damage the network. Childs claimed he did not want to provide sensitive access credentials to unqualified individuals who might damage "his" network.

In 2010, Childs was found guilty of felony network tampering and sentenced to four years in prison and \$1.5 million in restitution for the costs the city incurred in regaining control of the network. An appeals court upheld the verdict.

Was Childs a fanatic, holding on too tight for his own good, or a highly responsible network admin who would not allow his network to be mismanaged by people he considered to be incompetent? Consider these questions:

- Could something like this happen at your enterprise? How would you know that this problem was developing, *before* it became a serious problem?
- What safeguards do you have in place to prevent a single-point concentration of power such as this?
- What would you do if your organization found itself in this situation?

WHO MUST YOU TRUST?

Some people dream of going back to nature and living apart from the rest of humanity. They want to build their own cabins, grow or raise their own food, and live entirely off infrastructure they have built with their own two hands and a trusty ax. But who made that ax? Even if you can make a

hand-chipped flint ax from local materials, it is far from “trusty,” and the amount of wood you can cut with a flint ax pales in comparison to what you can cut with a modern steel ax. So if you go into the woods with a modern ax, can you truly say you are independent of the world?

If you work on the Internet, or provide some service to the Internet, you have a similar problem. You cannot write all of the code if you intend to provide a modern and useful network service. Network stacks, disk drivers, Web servers, schedulers, interrupt handlers, operating systems, compilers, software-development environments, and all the other layers needed to run even a simple Web server have evolved over many years. To reinvent it all from the specifications, without using other people’s code anywhere in the process, is not a task for the faint-hearted. More importantly, you couldn’t trust it completely even if you did write it all. You would be forever testing and fixing bugs before you were able to serve a single packet, let alone a simple Web page.

Neither can you build all of the hardware you run that service on. The layers of tools needed to build even a simple transistor are daunting, let alone the layers on top of that needed to build a microprocessor. Nor can you can build your own Internet to host it. You have to trust some of the infrastructure necessary to provide that service. But which pieces?

HOW MUCH DO YOU NEED TO TRUST?

To determine how far your trust needs to extend, start with an evaluation of your service and the consequences of compromise. Any interesting service will provide some value to its users. Many services provide some value to their providers. What is valuable about your service, and how could that value be compromised?

Once you have a handle on those questions, you can begin to think about the minimum of components and services needed to provide such a service and which components you have to trust.

Writing your own software can be part of this exercise, but consider that the bulk of security derived from that is what is known as “security through obscurity.” Attacks will fail because attackers don’t understand the code you have built—or so some think. If you choose the path of obscurity as a strategy, you’re betting that no one will show interest in attacking your service, that your programmers are better than others at writing obscure code in a novel way, and that even if the code is obscure, it will still be secure enough that someone determined to break through it will be thwarted. History has shown that these are not good bets to make.

WHO IS EVERYBODY ELSE TRUSTING?

A better approach might be to survey the field to see what others in similar positions are doing. After all, if most of your competitors trust a particular software package to be secure, then you are all in the same situation if it fails. There are variables, of course, because any software, even the best, can be untrustworthy if it is badly installed or configured. And your competitors might be mistaken.

A variation on this approach is to find out which software all of your competitors *wish* they could use. Moving to what they use now could leave you one generation behind by the time you get it operational. On the other hand, moving to one generation ahead could leave you open to yet-undetected flaws. The skill is in choosing wisely.

HOW ARE THOSE SERVICES EVALUATED FOR SECURITY?

Whatever components or services you choose, consider how they have been tested for

trustworthiness. Consider these principles attributed to Auguste Kerckhoffs, a Dutch linguist and cryptographer, in the 19th century:

- The system should be, if not theoretically unbreakable, then unbreakable in practice.
- The design of a system should not require secrecy, and compromise of the system design should not inconvenience the correspondents.

Kerckhoffs was speaking of cipher design in cryptosystems, but his two principles listed here can be applied to many security issues.

When considering components for your enterprise, do they live up to Kerckhoffs's principles? If they seem to, who says that they do? This is one of the strongest cases for open-source software. When done properly, the quality and security of open-source code can rival that of proprietary code.²

For services that you wish to subscribe to, consider how often and how thoroughly they are audited, and who conducts the audits. Do the service providers publish the results? Do they allow prospective customers to see the results? Do the results show their flaws and describe how they were fixed or remediated, or do they just give an overall thumbs-up?

THINKING ABOUT THE BAD CASES FIRST

The legendary Fred Brooks, he of *The Mythical Man Month*,¹ famously said: "All programmers are optimists." Brooks meant this in terms of the tendency of programmers to think that they can complete a project faster than it will actually take them to do it. But as ACM's own Kode Vicious is wont to point out, there is a security implication here as well. Developers often code the cases that they want to work first and, if there's enough time, fill in the error-handling code later, if at all.

When you are worried about security issues, however, reversing the order of those operations makes a lot of sense. If, for example, your application requires a cryptographic certificate to operate, one of the first issues a security programmer should think about is how that certificate can be revoked and replaced. Selecting certificate vendors from that perspective may be a very different proposition from the usual criteria (which almost always emphasize cost). Building agile infrastructure from the start, in which the replacement of a crypto cert is straightforward, easy to do, and of minimal consequence to the end user, points the way toward a process for minimizing trust in any one vendor.

Developing an infrastructure that makes it easy to swap out certificates leads to the next interesting question: How will you know when to swap out that bad certificate? Perhaps the question can be turned around: How expensive is it to swap out a certificate: in money, effort, and customer displeasure? If it can be done cheaply, quickly, easily, and with no customer notice, then perhaps it should be done frequently, just in case. If done properly, then a frequent certificate change would help limit the scope of any damage, even if a problem is not noticed at first.

But here there be dragons! Some might read the previous paragraph and think that having certificates that expire weekly, for example, eliminates the need to monitor the infrastructure for problems, or the need to revoke a bad certificate. Far from it! All of those steps are necessary as well. Security is a belt-and-suspenders world.

An infrastructure that is well monitored for known threats is another part of the trust equation. If you are confident that your infrastructure and personnel will make you aware of certain types of problems (or potential problems), then you can develop and practice procedures for handling those problems.

That covers the “known unknowns,” as former Secretary of Defense Donald Rumsfeld⁶ said, but what about the “unknown unknowns”? For several years Heartbleed was one of these. The fault in OpenSSL was present and exploitable for those who knew of it and knew how to do so. As of this writing, we do not know for certain if anybody did exploit it, but had someone done so, the nature of the flaw is such that an exploit would have left little or no trace, so it is very difficult to know for sure.

There are two major kinds of “unknown unknowns” to be aware of when providing a network service. The first are those unknowns that you don’t know about, but somebody else might know about and have disclosed or discussed publicly. Let’s call them “discoverable unknowns.” You don’t know about them now, but you can learn about them, either from experience or from the experiences of others.

Discoverable unknowns are discoverable if and only if you make the effort to discover them. The pragmatic way to do this is to create an “intelligence service” of your own. The Internet is full of security resources if you care to use them. It is also full of misdirection, exaggeration, and egotism about security issues. The trick is learning which resources are gold and which are fool’s gold. That comes with practice and, sadly, often at the cost of mistakes both big and small.

A prudent, proactive organization has staff and budget devoted to acquiring and cultivating security resources. These include someone to evaluate likely Web sites, as well as read them regularly; subscriptions to information services; membership in security organizations; travel to conferences; and general cultivation of good contacts. It also includes doing favors for other organizations in similar situations and, if possible, becoming a good citizen and participant in the open-source world. If you help your friends, they will often help you when you need it.

The second type of unknowns can be called “unexpected unknowns.” You don’t know what they are, you don’t even know for sure that they exist, so you are not on the lookout for them specifically. But you can be on the lookout for them in general, by watching the behavior of your network. If you have a way of learning the baseline behavior of your network, system, or application, then you can compare that baseline to what the system is doing now. This could include monitoring servers for unexpected processes, unexpected checksums of key software, files being created in unusual places, unexpected load changes, unexpected network or disk activity, failed attempts to execute privileged programs, or successful attempts that are out of the ordinary. For a network, you might look for unusual protocols, unexpected source or destination IP addresses, or unusually high- or low-traffic profiles. The better you can characterize what your system is supposed to be doing, the more easily you can detect when it is doing something else.

Detecting an anomaly is one thing, but following up on what you’ve detected is at least as important. In the early days of the Internet, Cliff Stoll,⁸ then a graduate student at Lawrence Berkeley Laboratories in California, noticed a 75-cent accounting error on some computer systems he was managing. Many would have ignored it, but it bothered him enough to track it down. That investigation led, step by step, to the discovery of an attacker named Markus Hess, who was arrested, tried, and convicted of espionage and selling information to the Soviet KGB.

Unexpected unknowns might be found, if they can be found at all, by reactive means. Anomalies must be noticed, tracked down, and explained. Logs must be read and understood. But defenses against known attacks can also prevent surprises from unknown ones. Minimizing the “attack surface” of a network also minimizes the opportunities an attacker has for compromise.

Compartmentalization of networks and close characterization of regular traffic patterns can help detect something out of the ordinary.

WHAT CAN YOU DO?

How can issues of trust be managed in a commercial, academic, or industrial computing environment?

The single most important thing that a practitioner can do is to give up the idea that this task will ever be completed. There is no device to buy, no software to install, and no protocol to implement that will be a universal answer for all of your trust and security requirements. There will *never* come a time when you will be done with it and can move on to something else.

Security is a process. It is a martial art that you can learn to apply by study, thought, and constant practice. If you don't drill and practice regularly, you will get rusty at it, and it will not serve you when you need it. Even if you do become expert at it, an attacker may sometimes overpower you. The better you get at the process, however, the smaller the number of opponents that can do you harm, the less damage they can do, and the quicker you can recover.

Here are some basic areas where you can apply your efforts.

KNOW WHOM YOU TRUST AND WHAT YOU TRUST THEM TO DO

Though it is an overused phrase, "Web of Trust" is descriptive of what you are building. Like any sophisticated construction, you should have a plan, diagram, or some other form of enumeration for which trust mechanisms are needed to support your enterprise. The following entities might be on such a plan: data-center provider (power, A/C, LAN); telecommunications link vendors; hardware vendors; paid software vendors; open-source software providers; cryptographic certificate suppliers; time-source suppliers; systems administrators; database administrators; applications administrators; applications programmers; applications designers; and security engineers.

Of course, mileage may vary, and there may be many more entities as well. Whatever is on the list you generate, perform the following exercise for each entry:

- Determine whom this entity trusts to do the job and who trusts this entity.
- Estimate the consequences if this entity were to fail to do the job properly.
- Estimate the consequences if this entity were a bad actor trying to compromise the enterprise in some way (extract information without authorization, deny service, provide bad information to your customers or yourself, etc.).
- Rate each consequence for severity.

KNOW WHAT YOU WOULD DO IF ANY OF THOSE ENTITIES LOST YOUR TRUST

Now that you have a collection of possible ways that your enterprise can be affected, sorted by severity, you can figure out what you would do for each item. This can be as simple or complicated as you are comfortable with, but remember that you are creating a key part of your operations handbook, so if your plans cannot be turned into actions when these circumstances occur, they will not be worth much.

Here are some examples of the kinds of consequences and actions that might be needed:

- A key open-source package is discovered to have a serious bug and must be replaced with a newer, bug-fixed version; replaced with a different package with the same API; replaced with a different

package with a different API; or mitigated until a fix can be developed. Your plan should be a good guide to handling any of these situations.

- A key systems administrator has been providing network access to a potentially unfriendly third party. You must: determine the extent of information lost (or was your information modified?); determine if any systems were compromised with backdoor access; determine which other systems under the sysadmin might be affected; and figure out the best way of handling the personnel issues (e.g., firing, transfer, or legal action).
- A key data center is rendered unusable by a disaster or attack. You must: shift to a standby reserve location; or improvise a backup data center.

PRACTICE, PRACTICE, PRACTICE

Having a plan is all very nice, but if it's in a dusty file cabinet, or worse yet, on a storage volume in a machine that is made unavailable by the circumstances you are planning for, then it doesn't help anybody. Even if the plan is readily available, carrying it out for the first time during a crisis is a good way to ensure that it won't work.

The best way to make sure that your plan is actionable is to practice. That means every plan needs to have a method of simulating the cause and evaluating the result. Sometimes that can be as easy as turning off a redundant server and verifying that service continues. Other problems are more complex to simulate. Even a tabletop exercise, in which people just talk about what is needed, is better than never practicing your contingency plan.

Practice can also take the form of regular operations. For example, Heartbleed required many service providers to revoke and reissue certificates. If that is a critical recovery operation for your enterprise, then find a way to work that procedure into your regular course of business, perhaps by revoking and reissuing a certificate once a month.

Other operations can also benefit from practice, such as restoring a file from backups; rebuilding an important server; transferring operations to a backup data center; or verifying the availability of backup power and your ability to switch over to it.

SET MOUSETRAPS

The most important step in defending against attackers (or Murphy's Law) is learning that you have a problem. If you understand your trust relationships—who is trusted with what and who is not trusted—then watching for violations of those relationships will be very instructive. Every violation will probably fall into one of these categories:

- An undocumented but legitimate trust relationship. This might be sysadmins doing their assigned work, for example, but that work was improperly overlooked when building the trust map.
- A potentially reasonable but unconsidered potential trust relationship that must be evaluated and either added to the trust map or explicitly prohibited—for example, a sysadmin doing unassigned but necessary work to keep a system operational.
- An unreasonable or illegitimate use.

The only way to know which case it is will be to investigate each one and modify your trust map accordingly. As with all things of this nature, mousetraps must be periodically tested to see if they still work.

VET YOUR KEY PEOPLE

Often, trusting a systems administrator takes the form of management saying to sysadmins “Here are the keys to everything,” followed by more-or-less blind trust that those keys will not be abused. Or, to quote science fiction author Robert Heinlein: “It’s amazing how much mature wisdom resembles being too tired.” That sort of blind trust is asking for trouble.

On the other hand, tracking sysadmins closely and forcing them to ask permission for every privileged operation they wish to perform can hobble an organization. Chances are good that both the sysadmins and the granters of permission will grow tired of this, and the organization will move back toward blind trust.

A good way to navigate between these two rocky shoals is to hire good people and treat them well. Almost as important is communicating with them to reinforce the security and trust goals of your organization. If they know what must and must not be done and, at least in general principle, why those constraints are good, then the chances are greater that they will act appropriately in a crunch.

LOG WHAT THEY DO. HAVE SOMEBODY ELSE REVIEW THOSE LOGS REGULARLY

Good people can make mistakes and sometimes even go astray. A regular non-privileged (in the security sense) employee should have a reasonable expectation of workplace privacy, but a systems administrator should know that he or she is being watched when performing sensitive tasks or accessing sensitive resources. In addition, sysadmins should perform extremely sensitive tasks with at least one other person of equal or higher clearance present. That way, someone else can attest that the action taken was necessary and reasonable.

Wherever possible, log what the sysadmins do with their privileges and have a third party review those logs regularly for anomalies. The third party should be distant enough from the systems administrators or other employees given trusted access that no personal or professional relationships will obscure the interpretation of the logs.

Investigate what you suspect and act on what you find. Let your trusted people know in advance that that is what you will do. Let them know that their positions of responsibility make them the first suspects on the list if trust is violated.

MINIMIZE YOUR WINDOWS OF VULNERABILITY

Once you know the ways in which you can be vulnerable, develop plans to minimize and mitigate those vulnerabilities. If you can close the hole, then close it. If you can’t close it, then limit what can be done through the hole. If you can’t limit what can be done, then limit who can do it and when it can be exploited. If you can’t limit anything, then at least measure whether an exploit is taking place. You may not have a perfect solution, but the more limits you put on a potential problem, the less likely it is that it will become a real problem.

LAYER YOUR SECURITY

When it comes to trust, you should not depend on any one entity for security. This is known as “defense in depth.” If you can have multiple layers of encryption, for example, each implemented differently (one depending on OpenSSL, for example, and the other using a different package), then a single vulnerability will not leave you completely exposed.

This is good reason to look at every component of your enterprise and ask: What if these components were to be compromised?

PRACTICE BEING AGILE

If a component were compromised, how would you replace it, and with what? How long would it take to switch over? Theories don't count here. You need to be prepared to switch packages or vendors or hardware in order to be adequately safe. How long will it take your purchasing department to cut paperwork for a new license, for example? How long to get that purchase order signed off? How long for the vendor to deliver?

This is not work you can do once and think you are ready. You need to revisit all components regularly and perform this kind of analysis for each of them as circumstances change.

LOOK AT YOUR NETWORK AS AN ATTACKER WOULD

Know the "as-built" configuration of your network, not just the "as-specified." Remember that the as-built configuration can change every day. This means you have to have people to measure the network, and tools to examine it. What network services does each component provide? Are those services needed? Are they available only to the places they are needed? Are all of the components fully patched? Are they instrumented to detect and report attack attempts? Does someone read the logs? What is the longest period of time between when an attack happens and when somebody notices it? Are there any events (such as holidays) when the length of time an attack goes unnoticed might increase?

The Internet abounds with free or inexpensive software for security analysis. These are tools often used by attackers and defenders. There is something to be learned by looking at your network through the same tools that your attackers use.

TRACK SECURITY ISSUES AND CONFIRM THAT THEY GET FIXED

If you find a problem, how is it tracked? Who is responsible for getting it into the tracking system, getting it to someone who can fix it, and getting it fixed? How do you measure that the problem is present? Do you measure again after the fix is applied to ensure that it worked?

DEVELOP YOUR OWN SECURITY INTELLIGENCE RESOURCES

Does your organization have personnel who track the technology used for potential security issues? How often do they check? Are they listened to when they report a problem?

Any equipment, software, vendors, or people you depend on should be researched on a regular basis. Quality security-focused Web sites exist, but they are often surrounded and outnumbered by those with products to sell or misinformation to distribute. Having staff gain the expertise to distinguish the good from the bad is extremely valuable.

PLAN FOR BIG-TICKET PROBLEMS

If you run a networked enterprise, whether you provide a public, private, or internal suite of services, you will find that trusted services *will* fail you, sooner or later. Repeatedly. How you respond to those failures of trust will become a big part of your company's reputation. If you select your vendors, partners, and components wisely, seriously plan for responses to trouble, and act on your plans when the time comes, then you will fare much better in the long run than those whose crisis planning is filed under "Luck."

CONCLUSION

The problem of trust is not new. If anything, the only *new* part is the mistaken impression that things can be trusted, because so many new things seem to be trustworthy. It is a sometimes-comforting illusion, but an illusion nonetheless. To build anything of value, you will have to place your trust in some people, products, and services. Placing that trust wisely is a skill that is best learned over time. Mistakes will abound along the way. Planning for your mistakes and the mistakes of others is essential to trusting.

It is generally better, faster, and safer to take something that meets good standards of trustworthiness and add value to it—by auditing it, layering on top of it, or adding to the open source—than it is to roll your own. Be prepared to keep a wary eye on the components you select, the system you include them in, and the people who build and maintain that system. Always plan for trouble, because trouble will surely come your way.

You must have some trust if you want to get anything done, but you cannot allow yourself to be complacent. Thomas Jefferson said, “Eternal vigilance is the price of liberty.” It is the price of security as well.

ACKNOWLEDGMENTS

Thanks to Jim Maurer at ACM for requesting this article and George Neville-Neil for reminding me of the Frederick Brooks quote. Special thanks to my wife, Yuki, and my kids for putting up with me as I grumbled to myself while writing.

REFERENCES

1. Brooks Jr., F. P. 1975. *The Mythical Man-Month*. Addison-Wesley.
2. Coverity. 2013. Coverity Scan report finds open source software quality outpaces proprietary code for the first time; <http://www.coverity.com/press-releases/coverity-scan-report-finds-open-source-software-quality-outpaces-proprietary-code-for-the-first-time/>.
3. Dilhac, J.-M. The telegraph of Claude Chappe—an optical telecommunication network for the XVIIIth century: 7; <http://www.ieeehn.org/wiki/images/1/17/Dilhac.pdf>.
4. Hall, E. C. 1996. *Journey to the moon: the history of the Apollo guidance computer*. Reston, VA: AIAA.
5. McMillan, R. 2008. IT admin locks up San Francisco’s network. *PCWorld*; <http://www.pcworld.com/article/148469/article.html>.
6. Rumsfeld, D. 2002. Press conference (February 12); <http://www.c-span.org/video/?168646-1/DefenseDepartmentBriefing102>.
7. Stephenson, N. 1995. *The Diamond Age*. Bantam Spectra.
8. Stoll, C. 1989. *The Cuckoo’s Egg*. Doubleday.
9. Tehranipoor, M., Koushanfar, F. 2010. A survey of hardware Trojan taxonomy and detection. *IEEE*; <http://trust-hub.org/resources/36/download/trojansurvey.pdf>.
10. Thompson, K. 1984. Reflections on trusting trust. *Communications of the ACM*, 27 (8); <http://cm.bell-labs.com/who/ken/trust.html>.
11. Venizia, P. 2008. Sorting out the facts in the Terry Childs case. *CIO*; http://www.cio.com.au/article/255165/sorting_facts_terry_childs_case/?pf=1.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

THOMAS WADLOW is a network and computer security consultant based in San Francisco. He enjoys the many cafes there, but never, ever uses his keys to hold an outdoor table.

© 2014 ACM 1542-7730/14/0500 \$10.00