# CHAPTER 1

# The Foundation of Civilization

*"The value of a thing sometimes lies not in what one attains with it, but in what one pays for it—what it* costs *us."*

*—Frederick Nietzsche*

For the city of London, 1854 was a dreadful year. An outbreak of cholera, the third in 20 years, claimed over ten thousand lives. Six previous city Commissions failed to adequately address London's growing sewage problem, leaving the entire metropolitan area—more than one million people—subject to the vagaries of overflowing cesspools, ill-constructed sewers, contaminated groundwater, and a dangerously polluted Thames River. Considering London was one of the most populated cities at the time and depended heavily on the Thames River, inaction had unfortunate consequences. Sadly, thousands of deaths could not properly motivate Parliament to overcome numerous bureaucratic and political obstacles required to address the crisis.

It was not until an inordinately hot summer in 1858 that the stench of the Thames so overwhelmed all those in close proximity to the river—particularly members of Parliament, many of whom still believed cholera to be an airborne rather than a waterborne pathogen—that resistance finally subsided. The "Great Stink" served as impetus to the largest civic works project London had ever seen.[1]

For the next ten years, Joseph Bazalgette, Chief Engineer of the Metropolitan Board of Works, constructed London's newer and larger sewer network against imposing odds. Despite Parliament's hard-won support and a remarkable design by Bazalgette himself, building a new sewer network in an active and sprawling city raised significant technical and engineering challenges.

Most obvious among these challenges was excavating sewer lines while minimizing disruption to local businesses and the city's necessary daily activities. Less obvious, but no less important, was selecting contracting methods and building materials for such an enormous project. Modern public works projects such as the California Aqueduct, the U.S. Interstate highway system, or China's Three Gorges Dam elicit images of enormous quantities of coordination and concrete. Initially, Bazalgette enjoyed neither.

Selecting suitable building materials was an especially important engineering decision, one that Bazalgette did not take lightly. Building materials needed to bear considerable strain from overhead traffic and buildings as well as survive prolonged exposure to and immersion in water. Traditionally, engineers at the time would have selected Roman cement, a common and inexpensive material used since the fourteenth century, to construct the extensive underground brickworks required for the new sewer system. Roman cement gets its name from its extensive use by the Romans to construct the infrastructure for their republic and empire. The "recipe" for Roman cement was lost during the Dark Ages only to be rediscovered during the Renaissance. This bit of history aside, Bazalgette chose to avoid Roman cement for laying the sewer's brickwork and instead opted in favor of a newer, stronger, but more expensive type of cement called Portland cement.

Portland cement was invented in the kitchen of a British bricklayer named Joseph Aspdin in 1824. What Aspdin discovered during his experimentation that the Romans did not (or were not aware of) was that by first heating some of the ingredients of cement—finely ground limestone and clay—the silica in the clay bonded with the calcium in the limestone, creating a far more durable concrete, one that chemically interacted with any aggregates such as stone or sand added to the cement mixture. Roman cement, in comparison, does not chemically interact with aggregates and therefore simply holds them in suspension. This makes Roman cement weaker in comparison to Portland cement but only in relative, not absolute terms. Many substantial Roman structures including roadways, buildings, and seaports survived nearly 2,000 years to the present.

It is the chemical reaction discovered by Aspdin that gives Portland cement its amazing durability and strength over Roman cement. This chemical reaction also gives Portland cement the interesting characteristic of gaining in strength with both age and immersion in water.[2] If traditional cement sets in one day, Portland cement will be more than four times as hard after a week and over eight times as hard in five years.[3] In choosing a material for such a massive and important project as the London sewer, Portland cement might have rightly appeared to Bazalgette as the obvious choice. There was only one problem: Portland cement is unreliable if the production process varies even slightly.

*Portland cement might have rightly appeared to Bazalgette as the obvious choice. There was only one problem: Portland cement is unreliable if the production process varies even slightly.*

The strength and therefore the reliability of Portland cement is significantly diminished by what would appear to the average observer as minuscule, almost trivial changes in mixture ratios, kiln temperature, or grinding process. In the mid-nineteenth century, quality control processes were largely nonexistent, and where they did exist were inconsistently employed—based more on personal opinion rather than objective criteria. The "state of the art" in nineteenth century quality control meant that while Portland cement was promising, it was a risky choice on the part of Bazalgette. To mitigate any inconsistencies in producing Portland cement for the sewer project, Bazalgette created rigorous, objective, and some would say draconian testing procedures to ensure each batch of Portland cement afforded the necessary resiliency and strength. His reputation as an engineer and the success of the project depended on it.

Bazalgette enforced the following regimen: Delivered cement sat at the construction site for at least three weeks to acclimate to local environmental conditions. After the elapsed time, samples were taken from every tenth sack and made into molds that were immediately dropped into water where the concrete would remain for seven days. Afterward, samples were tested for strength. If any sample failed to bear weight of at least five hundred pounds (more than twice that of Roman cement), the entire

delivery was rejected.[4] By 1865, more than 11,587 tests were conducted on 70,000 tons of cement for the southern section of the sewerage alone.[5] Bazalgette's testing methodology proved so thorough, the Metropolitan Board who oversaw the project eventually agreed to Bazalgette's request to construct sewers entirely from concrete. This not only decreased the time required to construct the sewerage, but eliminated the considerable associated cost of the brickworks themselves.[6]

Once completed, Bazalgette's sewer system saved hundreds of thousands of lives by preventing future cholera and typhoid epidemics.[7] The sewer system also made the Thames one of the cleanest metropolitan rivers in the world and changed the face of river-side London forever. By 1872, the Registrar-General's Annual Report stated that the annual death rate in London was far below any other major European, American, or Indian city, and at 3.3 million people (almost three times the population from the time Bazalgette started his project), London was by far the largest city in the world. This state of affairs was unprecedented for the time. By 1896 cholera was so rare in London, the Registrar-General classified cholera as an "exotic disease." Bazalgette's sewer network, as well as the original cement used in its construction, remains in use to this day. Given that Portland cement increases with strength over time, it is likely London's sewer system will outlive even some of Rome's longest standing architectural accomplishments such as the aqueducts and the Pantheon.

## *Software and Cement*

While Bazalgette's design of the sewer network was certainly important, in hindsight the selection and qualification of Portland cement was arguably the most critical aspect to the project's success. Had Bazalgette not enforced strict quality control on production of Portland cement, the outcome of the "Great Stink of London" might have been far different. Due to Bazalgette's efforts and the resounding success of the London sewer system, Portland cement progressed in a few short years from "promising but risky" to the industry standard used in just about every major construction project from that time onward.

Portland cement's popularity then, is due not just to its physical properties, but in large part to Bazalgette's strict and rigorous quality tests, which drastically reduced potential uncertainties associated with Portland cement's production. At present, more than 20 separate tests are used to ensure the quality of Portland cement, significantly more than Bazalgette himself employed. World production of Portland cement exceeded two billion metric tons in 2005, with China accounting for nearly half of that production followed closely by India and the United States.[8] This works out to roughly 2.5 tons of cement for every person on the planet. Without Portland cement, much of modern civilization as we know it, see it, live on it, and drive on it would fail to exist.

Cement is everywhere in modern civilization. Mixed with aggregates such as sand and stone, it forms concrete that comprises roadways, bridges, tunnels, building foundations, walls, floors, airports, docks, dams, aqueducts, pipes, and the list goes on. Cement is—quite literally—the foundation of modern civilization, creating the infrastructure that supports billions of lives around the globe. One cannot live in modern civilization without touching, seeing, or relying on cement in one way or another. Our very lives depend on cement, yet cement has proven so reliable due to strict quality controls that it has to a large extent disappeared from our field of concerns—even though we are surrounded by it. Such is the legacy of Bazalgette's commitment to quality: We can live our lives without thinking twice about what is beneath our feet, or more importantly, what may be above our head.

Civilization depends on infrastructure, and infrastructure depends, at least in part, on durable, reliable cement. Due to its versatility, cost-effectiveness, and broad availability, cement has provided options in construction that could not otherwise be attained with stone, wood, or steel alone. But since the 1950s, a new material has been slowly and unrelentingly injected into modern infrastructure, one that is far more versatile, cost-effective, and widely available than cement could ever hope to be. It also just so happens to be invisible and unvisualizable. In fact, it is not a material at all. It is software.

Like cement, software is everywhere in modern civilization. Software is in your mobile phone, on your home computer, in

cars, airplanes, hospitals, businesses, public utilities, financial systems, and national defense systems. Software is an increasingly critical component in the operation of infrastructures, cutting across almost every aspect of global, national, social, and economic function. One cannot live in modern civilization without touching, being touched by, or depending on software in one way or another.

*Like cement, software is everywhere in modern civilization.*

Software helps deliver oil to our cities, electricity to our homes, water to our crops, products to our markets, money to our banks, and information to our minds. It allows us to share pictures, music, thoughts, and ideas with people we might meet infrequently in person but will intimately know from a distance. Everything is becoming "smarter" because software is being injected into just about *every thing*. Software has accelerated economic growth through the increased facilities of managing labor and capital with unprecedented capacity. Hundreds of thousands of people if not millions owe their livelihoods to software. With its aid, we have discovered new medicines, new oil fields, and new planets and it has given us new ways of visualizing old problems, thereby finding solutions we might never have had the capacity, time, or ability to discover without it. With software we are able to build bridges once thought impossible, create buildings once thought unrealistic, and explore regions of earth, space, and self once thought unreachable.

Software has also given us the Internet, a massive world-wide network connecting all to all. In fact, connectedness in the twenty-first century is primarily a manifestation of software. Software handles the protocols necessary for communication, operates telecommunications equipment, bundles data for transmission, and routes messages to far-flung destinations as well as giving function and feature to a dizzying array of devices. Software helps connect everything to everything else with the network—the Internet—merely a by-product of its function. Without software, the network would be just a bunch of cables, just as a human cell without DNA would be just a bunch of amino acids and proteins.

Software is *everywhere*; it is everywhere because software is the closest thing we have to a universal tool. It exhibits a radical malleability that allows us to do with it what we will. Software itself is nothing more than a set of commands that tells a computer processor (a microchip) what to do. Connect a microchip to a toy, and the toy becomes "smart;" connect a microchip to a car's fuel injector, and the car becomes more fuel efficient; connect it to a phone, and the phone becomes indispensable in life's everyday affairs. Connect a microchip to just about anything, and just about anything is possible because the software makes it so. Software is the ghost in the machine, the DNA of technology; it is what gives *things* the appearance of intelligence when none can possibly exist.

The only aspect of software more impressive than software itself is the people that create software. Computer programmers, also known as software developers or software engineers, write the instructions that tell computers what to do. Software developers are in large part a collection of extremely talented and gifted individuals whose capacity to envision and implement algorithms of extraordinary complexity and elegance gives us search engines, operating systems, word processors, instant messaging, mobile networks, satellite navigation, smart cars, advanced medical imaging; the list goes on. As such, software is a human creation, and as a human creation it is subject to the strengths and foibles of humanity. This is where the similarities of cement and software become most interesting.

Software, like cement before it, is becoming the foundation of civilization. Our very lives are becoming more dependent on and subject to software. As such, the properties of software matter greatly: quality, reliability, security, each by themselves accomplish very little, but their absence faults everything else. Like Portland cement, software can be unreliable if production processes vary even slightly. Whereas variations in kiln temperatures, mixture ratios, or grinding processes can detrimentally affect the strength and durability of Portland cement after it has been poured, there are a host of similar, seemingly trivial variations in producing software that can detrimentally affect its "strength" when "poured" into microchips. It is up to humans to get the production process right.

Unlike Portland cement, for more than 50 years software of all types and function has been continuously released into the stream of commerce, plagued by design and implementation defects that were largely detectable and preventable by manufacturers, but were not. This has and does result in catastrophic accidents, significant financial losses, and even death. The trepidation over insufficient software manufacturing practices extends back to the late 1960s when the North American Treaty Organization (NATO) convened a panel of 50 experts to address the "software crisis." While the panel did not provide any direct solutions, the concept of a "software engineer" was developed as a means to more closely align software manufacturing with the engineering discipline rather than artistic creativity. The intent, as far as we can tell, was to remove the "rule of thumb" in the production of software and all the inconsistencies such approximation introduces. After 50 years, defining what actually constitutes the principles and practice of software engineering has not progressed far. What is clear, however, is that the unfortunate history of software blunders sullies the reputation of software in general and distorts the genius of software developers in particular.

*What is clear, however, is that the unfortunate history of software blunders sullies the reputation of software in general and distorts the genius of software developers in particular.*

Perhaps most frustrating is the inconsistent use of quality control measures by such a wide range of software manufacturers for such an extended period of time. Software is infinitely more complex than cement to be sure, but complexity does not entirely account for systemic, reoccurring software manufacturing defects. Quality control measures—even in the absence of a clear definition for software engineering—have been and are available specifically to address problems with software production.

Software has its own modern-day equivalent of Joseph Bazalgette: his name is Watts Humphrey. Humphrey is a fellow and research scientist at Carnegie Mellon University's Software Engineering Institute (SEI) and is often called the "father of software quality" having developed numerous methodologies since

the 1980s for designing quality and reliability into software products. In 2005, President George W. Bush awarded Mr. Humphrey the National Medal of Technology, the highest honor for innovation in the United States. The only problem in this story is that a significant portion of software manufacturers around the world still largely ignore or only superficially implement Humphrey's guidance. As a result, the Software Engineering Institute noted at the beginning of the twenty-first century that software was getting worse, not better. Such a proclamation augurs ill for civilization's newest foundation.

But if software quality were the only issue, perhaps we could discount the problem of low-quality software simply on the basis of "growing pains." After all, at 50 years old, some might argue software is still a relatively new phenomenon and that such failures in quality are understandable and even tolerable for such a young technology. When civil engineering was 50 years old, for instance, the brick had not even been invented yet.[9]

Yet when civil engineering was 50 years old, the profession was not building and connecting global infrastructure. Software's newness has not precluded it from being injected into nearly every aspect of modern civilization. That software connects everything to everything else magnifies even the smallest foibles in software production. This introduces a critical aspect of software vastly different from weaknesses in traditional building materials: once interconnected, even the smallest piece of insecure software may have global consequences. New or not, software needs to be worthy of its place.

Weaknesses or defects in software can not only result in a given software application failing for one reason or another (including no reason), but software defects can potentially be exploited by hackers, who, discovering or knowing the weakness exists, may use it to surreptitiously access and control a system from a continent away, stealing sensitive personal information such as credit cards or social security numbers or absconding with trade secrets or intellectual property. Such weaknesses could also be used to hijack computer systems and then turn those systems against their owners or against other nations and other peoples. In the end, insecure software is *right now* resulting in economic and social costs that are now well into billions of dollars per year with no sign of abatement. The trend is disturbing.

Understanding why this situation persists and seems to be only getting worse has important implications for modern civilization. In other words, new or not, society inevitably demands any technology used in the foundation of civilization, whether cement or software, should be given the time and attention foundations deserve. Bazalgette and his legacy expected no less; nor should we.

## *In the Shadow of Utility*

The litany of documented software failures is extensive and tragic.[10] It does not take much effort to find examples of software failures resulting in loss of life, limb, money, time, or property. The trend only promises to become worse as software becomes more critical to almost every aspect of modern life; yet, software manufacturers enjoy an astonishing amount of insulation from government oversight, legal liability, consumer retaliation, and indeed, as some critics have observed, engineering skill. A proven record of significant, costly, and deadly failures with no significant decline in use by its victims is baffling. On top of—in fact, despite—these shortcomings, victims (consumers, corporations, and governments included) lavishly spend on acquiring and defending a clearly defective product. Why?

*Software manufacturers enjoy an astonishing amount of insulation from government oversight, legal liability, consumer retaliation, and indeed, as some critics have observed, engineering skill.*

Why do software manufacturers continue to produce and consumers continue to purchase unreliable and insecure software?

Why do software users willingly and repeatedly accept licensing agreements that absolve software manufacturers of most forms of liability for any design or application defects that might result in injury, harm, or damages?

Why do governments make so few demands on software manufacturers while placing onerous compliance requirements on software buyers, who are least qualified to address the problems associated with software manufacturing?

Why should software not be subject to the same public policy concerns applied to other critical elements of national infrastructure?

Why do chickens cross the road?

Each of these questions is answered in part by this simple response: *to maximize utility*. We all do things that might appear perfectly acceptable in our own eyes that might appear perfectly crazy to someone else. A chicken crossing the road in the presence of drivers who may be willing to flatten the poor thing simply to interrupt the monotony of driving might appear rather crazy to an outside observer. In fact, from an economist's perspective, this is perfectly rational behavior on the part of the chicken so long as the chicken believes it will be better off for the crossing. Jumping out of an airplane with a parachute might seem perfectly crazy to observers, unless the skydiver believes they are better off for the jumping. Likewise, software buyers continuing to accept software licensing terms that put them at a distinct disadvantage legally, financially, or personally should the software fail might appear perfectly baffling, unless buyers believe they will be better off for the accepting.

Economists use the notion of *utility* to help explain why people behave the way they do. The concept of utility is a little like the concept of "happiness" only more general. I explain the concept of utility in more detail in Chapter 2, "Six Billion Crash Test Dummies," but sufficed to say, utility centers around the notion that most of us want to make our lives better, and that many of our life decisions are probably based on this desire. Software inarguably makes our life better, but like crossing the road or jumping out of an airplane or owning a swimming pool, everything has a cost.

It is not always the utility we get out of something or some activity that matters most, but how much it potentially costs us. Costs are not always obvious to the individual at time of "purchase" so to speak, and can be hidden or otherwise obscured. In general, cost can be measured in private terms, what it directly costs an individual to behave in a certain way, or measured in social costs, what it costs society for an individual to undertake a certain activity. The balance of private and social costs is the focus of many public policy efforts.

The private cost of smoking, for instance, is relatively low monetarily from an individual's view point, but can impose substantial social costs due to the prolonged medical services associated with caring for long-term chronic smokers. Imposing a cigarette tax is but one way to raise the private cost of an activity in order to deter the behavior, which thereby potentially reduces the social cost by reducing the total number of smokers in the population and how much they smoke.

People's evaluation of utility versus cost can lead to some fairly interesting situations. As a case in point, in the United States swimming pools kill or injure more children under the age of 14 than firearms. At 16 percent, accidental drowning was the second leading cause of injury-related death of children aged 14 and under in 2004 (car accidents ranked first); compare this with only 1 percent of children that died due to accidental discharge of firearms.[11] In fact, injury-related death due to accidental discharge of firearms ranks at the bottom of all other causes of death and injury among children including choking (17 percent), fire and burns (10 percent), and bicycle accidents, poisoning, and falls (each at 2 percent).

There are plenty of people, and parents in particular, who might forbid children playing at the home of a neighbor who possesses one or more firearms, but the likelihood of a child drowning at a neighborhood pool party is far higher than a child being injured or killed by the firearm of a neighbor. Yet few parents espouse an anti-swimming pool sentiment or join anti-swimming pool action groups as they would for firearms, even though statistics would certainly warrant such behavior. The rather simplistic answer to this incongruency is that a larger portion of the population sees the intrinsic utility of a swimming pool over and above the utility of possessing a hand gun. Yet a swimming pool incurs a much higher cost to both families and society than do firearms. Even things with obvious utility like a swimming pool can have a dark shadow.

Played out against this background of people's desire for utility (and not always recognizing the real cost), is the story of software. The questions at the start of this section really touch on the issues of self-interest and, more importantly, the incentives we have as individuals to undertake certain activities and the

utility we derive. Understanding incentives also gives us a possible foundation to address the issues of why software manufacturing seems to be in the state it is in. If it is up to humans to get the production processes for Portland cement and software correct, then it is just as important, if not more so, to understand why humans behave as they do. Incentives are a good place to start.

As such, *Geekonomics* is not so much the story of software told through the lens of technology, but through the lens of humanity, specifically the incentives for manufacturing, buying, and exploiting insecure software. Economics is simply one way of understanding why humans behave as they do. But if economics is generally described as "the dismal science," then software engineering is economics' freakish, serotonin-deprived cousin. Economics is positively cheery and approachable in comparison. To date, the discussion regarding software has been largely dominated by technology experts whose explanations largely serve to alienate the very people that are touched most by software. *Us*.

Yet the congress of these two disciplines tells an important and consequential story affecting both the reader's everyday life and the welfare of the global community. The issue of insecure software is at least as much about economics as it is about technology. And so I discuss both in this book. This book is not intended to be a comprehensive economics text, a litany of software failures (although this is sometimes inevitable), a diatribe as to how the world is coming apart at the seams, or a prophecy that civilization's ultimate demise will occur because of "bad" software. Prophesizing disaster is cliché. Bad things happen all the time, and forecasting tragic events does not require an exceptional amount of talent, intelligence, or foresight. If anything, the world tolerates disaster and somehow still makes progress. This does not mean valid threats to economic and national stability due to "bad" software are illusory or should be minimized. On the contrary, the story of insecure software has not been readily approachable and therefore not well understood. We cannot manage what we do not understand, including ourselves. Software is a ghost in the machine and, at times, frustratingly so. But as software is a human creation, it does need to remain a frustrating ghost.

My intent in this book is to give this story—the story of insecure software—a suitable voice so that readers from any walk of life can understand the implications. I promise the reader that there is not a single graph in this book; nor is there a single snippet of code. This story should be accessible to more than the experts because it is we who create this story and are touched by it daily. The consequences are too great and far-reaching for the average person to remain unaware.

The first task of *Geekonomics*, then, is to address the questions presented at the beginning of this section as completely as possible within the confines of a single book. This means some aspects may be incomplete or not as complete as some readers might prefer. However, if anything, the story of software can be entertaining, and this book is intended to do that as well as inform and enlighten.

The second and more difficult task of *Geekonomics* is to analyze what the real cost of insecure software might be. Swimming pools can have a high cost, but how costly is insecure software, really? This is a challenging task considering that unlike statistics regarding accidental drowning, good data on which to base cost estimates regarding insecure software is notoriously lacking and inaccurate for two reasons. First, there is presumed to be a significant amount of underreporting given that many organizations might not realize they have been hacked or do not want to publicly share such information for fear of consumer retaliation or bad publicity. Second, actual costs tend to be distorted based on the incentives of those reporting their losses. For some victims, they may tend to inflate losses in an effort to increase their chances of recovering damages in court. Other groups of victims might deflate costs in an effort to quell any uprisings on the part of customers or shareholders. Law enforcement and cyber security companies can tend to inflate numbers in an effort to gain more funding or more clients, respectively. Whatever the incentives might be for reporting high or low, somewhere within these numbers is a hint to what is actually going on.

The third and final task of *Geekonomics* is to identify current incentives of market participants and what new incentives might be necessary to change status quo. One alternative is always choosing to do nothing; simply let things work themselves out on their own, or more accurately, let the market determine what

should be done. This book argues against such action. Any intervention into a market carries with it the risk of shock, and doing nothing is certainly one way of avoiding such risk. But intervention is necessary when a condition is likely to degenerate if nothing is done. The magnitude of the risk is great enough and the signs of degeneration clear enough that new and different incentives are needed to motivate software manufacturers to produce and software buyers to demand safer, higher quality, and more secure software.

## *Fragile Analogies*

Writing a book is far easier than writing software. If the text in a book should have "bugs" such as ambiguities, inconsistencies, or worse, contradictions, you the reader might be annoyed, even angry, but you will still have your wits about you. Simply shrug your shoulders, turn the page and read on. This is because, as a human, you are a perceptive creature and can deal to a greater or lesser extent with the paradoxical and ambiguous nature of reality. Computers are not nearly so lucky. As Peter Drucker, a legendary management consultant, pointed out in *The Effective Executive* more than 40 years ago, computers are logical morons. In other words, *computers are stupid*. This is the first important realization toward protecting modern infrastructure. Computers are stupid because logic is essentially stupid: logic only does what logic permits.[12] Computers do exactly as they are instructed by software, no more and no less. If the software is "wrong," so too will be the computer. Unless the software developer anticipates problems ahead of time, the computer will not be able to simply shrug, turn the page, and move on.

Computers cannot intrinsically deal with ambiguity or uncertainly with as much deft and acumen as humans. Software must be correct, or it is nothing at all. So whereas humans live and even thrive in a universe full of logical contradictions and inconsistencies, computers live in a neat, tidy little world defined by logic. Yet that logic is written primarily by perceptive creatures known as software developers, who at times *perceive* better than they *reason*. This makes the radical malleability of software both blessing and bane. We can do with software as we will, but what we *will* can sometimes be far different than what we *mean*.

The radical malleability of software also poses additional explanatory complications. Software is like cement because it is being injected into the foundation of civilization. Software is also like a swimming pool because people opt to use it even though statistics tend to show the high private and social costs of its use. In fact, in this book, software is described to be like automobiles, DNA, broken windows, freeways, aeronautical charts, books, products, manuscripts, factories, and so on. Software might even be like a box of chocolates. You never know what you're going to get. With software, all analogies are fragile and incomplete.

Cement is an imperfect analogy for software, but so too is just about everything else, which means analogies used to understand software tend to break quite easily if over-extended. The radical malleability of software means any single analogy used to understand software will be somewhat unsatisfying, as will, unfortunately, any single solution employed to solve the problem of insecure software. As a universal tool, software can take far too many potential forms for any one analogy to allow us to sufficiently grasp software and wrestle it to the ground. This challenge is nowhere more obvious than in the judicial courts of the United States, which reason by analogy from known concepts.[13] This does not mean that software cannot be understood, simply that significantly more mental effort must be applied to think about software in a certain way, in the right context, and under the relevant assumptions. As such, this book may liberally switch between analogies to make certain points. This is more the nature of software and less the idiosyncrasies of the author (or at least, I would hope).

Finally, there are many different kinds of software: enterprise software, consumer software, embedded software, open source software, and the list goes on. Experts in the field prefer to distinguish between these types of software because each has a different function and different relevancies to the tasks they are designed for. Such is the radical malleability of software.

A fatal flaw of any book on software, therefore, is the lack of deference to the wild array of software in the world. The software in your car is different than in your home computer, is different than the software in space shuttles, is different than software in airplanes, is different than software in medical devices,

and so on. As such, one can argue that the quality of software will differ by its intended use. Software in websites will have different and probably lower quality than software in airplanes. And this is true. There is only one problem with this reasoning: Hackers could care less about these distinctions.

At the point when software is injected into a product and that product is made available to the consumer (or in any other way allows the attacker to touch or interact with the software), it is fair game for exploitation. This includes automobiles, mobile phones, video game consoles, and even nuclear reactors. Once the software is connected to a network, particularly the Internet, the software is nothing more than a target. As a case in point, two men were charged with hacking into the Los Angeles city traffic center to turn off traffic lights at four intersections in August 2006. It took four days to return the city's traffic control system to normal operation as the hackers locked out others from the system.[14] Given that more and more products are becoming "network aware;" that is, they are connected to and can communicate across a digital network, software of any kind regardless of its *intended* use is fair game in the eyes of an attacker. As William Cheswick and Steven Bellovin noted in *Firewalls and Internet Security*, "Any program no matter how innocuous it seems can harbor security holes…We have a firm belief that everything is guilty until proven innocent."

This is not paranoia on the part of the authors; this is the reality.

Therefore, I have chosen to distinguish primarily between two types of software: software that is networked, such as the software on your home computer or mobile phone, and software that is not. The software controlling a car's transmission is not networked; that is, it is not connected to the Internet, at least not yet. Though not connected to the Internet, weaknesses in this software can still potentially harm the occupants as I illuminate in Chapter 2. But it is only a matter of time before the software in your transmission, as with most all other devices, will be connected to a global network. Once connection occurs the nature of the game changes and so too does the impact of even the tiniest mistake in software production. That software has different *intended* uses by the manufacturers is no excuse for

failing to prepare it for an actively and proven hostile environment, as Chapter 3, "The Power of Weaknesses," highlights.

Finally, the radical malleability of software has moved me to group multiple aspects of insufficient software manufacturing practices such as software defects, errors, faults, and vulnerabilities under the rubric of "software weaknesses." This might appear at first as overly simplistic, but for this type of discussion, it is arguably sufficient for the task at hand.