Vendor Liability For Computer Security Defects

Benedict Dugan © 2003

Introduction

In a networked environment, both vendors and users of insecure software externalize the costs associated with network failures due to malicious mobile code. As networked computing resources become increasingly integrated into society, the expected cost of poor network security can be expected to increase super-linearly. Currently, software vendors are entitled to develop software containing security flaws without fear of legal repercussions. The lack of legal remedies, coupled with market flaws, does not tend to encourage vendors to produce more secure software.

Overview

This paper focuses primarily on one aspect of the network security problem, specifically the so-called buffer-overflow vulnerability. The paper is divided into three main sections. The first provides a technical description of the buffer overflow exploit, together with an overview of the exploit's past impacts and future risks. The next section explores potential solutions to the problem in the context of an abstract legal framework. The final section recommends a publicly enforced standards-based approach to reduce the number of initial software security defects. Additionally, strict liability for un-repaired defects in deployed software, cushioned by near-term immunity, will force vendors to move to alternate methods of software distribution.

Buffer Overflow Vulnerabilities and Exploits

All modern computers are *stored program computers*, that is, the instructions that make up a given program are stored in the computer memory alongside the data that the program will operate upon. This feature gives computers incredible flexibility in that they can simulate other kinds of machines. In other words, a stored program computer is a machine that can *become*

another machine described by the program that it currently executes. A program can be thought of as a description of a machine; a computer executing that program implements or instantiates that machine. It is this chameleon-like quality of the stored program computer that enables the buffer overflow attack. The goal of a buffer-overflow attack is to hijack the operation of a running computer. If, somehow, the attacker can "inject" code into a running program, they can assume the identity of that program. The attacker has two objectives: first, they need to inject a sequence of instructions into the memory space of a program, and second, they need to get the CPU to start executing those instructions. Unfortunately, in many cases, this is easier than it sounds.

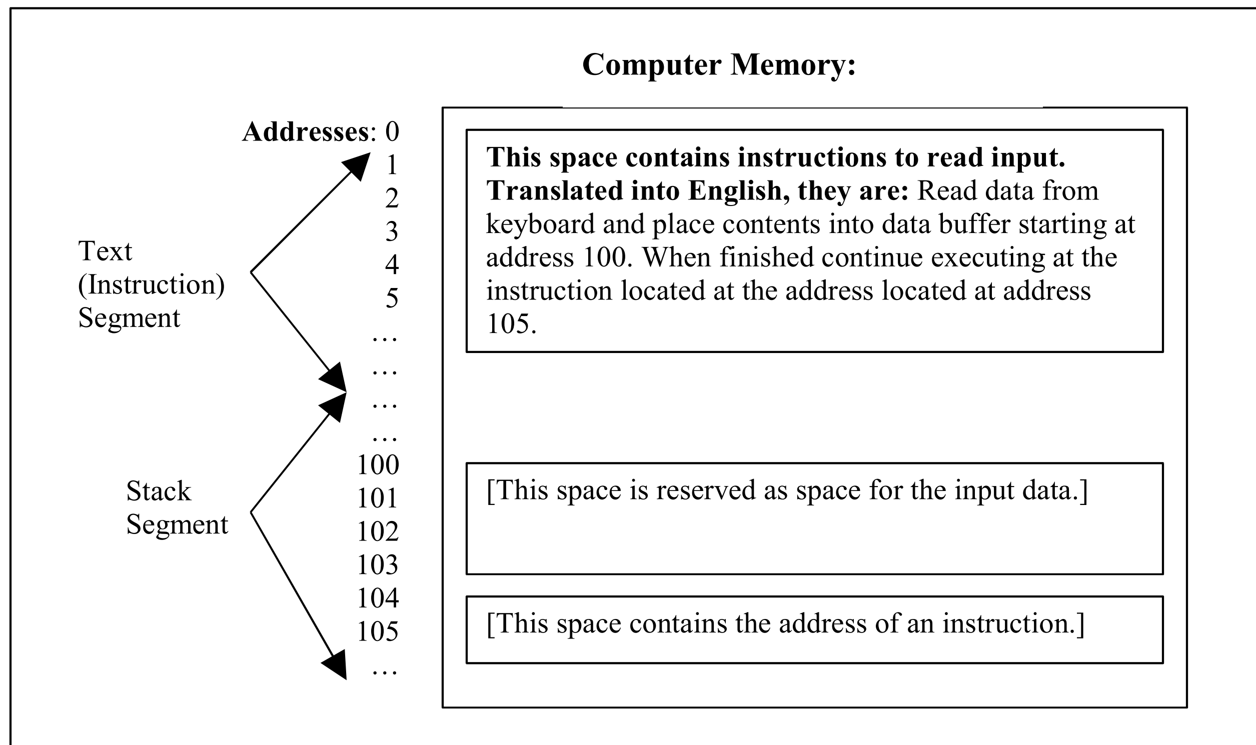
Achieving the first objective simply requires a means to input rogue instructions into the computer's memory. Since all interesting computer programs perform input, there appear ample opportunities to do this. Consider a program that first reads a username from a terminal into computer memory during a log-on procedure. Rather than inputting a user name, the attacker simply inputs the characters that correspond to the instructions they wish to have executed. These characters are typically read into a region of memory - called a buffer - for later processing, such as comparing the username to the database of account records. Once the instructions are read into memory by the computer, the attacker has successfully achieved the first goal of her attack.

At this point, the attacker has only injected malicious code into a data buffer managed by the running program. Without making the CPU execute those instructions, the program is not in danger. Getting the CPU to do this requires a small digression into the organization of a typical computer memory. Generally, a computer memory is logically comprised of (at least) two regions (sometimes called segments): a text segment, which contains instructions, and a stack segment, used to hold data that the program operates upon. In our example, the data buffer that contains malicious instructions is located in the stack segment. Diagram 1, below, illustrates the layout of a computer's memory, just before a buffer overflow attack takes place. It shows portions of both the

text segment and the stack segment, together with their contents. The text segment is labeled with a human readable version of the binary instructions that would reside there.

Diagram 1: Just Before the Buffer Overflow Attack.

The key to getting the CPU to execute the malicious code that the attacker will inject into the program lies in overflowing the input buffer and overwriting critical data adjacent to that buffer. In this example, the buffer resides in five sequential memory cells, starting at address 100. In the



105th cell is the so-called return address, which is the address of the instruction the CPU will execute after it finishes reading input from the user.[1]

Now, suppose that the attacker can input just the right sequence of characters that not only fills the buffer with malicious instructions, but also ultimately overflows that buffer and overwrites the return address that lies adjacent to it. The value used by the attacker to overwrite the return

[1] Return addresses are stored in the stack segment in order to implement procedure (subroutine) calls within programs. Saving the return address enables the continued execution of a program after a subroutine finishes executing.

address is the address of the beginning of the buffer, because that is where the first malicious instruction lives. Note that the input reading routine, when it finishes, fetches an address contained at memory location 105, and continues by executing the instruction located at that address. Generally, that will be the address of an instruction in the "real" program, but in this case, the attacker has overwritten it with the address of the first instruction of their instruction stream. Diagram 2, below, illustrates the situation in computer memory just after the buffer overflow has taken place. The changes have been emphasized in italics.

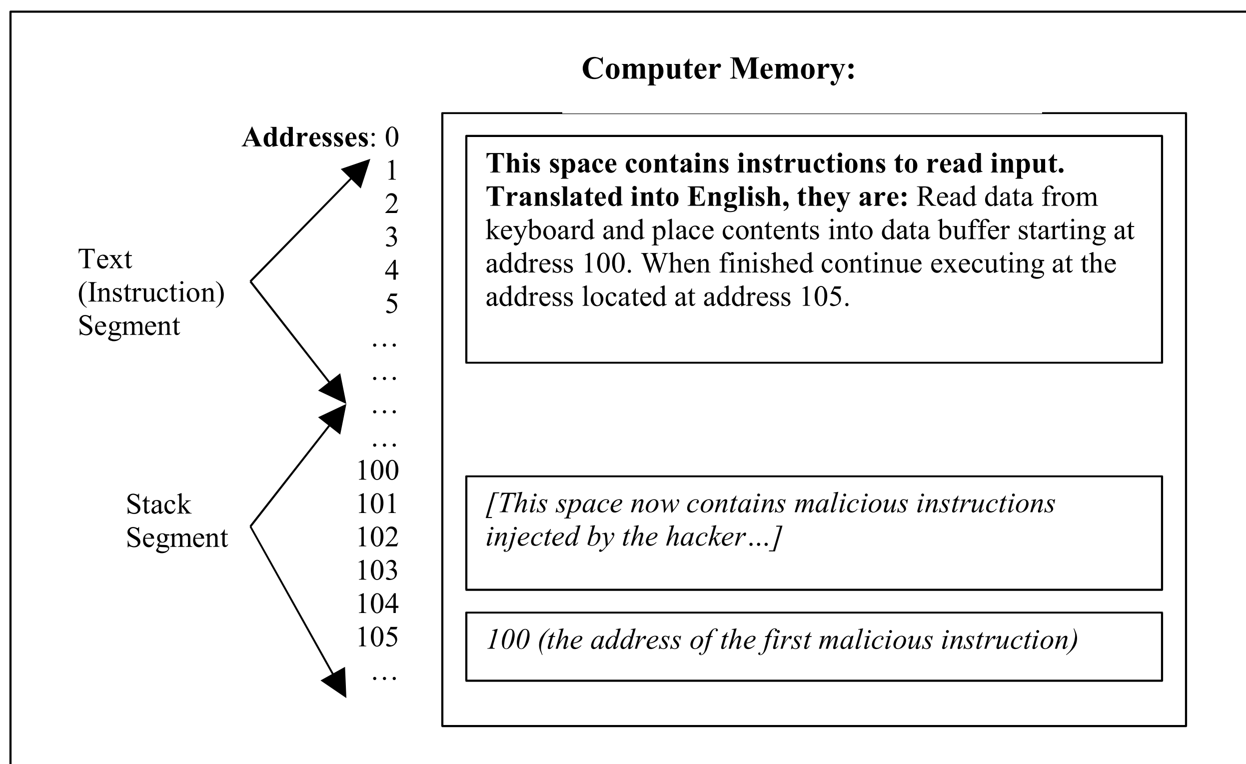


Diagram 2: Just After the Buffer Overflow

In this example, once the attacker has input her carefully constructed input sequence that overflows the buffer, control will be passed to her code, and she will now assume the identity and rights of the running program. This means that if the program is running at a level of elevated privilege, she - or rather the program, as her proxy - will not only be able to read, modify, or even

destroy files belonging to ordinary users, but also have control over privileged data within the system, such as password files.

Worms: Exploiting Buffer Overflow Vulnerabilities in a Networked Environment

In modern times, computers operate in a highly networked environment populated by client and server programs. Network connections provide a channel through which hackers can launch attacks by means of buffer overflows, or other vulnerabilities. The connectivity afforded by networks, coupled with the automation afforded by programmatic attacks, allow hackers to write self-replicating worms that spread across the Internet, overtaking one server after another. Assuming that the original code continues attacking other servers, it is easy to see that most worms will rapidly (geometrically) take over large numbers of machines. As we shall see below, these worm-driven, automated attacks can spread across the Internet in a pattern similar to that of an infection through biological populations.

The potential scale of an epidemic increases with a number of factors: the number of hosts on a particular network, the number and kinds of services running on each host, the number of security vulnerabilities, and the homogeneity of software and hardware. The number of connected hosts obviously describes an upper limit on the reach of the infection. Obviously, the degree of connectivity itself varies from host to host, depending on the number and kinds of services running on that host. A home machine, for example, will typically run fewer services than a corporate machine that handles e-mail, file-transfer, printing and so on.[2] The number of security vulnerabilities serves to constrain the channels through which infections can spread. While it is

[2] Of course, many systems are commonly shipped with unnecessary services enabled. For example, Microsoft IIS (the Microsoft web server) is enabled by default on standard installations of Windows 2000. As most home users will never have a need to run their own web server, and since Internet worms commonly target IIS, shipping systems with IIS disabled by default is perhaps the cheapest way to avoid the risk of a large class of infections.

easier said than done, eliminating all vulnerabilities will eliminate potential epidemics. As we shall see below, however, it may be possible to eliminate a broad class of vulnerability (in this case, buffer overflows) and sharply reduce the network's susceptibility to infection.

Finally, the homogeneity of the network plays an important role. A single worm can only successfully attack a single combination of software, operating system, and machine architecture.[3] Hence, the hacker's workload increases with the number of platforms. Of course, a world where the number of kinds of platforms equals the number of hosts may be inconceivable, but a dozen combinations is not out of the question.[4] The lesson here is that there is a strong analog between cyber-diversity and bio-diversity: just as mono-cultural agriculture is more susceptible to devastating insect or fungal attack, so is a monoculture of computer systems.[5]

Some Famous Worms in History

The grandaddy of all worms is the Morris worm of November 1988. This worm spread through a variety of mechanisms, but in large part relied on a buffer overflow vulnerability in the *finger daemon*, a UNIX server that provides information regarding users on its host machine. In

[3] This is so because if an attacker discovers a buffer overflow vulnerability in one version of a particular server (e.g. a web server written by vendor X, version Y, for operating system Z), they can compromise all machines on the network that are running that particular version of the web server. The reason is that the code and data layout will be identical across all identical versions of software running on binary-compatible hardware, with the same operating system. There are worms that can attack multiple platform combinations, but these can really be viewed as N separate worms, one each per platform.

[4] Just two to three vendors each of hardware, software, and operating system should yield eight to 27 possible combinations.

[5] This observation was made as early as 1988 in the aftermath of the highly publicized worm, the Morris Worm. "Diversity is good. Though the virus picked on the most widespread operating system use on the Internet and the two most popular machine types, most of the machines on the network were never in danger. A wider variety of implementations is probably good, not bad. There is a direct analogy with biological diversity to be made." Mark Eichin & Jon Rochlis, *With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988*, IEEE Symposium on Research in Security and Privacy (1989), *available at* http://www.mit.edu/people/eichin/virus/main.html.

1988, the Internet consisted only of 60,000 hosts, yet the worm managed to infect between 1000 and 6000 of them in a matter of hours. Dollar losses were estimated to run between $100,000 and $10 million.[6] In July of 2001, the Code-Red family of worms made its rounds and infected 360,000 hosts. The worm's population doubled every 37 minutes, adding 2000 new hosts per minute at its peak. This worm exploited a buffer overflow vulnerability in the Microsoft IIS (web-server) and cost businesses an estimated $2.6 billion.[7] In January of 2003, the SQLSlammer worm exploited a buffer overflow vulnerability to attack the Microsoft SQL Server. The Slammer worm was exceptionally virulent, infecting 75,000 hosts in just 15 minutes.[8] The worm's population doubled in size every 8 seconds; at it's peak, the worm scanned 55 million hosts per second. While the Slammer didn't actually destroy data, it did disrupt the operation of the database server and the network itself, resulting in ATM crashes, flight cancellations, and 911 disruptions.[9] The Slammer racked up $1 billion in damages by the time the dust had settled.[10]

---

[6] U.S. General Accounting Office, *Computer Security: Virus Highlights Need for Improved Internet Management* (June 1989), http://www.worm.net/GAO-rpt.txt.

[7] David Moore et al., *Code-Red: A Case Study on the Spread and Victims of an Internet Worm* (2002), http://www.caida.org/outreach/papers/2002/codered/.

[8] Such highly virulent worms are called "Warhol" worms because of their ability to infect most or all targets in a matter of minutes, giving truth to the artist's adage, "In the future, everyone will have 15 minutes of fame." Stuart Staniford, Vern Paxson & Nicholas Weaver, *How to Own the Internet in Your Spare Time*, Proceedings of the 11th USENIX Security Symposium (2002), *available at* http://www.icir.org/vern/papers/cdc-usenix-sec02/cdc.web.pdf.

[9] David Moore et al., *Inside the Slammer Worm* (2003), http://www.caida.org/outreach/papers/2003/sapphire2/.

[10] Robert Lemos, *Counting the Cost of Slammer* (January 31, 2003), http://zdnet.com.com/2100-1104-982955.html.

Annually, computer security breaches result in tens of billions of dollars of damage.[11] Worms - particularly worms that spread through buffer overflow vulnerabilities - are clearly not the only threat to computer security, yet they comprise a significant portion of the annual damage. Costs related to security breaches include the service disruptions, data destruction, time and effort required to recover compromised systems and purchase, install, and apply countermeasures and patches. For many organizations and home users, the cost of keeping up with, understanding, and applying the parade of patches that vendors release is significant.

Finally, it cannot be underemphasized that worms such as Code-Red and SQLSlammer caused damage only because they brought machines and networks to their knees. It would have been trivial for a hacker skilled enough to write such a worm in the first place to add a few more lines of code to delete the fixed disks on the machines they infected. In such cases, damages would have increased by one or more orders of magnitude. Moreover, as the number and connectivity of computers increases, and as society becomes increasingly reliant upon them, it is clear that the potential for large-scale disaster increases dramatically.[12]

Eliminating Buffer Overflow Vulnerabilities

Buffer overflow vulnerabilities comprise a significant portion of the known vulnerabilities listed in public databases.[13] There are a number of techniques that can be used to eliminate or

---

[11] Yochi Dreazen, *Workplace Security*, 9/29/03 Wall St. J. R4 (estimating cost to businesses of worm and virus attacks $17 billion).

[12] A simple formal model for understanding networked risk is developed in Appendix A. The intuition behind it has been expressed before the terrorist attacks of September 11, when President Clinton's terrorism czar, Richard Clarke observed the dangers of cyber terrorism: "I'm talking about people shutting down a city's electricity, shutting down 911 systems, shutting down telephone networks and transportation systems. You black out a city, people die. Black out lots of cities, lots of people die. It's as bad as being attacked by bombs." Quoted in Tim Weiner, *The Man Who Protects America From Terrorism*, 2/1/99 N.Y. Times A3.

[13] Buffer overflow vulnerabilities account for roughly one third of the total number listed in the CERT Vulnerability Database, http://www.kb.cert.org/vuls (337 out of 955, by the author's count

reduce the number of buffer overflow vulnerabilities in a given piece of software. At one extreme lie solutions that rely upon using "safer" implementation languages. The most commonly used programming language, C, not type safe -- that is, abuses of data such as buffer overflows are not guarded against. Modern programming languages such as Java are guarantee type safety, meaning that they attempt to detect such abuses either at compile time, run time, or both. While programs implemented in a type safe language can be said to be "immune" to the buffer overflow exploit, unless the programmer has explicitly handled the error condition, such programs will still terminate when an abuse of data (such as an over-long input) is discovered at run time. Termination with a meaningful error not a perfect solution, but at least such programs are not subject to being "hijacked" in the same manner as the equivalent C program. Of course, guaranteeing type safety comes at a cost, and consequently a program written in Java will tend to run slower than the equivalent program written in C or C++.[14] To the vendor who has a large, existing code base, the cost is that of rewriting all or much of their software in another programming language. Clearly, this might represent a monumental effort for some vendors, resulting in years of re-engineering effort.

Given that a vendor may be reluctant to use a language-based approach, engineering process-based solutions will at least help curb the problem. Code audits can flag potential buffer overflow problems in an existing C/C++ code base. These audits can be to some extent automated.[15] Potential weaknesses can then be carefully considered and possibly re-coded. Many

---

in November, 2003).

[14] While a program that runs 10 times slower seems bad, recall that the underlying hardware runs about 10 times faster every 5 or 6 years.

[15] David Evans & David Larochelle, *Improving Security Using Extensible Lightweight Static Analysis*, January/February 2002 IEEE Software 42.

organizations implement human code reviews as part of their engineering process. However, as deadlines loom, code reviews tend to lose out to the allure of running code.

Finally, there are dynamic techniques that are based on modifications to the C or C++ runtime system, which can catch large classes of buffer overflow exploits. The overhead of such solutions is typically under 10% on benchmark programs.[16] Of course, the advantage of such solutions is that they simply require a recompilation of existing C programs, which is a trivial task compared to re-implementing the entire program in a new language.

Legal Framework

Having established that buffer overflows are a substantial source of costs external to the software industry, we turn now to the question of whether and how best to distribute these costs. The protected entitlements model, pioneered by Calabresi and Melamid, provides a useful framework.[17] Here, the entitlement centers on the presence or absence of security defects in software: first, an entitlement to a secure software system, and second, an entitlement to ship insecure software. Under the model, entitlements can be protected either by liability rules or property rules. Entitlements protected by property rules can only be taken if the taking party agrees to the entitled party's asking price. Entitlements protected by liability rules can be taken, but the (formerly) entitled party must be compensated for their loss. Applying each type of rule to each entitlement yields four combinations:

1) An entitlement to secure software protected by a property rule. Selling insecure software is viewed as "nuisance." The vendor cannot sell insecure software unless the consumer allows

[16] Crispin Cowan et al., *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*, Proceedings of DARPA Information Survivability Conference and Expo (1999), http://www.cse.ogi.edu/~crispin/discex00.pdf.

[17] Guido Calabresi & A. Douglas Melamid, *Property Rules, Liability Rules, and Inalienability: One View of the Cathedral*, 85 Harv. L. Rev. 1089 (1988).

it. Consumers can enjoin the vendor, but the vendor can of course negotiate with consumers and "pay" them to accept insecure software.

2) An entitlement to secure software protected by a liability rule. The vendor may sell insecure software, but must compensate the consumers for damages that result.
3) An entitlement to ship insecure software protected by a property rule. Selling insecure software is not viewed as a "nuisance." Consumers can only stop the vendor from selling such software by paying the vendor not to sell it. This combination represents the status quo.
4) Entitlement to ship insecure software protected by a liability rule. Consumers can enjoin the vendor from shipping insecure software, but must pay damages to the vendor.

Property Rules

Calabresi and Melamed teach that in the realm of property rules, when we are certain that one party is the cheapest cost avoider, we should grant the entitlement to the opposite party. In the instant case, the entitlement has been assigned to the wrong party -- the software vendors. It lies with the wrong party because at least with respect to the buffer overflow problem in particular, and software internal security flaws in general, they are the cheapest - indeed, the *only* - cost avoider. As long as software is only shipped in binary format, the consumer has no ability to secure the defective software. Of course, the Coase Theorem tells us that even if we are wrong in assigning the initial entitlement, the parties ought to be able to negotiate an economically efficient result.[18] Here, that would mean that instead of avoiding the costs on their own, consumers would negotiate with vendors and pay them to produce secure software. Since the vendors are the cheapest cost avoider, the result will be economically efficient, because there must be some price between the value of secure software to users and the cost of producing secure software to vendors that will result in the delivery of secure systems.

[18] Ronald Coase, *The Problem of Social Cost*, 3 J. Law & Econ. (1960).

Unfortunately, the Coase theorem assumes zero transaction costs between the parties. Given that we live in a world where vendors are entitled to produce insecure software, there are no practical means by which consumers can, without more, collectively negotiate with the vendor to give up the entitlement. In contrast to the example of air pollution, the problem of insecure software is significantly more global. It is not just a matter of getting those who live near a pig farm together and hashing out what they would be willing to pay the farmer to stop raising pigs. Here there are millions of different users distributed around the world, many of whom will place starkly different prices on the value of security, giving rise to free-rider problems.

Given that there exist high transaction costs and hence no cheap way for consumers to negotiate for more secure software with vendors, we should see consumers doing the next best thing to obtain a secure system. This process is evidenced by the growth of the "security industry," which broadly speaking sells services and products to improve computer security. In many ways, however, the security industry is like the cold-remedy industry during flu season. The right thing of course might be for everyone to receive vaccinations, but since this doesn't happen, there exists an after-market for remedies. These remedies don't cure the problem, they just make people feel better after they have gotten ill. Anti-virus software and patches generally only fix problems once they are known. They clean up systems that are already broken, possibly after significant damage has occurred. Firewalls are only effective insofar as they protect a computer from incoming network connections to nonstandard ports. They will not, without more, protect an organization's web-server from a buffer overflow attack, because the server's very job is to accept connections on a particular port, and therefore the firewall must allow such connections to pass through.

Given that after-market security remedies haven't solved the problem, it is still not immediately clear why a market-based approach might not solve the problem without moving the

entitlement from its current place. In other words, different vendors ought to be able to compete on the basis of security features in the same manner that they would and do compete on the basis of any other features. Vendor X might advertise the technological countermeasures it has adopted in order to assure users that its software achieves a high standard of security. Vendor Y might make no representations about its software aside from the fact that it costs half as much as that from X. Consumers should then be able to choose whether they want software that is cheap and maybe dangerous or pricey and probably safer.[19]

At the surface, the reason the market hasn't solved the problem of internal security holes relates to lock-in and a near monopoly in the home operating systems market.[20] Over 90% of home users use a Microsoft operating system. Single vendors strategically design different software packages to interoperate with their own packages, but then hide the relevant interfaces that could enable other vendors to step in and offer competing products. Moreover, as long as installing software is a nontrivial task for home users, the cost of switching to software packages that did not ship with their machine remains relatively high. As long as there is no real competition (or the barrier to entry to competition remains unreasonably high), the market won't yield more secure software because there is simply no incentive to do so.

But the issue is deeper still. Even in a market with no barriers to entry to competitors and no switching costs to consumers, the simple fact that some vendors will create low-security products means that those consumers who purchase those offerings will be able to externalize the costs of their own poor security. Poor security gives rise to at least two separate risks: one to the

---

[19] *See e.g.* Michael Rustad & Lori Eisenschmidt, *The Commercial Law of Internet Security*, 10 High Tech. L.J. 213 (1995) (favors adopting UCC Article 2B to provide a framework for clarifying contract law with respect to software licensing, to give vendors and consumers the ability and freedom to allocate liability among themselves).

[20] Dan Geer et al., *CyberInsecurity: The Cost of Monopoly* (2003), http://www.ccianet.org/papers/cyberinsecurity.pdf.

internal integrity and operation of a computer system, and another to the external availability of the network. While buying expensive, higher security software may better protect data resident on a consumer's machine, it will do nothing to guarantee the availability of the network, because all of the consumers who have bought low security software provide attackers ample opportunity to clog up the operation of the network for, or even launch denial of service attacks against users who are running secure software. The Internet is subject to the tragedy of the commons, where a relatively small number of insecure and misbehaving hosts can seriously degrade a public good.[21] The Slammer worm demonstrates this principle: even though it only affected 75,000 hosts, it caused disruptions for millions of hosts on the Internet.

Finally, this insight explains why neither of the property rule combinations is sufficient in this situation. Currently, those who desire secure systems cannot possibly purchase the entitlement from the vendors, because as long as there are vendors who produce low security software, there will be users who choose to externalize the cost of their choices. And if the entitlement were moved to the users, it seems likely that for the right price, some users might still choose trade the entitlement by accepting software with poor security, again externalizing those costs. Moreover, the enforcement costs seem unmanageable. How could the courts enjoin potentially millions of users running low security systems? Of course, they would focus their energies upon the vendors of such systems, but even this is no easy task. For in contrast to the case of a pig farm - where the "problem" is obvious to anyone with a nose - the very act of deciding what comprises "poor" security is very difficult. In the limited case of the buffer overflow problem, it may be possible to make some pre-sale judgment about the relative security of two offerings, but in the general case, where the total security of a system depends on many factors - both social and technological - this sort of judgment is difficult to make until the software has spent some time in the field.

---

[21] Garrett Hardin, *Tragedy of the Commons*, 162 Science 1243 (1968).

Liability Rules

The second option suggested by the model is to move the entitlement to the consumer, and to protect it with a liability rule. Under this approach, consumers would be able to sue vendors for damages caused by security flaws in their software. Subclasses of this approach have been treated extensively in several academic papers.[22] Primarily, they focus on shoehorning the issue into one of the traditional private causes of action in tort. Rather than dissect each of the offerings in detail, it should suffice here to make observations that are general to all of them.

First, these approaches all suffer from high administrative overhead: finding and valuing the damage to all of the injured parties is difficult, because the current pattern of damage for Internet worms is to widely distribute a relatively small amount of harm. In a way, this would seem to be an ideal arena for mass tort litigation, because the costs of bringing the action by an individual consumer dwarf the damage they seek to recover. At present, adjudicating a mass tort in this arena will be difficult as long as the standard of due care and the tort defenses will vary widely over the many jurisdictions covered by the class.[23] Moreover, in a future inhabited by increasingly malicious worms that actually destroy data, the quantity of damage will vary greatly between individual users, making it perhaps even more difficult to accurately assess damages.

---

[22] Kevin Pinkney, *Putting Blame Where Blame Is Due: Software Manufacturer and Customer Liability For Security-Related Software Failure*, 13 Alb. L.J. Sci. & Tech. 43 (2002) (suggesting strict liability for vendors with defense of contributory negligence for users who fail to patch); Erin Kenneally, *Stepping On The Digital Scale: Duty And Liability For Negligent Internet Security*, Ann. 2002 ATLA-CLE 403 (suggesting imposition of duties upon vendors, service providers, and end-users, with the scope of the duty depending upon which party is best positioned to implement safeguards); Patrick T. Miyaki, *Computer Software Defects: Should Computer Software Manufacturers be Held Strictly Liable for Computer Software Defects*, 8 Santa Clara Computer & High Tech. L.J. 121 (1992) (arguing against liability due to negative economic impacts on the software industry).

[23] Robin Brooks, *Deterring the Spread of Virus Online: Can Tort Law Tighten the Net?*, 17 Rev. Litig. 343 (1998).

The second problem relates to potentially disastrous distributional effects of such a policy. In short, the cost of making a single mistake may be sufficient to drive a vendor out of business. With other modern technologies, such as railroads and airplanes, the potential damage caused by a single incident tends to be isolated and relatively much cheaper than the total benefit of the technology.[24] In the case of computing technology in a networked environment, however, the potential impact of failing to catch even a single buffer overflow vulnerability is the total failure of the network, with costs ranging into the billions of dollars. And as we have seen, in the foreseeable future, the costs could even include the loss of life on a large scale. Damages of this magnitude seem uninsurable and far beyond the capability of any vendor to pay. The reason is, of course, that the very qualities that make the Internet such a valuable resource - cheap and extensive connectivity - are the same qualities that allow a single hacker to bring it to its knees.

The last option suggested by Melamid and Calabresi is leaving the entitlement where it is, but protecting it with a liability rule. This would mean that if someone takes the entitlement, they would have to pay the vendor damages, presumably the cost to repair the software. An obvious way to implement such a solution would be for the government to step in, force companies to improve their security, but then pay them to make the improvements. The costs would then be distributed, presumably amongst the parties who benefit, via a tax of some sort. One nice thing about this is approach is that it is easier to calculate damages and bring the relevant parties to the table, when compared to the other liability rule. It also has the admirable distributional effect that it will tend to keep companies in business. However, it doesn't really provide the right incentive: at the extreme, a company can keep making defective software while someone else foots the bill for fixing it.

[24] Of course the events of 9/11 have upped the ante, with an order of magnitude increase in loss of life, coupled with the lingering, hard to calculate damages to the air travel industry in particular, and economic confidence in general.

Recommendation:

The suggested strategy is for a joint public and private process to create and publish a set of standards that provide a bright line rule for what it means for a vendor to develop secure software.[25] These standards will surely contain a mix of sophisticated automated tools or benchmarks together with engineering process requirements.[26] Automated tools can be used to perform sophisticated code audits at relatively low cost to those vendors who continue to use technologies that are susceptible to buffer overflow problems, such as C or C++. An agency administered civil penalty mechanism will enforce the standards against vendors who fail to follow them. Amazingly, even very small per-license penalties will quickly amount to very large awards.[27] Public funding, recouped through a software tax and penalties gathered through the enforcement program will be used to develop improved tools and processes. Finally, tax incentives can be used to encourage companies to start adopting implementation technologies that are demonstrably safer.

The above proposal should serve to significantly reduce the number of security holes, especially in the realm of something like the buffer overflow problem. In addition it has numerous benefits not found in a private enforcement scheme based simply upon a common law tort. First, the rules have bright lines. The advantage of setting standards is of course that it reduces the cost of compliance by engineers before the fact and administration of the law after the fact. The cost of compliance is relatively low; the penalties for not doing so are high and relatively straightforward

---

[25] The Food and Drug Administration already attempts to minimize software defects by requiring a formalized development processes together with design validation to help assure that the finished product works as intended. 21 CFR 820 (1996). See also Food and Drug Administration, *General Principles for Software Validation* (2002), http://www.fda.gov/cdrh/comp/guidance/938.pdf

[26] *See e.g.* Nancy Leveson, *Safeware: System Safety and Computers* (1995).

[27] Arguably, the penalty structure should track the nonlinear nature of damage in a networked environment to better reflect the amount of actual damage done.

for a court to apply. Second, the knowledge (better processes and better tools) generated through the public research component remains in the public domain. This is an acknowledgement that network security is a public good, and techniques to help guarantee it ought to remain public. In contrast, in a private scheme, every company is forced to develop or purchase this kind of knowledge, and keep it secret for fear of losing what has become a competitive advantage. Third, such a policy will effect a change of culture within the software development houses, where the pressure to market often plays a major role in determining the order in which known defects are fixed. Gone will be the day, hopefully, where known security holes receive lower priority to the latest features demanded by the marketing or sales department. Finally, by paying out bounties to hackers who discover and report security holes (which should have been caught by a company complying with the standards), this scheme may create incentives for hackers to perform socially beneficial work.

While this proposal should reduce the number of security defects, it does nothing to solve the problem of fixing deployed software after defects are discovered. As long as the parade of patches approach remains the preferred or only method of fixing software in the field, known defects will remain a popular avenue for attack. In many ways, this second problem - fixing defects in the field - is more vexing than that of improving the initial quality of software. Solving this problem calls perhaps for a radically new - or old - model of software distribution. In a world where home computers contain little or no software, software will be dynamically and transparently downloaded every time it is run. Since software only resides on a relatively small number of servers, patching software becomes a manageable problem. Ironically, this architecture is not too different from the centralized computing resources accessed by dumb terminals of the 1960s and 70s. Moving to such a model of computing may require something of a sea change in the industry, and will certainly take longer than improving initial software quality. Under this

policy, industry would be encouraged to make the transition by phasing in - after a period of years - an increasing measure of strict liability for unfixed software in the field. Given the difficulty of eliminating every potential security hole, many vendors ought to decide to avoid liability simply by changing their model of software distribution. Again, the program will provide for funding to research new architectures, or perhaps to relearn the lessons of old.[28]

In summary, the approach has two phases. Initially, a standards-based approach attempts to improve the situation in the short term, by forcing vendors to make building secure software a top priority, but forgiving them for defects that could not have been captured under the current state of the art. Ultimately moving to strict liability for unfixed, deployed defects acknowledges that improved engineering process and technology will never fix every problem, and creates a powerful incentive to develop alternate models of software distribution. Vendors can trivially escape liability under such a regime by never (permanently) deploying software.

Conclusion

The general case of computer security is a complicated problem. The complicated nature of the problem is no reason, however, not to attempt to engineer a coherent solution.[29] This paper has sketched a potential solution for the limited and fairly well understood problem of buffer overflow vulnerabilities. Improved security in the general case is of course more than improved technologies, meaning that more work will be required to in other areas.

---

[28] *See also* Stephanie Forrest et al., *Building Diverse Computer Systems*, 6th Workshop on Hot Topics in Operating Systems 67 (1997) (presenting an alternative to the above model by taking lessons from biodiversity seriously and attempting to build software not subject to the one-attack-fits-all problem plaguing current systems).

[29] A "Gee whiz, it's a big job" attitude seems even to have infected Amit Yoran, Cybersecurity Czar, when he states: "In concept, [holding software vendors liable] may make sense. But in practice, do they have the capability, the tools to produce more secure code?" David Bank, *Tech Executives Try to Ward Off Security Rules*, 12/3/03 Wall St. J. B5.

Finally, there remains a danger that we have not been asking the right question. Perhaps rather than asking whether and how to best shift the cost of security flaws, we should ask whether we, as a society, are willing to live with those costs, regardless of the benefits. Perhaps nuclear energy is an apt analogy: a technology with the promise of an endless supply of electricity but at the potential cost of terrible environmental disaster. In some nations, the people have decided that they would prefer not to live with those risks, regardless of how remote the risks and grand the benefits may be on paper. As computer technology becomes increasingly fast, embedded and connected, network failures will surely mean terrible things. At some point, merely deciding who should bear the cost of these terrible outcomes doesn't do justice to the deeper issue, which is whether we even want to live in a world where these remote-sounding risks of massive failure become reality. Complex, integrated technical systems tend to fail spectacularly. Often lost in the discussion is the fact that we are the ones in ultimately charge of building those systems, and maybe the answer to the question is that if there is no guarantee that there will be no spectacular failures, then we don't want them to get any more complicated, even if there exists a system for recovering the damages.[30]

[30] Perhaps the most controversial expression of this viewpoint is found in Theodore Kaczynski, *The Unabomber Manifesto* §172 (1995), http://hotwired.wired.com/special/unabom/list.html.

Appendix A: A Simple Risk Model for Thinking About Network Insecurity

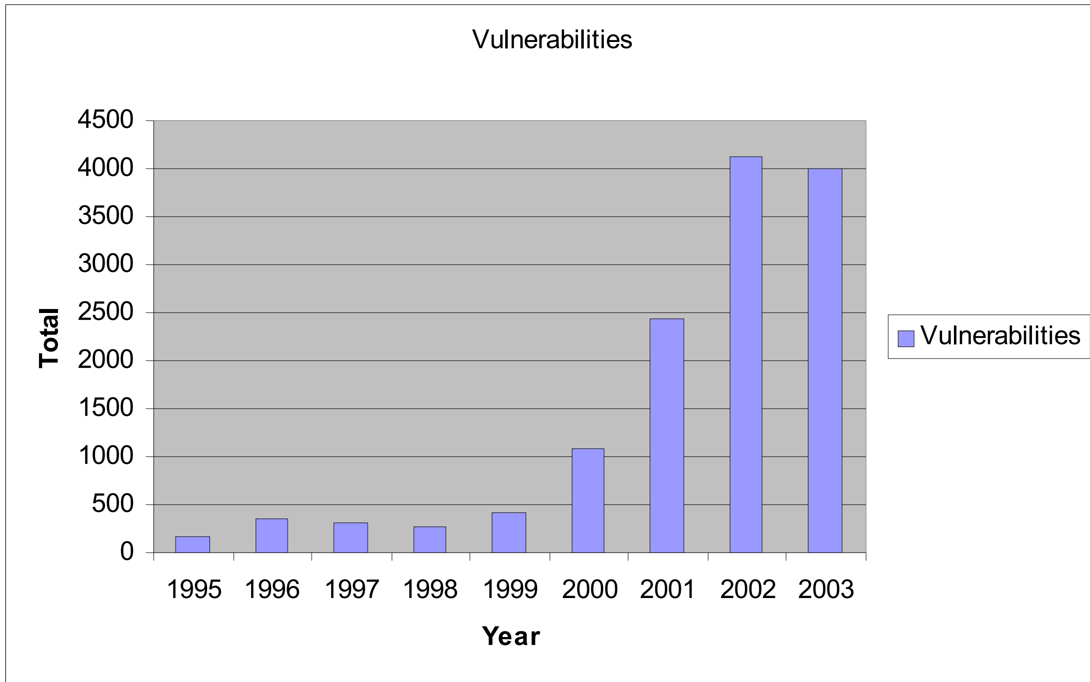
Risk is characterized as the probability, P, of an incident times the cost, C, of the consequences. $R = PC$. Metcalfe's law holds that the value, V, of a network increases with the square of the number of hosts, N, on the network. $V = N^2$. While this law is typically called upon to explain the benefits of a networked environment, it surely has a corollary, which is that the cost of a network failure, C, also increases with the square of the number of hosts, N, on the network. This is just common sense, for everything given by the network effect must also be capable of being taken away. $C = N^2$. Now substituting, we see that in a networked environment, risk can be expressed as the probability of an incident times the number of hosts. $R = PN^2$. This supports the intuition that the amount of damage caused by malicious mobile code will increase dramatically as the size of the network grows.

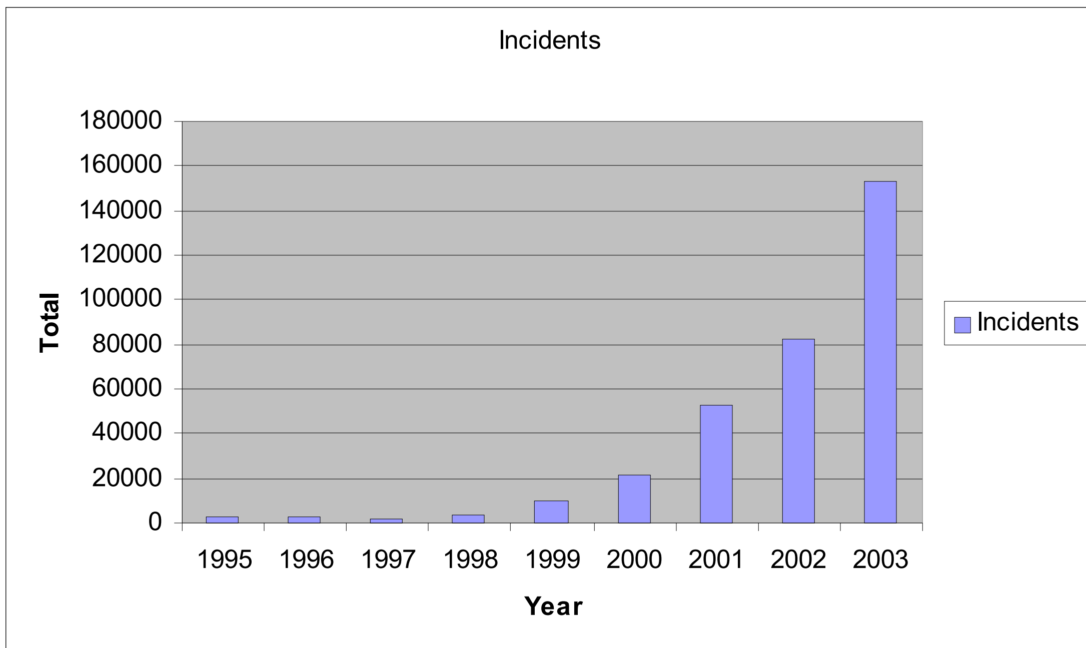
Next, we know that as software grows, the complexity of software increases by a square of the number of lines of code. The probability of a security hole (directly related to the probability of an incident) is directly proportional to the complexity of the software. Hence, $P = kL^2$, where L is the number of lines of code, and k is a constant factor relating the probability of a security hole to the probability of an incident. Now we again substitute into the risk model, yielding:

$$R = k\, L^2\, N^2$$

In English, this says that risk grows with the square of the size of the program times the square of the number of hosts on the network. Assuming 10% growth rates for both software size and network size, the model predicts that risk doubles roughly every 18 months. While this model is subject to much refinement, it provides a rough theoretical basis for understanding the risks associated with not properly managing the problem presented by network insecurity.

Appendix B: Total Vulnerabilities and Incidents Listed in the CERT Database.





Bibliography

David Bank, *Tech Executives Try to Ward Off Security Rules*, 12/3/03 Wall St. J. B5.

Robin Brooks, *Deterring the Spread of Virus Online: Can Tort Law Tighten the Net?*, 17 Rev. Litig. 343 (1998).

Guido Calabresi & A. Douglas Melamid, *Property Rules, Liability Rules, and Inalienability: One View of the Cathedral*, 85 Harv. L. Rev. 1089 (1988).

Ronald Coase, *The Problem of Social Cost*, 3 J. Law & Econ. (1960).

Crispin Cowan et al., *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*, Proceedings of DARPA Information Survivability Conference and Expo (1999), http://www.cse.ogi.edu/~crispin/discex00.pdf.

Yochi Dreazen, *Workplace Security*, 9/29/03 Wall St. J. R4.

Mark Eichin & Jon Rochlis, *With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988*, IEEE Symposium on Research in Security and Privacy (1989), *available at* http://www.mit.edu/people/eichin/virus/main.html.

David Evans & David Larochelle, *Improving Security Using Extensible Lightweight Static Analysis*, January/February 2002 IEEE Software 42.

Food and Drug Administration, *General Principles for Software Validation* (2002), http://www.fda.gov/cdrh/comp/guidance/938.pdf

Stephanie Forrest et al., *Building Diverse Computer Systems*, 6th Workshop on Hot Topics in Operating Systems 67 (1997)

Dan Geer et al., *CyberInsecurity: The Cost of Monopoly* (2003), http://www.ccianet.org/papers/cyberinsecurity.pdf.

Garrett Hardin, *Tragedy of the Commons*, 162 Science 1243 (1968).

Theodore Kaczynski, *The Unabomber Manifesto* §172 (1995), http://hotwired.wired.com/special/unabom/list.html.

Erin Kenneally, *Stepping On The Digital Scale: Duty And Liability For Negligent Internet Security*, Ann. 2002 ATLA-CLE 403.

Robert Lemos, *Counting the Cost of Slammer* (January 31, 2003), http://zdnet.com.com/2100-1104-982955.html.

Nancy Leveson, *Safeware: System Safety and Computers* (1995).

Patrick T. Miyaki, *Computer Software Defects: Should Computer Software Manufacturers be Held Strictly Liable for Computer Software Defects*, 8 Santa Clara Computer & High Tech. L.J. 121 (1992).

David Moore et al., *Code-Red: A Case Study on the Spread and Victims of an Internet Worm* (2002), http://www.caida.org/outreach/papers/2002/codered/.

David Moore et al., *Inside the Slammer Worm* (2003), http://www.caida.org/outreach/papers/2003/sapphire2/.

Kevin Pinkney, *Putting Blame Where Blame Is Due: Software Manufacturer and Customer Liability For Security-Related Software Failure*, 13 Alb. L.J. Sci. & Tech. 43 (2002).

Michael Rustad & Lori Eisenschmidt, *The Commercial Law of Internet Security*, 10 High Tech. L.J. 213 (1995).

Stuart Staniford, Vern Paxson & Nicholas Weaver, *How to Own the Internet in Your Spare Time*, Proceedings of the 11th USENIX Security Symposium (2002), *available at* http://www.icir.org/vern/papers/cdc-usenix-sec02/cdc.web.pdf.

U.S. General Accounting Office, *Computer Security: Virus Highlights Need for Improved Internet Managemen*t (June 1989), http://www.worm.net/GAO-rpt.txt.

Tim Weiner, *The Man Who Protects America From Terrorism*, 2/1/99 N.Y. Times A3.