Attack and Defense:

Improving Cybersecurity by 2014

Caroline Benner
Andrew Pardoe
Jack Richins
John Spaith
Santeri Voutilainen

CSE P 590TU: Information Technology & Public Policy

Professor Ed Lazowska
Stephen M. Maurer

December 10, 2004

Table of Contents

An Attack in 2014: A Walk Through

The date is October 23, 2014. Rep. Sandra Hill is running for U.S. Senate. The race is tight and could turn on the tiniest misstep. Rep. Hill has a secret: She has been diagnosed with the early signs of Alzheimer's disease the past summer. Based on her doctor's diagnosis of slow progression that can be treated with experimental drugs, Rep. Hill has decided to stay in the race.

Joe Cracker has an agenda: he does not want Rep. Hill elected to the Senate. He is strongly opposed to most issues Rep. Hill has promoted, especially increased control over rogue elements on the internet. Joe Cracker knows a little about software security and vulnerabilities, and has had plenty of time on his hands. Months earlier Joe had decided to find embarrassing information about Rep. Hill and publicize it in order to derail her campaign.

The previous week, on a visit to his doctor, Joe pocketed a small computer memory device that someone had left on the counter. Joe did not expect it to be of much use other than as more portable memory, but upon examining it at home he soon realized that it contained access code to the Central Medical Record Service (CMERECS). CMERECS was created 8 years earlier to allow authorized individuals access to the records of any patient under their care. Joe knew that although his doctor's office used the latest and greatest patient management application, the application vendor had not updated the encryption mechanism for nearly 10 years. The encryption algorithm was still on the list of approved algorithms for storing CMERECS access codes, but Joe knew it could be cracked. Joe set his two brand new high end computers to work cracking the encryption by doing nothing cleverer than just attempting every possible combination. Four days later Joe had the access codes. These access codes allowed him to connect to CMERECS and request the medical records for anyone in the nation with the requests appearing to have originated from his doctor's office. Under normal conditions the patient's approval would be required before the records could be released to a doctor, but the system had a loophole: For emergency services no approval was required. Joe requested the records for Rep. Hill using this emergency request procedure.

With the medical records in hand, Joe embarks on a plan he spent the last several months preparing; he will distribute Rep. Hill's medical records using a worm that attacks cell phones. Joe has a built a network of vulnerable cell phones with the help of a "surreptitious worm"[1] that spreads slowly from cell phone to cell phone by keeping a low profile and piggy backing on the phone users regular phone usage. Thanks to this behavior, the worm has thus far been able to avoid discovery, but that will change once Joe triggers it into high gear.

This initial worm, which Joe named PrepBoy, takes advantage of an unpublicized vulnerability in the operating system used in several data capable cell phones. As the worm spreads it sends a message to other cell phones indicating an update for the cell phone is available. Included in this message is a malformed value for the expected size of the update. This malformed value causes the cell phone software to attempt to contact a backup site listed in the original message. Joe included an address to an anonymous website on which he had previously placed his payload (a copy of PrepBoy). When the cell phone downloads PrepBoy, assuming it is the official update, the second vulnerability, a buffer overflow, is encountered and allows PrepBoy to take over the cell phone. Once on the phone, PrepBoy attempts to spread itself by sending a copy of the update message to other cell phone numbers that it has gathered from the cell phone's address book or numbers within the same telephone exchange. In order to avoid detection it only sends a few of these messages at a time, and only when the cell phone is connected to the data network as part of the user's regular activities.

With this network in place, Joe sends a message exposing Rep. Hill's Alzheimer's diagnosis, with a copy of the medical records, to the set of infected cell phones to which he initially sent PrepBoy. Upon receiving this message, PrepBoy notices a key word and kicks into high gear. It begins to blast the received message to all the other cell phones it has infected which in turn forward it to all the phones they have infected. This blast quickly grows into a tidal wave reaching all infected cell phones within minutes. The message also triggers another change in PrepBoy: PrepBoy starts to forward the message not only to other cell phones, but to any email address it finds on the cell phone.

Within a couple hours, not only the most vulnerable cell phones in the state, but also within the whole country have received the message. There is public outcry not only about the massive traffic generated by the worm, but also about Rep. Hill's decision to run for office even after she was diagnosed with Alzheimer's. With less than a week left before the elections, Rep. Hill's support drops dramatically and her opponent wins by a landslide. Joe's plan has succeeded; he has effectively ended Rep. Hill's career.

Introduction

The ease of exploits like this one will encourage more and more Joes to test the ability of software makers to provide secure software in the coming decade. Policy-makers must understand the limitations in our ability to make secure software and what we can do to change this situation.

In this paper, we explain to policy-makers what forces could be brought into play to improve computer security. We begin by evaluating the problems we're facing when it comes to computer security: we look at past, present and future threats. From there we offer the policy-maker a range of possible solutions for improving security, from technical, to policy to economic so the policy-maker is well-briefed in the various ways security can be improved. Our first solution is a technical one: we survey promising areas of research in designing secure systems and assess costs and benefits of these approaches. We then consider a policy solution, namely whether software engineering should be licensed as civil engineering is licensed. From there we look at liability: can you hold software vendors liable for their products? Finally, we conclude with a look at how we can design an independent lab to certify software. This lab would be useful because if the consumer could understand security via common, easily-understood ratings of how secure the software they buy is, this would encourage consumer demand for security. In concluding, we offer thoughts on how policy-makers might direct their dollars and law-making ability toward improving software security.

[1] Stuart Staniford, Vern Paxson, and Nicholas Weaver. "How to Own the Internet in Your Spare Time." Proceedings of the 2002 USENIX Security Symposium, San Francisco, CA, August 2002.

Software Vulnerabilities

Santeri Voutilainen

Theft, bombings, power outages, sweet talkers, worms, and bugs – these are just a few of threats faced by computer systems. Although computer systems are vulnerable to traditional physical and social engineering attacks, the vulnerabilities most closely associated with computer systems are software defects. The potential damage from software defects is arguably the greatest, especially when software in integrated and interconnected. Many of these defects are benign, but some can expose the system or data stored in the system to the ill will of malicious parties. There is no lack of software vulnerabilities as can be confirmed with a quick glance at any of the software security websites. The state of software vulnerabilities is such that there have been calls for action among the computer science community to devise new methods for preventing attacks. One such call is from Professor Jeannette Wing, Computer Science Department Head at Carnegie Mellon University. In “Beyond the Horizon: A Call to Arms” she calls for the computer science community to look beyond the current flaws and examine flaws of the future while acknowledging that today’s attacks and flaws are likely to remain[1]. Before examining processes and incentives to improve software quality, it is useful to examine the current attack vectors and speculate about future ones. We’ll discuss the past and future evolution of software defects, and also examine how integration and interconnectedness can increase the damage caused by attacks, including transferring them from one medium to another.

Software Coding Vulnerabilities

It is commonly accepted that all software has defects, or ‘bugs’ as they are known within the software development community. Although many defects can be considered benign or irritating – a spellchecker failing to identify a misspelled word or suggesting an inappropriate word as the correct spelling – and do not cause damage to the user’s work or system, other defects can have far more severe consequences. These defects can range from minimal loss of the user's work to the clearing of storage devices or, in arguably the worst case, enabling attackers to gain complete, uncontrolled access to the system. Such access allows the system to be used for the attacker’s own purposes, such as mounting further attacks. Examples of these most serious defects, which we’ll call exploitable defects, are numerous – the Morris Worm which paralyzed the internet in 1988 and the Code Red and Sapphire/Slammer worms of 2001 and 2003, respectively. What types of defects cause these most serious vulnerabilities in software and how do they occur?

Exploitable software defects first became widely known with the Morris worm in 1988, but they did not gain much traction until a posting on the BugTraq mailing in 1995[2] describing several such defects spurred a wave of reports of similar vulnerabilities. This type of defect is known as a buffer overflow, or stack smashing attack. In its simplest form, the defect is caused by the software’s failure to check the length of the data it receives and subsequent blind overwriting of its own control structures with data from the user/attacker. The root cause is that the software engineer trusts users of the software to pass only valid input and therefore does not check it for validity – if the engineer has designed that the maximum password size is 8 characters, she or he assumes that no one will provide a password with 9 characters or more. As companies and engineers have become more aware of these specific defects, more variations on the original defect have been shown to be exploitable. In most cases it had been widely believed that, although these variations existed, they could not be exploited for one reason or another. The original defect overwrote data in a software program's scratch area where it tracks, among other things, its next instruction steps. Overflows in other areas were thought harmless, but were eventually shown to be just as exploitable as the original defect[3]. With the last couple years it has been shown that an overflow of even a single character can be used maliciously[4].

One of the more recent variations is the integer arithmetic defect. In these defects an attacker is able to cause the software to perform an error in integer arithmetic. This may seem outrageous, as computers are computing machines, and except for faults in processors, should not be subject to arithmetic errors. However, computer systems are limited in the range of numbers they can store accurately. An integer in a computer system does not have a range from negative infinity to positive infinity, but rather a considerably smaller range that is dependant on the amount of memory used to store the value. For technical reasons, although engineers can choose from different sized integers, once a size is chosen it is fixed. An integer arithmetic defect occurs when the result of a calculation does not fit into the fixed size chosen by the engineer. In this case, the value is truncated at the front. A quick example illustrates this:

A city requirements form allows two digits for the minimum height of a security fence. The height was set to 72 inches in 1999 because no guard dog could jump that high. In 2004 a new breed is introduced that can jump 40 inches higher than the previous best jumper. A clerk is instructed to add 48 inches to the minimum value. The clerk uses long addition and changes the first digit to a 0 and the second to a 2. When the clerk attempts to write a 1 for the third digit he finds there is no room for it and just leaves it out, leaving 20 where there should have been 120. The next day a builder comes to check the minimum height for the fence he's building and reads 20 inches. Once the fence is finished the dogs are placed inside and immediately escape because the fence is so low that most dogs can jump over it.

Until recently it was widely believed that arithmetic errors could not lead to exploits[5,6], but they have now shown to be exploitable[7].

Although variations of these defects have been widely known about since 1988, and definitely 1995, even the earliest variations are still very common in modern software system. A search of the Secunia.com security advisory database lists 22 new buffer overflow security advisories for the first 23 days of November 2004[8]. Some of these vulnerabilities are in mature software such as WinAMP[9] and Microsoft Internet Explorer[10]. If mature software products still contain undetected buffer overflows, it is unlikely that these defects will disappear anytime soon. These existing defect types will also be joined by new types of defects as attackers look for new ways to attack.

New defects in the future may be variations or evolutions on current vulnerabilities, as the above are variations on the original buffer overflow defect, or they may take completely new approaches. Although the future is impossible to predict, we can make educated guesses based on attack vectors that are currently either impractical or still only theoretical.

An example of a new attack vector is the algorithmic defect. Algorithmic defects are weaknesses in algorithms, which can cause the algorithm to perform incorrectly or, in the case of security algorithms, reveal the secrets that the algorithm was intended to protect. Although their elusiveness may account for the relatively low numbers of algorithmic exploits, algorithmic flaws nevertheless cannot be ignored. Once found and abused, algorithmic defects can cause a great deal of harm, precisely because software engineers trust algorithms to operate safely. Examples of algorithmic flaws are the Needham-Schroeder authentication protocol and several digital watermarking algorithms. The Needham-Schroeder authentication protocol claimed to allow two parties to reliably prove their identities to each other – such as a customer and a bank during phone-banking. The bank needs a guarantee that the customer is who he claims to be. Similarly, the customer wants a guarantee that the entity with whom he makes a transaction is not a fraudster who will turn around and use information given in confidence to empty out the customer's account. The Needham-Schroeder algorithm contained a flaw that escaped detection for 17 years and allowed an intruder to impersonate one of the participants in the protocol[11]. Such a flaw in widely used algorithms would have devastating implications.

In addition to paying attention to buffer overflow defects and algorithmic defects, software engineers will also need to prepare for advances in computing power and even power consumption levels in order to fully protect their software from exploits.

The Data Encryption Standard (DES) was selected as the data encryption standard by the United States government in 1976. Although controversial from the start it was reaffirmed as the standard as late as 1998[12]. Also in 1998 the Electronic Frontier Foundation developed a $250,000 chip that could crack the DES encryption in a little over two days[13]. This was a brute force attack – try all combinations to a combination lock until the lock opens. As computing power increases over time – Moore's law states that computing power doubles every 18 months – brute force attacks become more viable attack vectors especially on older, weaker standards such as the DES and software systems using these standards will be vulnerable to eavesdropping by attackers on communications or data that should be secure.

An attack vector currently at the theoretical stage is Differential Power Analysis (DPA). DPA can break encryption keys by monitoring the power usage of a processor as it encrypts or decrypts a message[14]. Although currently DPA is mostly at a theoretical level, it is possible that it will become feasible in the future. Even though it is possible to defend against DPA using mechanical/electronic measures, it is also possible to defend against using software measures that mask the actual power consumption levels. Failure to protect a software system against DPA, especially in secure systems, should be considered a failure similar to any other software defect that can lead to an exploit.

From buffer overflows to DPA, software defects are of limited use on isolated systems as the effects of an attack are generally limited to that individual machine. Similarly, the effort required to mount an attack on isolated machines is far greater than the benefit to an attacker, except perhaps in cases of espionage. Just as integration and interconnectedness increases the value of computing systems, they also allow attacks on a larger scale and at more tempting targets.

Integration and Interconnectedness Vulnerabilities

Common wisdom says that only a fully isolated system can be truly secure[15]. With integration and interconnectedness, systems become harder to secure. Previously benign failures can, at worst, become life threatening when integrated in an application that has control of physical or physiological events. Software systems are presently already integrated in many devices; digital video recorders, car environmental systems, heart pacers, cell phones, medical equipment and military system all have software components and many are connected to other systems. In many cases these software systems are not based on specialized products, but on off-the-self software such as versions of Microsoft Windows or Linux. Systems are also more interconnected; cell phones can connect to data networks, more government and commercial systems are linked to the internet, and even cash machines can be reached from the internet. This integration and interconnectedness will increase in the future as much of the value realized from computer systems comes from this very integration and interconnectedness. This integration and interconnectedness exposes these systems to more attacks by making the systems more accessible and provide more tempting targets.

Integration of software systems into other products has already shown to be problematic and even life threatening. The Thai Finance Minister nearly baked in his bulletproof car because the car's environmental control software failed and he could not open the doors or windows. This case also shows how a previously benign failure can be life threatening in an integrated system. The software in the car was based on software also used in Personal Digital Assistants (PDAs). Had the failure occurred in a PDA, the user could have easily restarted the PDA and lost, at most, the work that had not been saved. In an integrated system, such as the car's environmental control system, a fault that in a PDA is an annoyance can become critical. The requirements of the software have dramatically changed based on the usage change.

Other examples of integration and interconnectedness abound. Software malfunctions partially contributed to the East Coast power outage of August 14, 2003 where alarm software was disabled by erroneous input[16]. In January of 2002, the Bellevue, WA 911 response center was rendered unavailable due to the outbreak of the Slammer/Sapphire internet worm[17]. The same worm disabled many Bank of America ATMs which were interconnected in a manner that allowed an internet based worm to disrupt the functioning of the ATMs[18]. The phone system is no longer connected just to the traditional phone network but also to data networks. Cell phones are becoming data devices and Voice-Over-Internet-Protocol (VOIP) based phone systems are starting to compete with traditional land-line phone systems. The phone traffic in these systems travels over the internet until converted to a phone line within the customer's location, exposing the VoIP phone system to the spectrum of internet based attack methods, and reducing the cost of attacking a phone system.

What does the future of integration and interconnectedness look like? Integration will continue with software systems becoming smaller and located in an ever increasing number of appliances. Cable TV boxes are now starting to have more extensive capabilities, such as integrated network connections and built-in web browsers. Medical professional can already observe and diagnose patients as well as assist and guide medical procedures remotely using video conferencing systems[19]. Future systems may allow for procedures to be performed remotely – doctors already benefit from local "remote" procedure when operating with the help of miniature cameras and tools inserted through small incisions. The risks of a vulnerability in these telesurgery systesm are life threatening. Could a script kiddie in the future cause the death of patient by sending malicious instructions to the operating system? RFID technology will ease inventory tasks but will also open to the door for remote snooping. It is conceivable that malicious RFID tags could exploit vulnerabilities in software such as those discussed in the first section. Refrigerators will have integrated software systems that will allow applications such as tracking food usage using afore mentioned RFIDs. The system could automatically place orders as current supplies are exhausted. If these systems are tied to the refrigerator's temperature controls, could it be possible for someone to cause the food in the refrigerator to spoil?

The integration and interconnectedness of software systems increases the number of factors to consider when designing these software systems as the software may operate in environments for which it may not have original been designed. The interconnectedness also makes the software systems more tempting targets as they are now reached easier with easily available and useable tools and for the limited cost of an internet connection. This level of requirements has traditionally been limited to very few systems, such as medical devices in a disconnected environment and space flight systems. With the potential exception of space flight systems, even these systems could not be guaranteed to work outside of their intended environments. As off-the-shelf commercial systems enter more of these specialized areas, they must be secured against the threats since the progress of integration and interconnectedness can not be stopped without greatly reducing the potential benefits from these systems.

Conclusion

With the status-quo, existing vulnerabilities are likely to continue to exist as buffer overflows have not been eradicated in the 16 years since they first came to wide spread attention. Just as integer overflows have evolved from buffer overflows, new threats such as DPA will continue to be discovered. The effects and reach of these vulnerabilities will also grow as the increase in software systems integration and interconnectedness continues. What can be done to break out of this status-quo and eliminate at least some of the vulnerabilities?

---

[1] Wing, Jeannette. "Beyond the Horizon: A Call to Arms" IEEE Security and Privacy. November/December 2003, pp. 62-67.

[2] Lopatic, Thomas. "Vulnerability in NCSA HTTPD 1.3." Online posting. 13 Feb. 1995. BugTraq. Retrieved 17 Nov. 2004 <http://www.securityfocus.com/archive/1/2154>

[3] Conover, Matt. "w00w00 on Heap Overflows." w00w00 Security Development. Retrieved 20 Nov. 2004 <http://www.w00w00.org/files/articles/heaptut.txt>

[4] "Buffer overflow". InformationBlast.com. Retrieved 20 Nov. 2004 <http://www.informationblast.com/Buffer_overflow.html>

[5] Howard, Michael. "Reviewing Code for Integer Manipulation Vulnerabilities" Microsoft Corp. 28 April 2003. Retrieved 17 Nov 2004 <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure04102003.asp>.

[6] Seltzer, Larry. "Integer Overflows Add Up to Real Security Problems". eWeek.com. 11 Mar. 2004 Retrieved 17 Nov 2004 <http://www.eweek.com/article2/0,1759,1545382,00.asp>

[7] "FreeBSD fetch utility Integer Overflow Vulnerability". Secunia.com 18 Nov. 2004. Retrieved 23. Nov. 2004 <http://secunia.com/advisories/13226>

[8] Secunia.com. Search Advisory, Vulnerability, and Virus Database. Retrieved 23 Nov. 2004. < http://secunia.com/search/?search=buffer+overflow>

---

[9] "Winamp 'IN_CDDA.dll' Buffer Overflow Vulnerability". Secunia.com 23 Nov. 2004. <http://secunia.com/advisories/13269/>

[10] "Internet Explorer IFRAME Buffer Overflow Vulnerability". Secunia.com 23 Nov. 2004. <http://secunia.com/advisories/12959/>

[11] Lowe, Gavin. "Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR". Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 1055 (1996): 147-166.

[12] "Data Encryption Standard." 19 Nov. 2004. Wikipedia, The Free Encyclopedia. 19 Nov. 2004. Retrieved 22 Nov. 2004 <http://en.wikipedia.org/wiki/Data_Encryption_Standard>

[13] EFF: DES Cracker Project. Electronic Frontier Foundation. Retrieved 19 Nov. 2004. <http://www.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/>

[14] Kocher, Paul, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis" Lecture Notes in Computer Science. 1666 (1996): 388-397.

[15] Although DPA shows that it is hard to make a computer truly isolated.

[16] U.S.-Canada Power System Outage Task Force. "Finald Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations". Natural Resources Canada. Retrieved 23 Nov. 2004. <http://www.nrcan.gc.ca/media/docs/final/finalrep_e.htm>

[17] Schneier, Bruce. "Blaster and the great blackout." Salon.com 16 Dec 2003. Retrieved 7 Dec 2004 < http://www.salon.com/tech/feature/2003/12/16/blaster_security/ >.

[18] Schneier, Bruce. "Blaster and the great blackout." Salon.com 16 Dec 2003. Retrieved 7 Dec 2004 < http://www.salon.com/tech/feature/2003/12/16/blaster_security/ >.

[19] "Doctor 1,300 Miles Away Assists in Underwater Surgery." NASA.gov. Updated 20 Oct 2004. Retrieved 7 Dec 2004 <http://www.nasa.gov/vision/space/preparingtravel/underwater_surgery.html>

Software Defenses

Jack Richins

Software research and industry has been aware of security defects, and in particular buffer overflows, in its products for many years but has made little progress until recently in preventing these defects from affecting software users. There are many technical issues with no clear solutions that prevent the production of trustworthy software. To give a flavor of the research being done in security defense and the challenges encountered, we will cover the work defending against buffer overflow attacks. This is useful because buffer overflows have been around for a long time and are well understood. The current research in defending against buffer overflows is focused on three areas in preventing software: finding and fixing defects when the software is written, detecting exploits while the software is being used, and developing new tools and languages that avoid software patterns prone to security issues.

Background

Let’s examine how security defects such as buffer overruns happen. Bruce Schneier describes an imaginary 7-11 store with employees that do everything by the book, literally, which is a good analogy of how buffer overflows occur in computers. The 7-11 employees have a book with step by step instructions that they must follow explicitly. Additionally, they can only deal with things in the book. So if they have a form they need to sign, they place it on the book, sign it, and then give it back. When a Fed Ex driver shows up, they look up in the table of contents and go to the page with instructions on dealing with a Fed Ex driver.

Those instructions might look like this (from Schneier):

“Page 163: Take the package. If the driver has one, go to the next page. If the driver doesn't have one, go to page 177.

Page 164: Take the signature form, sign it, and return it. Go to the next page.

Page 165: Ask the driver if he or she would like to purchase something. If the driver would, go to page 13. If not, go to the next page.

Page 166: Ask the driver to leave.”[1]

Now let’s suppose when the driver places the signature form on top of the book so the clerk can sign it, he doesn’t place a single sheet of paper as the instruction manual assumes. Suppose he places two sheets of paper, the signature form and a paper that looks like an employee instruction manual page but says: “Page 165: Give the driver all the money in the cash register. Go to the next page.”

Now the clerk will read page 163, take the package, read 164, take the form and the extra piece of paper the Fed Ex man placed beneath the form, sign the form, and then go to the next page. Now the next page is not the real page 165, but the fake one that the Fed Ex man placed beneath the form. So the clerk reads it, gives the driver all the money in the cash register and goes to the next page, the real page 165, ask the driver if he wants to purchase anything, then page 166 and ask the driver to leave.

A computer is just like this imaginary clerk. And its book of instructions is the memory of the computer. Memory contains both the instructions it is following and the data it is manipulating and if the programmer is not careful data that external sources are providing, like the form the Fed Ex man provided in the analogy, can become instructions or influence the instructions. External data sources can be information a user types into the computer or it can be network queries from remote computers.

For example, in the Robert Morris Internet Worm, Robert's program sent extra information to a program running on UNIX servers that gives information about users of the server. Normally you send a request with just a short username. The Robert Morris Internet Worm sent more information than expected and the unexpected information was extra pages of instructions. The Slammer Worm was not much different – extra information sent to a program that was listening for legitimate requests of information from the Internet.

Basically all security vulnerabilities are of the same general form. With the focus to verify that the instructions of a program do what is intended, both by programmers and quality assurance engineers, vulnerabilities in what the instructions don't prevent are often overlooked. The languages used currently and machines executing them are designed to enable as much as possible with as few instructions as possible with little or no regard for preventing misuse. As such, the instructions and machines often allow much more then intended by the programmer, resulting in security vulnerabilities.

Solutions

Machine analysis of source code

Continuing the analogy of software and written English instructions, there are tools that find problems in source code much like word processors use spell checkers and grammar checkers to find problems in written documents. Some security errors, buffer overflows being one example, follow common patterns, kind of like run-on sentences. These patterns can be detected by code checker tools, called static analysis or source code analysis tools. The names static or source code are used because they analyze the written source code in contrast to dynamic analysis tools which analyze code as it is actively running on a computer.

Essentially tools are scanning the written source code for patterns that violate “do not’s” that security experts have identified[2]. Unfortunately, even at their best they have one glaring flaw. Returning to the word processing analogy, most word processors can identify run-on sentences by recognizing repeated conjunctions between clauses in a single sentence but few commercial programs can tell you how to correct your sentence while retaining your intended meaning. To validate that software is truly secure, source code analyzers should go beyond simple pattern detection and proactively suggest better patterns to use or how to correct the vulnerable source code[3].

Even without these improvements, source code analysis tools offer much. It is easy for programmers to make these kinds of mistakes, speaking of security errors in general and not just buffer overflows, with current development tools. It is possible for trained engineers to review others code and attempt to find errors, but such code reviews are laborious, time consuming, and prone to mistakes even by well trained engineers. Additionally, engineers sufficiently trained for this are rare and expensive. So these source code analysis tools enable a level of review of source code that is currently not economically feasible.

Unfortunately, these programs will find “false positives” - lines of code they think might be defects but are not. When the rate of false positives is too high compared actual defects found, the engineers lose trust in the systems and may not fix or believe reports of other actual errors. Further, it takes them longer to review the output of such systems and these systems lose one of their advantages, their speed.

“False negatives” are also possible; cases where there are real defects, but the tool does not recognize the defect. This can happen for multiple reasons. There can be defects that are so difficult to detect that attempting to detect them results in too many false positives to be useful, so the tool intentionally ignores them. Or there maybe defect patterns that attackers discover before security experts do and are able to add to static analysis tools.

Tevis and Hamilton survey several static analysis tools in “Methods For The Prevention, Detection And Removal of Software Security Vulnerabilities”. Their summary of these tools is they focus too much on UNIX applications to the exclusion of Windows and Macintosh software, still require a significant level of expert knowledge, and only cut down about a fraction of the manual code analysis that must be done. However, even with these limitations, they do help with code analysis, focus the analyst’s attention on more severe problems through prioritization features, and find real bugs in minutes that would have taken longer. However, none of the current checkers detect as many problems as manual analysis will uncover.[4]

Runtime analysis of code

Runtime or dynamic code analysis takes place while the program is running on the end users machine. It has the advantage of seeing real data and knowing exactly what is happening on the system. It has the disadvantage of consuming memory and CPU time that would normally be used for doing real work. Referring back to our 7-11 example of buffer overflows, it would be like having an appendix in the manual of instructions that the employee must constantly to refer to before and after certain instruction in the manual to help ensure an error has not occurred.

This has the advantage of more information available about what is actually happening. But it has the disadvantage of slowing many operations down. I.e., for every instruction in the main section of the employee handbook, there may be an instruction in the appendix that needs to be performed to validate things are working properly.

Despite these performance concerns, some levels of runtime analysis are becoming quite common. For example, code “canaries”, named after the canaries used by coal miners to detect poisonous gases[5], are being inserted automatically by development tools used in Linux and Windows application development[6]. These code canaries are used to detect buffer overruns at runtime – special checking code is also created to verify the canaries are still in place after certain operations to make sure the buffers have not been overrun.

Most runtime detection tools have two shortcomings. First, they require recompilation, meaning the source code must be used to regenerate machine code using new tools. Second, most tools stop the running program when they detect an error. This is deemed most secure because it avoids trying to recover from an attack where you may not be able to trust the system. However, this increases the risk of denial of service attacks as all buffer overflow attacks and other security vulnerabilities these tools detect can be used as a means of stopping the program and preventing legitimate uses of the program.[7]

For example, say Microsoft Internet Explorer 2012 has runtime buffer overflow detection and a buffer overflow vulnerability in how it processes web pages. A hacker might, unbeknownst to Microsoft, alter the MSN home page so it causes a buffer overflow and attempts to force Internet Explorer to send all passwords and credit card numbers stored on the computer to the hacker. Without the runtime buffer overflow detection, any user that directed Internet Explorer to the MSN home page, the default home page for Internet Explorer, would be unknowingly sending the hacker their passwords and credit card numbers. With the runtime buffer overflow detection, every time they tried to start Internet Explorer and it tried to load its home page it would crash. This would prevent anyone using Internet Explorer and MSN as their homepage from browsing the internet at all. While the runtime detection protects the users passwords and credit card numbers, the hacker is still able to prevent these users from using their web browser to surf the internet.

There has been research into making such failures invisible to the user. In other words to save enough information that after terminating the program the runtime environment or operating system can restore program so the user does not lose any work. This would be done by recording enough of the work the program is doing while it's running to recover should it fail.

Unfortunately there is a dilemma – if the work is recorded that caused the failure, then the failure will happen again when the operating system or runtime environment tries to recover. So such a recover mechanism must record as little work as possible to avoid recreating the error. These two principles, the principle of recording all work so there is no loss of work and the principle of not recording work so you don't repeat the defect, are in direct conflict in all but the simplest types of defects.[8]

Research suggests though that runtime environments or operating systems might be able handle simple defects, which are estimated to be about 10% of failures. Further, there might be ways of providing aids to applications so in cooperation with the runtime environment or operating system programs can fail and recover with little work lost. For example, referring to our Internet Explorer 2012 scenario, when trying to start the second time, Internet Explorer might avoid the MSN home page and inform the user there is a problem with loading it rather than crashing repeatedly.[9]

Although runtime detection of security defects is not capable of completely protecting a system, as computers get faster they are becoming a more reasonable trade off for the protection they do provide. Even if they only prevent 10% of the errors at a 5% performance penalty, that's a good tradeoff when you aren't using that 5% extra performance anyways. Particularly as computers are shifting to having multiple processors rather than a single fast one, more processor intensive solutions like having two versions of a program run simultaneously and check each other's results becomes more reasonable.

Other Languages

Since many of these defects being used to compromise systems are specific to languages that directly manipulate memory, one suggestion has been to use other languages. One step towards this can be seen in Java, C#, and some other .Net languages. These language systems manage memory automatically for the programmer and to the degree they manage it correctly they are free from defects like buffer overflows.

However, even these languages have inherent defects. For example, in the .Net framework there is a way of specifying code to be executed when something unexpected happens. This language feature, if used in correctly, can be used to reroute execution. For example, if the code that verifies a password generates an exception while running that it did not expect and does not handle, it may continue executing as if the password was correct[10].

Tevis and Hamilton propose an even more revolutionary change from imperative languages to functional languages. Functional languages are more purely rooted in mathematics and as such are easier to mathematically prove correct.[11]

Programs written in imperative languages, the most commonly used in commercial software, are a series of command statements. When run, imperative programs execute these instructions. Programs written in functional languages are composed of definitions of functions and are executed by evaluating the function definitions.

Historically, functional languages were criticized as being slow, but with faster processors that has become less of an issue. There still remains a chicken and egg problem. Most imperative languages have several editor programs, debugging programs, compilers, and optimizers to choose due to a long history of being used for commercial software and therefore having a large market for these tools. Functional languages which have a much smaller market, lack such quality and quantity of supporting tools. Of course without such supporting tools, it is unlike functional languages will ever be used much commercially and develop a market for these supporting tools.

Further, there is not the industry wide knowledge of functional languages among software engineers that exists with imperative languages. Although functional languages are frequently taught in college, that's usually the last a software engineer encounters functional programming languages.

For these reasons, conversion to functional languages would require a massive retraining of most software engineers and retooling as most tools used to generate commercial software are designed around imperative languages such as C, C++, Java, and C#. This would prove very costly in terms of engineers' time. And though functional languages would probably improve many of the ills being experienced with imperative languages, they certainly would not be defect free and with extensive use we might discover they have their own set of unique defects.

Hybrid Approach

There are also some hybrid approaches being explored. One promising example is Microsoft's PREfix code analysis tool. It analyzes the source code like a static analysis tool, but then uses this analysis to build a simplified model of how the code will run. Then it analyzes this model of runtime behavior to predict defects. Since it is not run at runtime it does not impact performance. Further, since it is a simplified model of behavior, it can test many different execution options more quickly than trying to test every possible ordering of instructions. But to the degree it accurately models the program it is as accurate as runtime analysis tools in finding real defects and avoiding false positives. It has proven itself sufficiently useful to see wide scale use at Microsoft on large bodies of code on a daily basis.[12]

Conclusions

A surprising conclusion of the work on PREfix was that how the information was presented to the developer was very influential in improving the rate of fixing the defects correctly found by PREfix[13]. As mentioned earlier, one of the defects of many static tools was even after find defects, they did not aid the programmer in correctly understanding and fixing the defect. Within the context of static analysis tools, this seems the area likely to yield the most immediate improvements.

Runtime detection seems to continually be a trade-off of security for performance. As performance increases and security becomes increasingly important, this is likely to become a more attractive option. However, a solution to denial of service attacks on runtime hardened systems is needed to make these systems truly trustworthy.

The idea of spontaneously changing languages used in commercial development seems unlikely, but it is possible that as the security benefit of certain language features can be proven they will slowly make their way into languages currently in use. So an evolutionary path to more secure languages and tools may be possible and should be further researched and adoption by industry encouraged.

At present, it does not appear any area of research promises a magic bullet to our current security problems. But they all show promise of shoring up our defenses and making progress in securing our systems. It is discouraging that we still have buffer overruns, but as pointed our earlier, there are signs of progress. Buffer overruns are being found more quickly and fixed. Complete elimination of these types of errors appears unlikely. It is more feasible that simultaneous improvements on multiple fronts will improve the overall security of software and internet users.

[1] Schneier, Bruce. Secrets and Lies, Digital Security in a Networked World. Wiley Computer Publishing, NY, NY, 2000, pp. 207-210.

[2] Tevis, Jay-Evan J., and John A. Hamilton, Jr. "Methods For The Prevention, Detection And Removal Of Software Security Vulnerabilities" Presented at ACM Southeast Conference '04, April 2-3, 2004, Huntsville, AL, USA, pp. 1.

[3] Tevis, Jay-Evan J., and John A. Hamilton, Jr. "Methods For The Prevention, Detection And Removal Of Software Security Vulnerabilities" Presented at ACM Southeast Conference '04, April 2-3, 2004, Huntsville, AL, USA, pp. 2.

[4] Tevis, Jay-Evan J., and John A. Hamilton, Jr. "Methods For The Prevention, Detection And Removal Of Software Security Vulnerabilities" Presented at ACM Southeast Conference '04, April 2-3, 2004, Huntsville, AL, USA, pp. 4.

[5] ON THIS DAY | 30 | 1986: Coal mine canaries made redundant. 22 Nov. 2004. <http://news.bbc.co.uk/onthisday/hi/dates/stories/december/30/newsid_2547000/2547587.stm >

[6] Silberman, Peter, and Richard Johnson. “A Comparison of Buffer Overflow Prevention Implementations and Weaknesses” 22 Nov. 2004. <http://www.idefense.com/application/poi/researchreports/display?id=12 >

[7] Wilander, John, and Mariam Kamkar. “A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention” 10th Network and Distributed System Security Symposium (NDSS), 2003, pp. 10.

[8] Lowell, David E., Subhachandra Chandra, and Peter M. Chen. “Exploring Failure Transparency and the Limits of Generic Recovery” Appears in the Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)

[9] Lowell, David E., Subhachandra Chandra, and Peter M. Chen. “Exploring Failure Transparency and the Limits of Generic Recovery” Appears in the Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)

[10] Kolawa, Dr. Adam. “Banish Security Blunders with an Error-prevention Process” 22 Nov. 2004. <http://www.devx.com/security/Article/20678/0/page/4 >

[11] Tevis, Jay-Evan J., and John A. Hamilton, Jr. “Methods For The Prevention, Detection And Removal Of Software Security Vulnerabilities” Presented at ACM Southeast Conference ’04, April 2-3, 2004, Huntsville, AL, USA, pp. 4-5.

[12] Bush, William R., Jonathan D. Pincus, David J. Sielaff. “A Static Analyzer for Finding Dynamic Programming Errors” Intrinsa Corporation, Mountain View, CA, USA, pp. 16.

[13] Pincus, Jon. “User Interaction Issues In Detection Tools” 22 Nov. 2004, pp. 61 <http://www.research.microsoft.com/specncheck/docs/pincus.pdf >

Licensing Software Engineers

John Spaith

An old saying tells us that “It is the poor craftsman who blames his tools.” There are clearly many promising software development languages and tools to aid in writing more secure programs. But any technological breakthrough in this area will be for naught if the software engineer is incompetent. Thoughtful Roman engineers, using crude tools and processes by modern standards, created structures that have lasted for over two thousand years. Will people two thousand years from now wax poetic about their robust software, still in use, created by thoughtful American engineers? What can the government, standards bodies, and industry do to raise engineering practices so that software created today is more reliable and more secure today? One area of particular interest is whether software engineers should be licensed, as they are in other fields of engineering.

Licensing should not be confused with certification. Licensing has considerably more strict requirements. Certification is “an occupational designation issued by an organization that provides confirmation of an individual's qualifications in a specified profession or occupational specialty[1].” Software vendors, such as Microsoft and Sun, offer certifications for their products. While certification is voluntary, “Under licensure laws, it is illegal for a person to practice a profession without first meeting state standards.[2]” Licensing is granted by governments, not companies or organizational bodies. Furthermore, licensed engineers in other professions may be legally liable if their work causes harm[3].

Software engineering is a relatively new field and software engineers often must invent everything as they go along. Should governments require licensing for software engineers, however, other fields of engineering that already require licensing would provide valuable models.

Engineering Licensing

An engineer outside the software industry – such as a civil engineer - who wishes to receive a professional license must undertake a grueling process. As a prerequisite, he must have an engineering degree from a program accredited by EAC/ABET. When he is a senior in college typically, he will take a Fundamentals of Engineering (FE) exam. This exam covers basic engineering, science, and mathematics that all engineering disciplines must be familiar with. After at least four years of professional engineering practice - typically working under the guidance of a professional engineer (PE) – the aspiring engineer may take the Principles and Practice examination. This test focuses on his discipline of engineering in more depth. It takes three days to complete and is comparable in length and breadth to the bar exam that lawyers take. After the engineer successfully passes the test, he is licensed and can now officially be called a professional engineer (PE). After completing the final test, he must take additional courses to stay current in his field. The entire licensing process is managed by state governments, not by a company or professional society[4].

The non-licensed engineer may work as an engineer, even if they never seek licensing. However, the scope and authority that he has without a license is limited. Among other limitations, the non-licensed engineer cannot be a consultant in private practice, sign off on projects, or act as witnesses in a court of law[5]. Being able to sign off on the work of other engineers is of particular importance. The engineer who is not licensed has his ability to attain technical leadership drastically reduced[6].

An integral component to the licensing program is training and testing in ethics. Engineering a bridge or fire escape is, after all, a matter of life or death. Yet the PE must operate in a market based economy where frequently non-technical managers may want to take short-cuts. The ethical standards bodies of professional engineering societies not only provide guidelines for dealing with various moral dilemmas the engineer may face, but also have the authority to censure him and even revoke his license should he behave unethically[7]. Furthermore, a licensed engineer may be held legally liable for faulty work done under his direction. Should the concern for his fellow man not provide the engineer sufficient motivation, the risk of being publicly held accountable for his actions may.

Arguments for Licensing Software Engineers

Software engineering author Steve McConnell begins his book <u>After the Gold Rush</u> with the story of a 1937 boiler explosion that killed 300 school children. After this the state of Texas began requiring licensing for engineers. McConnell ominously points out that this boiler is controlled by software today[8]. Software is everywhere - embedded in factory robots, aiding air traffic controllers, and running stock exchanges being just a few examples. <u>After the Gold Rush</u> was published in 1999, when the greatest threat was arguably incompetence on the part of an individual programmer. The word security does not even appear in its index. The threat that lurks in the form of individuals, criminal organizations, and hostile foreign governments that are just a buffer overflow away from wreaking havoc on society is much better understood by the public than just five years ago.

How would licensing software engineers help mitigate security threats? To understand this, it is first necessary to understand what the software engineering licensing would be like. The implementation, as envisioned by McConnell at least, would be similar to traditional engineering licensing today. None of the infrastructure required to support these policies is in place at present, however.

The software engineer would have to graduate from an accredited four year university with a specialization in software engineering. The focus of this program would be engineering of software, not the more theoretical aspects of computer science that most colleges focus on today[9]. In his senior year, the aspiring software engineer would take a Fundamentals of Engineering exam. The engineer would then work for a few years in industry before being allowed to take a Principles and Practice examination that was specific to software engineering. The IEEE currently has a software engineering certification (not a license) that could serve as a model for this exam[10]. After completing the Principles and Practice examination, the licensed engineer would take continuing learning courses and also be bound by the ethical considerations of his profession[11].

Most software engineers would not be licensed. McConnell estimates the number to be between five to ten percent[12]. But these five to ten percent would form the core of technical leadership for the profession.

The arguments in favor of requiring some engineering of software engineers are compelling. Just as a licensed civil engineer is required (legally and morally) to understand all threats to the bridge he signs off on, so a professional software engineer would understand all threats to the program running that he writes. This would most certainly entail detailed knowledge of hacker's methods and processes to protect against them. Just as whether a bridge is safe or not is not left to the whims of an MBA, so to could software security be placed into more reliable hands. Today it is the program manager who is more focused on business than technical issues who decides when software is "secure enough" to ship. In the future it could be an engineer focused purely on technical issues.

There have been great strides in software engineering practice in the last thirty years. Countless books and papers have been written to advance the state of the art in the fields of productivity, reliability, and security. Yet many best practices are not widely disseminated, frequently because engineers do not keep pace with the changes. The situation has been compared to having seventy five percent of modern doctors continue to use leeches because they were unaware of penicillin's existence[13]. Part of retaining an engineering license entails continuing learning. Educating an elite corps of software engineers with new, safer development models, could help bring the state of the art into common practice.

The software industry often evokes the image of two guys working in a garage, inventing "the next big thing". Licensing would radically change the field, making it much more rigid and professional. With the stakes so high and the malicious users so numerous, proponents of licensing claim that this change is much needed.

Arguments against licensing

Software engineers are not in general in favor of being licensed. Many reasons are given, many of which are emotional. Software engineers typically enjoy the freedom that their profession affords them to be creative. Governments and professional societies forcing formal engineering processes on software engineers are seen as ending the “good times.” More reactionary arguments against licensing are based on worries of abuse by the certification board that could de-certify engineers who did not fit into a certain mold, just as the New York law board in the 1960’s decertified lawyers opposed to the Vietnam War[14].

None of these emotional arguments should be taken too seriously. Society frankly does not care whether civil engineers are happy being licensed or not, nor should it care about the feelings of software engineers. And the potential for abuse exists in any organization, a software licensing foundation being no exception. If, however, a licensing program is ever forced upon software engineers by a government body, these responses must be anticipated. An educational program would be essential.

There are more reasonable arguments against licensing that should be considered. Some of the best are literally in front of our eyes. This paper was written with Microsoft Word, a technological marvel that is taken for granted. Though earlier versions of Word were unreliable, thanks to technological progress and market pressure Microsoft has produced an extremely reliable program. Its software has improved without licensed engineers.

The security of software - and Microsoft products in particular - is another question. Still, great progress is being made <u>without</u> government intervention. With viruses widely reported in the media and understood by the public, companies understand that they must react to market demands. Microsoft spent about one hundred million dollars retraining its engineers to write more secure code[15]. The effort seems to have paid off. Iain Mulholland, manager of Microsoft's security response center, said that, "There were only nine critical or important vulnerabilities in Windows Server 2003 within the first 292 days of release, compared to 38 vulnerabilities discovered in Windows 2000 in the same time period.[16]"

As the technology industry rapidly innovates, so do the techniques available to hackers. A class of attack that has recently become more widely publicized is the integer overflow attack. Microsoft engineers (all of whom had security training) on the core Windows CE Networking team spent two weeks fixing potential integer overflow issues in their code. Almost all of this potentially vulnerable code has been shipping for years. Even had all of the engineers been licensed, had the threat of integer overflows been unknown at the time of their training then their licensing would have been useless. Building a bridge, on the other hand, is an extremely well understood process that engineers have been doing since ancient times.

The model of the two guys in the garage inventing the "next big thing" may not be something we should be too quick to move away from. Despite the tremendous progress they have made in the last decades, computing is still in its infancy. Unimaginable advances lie in the future, both in theoretical aspects of computer science and in best practices of developing software. Formal bodies such as licensing boards, however well intentioned, could slow the pace of innovation if they set the barriers to entering the software field too high. Perhaps the next big thing to come out of a garage will make computing infinitely more reliable and secure than it is now, which is ultimately the goal that licensing seeks to achieve as well.

Conclusions on Licensing

The impact that software engineering has on the world is tremendous. Allowing the field to continue as it is – with a haphazard, get to market, two guys in the garage mentality – could be courting disaster. That more professionalism and responsibility is needed at all levels of the profession to ensure more secure software is clear. Whether licensing engineers is the best way to do this is not. Licensing may help the systems of today be more secure. But aren’t market forces already causing this? And how licensing could be applied to a rapidly changing field such as software engineering is far from clear. The metrics to determine when software is reliable or secure are nowhere near as advanced when a bridge is secure.

Perhaps a hybrid approach is best, where mission critical software such as that embedded inside boilers should require professional engineering practices and licensed software engineers. Requiring a wider array of software projects to use licensed engineers involves a careful consideration between the public’s short-term convenience and safety and the long-term good that comes with increased innovation.

[1] "CSDP: Is Certification for You?" IEEE Computer Society, 2004. http://www.computer.org/certification/cert_for_you.htm
[2] "CSDP: Is Certification for You?"
[3] McConnell, Steve. After the Gold Rush. Microsoft Press, November, 1999. Redmond, WA. pg 106.
[4] Cottingham, J. Richard. "Engineering Licensing Is Important for Your Future" http://www.jobweb.com/Resources/Library/Careers_In/Engineering_41_01.htm
[5] Cottingham.
[6] McConnel, pg 107.
[7] American Society of Civil Engineers. "Guidance for Civil Engineering Students on Licensing and Ethical Responsibilities". September 2001. http://www.asce.org/pdf/ethics_student_guide.pdf
[8] McConnel, pg ix.
[9] McConnel, pg 117.
[10] "CSDP: Is Certification for You?"
[11] McConnell, pg 139.
[12] McConnel, pg 104.
[13] McConnel, pg 150.
[14] Tom DeMarco. "On Certification: Letter to the Editor". Cutter IT Journal, 1998. Arlington, MA. http://www.systemsguild.com/GuildSite/TDM/certification.html
[15] LeClaire, Jennifer. "Microsoft and the New Science of Security Flaws". E-Commerce Times, September 26, 2004. http://www.ecommercetimes.com/story/19514.html
[16] Chapman, Siobhan. "AusCERT: Microsoft's security journey". itNews Australia, May 26, 2004. http://www.itnews.com.au/msoft_storycontent.asp?ID=9&Art_ID=19766

Liability for Software Defects

Andrew Pardoe

Complexity of systems and prevalence of failure

Who is responsible for these software and hardware deficiencies which can lead to catastrophic failures and exploits? Why haven't they fixed these bugs? Shouldn't someone be held liable for these problems?

Computers and software are immensely complex engineering feats. Windows Server 2003 was written by about 5000 developers working on over 50 million lines of code.[1] Intel's "Prescott" microprocessor contains 330 million transistors in an area the size of a fingernail.[2] As security expert Bruce Schneier points out, complexity is the worst enemy of security.[3] But consumers want features and features increase complexity. A word-processing program which doesn't print is still effective as long as you can save your file and load it into a printing program. But having the printing functionality in the word-processing program itself is a useful—but less secure—feature which customers demand. Computer programming pioneer Nicklaus Wirth summed it up nicely: “Increasingly, people seem to misinterpret complexity as sophistication, which is baffling—the incomprehensible should cause suspicion rather than admiration.”

Computer users have come to expect that computers are naturally prone to failure. We have come to accept that a computer is an imperfect tool and will work around any problems we encounter. Users will in fact make excuses for bad computer program designs rather than blame the programmers and design engineers.[4] One reason users infrequently demand perfection is that we often use computers to do things we could easily do without computers: the proper functioning of the computer is essentially unimportant because we can do what we need to do without the aid of the computer or we know of some “workaround” to force the machine to give us the desired results despite what it “thinks” is correct. Other users are willing to forgive imperfect designs and implementations to get their hands on new tools and functionalities.

What happens when we rely on computers to do more important tasks? A software error in General Electric's XA/21 system contributed to the extensive and immediate spread of the 14 Aug. 2003 blackout which affected much of the northeastern American continent.[5] When a software error (or hardware error) affects so many people should someone be held liable?

Example: a consumer-grade software tool

At first blush it seems obvious that an individual or corporation which produces faulty software should be held liable for the errors. We have product liability laws and computer systems—hardware and software—are just another product. In fact the IRS ruled in 1985 that authors of tax software may be liable for advice given to taxpayers.[6] Now that the industry has matured, however, the IRS no longer addresses the issue of liability for software errors in tax preparation software and in fact allows a few math errors to exist. The IRS's current guidelines for e-File providers specify that the software used to file tax returns must pass a test which verifies, in part, that “returns have few validation or math errors.”[7] One popular consumer-grade tax software package, TaxCut 2003, includes a standard license exempting the publisher from any liability and makes clear that using the software does not make Block (the publisher of TaxCut 2003) the consumer's tax preparer.[8] This kind of license is ubiquitous in consumer software freeing the publisher of software from any legal liability.

This kind of end-user license may not be enforceable due to an implied warranty of merchantability.[9] Despite the IRS's requirement of “few math errors” one would expect that tax-preparation software would do math correctly. Yet there is very little case law explicitly addressing the liability of software vendors. Because of this legal ambiguity large software vendors are advocating the adoption of a modification to the Uniform Commercial Code Article 2B called the Uniform Computer Information Transactions Act (UTICA). UTICA, if adopted by the individual states, would free software vendors from the responsibility to guarantee that their software works.[10] This law is, understandably, popular with software vendors and equally unpopular with consumer advocacy groups.

Liability for negligence

How do we determine who should be liable for software errors? In the case of United States vs. Carroll Towing Co. Judge Learned Hand stated a formula to test a company's duty to see that their product does not cause harm. This formula was a function of three variables: P, the probability of failure, L, the gravity of resulting injury given a failure, and B, the burden of adequate precautions. If B is smaller than the product of P and L ($B < PL$) then the corporation should be found to be negligent.[11]

What is the probability of failure? Few programmers would claim to have written a program which is free of bugs. One of these bold few was Dr. Donald Knuth author of (amongst other things) the TeX typesetting software. Dr. Knuth said in the preface to his book *TeX: The Program* that he “believe[s] that the final bug in TeX was discovered and removed on November 27, 1985.” And yet bugs were still being uncovered--though very infrequently--ten years later.[12] TeX is a relatively small program. At over 50 million lines of code (for Microsoft Windows) or 330 million transistors (for an Intel microprocessor) existing computer software and hardware is clearly too complex to be completely tested.

The more interesting question is the probability of a failure which causes harm. This approaches Judge Hand's second question: what is the gravity of resulting injury given a failure? While a physical product is normally intended for one single purpose software is often used for purposes which the original vendor never imagined possible. For example, Lotus 1-2-3 spreadsheet software is used for nuclear reactor core simulation and the design of a lunar base.[13]

It is impossible to completely test any reasonably complex software program and even more impossible to test it in usage situations for which the software was not originally designed. The manufacturer of wrench would not be held liable for an injury stemming from the use of that wrench as a hammer or screwdriver. If a program appears to work for a given purpose, why should one not use it for that purpose? Should the vendor have tested it in that scenario which may not have even existed when the program was written?

Another problem is that computer systems force us to devine the purpose of interfaces constantly. For example, when one uses the “trash bin” on a MacIntosh computer the obvious metaphor is that this virtual piece of paper is going into the virtual trash bin. But is it also obvious that the virtual trash bin needs to be emptied by the user? Or should the user be able to assume that the computer will take care of it for her? If a user puts a sensitive document into the trash bin where it is discovered by another user who should be responsible for any malicious usage of the document? People who don't fully understand tools still need to use the tools but present a security risk to themselves and others. Testing can't cover all ranges of user expertise.

Likewise it's difficult to ascribe liability to a software vendor when software experiences such dynamic application during—and after—its intended lifetime. Think of the Y2K “crisis” which resulted from the fact that software vendors had used two digits to represent a four-digit year. While we look back and say it was stupid to write a program in 1975 that wouldn't be able to handle the year 2000 we can easily imagine that the software developers of 1975 never thought their programs would still be in use in 25 years.

Burden of software testing

What is the burden of adequate precaution as it applies to testing computer software and hardware? Bruce Schneier comments that “on the one hand, it's impossible to test security. On the other hand, it's essential to test security.”[14] It’s impossible to demonstrate that a system has no security faults. One can prove that there are faults by demonstrating the fault. One can hypothesize the likely existence of a fault by examining the system code. But proving that a system is absolutely secure is like proving the null hypothesis: you can't prove the nonexistence of something you can't find.

Testing clearly adds more cost to the development cycle in terms of total time spent working on a product. Provably thorough testing would add an infinite amount of work to the development cycle. Is the incentive to produce software high enough to include the cost of adequate testing?

Consider on one side of the spectrum the open-source free software project. What would be the consequences if the developers of “Open Tax Solver” (an actual open-source tax software project) were held liable for program errors in the same way that a professional tax advisor is liable for incorrect advice?

Software developers generally create software because they have identified a problem they want to solve. As Eric Raymond wrote, “Every good work of software starts by scratching a developer's personal itch.”[15] The programmer had a problem to solve and wrote a program to solve the problem. Having already done the work and not having intended to profit from the work the programmer releases the code for other people to use.[16] Because the developer has minimal incentive to release software for other people to use the warranty expressed by most open-source licenses can be reduced to “if it breaks, you get to keep both pieces.”

For “Open Tax Solver”, then, we can assume that the burden of adequate precaution is unreasonably high for the software developers. No one is paying for the software, thus, no one should hold the developers responsible for faults in the software. In another way of thinking, “you get what you pay for.” On the other hand, the correct monetary difference between free software and for-profit software is unknowable. Given that the marginal cost of software is zero, however, at what price point can we say that a commercial developer should be liable for defects in the software? If Microsoft charges $300 for Windows can we say that the difference between the cost of Windows and Linux is adequate to demand adequate security testing? What portion of the $300 purchase price represents testing for the security bug which affects you?

Incentives for quality assurance testing

The purpose of holding manufacturers liable for negligence is to provide an incentive for manufacturers to produce products of sufficient quality to not pose an unreasonable threat to the users of those products. But product liability generally applies to manufacturing defects as opposed to design defects. Software flaws almost never result from manufacturing defects (as a manufacturing defect—such as an error in the CD replication process—is more likely to make a program not function at all than to function incorrectly.) Hardware manufacturing defect, such as bad memory or a defective connection, generally produce unpredictable errors. The design process cannot account for all possible errors. Quality assurance testing cannot trace all possible execution routes through a software program with all possible inputs. And the designer of a hardware or software product often cannot predict all of the execution environments in which the product will be used. Interactions between software programs or hardware components often reveal bugs that weren't revealed in the normal course of testing.

Companies and individuals have a dual incentive in producing quality software. The first incentive is to produce the software, the second to produce a quality product. There is a balance between features and bugs which exists from the very beginning of the development cycle. Fixing every bug—or deciding exactly what represents a bug—would prevent most products from ever shipping. Consumers pick a products based upon a balance between features and usability. The markets do a good job of sorting out which products are of sufficient quality to be usable. When there is a threat of lost sales due to security problems companies will respond or go out of business. When consumers demand it “security” has become a feature instead of just an added expense.

Future scenarios

The future holds even more uncertainty regarding product liability for computerized products if for no other reason than our increasing reliance on computers in everyday life. Some producers of computerized technologies are being exempted from defect liability: the SAFETY Act of 2002 (Support Antiterrorism by Fostering Effective Technologies) provides liability protection for sellers of “Qualified Anti-Terrorism Technologies.”[17] Some producers, however, are being informed of their liabilities up front: the U.S. Food and Drug Administration has published guidelines for computerized systems used in clinical trials[18] and informed manufacturers of medical devices that they are liable for embedded computer errors in medical devices with regards to the Year 2000 problem.[19] The decision to exempt Qualified Anti-Terrorism Technologies from liability should lead to an atmosphere where innovation is unfettered. And the liability ascribed to medical devices will protect patients from harm. But terrorism technologies could injure people and medical devices could certainly use innovative solutions. What’s the right balance of liability and incentive to innovate?

The prevalence of embedded computer systems causes even more problems for the field of liability for computer flaws and security exploitations. Previously manufacturers would use custom-designed hardware and software systems to control consumer products ranging from blenders to automobiles. Now it is often cheaper to use a general-purpose microprocessor and an off-the-shelf operating system such as Embedded Linux or Windows CE to control the consumer product. Who is liable when a flaw or security exploit compromises these products? The flaw in General Electric's XA/21 system which contributed to the northeast blackout is a known problem in the Linux kernel.[20] The SAFETY act protects sellers of qualified anti-terrorism technologies and the FDA regards a medical device as a single unit regardless of the origin of embedded hardware or software. In these cases the makers of the final product, be it an anti-terrorism technology or a medical device, are held liable (or not liable) for the final product.

[1] Thurrott, Paul. "SuperSite for Windows." 20 Nov. 2004 <http://www.winsupersite.com/reviews/winserver2k3_gold2.asp>

[2] Intel Corporation. "Intel Unviels World's Most Advanced Chip-Making Process." 20 Nov. 2004 <http://www.intel.com/pressroom/archive/releases/20020813tech.htm>

[3] Schneier, Bruce. Beyond Fear: Thinking Sensibly about Security in an Uncertain World. New York: Copernicus Books, 2003. p. 90

[4] Norman, Donald A. *The Psychology of Everyday Things*. New York: Basic Books (Harper Collins), 1988. pp. 34-36.

[5] Poulsen, Kenneth L. "Software bug contributed to blackout." *Risks Digest* 23.18. 20 Nov. 2004 <http://catless.ncl.ac.uk/Risks/23.18.html#subj1>

[6] Schauble, Paul. "Software developer's liability." *Risks Digest* 3.4. 20 Nov. 2004 <http://catless.ncl.ac.uk/Risks/3.04.html#subj2.1>

[7] Internal Revenue Service. *Handbook for Authorized IRS e-file Providers of Individual Income Tax Returns.* 20 Nov. 2004 <http://www.irs.gov/pub/irs-pdf/p1345.pdf>

[8] Block Financial Corporation. *TaxCut End-User License Agreement.* 20 Nov. 2004 <http://www.taxcut.com/license/TCB2003.pdf>

[9] Kaner, Cem. *Bad Software: What to do when Software Fails.* New York: Wiley Computer Publishing, 1998. 178-181.

[10] IDG.net. “UTICA: Summary Information.” 20 Nov 2004 <http://cgi.infoworld.com/cgi-bin/displayStory.pl?/features/990531ucita_home.htm>

[11] Landau, Philip J. “Comment: Products Liability in the New Millenium: Products Liability and the Y2K Crisis.” *The Richmond Journal of Law and Technology* VI:2. 20 Nov. 2004. <http://law.richmond.edu/jolt/v6i2/note4.html>

[12] Kinch, Richard K. “An example of Donald Knuth’s Reward Check” 20 Nov. 2004 <http://truetex.com/knuthchk.htm> Interestingly, Dr. Knuth pays a small bounty to every person who finds a bug in his software, documentation or books.

[13] Heckman, Corey. “Two Views on Security Software Liability: Using the Right Legal Tools”. *IEEE Security & Privacy*. 1:1 (2003): pp. 73-75.

[14] Scheier, Bruce. *Beyond Fear*. p. 228

[15] Raymond, Eric. The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. Sebastapol, CA: O’Reilly, 1999. p. 32.

[16] The secondary effect of getting contributions of source code to make the program better without extra effort by the original author is often not seen in open-source software. Usually the number of contributors is extremely small compared to the number of users.

[17] Office of the Secretary, Department of Homeland Security. *Federal Register* 68:200. (2003).

[18] Office of Regulatory Affairs, Department of Health and Human Services. “Guidance for Industry: Computerized Systems Used in Clinical Trials” 20 Nov. 2004 <http://www.fda.gov/ora/compliance_ref/bimo/ffinalcct.htm>

[19] Assistant Secretary for Legislation, Department of Health and Human Services. “Statement on Medical Devices and the Year 2000 Computer Problem by William K. Hubbard” 20 Nov. 2004 <http://www.hhs.gov/asl/testify/t990525a.html>

[20] Bovet, Daniel and Cesati, Marco. *Understanding the Linux Kernel.* Sebastapol, CA: O’Reilly, 2001. pp. 474-474. The XA/21 system had a bug due to a race condition which is a well-documented problem in the Linux kernel’s swapping of pages in memory.

A Lab to Certify Software

Caroline Benner

Software companies do not make secure software because consumers do not demand it. This shibboleth of the IT industry is beginning to be challenged. It is dawning on consumers that security matters as computers at work and home are sickened with viruses, as the press gives more play to cybersecurity stories, and as software companies work to market security to consumers. Now, consumers are beginning to want to translate their developing, yet vague, understanding that security is important into actionable steps, such as buying more secure software products. This, in turn, will encourage companies to produce more secure software. Unfortunately, consumers have no good way to tell how secure software products are.

Security is an intangible beast. When a consumer goes to buy a new release of Office, he can see that the software now offers him a handy little paper clip to guide him through that letter he is writing. He understands that Microsoft has made tangible changes, perhaps worth paying for, or perhaps not, to the software. Security does not lend itself to such neat packaging: Microsoft can say it puts more resources into security, but should the consumer trust this means the software is more secure? How does the security of Microsoft's products compare to that of other companies?

To help the consumer answer these questions, one idea that has been floated is to create an independent lab to test and rate software on security. A rating system would allow consumers to vote with their dollars for more secure software. It could also pave the way to assigning legal liability to companies for software defects. The point of rating software would be to tip a software company’s cost-benefit analysis in favor of providing security, be that via the threat of liability—increasing the cost the company incurs for selling insecure software—or via increased consumer demand for secure software—increasing the benefit the company receives for making their products secure.

A certifying lab seems to be an ingenious solution. The consumers who buy the software and the lawyers who would assign liability do not know what secure software looks like so they will trust the software security experts running the lab to tell them. The trouble is, those software security experts do not know what security looks like either.

Why is software security so hard?

Quite simply, software security experts do not know whether a piece of software is secure because it is generally impossible to ascertain whether a piece of software has a given property for all but the most simple classes of properties. Security, needless to say, is not one of those simple properties. Why is this true? Consider first what security vulnerabilities are. Security vulnerabilities are typically errors—bugs—made by the designers or implementers of a piece of software. These bugs, many of them very minor mistakes, live in a huge sea of code, millions of lines long for much commercial software, that is set up in untold numbers of different environments, with different configurations, different inputs and different interactions with other software. This creates an exponential explosion in the number of things we’re asking the software to be able to do without making a mistake.

So where do bugs come from? Bugs are either features that the software's designers planned for the software but that did not make it into the final product, or undesired features that the designers did not ask for, but that became part of the final product anyway. It's very easy for a programmer to introduce the latter type of error, which is where most security vulnerabilities sneak in. Perhaps Roger the developer writes perfect code—an impossibility in and of itself since humans make mistakes. Now Joe next door writes another piece of code that uses Roger's code in a way Roger did not expect and this can introduce bugs. Or Roger's code, which was written on Roger's computer with its unique configurations works fine on Roger's computer but not fine on Joe's next door.

If you can't avoid making errors, can you catch them? The best we can do is test the software, and testing will never catch all errors since you need to test for an unlimited number of possible things the software might do. If you want to keep your cat fenced up in your backyard, you have to anticipate all possible ways the wily cat might escape. You can climb on the roof and guess whether the cat would be willing to jump off it over the fence. You can crouch down and guess whether a cat can slink under the fence. And still, the cat will probably escape via the route you overlooked. It is very hard to prove that something is not going to happen, that the cat cannot possibly escape the yard, that a piece of software has no vulnerabilities.

Two other problems make building secure software difficult and unique from other complicated engineering tasks, such as building a bridge. First, there is often little correlation between the magnitude of error in the software and the problems that the error will cause. If you forget to drill in a bolt when building a bridge, the bridge is not likely to collapse. However, if you forget to type a line of code, or even a character, you may well compromise the integrity of the entire program.

The second problem is that software is constantly being probed for vulnerabilities by attackers because spectacular attacks can be carried out while incurring little cost; they require minimal skill and effort, and the risk of detection is quite slim. A bridge's vulnerabilities are not easy to exploit—it would be a hassle to try to blow up a bridge—and so the payoff for trying is not worth it to most.

Trying to measure security

Software's complexity makes it difficult not only to avoid making mistakes but also to devise foolproof ways to detect these mistakes: to measure how many of them, and how consequential, they are. However, computer scientists have devised a few methods for taking educated guesses at how secure a piece of software is.

Evaluating security can be attempted by looking at the design of the software and the implementation of that design. Experts can study the plan, or specification, for the software, to see if security features, such as encryption and access control, have been included in the software. They can also look at the software's threat model, its evaluation of what threats the software might face and how it will handle those threats.

On the implementation side, the experts could evaluate how closely the software matches its specification, which is a very difficult thing to do well. They can also look at the process by which the software developers created the software. Many computer scientists believe that the more closely a software development process mirrors the formal step-by-step exactitude of a civil engineering process, the more secure the code should be. So the experts can evaluate if the development team documented its source code so others can understand what the code does. They can see if a developer's code has been reviewed by the developer's peers. They can evaluate if the software has been developed according to a reasonable schedule and not rushed to market. They can look at how well the software was tested. The hope, then, is if the developers take measures like these, the code is likely to have fewer mistakes and thus will be more secure.

The experts might also test the software themselves. They might run programs to check the source code for types of vulnerabilities that we know occur frequently, such as buffer overflows. Or they might employ human testers to probe the code for vulnerabilities.

Finally, the experts could follow the software after it has been shipped, tracking how often patches to fix vulnerabilities are released, how severe a problem the patch fixes and how long it takes the company to release patches from when the vulnerability was found.

Creating a certifying lab: Requirements

For a lab to inspire confidence in its ratings, the lab must meet several requirements.

1. Reasonable tests for measuring security

The first requirement for an independent lab would be to decide on which of the above methods it would choose to evaluate security. The lab's method must produce a result that enough people believe has merit. Some security experts argue that the methods we currently use to evaluate software aren't good enough yet and so there *are* no reasonable tests for measuring security.[1] Talk of a lab, they say, is premature: Even if we can find some security vulnerabilities—which we can using these methods—there will always be another flaw that is overlooked. Therefore, any attempt to certify a piece of software as "secure" is essentially meaningless since such certification can't guarantee with any degree of certainty that the software cannot fall victim to devastating attack. Others argue that since we know how to find some security vulnerabilities, certifying software based on whether or not they have these findable flaws is better than doing nothing.[2]

Assuming for the moment that we agree with the computer scientists who believe doing something is better than doing nothing, which of these methods should the lab use? Ideally, if the only consideration were to do the best job possible in estimating whether the software is secure, the lab would look at all of them. But in the real world, the lab will face constraints such as the time and money it takes to evaluate the software, and access to proprietary information such as source code and development processes.

Our current best attempt to certify software security works by employing experts to look at documentation about the software to evaluate the design and implementation of the software. This process places special emphasis on looking at security features such as encryption, access control and authentication. Software companies submit their products to labs which use this scheme, called the Common Criteria, to evaluate their software. A good analogy for how this works is to consider a house with a door and a lock. The

[1] For example, Eugene Spafford, Professor of Computer Science, Purdue University; Jeannette Wing, Chair, Department of Computer Science, Carnegie Mellon University.

[2] Maccherone, Larry, Manager of Software Assurance Initiatives, Cylab, Carnegie Mellon University, Interview, November 11, 2004

Common Criteria try to examine how good that lock is: Is it the right lock? Is it installed properly? Critics of the Common Criteria say a major shortcoming of this method for evaluating software is that it relies too much on the documentation of the design and implementation and essentially ignores the source code itself which can be full of vulnerabilities. Figuring out if the lock on your door is a good one is hardly useful if the bad guys are poking holes through the walls of your house, which is what flawed code lets you do. Another problem with the Common Criteria is that it also looks at the software in a very limited environment—it evaluated Windows for use in a computer with no network connections and no remote media[3]—too limited to be meaningful in any general sense.

Another approach to devising methods to measure software security is under development at the Cylab at Carnegie Mellon University in partnership with leading IT companies. The Cylab is working on a plan to plug holes in the walls of the house by running automated checking tools—programs that run against a piece of software's source code much like spell-checkers—to catch three common security vulnerabilities, including buffer overflows. According to Larry Maccherone, Manager of Software Assurance Initiatives of Cylab, 85 percent of the vulnerabilities cataloged in the CERT database, a database run by CMU's Software Engineering Institute which is one of the more comprehensive collections of known security vulnerabilities, are of the kind that the Cylab's tools will test for. The flaw with this method for evaluating software is that it is only looking at a part of the problem: In this case, we are ignoring the locks on the doors.

[3] Spafford, Interview, December 2, 2004

2. A meaningful rating system

Once the lab decides on which methods it will use to evaluate the software's security, it needs to devise a way to convey information about its findings to consumers. Rating the software is obvious solution.

Any rating system for the software must say how secure the lab thinks the software might be, but at the same time, not give users a false sense of security, so to speak. The lab should make clear that the ratings are simply an attempt to measure security and can never guarantee the software is secure.

In addition, the descriptors the ratings system uses to describe how secure a piece of software is must make sense to the average consumer, and to those assigning liability, and at the same time map to metrics intelligible to computer scientists. A simple solution is to pass or fail the software, as Maccherone proposes. However, translating what passing means and offering the appropriate caveats as described above, is no mean feat. For the average user you might try to explain that that the tools the Cylab would use to scan the source code are intended to find vulnerabilities that make up 85 percent of the vulnerabilities present in the CERT database, and that different pieces of software contained certain numbers of these vulnerabilities. However, as McGraw points out, the average user, much like Gary Larson's cartoon dogs, would hear "Blah blah blah CERT database." Confused, the user might then ask: "Does this mean my computer is going to turn into a spam-sending zombie or not?" This is a perfectly fair, and unanswerable, question. Spafford adds that it would be difficult to present meaningful information even to sophisticated users like system administrators.

Hal Varian, Professor of Economics at Berkeley, suggests ratings might go beyond a general pass/fail system and note the software is “certified for home use” or “certified for business use.” This scheme too would confront the same problem: What does it mean for a piece of software to be certified for home or business use and how do you convey that meaning to consumers? Maccherone worries that it would be difficult to differentiate between certification for different types of use at the code level.

The Common Criteria, the primary certification system in use today, makes no attempt to make its ratings meaningful to the average consumer: The ratings are largely intended for use by government agencies. Vendors who want to sell to certain parts of the U.S. government must ensure their products are certified by the Common Criteria. Companies do tout their Common Criteria certification in marketing literature, but this likely means nothing to most consumers.

3. Educated consumers

Consumers then, would need to know about and value the ratings. Press attention, the endorsement of the ratings by leading industry figures and computer scientists, and more consumer education on why security is important would help here.

4. Critical mass of participating companies

A critical mass of companies would need to participate in the certification process for it to be useful. There are various ways to convince companies to participate. Industry leaders might take the initiative and participate in creating and deploying the ratings for the benefit of the industry as a whole, like the leading IT companies partnered in the

Cylab would do. According to Maccherone, industry giants like Microsoft have every interest in promoting industry-wide security evaluations. Big vendors want to see the security problem solved, and they would like to be involved in helping develop standards for measuring security rather than have security standards handed to them.

Alternatively, government policy might mandate that certain government agencies only use software that has been certified, which is how the Common Criteria certifying process works.

5. Costs in Time and Money

It must not be prohibitively expensive to submit your software for review so the smaller players in the industry can afford to be included. This is a major complaint about the Common Criteria labs: It’s too expensive, which bars all but the biggest companies from having their software evaluated.

As new releases in software come with some regularity, the review process must take place quickly enough for the software to have time on the market before its next version comes out. Critics fault the Common Criteria for being too slow as well.

The Cylab, because it relies on automated tools, might well delivered speedier and cheaper results than a process like the Common Criteria which depends on human experts. Then again, a major problem with automated tools is that each of their finds needs to be evaluated by a human since the number of false positives they uncover is huge. Spafford noted that in one test he performed, the tools uncovered hundreds of false positives in ten thousand lines of code. Windows is 60 million lines of code long. How much would handling these false positives increase the costs in time and money for the lab?

6. Accountability and independence

Finally, the lab must be accountable and independent. It needs to be held responsible for its scoring process, and it must be able to evaluate software without regard for whose software it is evaluating and who is funding the evaluation. Jonathon Shapiro, Professor of Computer Science at Johns Hopkins University, points out that the Common Criteria labs are not as independent and accountable as they could be: He says that companies are playing the labs off each other for favorable treatment. While the Cylab's reliance on tools might increase its ability to be independent—tools give impartial results—again, humans need to be involved to evaluate the tools' findings. Processes with human intervention must be structured so they maintain independence and accountability.

Is a lab worth doing now?

So would it be more useful to have a lab that can test for some flaws than it would be to do nothing? Would widely publicizing the Common Criteria ratings—assuming they are meaningful, which many experts doubt—or hyping the Cylab's certification when it goes live to consumers inspire companies to make certifiable software? It seems it would, at least as long as consumers perceived their computers to be "safer" than they were before buying certified software. However, if consumers began to feel their computer was no more secure for having bought certified software, the certification system would quickly fall apart. (How consumers might perceive this to be true is another question entirely—most consumers probably believe their computers are pretty secure right now anyway.) It is for this reason that we may just have one shot at making a workable lab. Perhaps then, we should wait until we can be reasonably confident that certification means a piece of software is really more secure than its uncertified rival.

Rather than encouraging the creation of a lab now, policy-making power then is probably best spent on funding security research so we can come up with metrics that would be meaningful enough for a lab to use. Spafford points out that research into security metrics, as well as into security more broadly, is woefully under-funded. He believes we need to educate those who hold the purse strings that security is about more than anti-virus software and patches. Security also about thinking of ways to make the software you are about to build secure, not just trying to clean up after faulty software. Funding for research into better tools, new languages, and new architectures is probably the best contribution policy-makers can make toward improving software security.

Katzke, Stuart W., 2004, Senior Research Scientist, National Bureau of Standards and Technology [Interview] November 12.

Maccherone, Larry, 2004, Manager of Software Assurance Initiatives, Cylab, Carnegie Mellon University [Interviews] November 11, and November 18.

Maurer, Stephen, 2004, Lecturer, Goldman School of Public Policy, University of California, Berkeley [Email exchanges] November 10-20.

McGraw, Gary, 2004, Chief Technology Officer, Cigital Corporation [Interview] November 30.

Razmov, Valentin, 2004, Graduate Student in Computer Science, University of Washington [Interview] November 4.

Shapiro, Jonathan, 2004, Assistant Professor of Computer Science, Johns Hopkins University [Interview] November 17.

Spafford, Eugene, 2004, Professor of Computer Science, Purdue University [Interview] December 2.

Varian, Hal, 2004, Professor of Economics, Business, Information Systems, University of California, Berkeley [Email exchange] November 15.

Wing, Jeannette, 2004, Chair, Department of Computer Science, Carnegie Mellon University [Interview] November 12.

Conclusion

The ultimate solution to making software more secure is not solely a technical one: new technologies, such as tools, and more skilled engineers are important, but insufficient by themselves. Even with great brain-power and expense focused on security, software is unlikely to be completely defect free. Other solutions, be they legal, economic or policy, are needed to push organizations and society toward devoting more resources to improving the security of our networked systems as a whole.

Policy-makers can use their law-making power and their power over purse-strings to encourage organizations to devote more resources to improving software security. They can threaten software vendors with a stick by enacting software vendor liability laws, but adding risk to the development of a cutting-edge technology can stifle innovation. They can encourage the market to push vendors toward making more secure software by, say, having the government refuse to purchase any software that has not been certified by an independent lab. Or they might extend carrots to help organizations improve the technology and technical skills behind their software: grants to universities for research into better tools and languages, or to start software engineering programs. Until consumers and producers deem security to be a feature, however, it will be no more than an extra expense in the development cycle.

There is no one easy answer to software security. Our paper has described several possible approaches for improving it and each has problems. Software is inherently error prone and costly to secure. Vendor liability might stifle innovation. Licensing too. And since we do not currently have the ability to create an independent lab that gave can meaningful ratings, it might be best not to try. We hope after reading this paper that policy-makers are now a bit better equipped to avoid poor decisions on this issue.