"Software Patents" — IT–Security at Stake?

Dipl.-Inform. Robert A. Gehring

(rag@cs.tu-berlin.de)

Technical University of Berlin

Department of Computers and Society

22 Feb 2002

Abstract

This paper [1] presents the thesis that insecurity of software is due to interaction of technological and legal shortcomings, fostered by economic rationality. Ineffective liability laws further the distribution of unreliable and insecure software. Copyright protection for software hinders the quality improvement. Patent protection encourages the use of proprietary instead of standard technology. Open source software development is proposed as a starting point for a risk management strategy to improve the situation. Existing patent laws should therefore be modified to include a "source code privilege". [Lutterbeck et al. 2000]

Introduction

There is an ongoing legislative process of enhancing intellectual property rights everywhere in the industrial world. This primarily refers to the developments in the legal field of copyright protection and in the field of patent protection.

New legislative measures are regularly complemented by technical measures developed by hardware and software producers to enhance the safety of the intellectual

[1] Full version of the contribution prepared for the international congress "Innovations for an e-Society. Challenges for Technology Assessment", October 17-19, 2001, Berlin, Germany. This full length version (v. 1.2, 22 Feb 2002) is available on the internet via http://ig.cs.tu-berlin.de/ap/rg/index.html. Minor changes were made (compared to v. 1.1) affecting spelling, layout and a missing reference (COM 2001).

property of their respective owners as well as to enable new business models such as “pay–per–use”. To the technical measures themselves is given legal protection by corresponding accommodated laws, often brought about by private law making procedures:

> *«[T]he tradition in copyright legislation involves getting a bunch of copyright lawyers to sit at a bargaining table and talk with one another [...]»*
> [Litman 2001, 31]

To point to just a few examples, one could name the Content Scrambling System (CSS), a technology to protect motion pictures on Digital Versatile Disc (DVD), and the U.S. Digital Millennium Copyright Act (DMCA), the corresponding law to defend content protection technology against being circumvented.[2] In Europe, the Commission of the European Community earlier this year issued the Directive 2000/29/EC. [COM 2001] This directive enhances the copyright protection for the information society. It contains the outlines for national legislators to create the European counterpart to the DMCA. The roots of this international development in legislation lie in the World Intellectual Property Organization (WIPO) treaties of 1996, the WIPO Copyright Treaty (WCT) and the WIPO Performances and Phonograms Treaty (WPPT).

At the same time the public debate about the consequences of such legislation intensifies. Fears are expressed that the new legal restrictions will impede the innovation process, stifle the freedom of speech and science and last but not least put a heavy burden on the enterprises which will have to pay high license fees to gain access to state–of–the–art technology.

The problem of how the security of information technology in general is affected by laws and technical measures is somewhat out of focus, at least when considering the political and legal process that lead to the new legislation. If for example we take the above mentioned WIPO treaties, we must state that the signing nations agreed upon those treaties without comprehensive consideration of IT security requirements.

Lobbyists of the right holders put heavy pressure on the politicians to enact laws to protect their commercial interests. As the topic of security is approached, the right holders worry about the safety of their respective intellectual property assets but actually don’t care much about the security of the underlying information infrastructure.[3]

[2] Latest efforts by the content industry resulted in the proposal for a new law titled "The Security Systems Standards and Certifi cation Act". [Yoshida/Leopold 2001]

[3] See, e.g., the comment of the Publishing Company Reed Elsevier, Inc., in response to 65 FR

This attitude poses a critical problem for security in a networked world: Laws that protect insecure software are going to protect network insecurity at the same time. The more the institutions of society get electronically networked the more they get at risk —with unbalanced laws.

Favored by such laws, we find ourselves confronted with the uncomfortable situation that we have to use unchecked software to be able to access certain kind of information at all. If the software contains security weaknesses, we don't have the right to learn about them.

And to make things worse there are no effective liability rules in the field of software as they are in other fields of technology. Existing liability legislation doesn't fit the peculiarities of a technology that—according to the current scientific knowledge—is inherently faulty but cannot be given up.

Consequently, there is no precedent worldwide where a distributor of insecure mass market software had to pay for the damages the users suffered from the insecurities of the product (e.g. for data loss) when they were hit by a computer virus.

This paper will describe the technical, economical and legal background for the problematic development and make a proposal of how to possibly escape the unsatisfying situation.

Highlights from the real life of unreliable and insecure software

In 1999, the so called «I love you» virus caused worldwide damages of about 12 billion dollars. That was more than twice the damage that was counted in 1998 due to all viruses together. In 2000, the same virus caused damages of not less than 6.7 billion dollars within 2 weeks. [McAfee 2001]

There are yet no comparable figures available for damages caused by viruses in 2001. But the reports from all over the world about the effects of the latest viruses—«Code Red II» or «SirCam» for example—make it likely that the before mentioned damages of earlier years are excelled.

35673 (5/9/00): *«Certainly, the DMCA has helped to make computer networks safer—but by no means risk-free—places to distribute copyrighted works.»* 65 FR 35673 asked for public comments on the intended agreement on the Uniform Computer Information Transaction Act (UCITA) in the U.S. [Reed Elsevier 2000]

And viruses are only a part of the problem. Everyday we read in the news about the latest break-in into a server of a well or lesser known provider. Time and again we hear of stolen credit card data, defaced websites and deleted files. We've been accustomed to such messages because they are ever present.

The burden of dealing with unreliable software sums up to estimated US$78 billions a year in the U.S. industry alone. [Levinson 2001]

As the following analyses will show, the insecurity of software is not due to conspiracy of fate but rather due to interaction of technological and legal shortcomings, fostered by economic rationality.

Technical reasons for insecure software

In short the classical software development process and its limitations should be described here to provide a basic understanding of where the technical problems are located.

Incomplete specifications

A software product development starts with a more or less detailed description of the tasks the program has to do. Such a description is derived from a systems analysis of the environment where the software is to be deployed —its market. The so called functional specification derived from the analysis contains instructions about what code the programmers will have to write, in which programming languages, how the different parts are designed, how they interact and what external components are to be included. The functional specification presents the building plan for the software.

In addition, the input/output–relation of every functional part of the program should be described in order to derive sufficient test instructions thereof. Albeit in practice it is not unusual to start coding with incomplete functional specifications. This is due to changing functional requirements that occur during the ongoing coding and testing process.

Measured by real world conditions, a functional specification is sometimes incomplete with respect to functional requirements. And it is nearly always incomplete when it comes to security considerations:

«[M]odern systems have so many components and connections—some of

> *them not even known by the systems' designers, implementers, or users—that insecurities always remain.»* [Schneier 2000, xii]

Incomplete testing procedures

All tasks and situations the system analysts and designers are able to anticipate on principle can be included in the functional specification. A software that is written in perfect compliance with its functional specification and has passed all the necessary tests is called correct. It is called correct, but nothing more. No serious programmer would claim that a certain piece of software is secure because it is tested to be correct. As IT security expert Donald Pipkin puts it:

> *«The developers are so in tune with what it should do, they cannot see what it might be able to do.»* [Pipkin 2000, 75]

For the system designers it is almost impossible to take into account all circumstances that a software product may be confronted with in its concrete place of application. That is the more true when we talk about mass market software that is used on standard PCs altogether with many other software products and hardware configurations.

Beside the unavoidable blind spots in the functional specification, there is another problem in the process of software writing: software is designed and written by humans. And as with every writing human beings are involved in, mistakes are made and need to be found and fixed. The mistakes can be made in the functional specification - as a wrong modelling of the requirements. Mistakes can be made in the program code, in the structure of the program or they can be made in the installation procedure, or ... There are (too) many ways of making mistakes. It is a limitation in the development process owed to the imperfect human mind.

It is true, a lot of mistakes are detected during the testing procedures. But not all errors can be found through testing procedures. It is impossible to foresee all input variants to a complex program and to check the according reaction of the software. Thus—in a faulty reaction—the testing procedure can only show the presence of errors in the program. But it cannot show the absence of errors. [Floyd 1997, 664][Kaner 1997a]

Errors as a property of human-computer interaction

If we would go further and include the user's expectations in the definitions of error and failure, as it was sometimes done in the earlier days of software development,

the problem gets even worse. Following the definitions of Myers [Myers 1976, 6], whereby *«A software error is present when the software does not do what the user reasonably expects it to do.»* and *«A software failure is an occurrence of a software error.»*, the software developers would have to speculate about the limits of rationality of their users in order to derive the program specification thereof. In mass market software production where developers and users don't know each other this seams an almost hopeless undertaking. And so concludes Myers (ibid):

> *«The reader should now be able to grasp an elusive characteristic of software reliability: software errors are not an inherent property of software. That is, no matter how long we stare at (or test, or "prove") a program (or a program and its specifications), we can never find all of its errors. We may find a few errors, such as an endless loop, but, because of the basic nature of software errors, we can never expect to find them all. In short, the presence of an error is a function of both the software and the expectations of its users.»*

These all are unavoidable technical limitations of the testing process.

Economic reasons for insecure software

Proprietary software is a product for a market. A product is only of value to its seller when a buyer can be found. No one will buy a product that is too expensive and therefore software prices are limited by market demand.

Avoiding costs

But the development of a software product, its testing and debugging is expensive. Today, about 40-50% of the development costs of a software product are caused by testing and debugging. [Sommerville 2001, 24] And testing and debugging costs grow faster than linear if more test conditions and environments are checked.

If the expected costs of liability for an unsafe product in sum are lower than the expected costs of a more complete testing and debugging process, to deliver an unsafe product will be preferred by the software producer.

> *«[T]esting is largely a problem in economics.»* [Myers 1976, 176]

Ineffective (legal) sanctions [4] for the distribution of insecure software make it a real bargain for software producers to ship their products in an unreliable state.

> *«That's the way vendors make money. They push products on the market before they've been adequately tested, demand payment up front and then are often not available to deal with the sequelae of poorly performing products.»* [Levinson 2001]

Why should software producers leave out this opportunity?

The bad influence of network externalities

Bexond that, the so called network externalities [Shapiro/Varian 1999] have the greatest influence on the behavior of software producers. A software distributor has to bear in mind that software as a good applied in virtual networking environments is affected by positive feedback effects: The more customers use the same software product the greater is the value of the software to the individual user. The more users deploy the same software, the more communication partners are there to share files and data, to communicate with each other. This drives the potential users of software to buy the product they believe to become the dominating one and to keep its position. Thereby they get the most value for their money.

Positive feedback works to the advantage of the largest supplier. Regarding the market, positive feedback cycles often end in a "winner–take–all" [market] situation. [Shapiro/Varian 1999] Incentives to be the first on the market and to establish one's own products as de facto standards are very high. If lowering security precautions gives an advantage, dominant market players will do so.[5]

Controlling standards

In network economics, controlling standards is of greater importance to a commercial success than to deliver a better product. A short time-to-market to exploit the

[4] Recognizing this problem, some customers are beginning to withhold payments for the delivered software until the vendor establishes a reliable state. But new legislation in software contract law (UCITA) may soon block this way. [Levinson 2001] Beside that, such measures are only feasible for heavy-weight customers.

[5] Some IT specialists identify this combination of market dominance and lack of security awareness as the paramount problem. Cf., for example: **Microsoft: A Proven Danger to National Security: CONCLUSIONS OF REALITY.** [Forno 2000]

first-mover advantage is of significant importance to establish a software product in the market and to profit from the network effects. More sophisticated testing and debugging procedures would prolong the time of introduction of a new product (to the market) and thus decrease the probability of commercial success. Faster and often less sophisticated testing procedures allow for a shorter time-to-market, thus leading to a competitive advantage.

As mentioned before, the software producers try to avoid testing and debugging time as far as possible to raise their profits. No complete testing and debugging process for a complex software would ever be profitable, since the price for the product would be prohibitively expensive. [Myers 1976, 176] And because of the lack of effective legal liability rules, the incentives to avoid costs for testing and debugging are high. Often the users are involuntarily "employed" as software testers to find out and report about problems. They, the users, have to invest time and money to find out about errors. [Pipkin 2000, 75]

Limited supply of service

By paying a service fee, a customer perhaps can buy more attention and service of the supplier. But service as a marketable good obeys the rules of the market, too. Since the service capacity of a mass market software producer is limited regarding the millions of customers, the amount of available service is limited. And scarce goods are pricy. Only a minority of customers (i.e. mass market software users) will be in a position to buy the necessary service. But in a networked world, the overall systems security cannot significantly be increased, when more secure systems are available to—and installed by—only a minority, and are interconnected with insecure systems.

Raising demand for service

Looking at the other end of the software market, to the vendors of tailor-made software, the findings are by no means better. Their business model includes profits from selling service and support. Delivering perfect products that could be used for many years without service would threaten their business.

> *«So it's in the manufacturer's best interest, at least financially, to make products that need maintenance and that have to be continually improved with successive updates, patches and versions [...]»* [Levinson 2001]

The price of convenience

From the analysis of the daily break-ins we learn that most often errors in the software or mistakes in the operation or configuration of the software lead to vulnerability and insecurity. Regularly, complex software is delivered in an insecure state. And even if it would be possible to configure the software in a securer way it is often avoided to do so "for the convenience of the customers"[6]. By lowering the necessary effort of a customer to use a software product, a producer creates a competitive advantage for his own product over a potentially more secure alternative that requires more struggling. Thus, insecure software knowingly is distributed in millions of copies and applied within network environments as a means of competition. It supplies an ideal basis to hackers for successful attacks of all kind.

A market with asymmetric information

Asymmetric information between producers and buyers within the software market makes another contribution to the problem of flawed and insecure software.[7]

Software is a so called experience good. Potential buyers don't have reliable information about whether the software they intend to buy meets their specific needs and is of appropriate quality. They first have to buy the product and can rate its quality only after having deployed it. At this time, the software producer already has made his profit. It cannot be in his business interests to provide information about weaknesses of his product to potential customers before they buy it.

Additionally, the reverse engineering of software with the aim of quality evaluation usually is declared an unlawful activity by copyright laws (see next section). Because no one is allowed to gather quality information about a certain software product by reverse engineering the code, there cannot be a provable serious source of quality information to serve as a basis for rational consumer choice. A competition for quality is disabled as long as reverse engineering software distributed in binary form is deemed unlawful. It would be irrational behavior of customers to prefer to buy more expensive software because its vendor's claims of higher quality since they cannot be verified.

[6] Cf. for example: [Akerlof 1970].

[7] Markets with asymmetric information leadig to adverse selection are very common as we know today. *George A. Akerlof*, who first worked on this topic [Akerlof 1970], together with *A. Michael Spence* and *Joseph E. Stiglitz* received this years' Nobel Prize for economic sciences for their contributions to the problem of asymmetric information markets.

The claims of quality and security may be right, but the customer cannot judge them. According to economic models of customer behavior, he should and will decide to buy the cheaper product that may or may not be of less quality.

Dependent certification is no solution

Certification of products, often proposed as a solution to this dilemma, won't work if neither the producer of the software nor the certification authority will have to bear the costs of ill–certified software. Instead, as security expert Ross Anderson explained [Anderson 2001], the certification process probably will be—and in reality quite often is—adapted to the needs of the software producer. Certificates as marketable goods of the "evaluation market" follow market rules. Certification authorities will offer what their customers, the software producers, are willing to buy.

To be certified, a given software product must fulfill the criteria for the certification. It is reasonable to expect that the scrutiny of the certification process will at a certain level meet the demands of the software producer. And the demands of the software producer will be derived from its willingness to remove detected errors and security flaws. If the certification authority won't certify a flawed product, the software producer will look after another certification authority. In this case, the certification authority couldn't sell its "product", the certificate. In order to make a deal it has to lower its requirements. One can very well imagine the race for the weakest certification process that gives both, software producer and certification authority, the best buy.

That is the simple economic rationale behind commercial software development and its built-in preference for insecure software: It is a perfectly rational behavior for a commercial software producer to distribute unsafe products as long as it is to his advantage.

Legal obstacles to better software quality and security

The lack of effective liability laws that motivates the production and distribution of unreliable and insecure software products was mentioned before. It is a result of the political process that brought legal protection for software along.

From a legal point of view, software is treated as some kind of literary information, a kind of written speech. [Raskind 1998] That place was chosen following consid-

erations of the linguistic nature of the source code of programs. And in general you cannot be held liable for distribution of literary information that contains errors. That rule applies not only to books but to software too. [Kaner 1996] There is no special liability law to be applied in cases involving mass market software. That is why common liability laws should be applied.

The classical approach to liability is ineffective for software

Common liability laws follow a "fault–based" approach [Kaner 1996]: You can be held liable if you deliver a defective product but only if (a) you have not obeyed your professional obligations or (b) you have bound yourself by contract to deliver a product in a certain state of usability. Option (a) is hardly a way to liability because of the shared view among experts that software in general cannot be written error–free. The exception proves the rule. Option (b) is usually avoided by software producers. Whatever mass market software license is considered, one will come to realize that there are no promises of usability. Sophisticated software contract law like the UCITA in the U.S. even grants the software producers liability restrictions if they knowingly distribute defective products. [IEEE 2000a]

No *sui generis* law

Instead of the development of a sui generis law for software that would have equilibrated the interests of software producers and software usersand of the public of courseexisting laws have been extended in order to cover the demands of software producers solely. The regularly raised demands of legal and technical experts to respect the specifics of the software development process and its challenges have just as regularly been rejected in the name of the software producers.

Worldwide, the predominating legal protection for software is to be found in copyright laws. In contrary to other linguistic works, e.g. books, software is intended to be used as part of a machine. Actually it can be considered a machine itself. [Samuelson et al. 1994] Software contains instructions that are meant to be executed on a computer and thereby builds a machine that does something more or less useful. Classical copyright law was neither intended to protect the behavior of a machine nor to prevent malfunction of a machine.

To protect software as (part of) a machine, special rules have been included in

copyright laws. Some of these rules oblige the user not to do certain things with the software he bought. One rule is of paramount importance to the quality and security of software, better to the lack thereof: the prohibition of reverse engineering.

Repair is unlawful

Copyright protection for software contains a broad ban—with only a few exceptions—on reverse engineering. Reverse engineering made illegal means that it is unlawful to reconstruct a human-readable form of software from the binary code form in which it is distributed, even for most honest purposes.[8] Usually, there is an exception for enabling interoperability and, perhaps, for the remove of errors that disable the intended deployment of the product; but sometimes not even this.

There is no exception for security inspection and/or enhancement. No one is legally in the position to inspect a software product delivered in binary form for malicious code—such as a Trojan horse, e.g.—be it placed intentionally or by accident. Such malicious code may endanger the security of the user's system. But the laws don't give him a chance to know about it and to prevent it.

There is no exception for to remove minor errors that may interfere with other software products or may, in the future, present a security hole. In the case of an accident with the software, the customer is doomed to wait for the help of the producer that may come or may not. Self–help of software users regularly is declared an illegal activity.

Many liability laws require the victim to prove that the cause of a suffered damage lies in the software product and could have been avoided by the producer through better testing and debugging. Such a proof is not to be established without extensive reverse engineering.

The prohibition of reverse engineering additionally furthers the above mentioned lack of market transparency and thus hinders the development of a market for secure software products. [IEEE 2000b]

Patent protection encourages binary distribution

The second important legal hindrance to the enhancement of quality and security of software products consists in the increasing patent protection for concepts imple-

[8] For the consequences see, e.g. [Kaner 1998].

mented in software.

While copyright protection makes it illegal to reverse engineer the product, it cannot prevent the ideas underlying a certain software product from being taken and reimplemented. Copyright protection only outlaws literary copying, not reuse of ideas. For a competitor it is therefore possible to offer a compatible product written on his own and with enhanced security features. Potential customers should have a choice regarding security needs between two competing but compatible products. The compatibility aspect is of high importance because of the earlier mentioned network externalities.

But copyright protection is not the only protection method for intellectual property in software. Software can in part be protected by patenting parts of its underlying technology.[9]

Patent laws give the patent holders the exclusive rights to the patented technology—in every possible implementation—and do not allow the offering of compatible technology without a license. Patent protection often bars competitors from the market if core parts of a standard technology are protected by patents. Since network externalities have great influence, the incentives are high, not to license technology to competitors. The hope is to let the positive feedback effects work to one's own advantage and thereby become dominating on the market.

Patent protection weakens security

Patent protection for software has at least three important implications for IT security.

> (1) Because of the absolute legal protection that patent law provides, compatible technology from competitors may be blocked. A faulty implementation of a certain technology may well become the single one solution available on the market.
>
> (2) Secure technology itself may be patented. In such cases, no software producer is allowed to include functional equivalent technology within his products without license. Thereby the fast spread of secure technology can be hindered.
>
> (3) Business models with a core idea of securing systems may be patented.

[9] Patent laws all over the world differ with regard to the preconditions for patentibility. Common is a requirement of novelty to constitute an invention. Alone, there is no common understanding of novelty.

> Actually, it already happened.[10] In effect, one has to acquire a license in order to make systems safe or to fix security flaws in a certain way —regardless of a possible emergency. Broad claims of granted patents, being common today, spoil the chances of wide deployment of latest security measures. The patent laws don't know restrictions for security requirements. Rarely applied, compulsory licensing rules provided hitherto no solution.

We can conclude that the more software technology is protected by patents, the higher is the probability for certain faulty software products to become very common. Possibly, they become the dominating ones in their respective category.

To make things worse, copyright protection and patent protection can be—and in practice are—combined. To provide a "family" of compatible software products, partially protected by patent law, complementary protected by copyright law, and to keep the software incompatible with competitor's products by preferred usage of proprietary, legally protected standards gives a software producer a unique competitive advantage. And that easily leads to a dominant or even a monopoly position.

Proprietary instead of open standards

To reach and to keep his unique position, the software producer rejects to use standard technologies that are tested for security. He does so because he wants to exclude potential competitors. Instead, proprietary solutions are preferred, even if they are known error-prone and insecure. We know many examples from the latest past: SDMI, CSS, Adobe's eBook encryption, Microsoft's proprietary modifications of the kerberos authentication protocol and so forth.

Until today, there is no legislative answer to all the presented questions. We will have to look elsewhere to overcome the problem.

[10] See, e.g. Finjan Software, Inc. (San Jose, CA), U.S. Pat. 6,167,520 (26 Dec 2000): System and method for protecting a client during runtime from hostile downloadables; McAfee.com Corporation (Santa Clara, CA), U.S. Pat. 6,266,774 (24 Jul 2001): Method and system for securing, managing or optimizing a personal computer.

How to improve software quality and system security?

Given the important inherent limitations of the software development process as stated above, it would be unreasonable to expect software to be error-free and perfectly secure. Complex software is faulty, that is the opinion of all experts in the field. [Floyd 1997, 644][Kaner 1997a] The computer science does not know of any systematical and feasible method to develop complex software in a way that it is written provably error-free and secure from the beginning on. That is the simple truth.

The need for a risk management strategy

In view of the error–prone development process which leads to faulty software, which in turn leads to insecure systems, it is time to ask for an adequate risk management strategy to serve the public interest in system security.

Such a risk management strategy has to be constructed in a manner that it will reduce the risks coupled with the use of software in the long term. That means to reduce the number of errors in the code, to reduce the scale of security weaknesses and to limit the harmful consequences of expected security breaches. While the first two requirements refer primarily to the enhancement of the software development process, the third point refers to the way security breaches are handled by and for the user.

Standard components and peer review

The best method we know so far to enhance the quality of software is the deployment of well tested standard components combined with a process of peer review by experts. The creation of more secure software requires the step–wise improvement of the software in order to fix detected weaknesses. [Sommerville 2001, 566] Software has to “mature” in order to become (more) reliable.

Peer review plays a crucial role in this quality management process. Within the peer review process, a number of qualified persons scrutinizes the written code, the documentation, the compliance with the functional specification and, where affordable, the whole system within its working environment.

The last demand is the hardest one to fulfill in the field of standard mass market software. No single software producer is able to observe all the circumstances his product is deployed under. Therefore, commercial software producers wait for the

reactions of their customers confronted with a error-prone product. And sometimes the software producer publishes so called service packs containing some corrections for the software. The delay between the detection of an error or security weakness and the availability of a service pack depends solely on the suppliers subjective estimation of its relevance. From a security point of view, the delay should be kept as short as possible.

We need a better peer review process than what a single software producer is able to provide. It must be scalable to the magnitude of the ever more ubiquitous internet. It has to have short reaction time cycles, and it must be transparent. The latter is of paramount importance for the security evaluation of the software. And more security is what is badly needed.

Security as a process

Security needs to be thought of as a "process". *«And if we're ever going to make our digital systems secure, we're going to have to start building processes.»* [Schneier 2000, xii] [11] And that process-building must be kept alive over the time a software product stays in use. The security process must again and again be adapted to reflect experiences and changing environmental conditions. [Pipkin 2000, 17f]

Environmental conditions are nowhere the same. They differ from system to system and, to put it blundly: There cannot be a "one–fits–all" approach. Security is to be tailor–made to reach the required level of efficacy. This is a critical point within a networked world. A network is only as secure as it's least secure part. The "weakest link in the chain" will determine the security of the network, not the strongest one.

Open source development as a starting point

To the current knowledge, there is only one peer review process that fulfills the mentioned quality requirements and at the same time supplies the basis of a security process. This is the open source software development model.[12]

[11] Similarly [Pipkin 2000, xx].

[12] I will use the term open source in a generic manner and not verbally differentiate between *free software* and *open source software*. The interest is directed to the technical and legal possibilities and from this point of view both, free and open source software, provide similar features. For further information on the difference between both types of software refer to: `http://www.fsf.org` and `http://www.opensource.org`.

The open source software development process is based on the unrestricted availability of the software's source code under a number of different copyright licenses. The source code is distributed on the internet and on other media. Software developers all over the world—scientists, employed professionals as well as ambitious amateurs—take this code basis, test, enhance and secure it. If no code can be found that fits one's needs, someone usually starts writing code from scratch.[13]

Since the source code of the program is publicly available, there is no need for reverse engineering to be able to understand how the program works. This is not the solution but the decisive prerequisite a number of security experts demand in order to make secure systems available. [Schneier 2000, 343f][Pfitzmann et al. 2000]

Fast reaction in case of a security incident

Above all, the available source code enables users at work and at home to fix security weaknesses as soon as they are detected. Users of open source software automatically obtain a license to modify the software according to their needs. Thus they are able to carry out modifications in the software strengthening the security of the system. In the case they lack the necessary qualification, they are able to entrust someone else to do so.[14]

Last but not least, open source software promotes the use and spreading of open standardsas well as the reuse of well tested componentsas it is propagated by software experts. [Gamma et al. 1995, 1] Both properties are suitable to foster the creation of less vulnerable and more reliable software. In fact, statistics show that open source software is less prone to attacks and outages than proprietary software.[15]

A competition for quality

There are more technical advantages of software that is available in source code. But what about the economics of security? How are the earlier recognized economic short-

[13]For further information on the open source development process see, e.g.: [Raymond 1999] and [DiBona et al. 1999].

[14]In recognition of this huge advantage of open source software, the U.S. National Security Agency chose an open source operating system as a basis for the creation of a secure operating system. [NSA 2001]

[15]On the internet, a variety of mailing lists deals with matters of security and insecurity of software. ISN and Bugtraq are just two of the more prominents. Consistently, those lists give evidence for the thesis that open source software is proportionally less vulnerable than proprietory software.

comings of proprietary, closed source software resolved with open source software?

Firstly, the availability and use of open standards removes the bias in favor of a dominant market player and its proprietary technology. Other competitors with possibly more reliable products get a chance to gain market shares without the artificial compatibility problems. The unwelcome results of network externalities can be minimized.

Secondly, the market transparency grows. Potential customers see themselves in a position to be able to collect neutral information on product quality. That information can be based on the quality of the code itself instead of mere marketing statements of the software vendor. A producer-independent certification process could be established and fulfill the customers demands for information.

Thirdly, the huge amount of open source code that is available without royalty fees, allows the development of security software and services at low costs. Thereby, the market supply could be better adjusted to the demand of the market instead of being derived from a vendor's production capacity or market strategy.

Fourthly, confronted with competing offers of quality open source software, vendors of proprietary software may well be forced to undertake efforts to increase the quality of their products. Otherwise, they may lose the competition in the long run.

Evidence of "quality is possible" before the court

Beside the technical and economical advantages, open source software could play a role in the improvement of the legal treatment of software liability questions.

In form of publicly available reference implementations of technology, a quality standard may be established. Before a court, such a quality standard could serve as a means of prima facie evidence that a certain level of software quality and security actually can be achieved. Based on such evidence, liability claims could be made legally enforceable and thereby put firm pressure on software producers to dedicate more resources to quality assurance and service. The liability gap with regard to software could be overcome.

Given all these insights, open source software shows reasonable qualities to be preferred as a development and distribution model in comparison to proprietary ones.

But this model is in danger. The gold–rush in software patenting we could see grow over the last years may well stall what looks yet so promising. There are serious problems connected to the patent protection for software.

The patent threat

Without the appropriate license, patent protection prevents the use of patented technology within open source software. Due to the decentralized development model of open source software, there is no single authority as a contractee in license negotiations. But without a valid license, patent holders can demand the removal of open source software from the internet.

Moreover, open source software developers are particularly vulnerable to patent litigation because the code of the programs is open to everyone to inspect for patent infringement. On the contrary, the vendors of proprietary software, which distribute their products in binary form, are protected against such search by the ban of reverse engineering in copyright laws. Put simply, patent law favors those who (try to) hide patent infringing code through binary distribution.

And the uninhibited patenting of nearly every trivial idea if only expressed in patent lawyers terms and implemented in software—as favored today by patent offices and courts all over the world—has led to the much criticized situation that complex software normally should not be written any longer without extensive patent search. But patent search is only meaningful if conducted with detailed knowledge of the system of patent databases, of the classification systems and how to read patent files. One needs to have assistance of an experienced patent lawyer to be on the safe side. In short, it is a job for a patent specialist, not for the average programmers.

Large software producers do have a patent department at their hands. The average open source programmer doesn't have such support. Thus, he is not in the position to avoid unintentionally writing code that possibly may infringe someone else's patent claims. Beyond that, he is financially not in the position to defend against asserted patent litigation, even if the complaints are unreasonable.

Small and medium enterprises which develop and distribute open source software are easily being driven out of the market by larger competitors that can afford a patent litigation suit.

To summarize: The open source software development process encourages the use of the best software engineering principles we know today to achieve high quality and security of software and computer systems. This is increasingly acknowledged within the scientific community and the business world.

However, software patents present a real threat for the open source software development and distribution model. Unrestricted possibilities of enforcing patent claims

on software against open source developers and/or distributors would mean to put IT security at stake.

Prospect and recommendation

Perhaps, the proposal of a “source code privilege” to be included in patent laws (as presented last year in the short expertise on “Security in Information Technology and Patent Protection for Software Products: A Contradiction?”, Commissioned by the Federal Ministry of Economics and Technology), may show a way to a solution. [Lutterbeck et al. 2000]

The core proposal suggests (Recommendation PP-1):

> *«The use of the source codes of computer programs must be granted privileged status under patent law. The creation, offering, marketing, possession, or introduction of the source code of a computer program in its various forms must be exempted from patent protection (source code privilege).»*

We should not allow the open source software development model to be destroyed in the name of the business interests of patent holders. We should pay the required attention to the security needs of today’s information infrastructures.

References

[Akerlof 1970] Akerlof, George A.: **The Markets for "Lemons" - Quality Uncertainty and the Market Mechanism**, in: Quarterly Journal of Economics, 1970, Vol. 84, No. 3, pp 488-500.

[COM 2001] **Directive 2001/29/EC of the European Parliament and of the Council of 22 May 2001 on the harmonisation of certain aspects of copyright and related rights in the information society**, Offi cial Journal L 167, 22/06/2001 P. 0010 - 0019, on the internet: `http://europa.eu.int/eur-lex/en/lif/dat/2001/en_301L0029.html` [22 Feb 2002].

[Anderson 2001] Anderson, Ross: **Why Information Security is Hard - An Economic Perspective**, 2001, on the internet: `http://www.cl.cam.ac.uk/ftp/users/rja14/econ.pdf` [28 Aug 2001].

[DiBona et al. 1999] Dibona, Chris; Ockman, Sam; Stone, Mark: **Open Sources. Voices from the Open Source Revolution**, O'Reilly, Sebastopol, CA, 1999.

[Floyd 1997] Floyd, Christiane: **Softwaretechnik** [Software engineering]**, 14.2.1 Eigenschaften von Software** [Properties of software], in: P. Rechenberg, G. Pomberger (eds.): Informatik-Handbuch, Carl Hanser Verlag, München, Wien, 1997.

[Forno 2000] Forno, Richard: **Microsoft: A Proven Danger to National Security: Conclusions of Reality**, Essay #2000-4, 15 May 2000, on the internet via: `http://www.info-sec.com/internet/00/MSFOR_natsec.pdf` [17 Oct 2001].

[Gamma et al. 1995] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: **Design Patterns. Elements of Reusable Object–Oriented Software**, Addison-Wesley, Reading, MA, 1995.

[IEEE 2000a] IEEE U.S.A.: **Opposing Adoption of the Uniform Computer Information Transactions Act (UCITA) By the States**, Approved By the IEEE-USA Board of Directors, Feb. 2000(a), on the internet: `http://www.ieeeusa.org/forum/positions/ucita.html` [28 Aug 2001].

[IEEE 2000b] IEEE U.S.A.: **Recommended Amendments to Virginia Uniform Computer Information Transactions Act**, (Title 59.1, Chap. 43, § 59.1-501.1 et. seq.), 17 Oct 2000(b), on the internet: `http://www.ieeeusa.org/forum/POLICY/2000/00oct17.html` [28 Aug 2001].

[Kaner 1996] Kaner, Cem: **Liability for Defective Content**, in: Software QA, 1996, Vol. 3, No. 3, p. 56, on the internet: http://www.badsoftware.com/badcont.htm [28 Aug 2001].

[Kaner 1997a] Kaner, Cem: **The Impossibility of Complete Testing**, in SOFTWARE QA, Volume 4, #4, p. 28, 1997(a), on the internet: http://www.kaner.com/articles.html [28 Aug 2001].

[Kaner 1997b] Kaner, Cem: **Software Liability**, 1997(b), on the internet: http://www.kaner.com/articles.html [28 Aug 2001].

[Kaner 1998] Kaner, Cem: **The Problem of Reverse Engineering**, in SOFTWARE QA, Vol. 5, #5, 1998, on the internet: http://www.kaner.com/articles.html [28 Aug 2001].

[Levinson 2001] Levinson, Meredith: **Let's Stop Wasting $78 Billion a Year**, in: CIO Magazine, October 15, 2001, on the internet: http://www.cio.com/archive/101501/wasting_content.html?printversion=yes [16 Oct 2001].

[Litman 2001] Litman, Jessica: **Digital Copyright**, Prometheus Books, Amherst, NY, 2001.

[Lutterbeck et al. 2000] Lutterbeck, Bernd; Horns, Axel H.; Gehring, Robert A.: **Sicherheit in der Informationstechnologie und Patentschutz fuer Softwareprodukte - ein Widerspruch?** [Security in Information Technology and Patent Protection for Software Products: A Contradiction?, Short Expertise Commissioned by the Federal Ministry of Economics and Technology], Berlin, December 2000, on the internet: http://www.sicherheit-im-internet.de/download/Kurzgutachten-Software-patente.pdf [28 Aug 2001].

[McAfee 2001] McAfee: **McAfee Lösungen**, 2001, on the internet: http://www.mcafeeb2b.com/international/germany/products/mcafee-solutions.asp [23.08.2001].

[Myers 1976] Myers, Glenford J.: **Software Reliabilty. Principles and Practices**, John Wiley & Sons, New York, NY, 1976.

[NSA 2001] National Security Agency (NSA): **Security-Enhanced Linux**, on the internet: http://www.nsa.gov/selinux/ [28 Aug 2001].

[Pfitzmann et al. 2000] Pfi tzmann, Andreas; Köhntopp, Kristian; Köhntopp, Marit: **Sicherheit durch Open Source? Chancen und Grenzen [Security by Open Source? Opportunities and Limitations]**, in: Datenschutz und Datensicherheit, 9/2000, pp 508-513.

[Pipkin 2000] Pipkin, Donald L.: **Information Security**, Prentice Hall PTR, Upper Saddle River, NJ, 2000.

[Raskind 1998] Raskind, Leo J.: **Copyright**, in: The New Palgrave Dictionary of Economics and The Law, Vol. 3, p. 478ff, Macmillan Reference Ltd., London, 1998.

[Raymond 1999] Raymond, Eric S.: **The Cathedral & The Bazaar. Musings on Linux and Open Source by an Accidental Revolutionary**, O'Reilly, Sebastopol, CA, 1999.

[Reed Elsevier 2000] Reed Elsevier, Inc.: **Response to 65 FR 35673**, on the internet: `http://www.loc.gov/copyright/reports/studies/dmca/reply/Reply005.pdf` [29 Aug 2001].

[Samuelson et al. 1994] Samuelson, Pamela; Davis, Randall; Kapor, Mitchell D.; Reichman, J.H.: **A Manifesto Concerning the Legal Protection of Computer Programs.** Symposium: Toward a third intellectual property paradigm, in: Columbia Law Review 94 (1994) no 8, p. 2308ff.

[Schmidt 2001] Schmidt, Jürgen: Sicherheitsrisiko **Microsoft. Die Kehrseite des Windows-Komforts [Security Risk Microsoft. The other side of Windows's comfort]**, pp 140-142, in: c't Magazin 21/2001.

[Schneier 2000] Schneier, Bruce: **Secrets and Lies. Digital security in a networked world**, John Wiley & Sons, Inc., New York, 2000.

[Shapiro/Varian 1999] Shapiro, Carl; Varian, Hal R.: **Information rules. A strategic guide to the network economy**, Harvard Business School Press, Boston, MA, 1999.

[Sommerville 2001] Sommerville, Ian: **Software Engineering**, 6th edition (german translation), Pearson Studium, 2001.

[Yoshida/Leopold 2001] Yoshida, Junko; Leopold, George: **Copy protection bill divides industry, Hollywood**, EETimes, October 1, 2001, on the internet: `http://www.eetimes.com/story/OEG20010928S0110` [16 Oct 2001].