11-30-2010

The Economics of Developing Security Embedded Software

Craig S. Wright
Charles Sturt University

Tanveer A. Zia
Charles Sturt University

DOI: 10.4225/75/57b5249fcd8af
8th Australian Information Security Mangement Conference, Edith Cowan University, Perth Western Australia, 30th November 2010

The Economics of Developing Security Embedded Software

Craig S Wright, and Tanveer A Zia
School of Computing and Mathematics
Charles Sturt University
Wagga Wagga, NSW, Australia
cwright20@postoffice.csu.edu.au
tzia@csu.edu.au

Abstract

Market models for software vulnerabilities have been disparaged in the past citing how these do little to lower the risk of insecure software. In this paper we argue that the market models proposed are flawed and not the concept of a market itself. A well-defined software risk derivative market would improve the information exchange for both the software user and vendor removing the often touted imperfect information state that is said to believe the software industry. In this way, users could have a rational means of accurately judging software risks and costs and as such the vendor could optimally apply their time between delivering features and averting risk in a manner demanded by the end user. It is of little value to increase the cost per unit of software by more than an equal compensating control in an attempt to create secure software. This paper argues that if the cost of an alternative control that can be added to a system is lower than the cost improving the security of the software itself, then it is uneconomical to spend more time and hence money improving the security of the software. It is argued that a software derivative market will provide the mechanism needed to determine these costs.

Keywords

Security, Derivatives, vulnerability market, software development, game theory

INTRODUCTION

This paper investigates the economics of security models and presents a hedge fund software risk derivative market approach to embed the security in software development. From the software security assurance perspective, the only practical approach to security assurance which is currently adapted by the security assurance industry is software vulnerability scanning and testing. This approach of security assurance requires the software to be physically present which means that we have already spent most of the budgeted time and cost in the development. If after performing those tests vulnerabilities are found, fixing those vulnerabilities would incur additional time and significant costs or may require the software to be totally scrapped and built from the scratch again. At this stage we can safely conclude that the current methods of assurance are time lagging and unless the security tests are passed, would always incur additional time and costs to ensure compliance to any information security policy or standard.

This paper starts with a review of the existing literature on the subject. The various market models that have been commonly applied are analysed against quantitative findings and current economic thought. The notion of vendor utility and incentives to minimise security flaws is introduced in section three. Here we demonstrate that there is an optimal point of vulnerability discovery. In this, it is argued that as some early bugs cost far less to discover than later vulnerabilities, an economic optimum has to exist. In section four, we argue that this optimised point is the position where it costs more to find more vulnerabilities than it would to use an alternative or external control. That is, if the cost of uncovering any additional software vulnerability exceeds the cost of an external control that could mitigate that control, the effort in discovering further flaws in the software is economically inefficient.

The paper finishes with an analysis of the impact of imperfect information on software vendor models. It is demonstrated that the introduction of a risk hedging derivative instrument would allow a market based appraisal of the cost of software flaws. Here, the market and end user could "bet" on the expected security and reliability of software. If the software proves to be more reliable than was expected, reputational effects are expected to increase the value of a vendor's future offerings. Users of software could also compare the total costs of the software plus expected reliability against both third party controls as well as against alternative vendors resulting in the economically optimised level of testing from vendors.

LITERATURE REVIEW AND RELATED WORK

Arora, Telang and Xu (2005) asserted that a market based mechanism for software vulnerabilities will necessarily underperform a CERT-type mechanism. The market that they used was a game theoretic *pricing game* (Nissan, Roughgarden, Tardos, and Vazirani (Eds.) 2007). In the model reported, the players in the market do not report their

prices. These players use a model where information is simultaneously distributed to the client of the player and the vendor. The CERT model was touted as being optimal. It relies on waiting until a patch was publically released and only then releasing the patch to the public. This ignores many externalities and assumes the only control is a patch in place of other alternative compensating controls.

Criminal groups have an incentive to maximize the time that vulnerabilities remain unknown as this extends the time that they have to exploit these bugs. Penetration testers have similar incentives as the trade secret of an unknown zero day vulnerability can provide them with a competitive advantage. This also goes to reputational motives for both parties. Consequently, the examined "market" model is in itself sub-optimal. It both creates incentives to leak information without proper safeguards and creates vulnerability black-markets. As criminal groups and selected security vendors (such as Penetration testers and IDS vendors) have an incentive to gain information secretly, they have an incentive to pay more for unknown vulnerabilities in a closed market. This means that a seller to one of these parties has a reputational incentive to earn more through not releasing information as the individual's reputation will be based on their ability to maintain secrecy.

This misaligned incentive creates a sub-optimal market. As a consequence, the market reported (Arora, Telang and Xu 2005; Kannan and Telang 2004) was sub-optimal to a CERT as this was an inefficient market and not that markets are less effective. The skewed incentivisation structure recommended in the paper was the source of the inefficiency and not the market itself. This simply highlights the need to allow efficient markets to develop rather than seeking to create these through design.

The other argument posed comes as a consequence of information asymmetry. It is asserted (Arora, Telang and Xu 2004) that software vendors have an informational advantage over other parties. The vendor does have access to source code (which is also available for Linux and other open source providers), but it can be proved that this does not provide the levels of information asymmetry that are asserted. Software vendors have a reputational input to their value (Cavusoglu, Cavusoglu and Zhang 2006).

Telang & Wattal (2004) did note that the market value of a software vendor influences through reputational costs and those vulnerabilities correlate significantly with a decrease in the companies traded price, a view supported by others (Cavusoglu, Cavusoglu and Zhang 2006).

> *"Vulnerability disclosure adversely and significantly affects the stock performance of a software vendor. We show that, on average, a software vendor loses around 0.63% of market value on the day of the vulnerability announcement. This translates to a dollar amount of $0.86 billion loss in market value. We also show that markets do not penalize a vendor any more if the vulnerability is discovered by a third party than by the vendor itself."*

TESTING SOFTWARE AS AN ECONOMIC CONSTRAINT

The results presented by Telang & Wattal (2004) demonstrate that a vendor has an incentive to minimize the vulnerabilities found in their products. If an excessive number of vulnerabilities continue to impact a vendor, their market capitalization suffers as a consequence. This itself is a strong argument that a vendor does not have an incentive to hide information (as third party vulnerability researchers cause an equal loss in capitalization). It has to be expected that any vulnerability known by the vendor will be uncovered. If the vendor fixes this flaw before release, the cost is minimized and at the limit approaches the cost of testing, (that is a zero incremental cost to that which would be expressed later).

If the vendor discovers vulnerability in the software they produce, the result is a *'strongly dominated'* motive to fix the bug. Hence, any remaining bugs are those that have not been uncovered by the vendor and which are less economical to find (through an increase in testing). It can thus be demonstrated that the vendor knows no more than the user at the point of software release as to the state of bugs in a product.

Testing is far less expensive earlier in the development cycle (RTI 2002). Figure 1 displays the expected utility of testing as the development progresses through an SDLC. Early in the process, the software developer has the greatest returns in testing and bug finding. As the development progresses, the returns are reduced as the process required and the costs associated with finding and correcting software vulnerabilities increases. Utility is a measure of relative value. As a result, the longer that a tester spends on discovering vulnerabilities, the lower the expected returns that will be achieved. The utility is lowest when the software has been shipped to the user. At this point, fixing flaws is an expensive process for both the user and the vendor. This leaves the optimal solution to find as many bugs as possible as early in the development process as is feasible. This contrasts with the increasing costs of finding bugs (Figure 2). This leaves the optimal solution for the vendor based on the discovery of as many bugs as possible as early in the development process

as is feasible. A bug discovered early in the process can cost as much as 10x less than one discovered later (Brooks 1995). It does not mean that all bugs or vulnerabilities will be found as the cost of finding additional vulnerabilities quickly exceeds the returns.

Figure 2 plots the incremental costs of finding each additional vulnerability in an in-house developed software application. These figures are derived from an analysis of project data and financial statements from 15 Australian companies between 2006 and 2009. 277 separate projects were analysed. The cost of discovering each additional vulnerability increases significantly and exponentially. This results in a markedly increasing overall increase in the mean or average cost of finding each vulnerability for each additional fault that is to be detected. Further analysis of this data is to be presented in a later study.

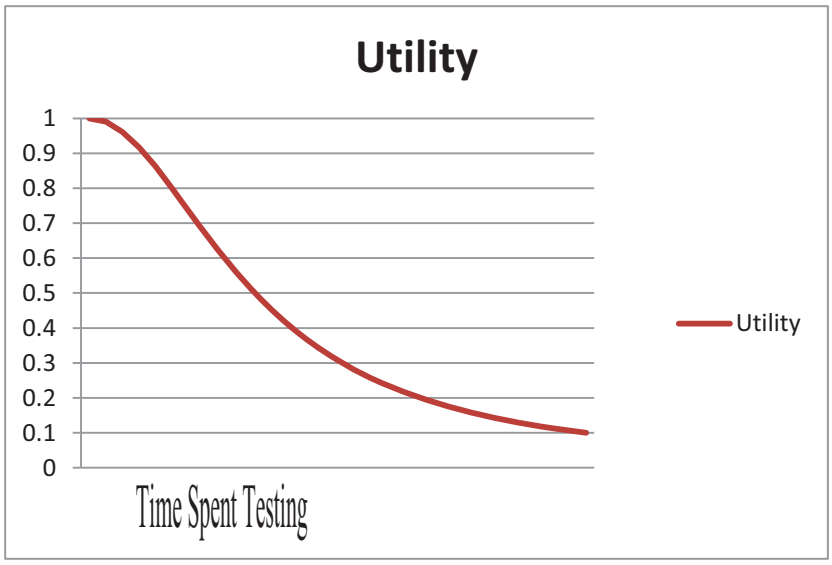

Figure 1: Decreasing utility of testing as the SDLC progresses

The widespread use of Open Source software (such as Linux) and the limited differential in discovering bugs using source code reviews demonstrates that the vast majority of software flaws (Cohen 2006) in commercial software are discovered early in the development process. It is highly unlikely that commercial software developers are less effective than open source developers and that the rate of errors is likely to be similar if not lower.

Adams (1984) noted that a third of all software faults take more than 5000 execution-years to manifest themselves. The secluded EAL6+ software sampled by Adams is not statistically significant over all software, but it does provide evidence of the costs. This also demonstrates why only two operating system vendors have ever completed formal verification. The "Secure Embedded L4 microkernel" by NICTA comprises 9,300 lines of code, of which 80% has been formally verified at a cost of 25 person years of work. The US$ 700 ploc costs for this exercise (the low estimate) demonstrates why formal verification is not a feasible solution for most software. This amount of time creates a great amount of expense for software development when it has to be completely conducted internally. An open market on the other hand distributes this expense in an optimal manner and provides a source of information as to the true cost of the vulnerability. This information is created as the true costs of developing patches and can be compared to alternatives where patching is more costly.

A vulnerability market provides information on discovery and vendor action (for instance Microsoft vs. Adobe vs. Linux etc) allowing clients to better select software vendors, mitigating the "*Market for Lemons*" (Akerlof 1970; Nissan, Roughgarden, Tardos and Vazirani (Eds.) 2007) that has been proposed. It has been demonstrated already that software vendors do not have more knowledge of bugs than users (or these would have been fixed prior to release) and hence do not have an advantage in information asymmetry when it comes to finding software flaws that may result through product interactions.

The market for lemons requires that the vendor knows the level of flaws better than the user. To many this may seem a common sense outcome, the vendor has access to source code, wrote the program and ran the development process. This is a flawed view as we have demonstrated as it is in the vendor's interest to mitigate (Davis and Holt 1993) vulnerabilities as early as possible. More importantly, the vendor is punished for bugs (Telang and Wattal 2004; Weigelt and Camerer 1988).

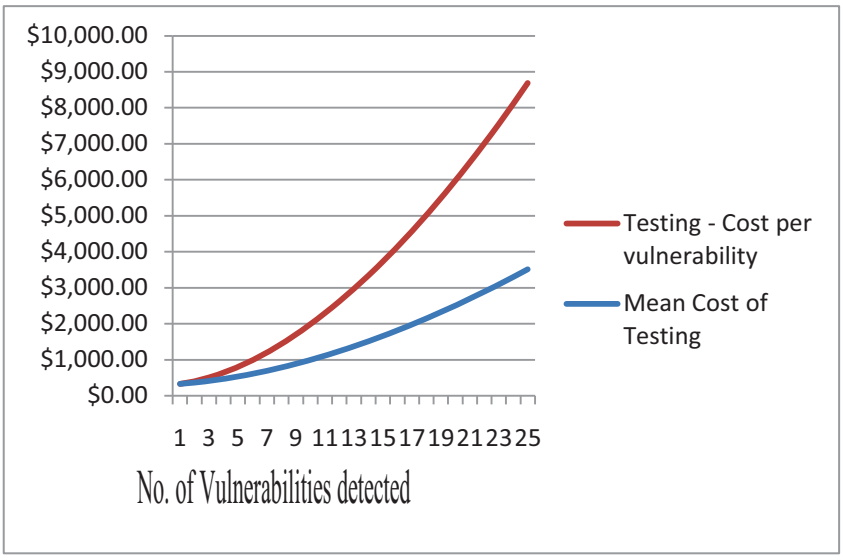

Figure 2: Each vulnerability costs more than the last to mitigate

A SOLUTION, SOFTWARE DERIVATIVE MARKETS

Our solution to the limited and sub-optimal markets that currently exist would be the creation of hedge funds for software security. Sales in software security based derivatives could be created on forward contracts. One such solution is the issuing of paired contracts (such as exist in short sales of stocks[8]). The first contract would be taken by a user and would pay a fixed amount if the software has suffered from any unmitigated vulnerabilities on the (forward) date specified in the contract. The paired contract would cover the vendor. If the vendor creates software without flaws (or at least mitigates all easily determinable flaws prior to the inception of the contract) the contract pays them the same amount as the first contract.

This is in effect a 'bet' that the software will perform effectively. If a bug is discovered, the user is paid a predetermined amount. This amount can be determined by the user to cover the expected costs of patching and any consequential damages (if so desired). This allows the user to select their own risk position by purchasing more or less risk as suits both the risk tolerance and the nature of the user's systems.

Such a derivative (if an open market is allowed to exist) would indicate the consensus opinion as to the security of the software and the reputation of the vendor. Such an instrument would allow software vendors and users to hedge the risks faced by undiscovered software vulnerabilities. These instruments would also be in the interest of the software vendor's investors as the ability to manage risk in advance would allow for forward financial planning and limit the negative impact that vulnerability discovery has on the quoted prices of a vendors capital.

Selective Survival

If the vendor creates the low feature version for $200 and the full featured secure version for $800, the user can choose the level of protection. Taking a survival model of Windows 98 and one for Windows XP (Campodonico 1994), the client can calculate the expected difference from each product. If Windows 98 has an expected compromise rate (without other compensating controls) of once per 6.2 days and Windows XP has an expected rate of 52.5 days when the firewall is enabled[9] we can calculate the cost per incident. If the user has a mean cost per incident of $320[10], we see that the Windows 98 option has an expected annual cost to the organisation of $19,040 ($18,840 damage plus $200) whereas Windows XP has an expected cost of $3,825 ($3,025 damage plus $800).

[8] Short selling involves an investor anticipating a decrease in the price or an item. This involves selling a chattel that the investor does not own through a contract agreement. If the goods sell higher than anticipated, the investor losses money as they have to purchase the goods at the (now higher) market rate.
[9] In each case we are presuming that the Centre for Internet Security (www.cisecurity.org) standards for installing the software has been reasonably met.
[10] Based on quantitative data to be published later this year.

Hence, most companies with a need to secure systems selected the more secure option and installed Windows XP over Windows 98. The initial costs did not reflect the overall costs to the organisation. Likewise, home users (who rarely calculate the costs of compromise) were more likely to install the cheaper option.

To extend the analogy, assume that a software vendor can create a perfect version of software for a mean cost of $20,000, a low estimate, and the flawed version at the standard costs of under $1,000. Where the expected damages do not exceed $19,000 per copy of software, we see that it remains in the user's interests to select the lower security option as the expected losses are lower. More generally, for an organisation with ***n*** users and an expected cost $C_S = nP_S$ (where P_S is the cost per user of the secure software product) against $C_I = nP_I$ (with P_I being the cost per user of the insecure software product), the cost of deploying either option is based on the expected losses for each option with a loss function defined as $D_S < D_I$. As such, if $(C_S - D_S) < (C_I - D_I)$ it is in the interest of the company to take the secure option. Where $(C_S - D_S) > (C_I - D_I)$, the economical solution is to install the less secure version of the software and self-insure against the loss.

As a means of maximizing the allocations of risk and costs, the vendor could offer liability-free and full-liability versions of their product, the later sold at a reduced price. The vendor could then control their risk using a hedging instrument as discussed earlier. A market evaluation of the risk being traded would provide better information than that which the vendor has alone (Not only does the market pooled knowledge feed into the price of the derivative, but the vendor's knowledge of their own product does as well). With this in mind, the user could also purchase the same derivative as the vendor. The cost to the user to purchase the software plus risk insurance would be less than purchasing the "more secure" version from the vendor as the vendor would hedge the risk it faces and add a transactional cost as well as a return. This return has to be incremented against the firms IRR (internal rate of return) in order to account for the cost plus the loss of an expected return from alternative investments.

Both parties are better off where the vendor produces the optimized version of their product and where the vendor does not assume liability for bugs.

Simply put, the vendor will provide a warranty which acts as an insurance policy for the user in cases where it can charge more for the provision of the warranty than it costs to provide the warranty. This cost could be either from a hedging instrument or through an investment in more testing. However, it must be noted that no increasing testing will make software invulnerable to all attacks and remove the ability for flaws to be discovered. As such, the cost of the hedged derivative will decrease, but will not go to zero.

The argument against this form of market is imperfect information. Many people argue that the software vendor has more knowledge than the user. This is untrue for several reasons. The vendor has no incentive to hide vulnerabilities as each vulnerability impacts the share price of the vendor through a decrease in capital (Telang and Wattal 2004). Next, the vendor would be competing on the same derivative markets as the users. The vendor's knowledge would be quickly and efficiently transferred through the market mechanism. The end result is that a derivatives based strategy does not allow for information asymmetry and would disprove entirely the assertion that software vendors operate a "*market for lemons*".

From this we can see that the lack of liability does not require that the software vendor does not have the appropriate incentive to provide the optimal amount of testing and hence security to the user, as well as to add the optimal levels of external controls. If more testing is cost effective in economically providing more secure software, that is if the additional testing is cost effective, the software vendor will conduct this testing whether it is liable for security flaws or not.

The incentive to provide security is no different than the incentive to provide other characteristics and features. A vendor will add software extensions that can reduce the overall security of a system if the user values these to a greater extent than their costs.

Safety (and security in general) is a tie-in good. Amenities cost something. Vendors provide amenities if consumers (users) are willing to pay the cost of obtaining these.

> "*Cars come with batteries and tires, but not always with stereo equipment or antitheft devices and rarely with driving gloves and sunglasses*" (Donald 2006).

What is demonstrated by those organisations with a defensive (and hence secure) coding regime is a lower variability in output. The mean coding time will remain similar, but the test time can be expected to be distributed differently for the organisation that codes with a defensive posture than that of the organisation that leaves too little time for testing. This increase in the standard deviation of produced results increases the risk to the all parties, both vendors and users.

Optimal Derivatives Design under Dynamic Risk Measures

The game theoretic approach to this can be modeled looking at the incentives of the business and programming functions in the organisation. Programmers are optimists (Brooks 1995). As Brooks (1995) noted, "*the first assumption that underlies the scheduling of systems programming is that all will go well*". Testing is rarely considered by the normal programmer as this would imply failure. However, the human inability to create perfection leads to the introductions of flaws at each stage of development.

In the model presented by Brooks (1995), as a program moves to a "Programming Product" and then to "a Programming Systems Product", there are incremental additions that extend the estimates of the programming team. This can be achieved through the addition of a "Programming System" with the addition of Interfaces and a System Integration Phase. At each phase these can be expressed as a function of effort expressed in the lines of code, l_c. We can express the mean effort $\overline{x}_{l_c}$ required to code a number of lines and the variability σ_{l_c} in achieving this outcome (where $\sigma = Variance^2$). This allows us to represent the coding effort as a representation of the stages of development:

$$F(x) = 9H\left(\overline{x}_{l_c}, \sigma_{l_c}\right) G\left(\overline{x}_{l_c}, \sigma_{l_c}\right) + \varepsilon \qquad (1)$$

In (1), $H\left(\overline{x}_{l_c}, \sigma_{l_c}\right)$ is the function of systematizing (Brooks 1995, p6) the code. The expression $G\left(\overline{x}_{l_c}, \sigma_{l_c}\right)$ is the productization of the code.

The derivative would require the firm to publish their productivity figures;

- Lines of code per programmer (mean value and standard Deviation); l_c, $\overline{x}_{l_c}$ and σ_{l_c}
- Bug-incidence figures (mean bugs per loc); $\overline{b}_{l_c}$ and σ_b
- Estimating and project rules/methodology
- Software design methodology
- Secure Programmer measures[11]. $\overline{t}$ and σ_t

Many see this as proprietary data and would be loath to share, but as more in the market take-up the use of such a derivative in managing their own risk, the incentives for others to follow increase. We can also demonstrate that as $\overline{t} \to \max_t$ and $\sigma_t \to 0$ that the volatility in coding σ_{l_c} and σ_b decrease with the number of reported bug incidents $\overline{b}_{l_c} \to 0$ as $\overline{t} \to \max_t$.

More metrics would be required based on the methodologies and coding practices used with different computer languages expected to exhibit different responses.

As the skill of the programming team ($\overline{t} \to \max_t$ & $\sigma_t \to 0$) increases through group training, secure coding practices and in-line testing, the volatility in coding $\sigma_{l_c} \to 0.$ the incremental returns on $\overline{t}$ diminish as $\overline{t} \to \max_t$ and the cost of training continue to grow linearly, that is, $Cost(t) = at + b$ where there is an initial cost plus a linear increase in the costs of developing the programmer skills. We can see that the optimal position for any software vendor is to have a skilled team that maximizes returns. At this point, additional training becomes economically unviable and does not create further returns for the organisation, this investment, however, may still be made by the individual programmer with returns to the organisation and also for further personal gains (investment in personal capital). We also see that it is

[11] e.g. SANS Coding Methodology could be used as such a baseline.

in the interest of the firm to minimize the variance in skill levels between programming staff (aim to have $\sigma_t \to 0$). This of course adds costs as junior staff would be less productive in that they would have to work with more senior staff in order to both cross train and to ensure that the junior employees maintain an acceptable level of production and testing. Figure 3 displays the expected utility of training staff in secure coding practices. The initial investment returns a large expected utility for each unit of training (t). However, the expected utility gained from training starts to decrease once the programmer has become competent.

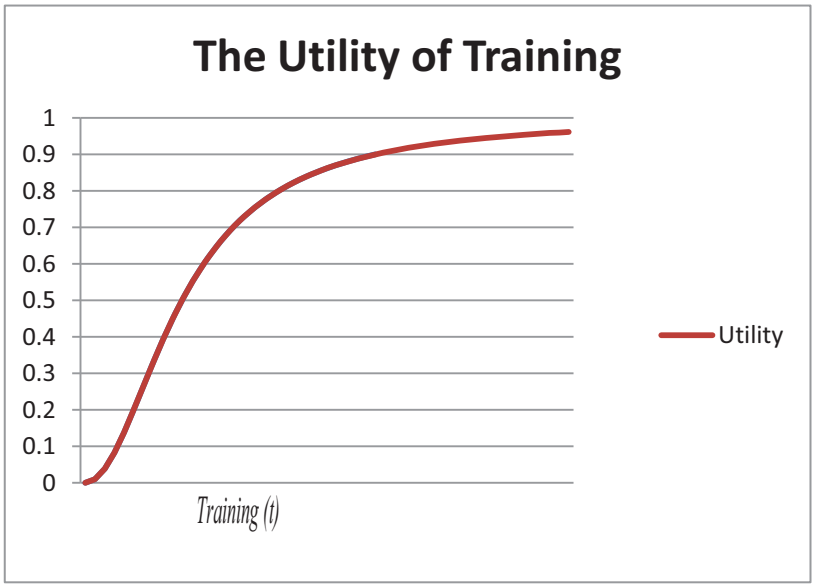

Figure 3: Training Software developers in Security adds utility

This exercise of joining junior and senior staff would create a return function depicted by Brookes (1995, p19) for a "*Time versus number of workers - task with complex interrelationships*". The output of the junior/senior staff unit would be lower than the senior staff level (initially) but would lower variability and increase the skill levels of the junior staff and hence over time $\sigma_t \to 0$. Due to staff turn-over and the increasing costs of maintaining more skilled programmers, this would also have a negative feedback on the vendor's production costs. This would also need to be optimized rather than maximized if the vendor is to maximize returns.

ANALYSIS: NO MORE LEMONS

The fallacy of the mainstream analysis of deflation can be used to demonstrate the fallacy of aligning a market for lemons to computer software. This game model has people refrain from spending when they anticipate falling prices. As people expect prices to fall, they believe that they will get a better price if they consume later (or that they will get more secure software for the same price).This drop in consumption results in a consequential price fall, and the whole cycle repeats. The argument made is that prices will spiral uncontrollably downward.

Robert Murphy (Murphy and Regnery 2009) addressed this issue and demonstrated its fallacy:

> *"One could construct an analogous argument for the computer industry, in which the government passes regulation to slow down improvements in operating systems and processing speed. After all, how can computer manufacturers possibly remain viable if consumers are always waiting for a faster model to become available? ... The solution to this paradox, of course, is that consumers do decide to bite the bullet and buy a computer, knowing full well that they would be able to buy the same performance for less money, if they were willing to wait… (There's no point in holding out for lower prices but never actually buying!)"* (pp. 68–9)

Some users do wait for patches to be proven and others use "*bleeding edge software*" (and some never patch). The model is more rightly based on a distribution of users centered on an install time slightly after the release date.

This model can help us combat George Akerlof's lemons model of the used-car market which has been applied to the computer industry. Akerlof argued that asymmetric information would result in the owners of good used cars being driven from the market. Contrary to the equilibrium suggested in this model, good used cars sell. One can state that just

because owners of good cars may not be able to obtain as high a price as they would like, it does not follow that they will refuse to sell at all.

Likewise, just as users do not know the state of security or how many bugs exist in software, software is still sold.

The "real balance effect": as prices fall, the value of money rises.

People's demand to hold money can be satisfied with less money as price deflation occurs. This in part explains why people will eventually spend, even when they expect prices to continue to fall. The same process applies to software. Users see value in the features and processes they purchase software for. These are offset against the costs which include the monetary costs as well as the costs of maintenance, patching, security, and fixing other flaws. Failure has a cost and this is incorporated in to the price of software to the user. The result is that users buy and install software even when they expect more bugs/vulnerabilities to be found.

It is also important that companies move from "Lines of Code per day" as a productivity measure to a measure that takes debugging and documentation into account. This could be something such as "Lines of clean, simple, correct, well-documented code per day". This also has problems, but it does go a long way towards creating a measure that incorporates the true costs of coding. The primary issue comes from an argument to parsimony. The coder who can create a small, fast and effective code sample in 200 lines where another programmer would require 2,000 may have created a more productive function. The smaller number of lines require less upkeep and can be verified far easier than the larger counterpart.

Such factors can be incorporated into a market based model where small clean code would be expected to be valued higher than a large code base with the same functionality as the cost of maintaining the former is less than the latter.

CONCLUSION

Just as car dealers buff the exterior and detail the upholstery of a used car, neglecting the work that should be done on the engine, software vendors add features. Most users are unlikely to use even a small fraction of these features, yet they buy the product that offers more features over the more secure product with fewer features. The issue here is that users buy the features and this is a less expensive option for the vendor.

The creation of a security and risk derivative should change this. The user would have an upfront estimate of the costs and this could be forced back to the software vendor. Where the derivative costs more than testing, the vendor would conduct more in-depth testing and reduce the levels of bugs. This would most likely lead to product differentiation (as occurred in the past with Windows 95/Windows NT). Those businesses will to pay for security could receive it. Those wanting features would get what they asked for.

At the end of any analysis, security is a risk function and what is most important is not the creation of perfectly secure systems, but the correct allocation of scare resources. Systems need to be created that allow the end user to determine their own acceptable level of risk based on good information.

It is argued [Donald 2006, Durtschi 2002] that negligence rules are required to force software vendors to act optimally. However, the effects of reputation and the marginal cost effect from reduced sales are incentives in themselves for the vendor to act optimally. The supply of imperfect information to the vendor through incomplete and inadequate feedback channels is a problem that can be solved in part through the creation of a market for vulnerability research. The effects of reputation on the vendor and the assignment of risk through this process result in fewer negative externalities then occur because of a legislative approach.

At present, it can be demonstrated that neither side (the user or the vendor) is informed as to the true risks and costs of software. The vendor does not know the situation of the user (for the most part) and the user may not have an easily accessibly means of determining the risk from software. Worse, neither side is likely to know the risk posed from integrating multiple software products. This model allows the end user to have an idea of the cost of security in software products. The market based trading of software derivatives allows the end user to select either the software product with the highest expected security or to introduce alternatives (such as third party add-ons, firewalls etc).

Such a market based model allows both the vendor and the end user to evaluate the costs of security inherent in software and to evaluate alternative controls against competing products.

REFERENCES

Adams, N.E., (1984) "Optimizing preventive service of software product," IBM Journal of Research and Development, 28(1), p. 2-14

Akerlof, George A., "The Market for 'Lemons:' Quality Uncertainty and the Market Mechanism," Quarterly Journal of Economics, August 1970, 84, 488-500.

Arora, A. & Telang, R. (2005), "Economics of Software Vulnerability Disclosure", IEEE Security and Privacy, 3 (1), 20-2

Arora, A., Telang, R. & Xu, H. (2004) "Optimal Time Disclosure of Software Vulnerabilities", Conference on Information Systems and Technology, Denver CO, October 23-2

Arora, A., Telang, R. & Xu, H. (2008), "Optimal Policy for Software Vulnerability Disclosure", Management Science 54(4), 642-6

Bacon, D. F., Chen, Y., Parkes, D., & Rao, M. (2009). A market-based approach to software evolution. Paper presented at the Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications.

Beach, J. R., & Bonewell, M. L. (1993). Setting-up a successful software vendor evaluation/qualification process for `off-the-shelve' commercial software used in medical devices. Paper presented at the Computer-Based Medical Systems, 1993. Proceedings of Sixth Annual IEEE Symposium on.

Brookes, F. (1995) "The Mythical Man-Month". Addison-Wesley

Campodonico, S. (1994). A Bayesian Analysis of the Logarithmic-Poisson Execution Time Model Based on Expert Opinion and Failure Data. IEEE Transactions on Software Engineering, 20, 677-683.

Cavusoglu, H., Cavusoglu, H. & Zhang, J. (2006) Economics of Security Patch Management, The Fifth Workshop on the Economics of Information Security (WEIS 2006)

Cohen,. J. (2006). Best Kept Secrets of Peer Code Review (Modern Approach. Practical Advice.). Smartbearsoftware.com

Davis, Douglas D., and Charles A. Holt, Experimental Economics, Princeton: Princeton University Press, 1993.

de Villiers, M (2005) "Free Radicals in Cyberspace, Complex Issues in Information Warefare" 4 Nw. J. Tech. & Intell. Prop. 13, http://www.law.northwestern.edu/journals/njtip/v4/n1/2

Dijkstra, E. W. (1972). "Chapter I: Notes on structured programming Structured programming" (pp. 1-82): Academic Press Ltd.

Donald, D. (2006), "Economic Foundations of Law and Organisation" Cambridge University Press

Durtschi, C., Hillison, W., & Pacini, C. (2002) "Web-Based Contracts: You Could Be Burned!" Journal of Corporate Accounting & Finance, Volume 13, Issue 5 , Pp 11 – 18.

Kannan K & R Telang (2004) 'Market for Software Vulnerabilities? Think Again.' Management Science.

Mills, H. D. (1971) "Top-down programming in large systems", Debugging techniques in large systems, R. Rustin Ed., Englewoods Cliffs, N.J. Prentice-Hall

Murphy, R. & Regnery, P. (2009) "The Politically Incorrect Guide to the Great Depression and the New Deal".

Nissan, N., Roughgarden, T., Tardos, E. & Vazirani, V. (Eds.) (2007) "Algorithmic Game Theory" Cambridge University Press, {P14, Pricing Game; P24, Algorithm for a simple market; P639 Information Asymmetry).

Nizovtsev, D., & Thursby, M. (2005) "Economic analysis of incentives to disclose software vulnerabilities". In Fourth Workshop on the Economics of Information Security.

Ounce Labs, 2. http://www.ouncelabs.com/about/news/337-the_cost_of_fixing_an_application_vulnerability

Ozment, A. (2004). "Bug auctions: Vulnerability markets reconsidered". In Third Workshop on the Economics of Information Security

Perrow, C. (1984/1999). Normal Accidents: Living with High-Risk Technologies, Princeton University Press.

RTI. (2002). The Economic Impacts of Inadequate Infrastructure for Software Testing. A report prepared by RTI for NIST. Retrieved from http://www.nist.gov/director/planning/upload/report02-3.pdf

Telang, R., & Wattal, S. (2004) "Impact of Software Vulnerability Announcements on the Market Value of Software Vendors – an Empirical Investigation" http://infosecon.net/workshop/pdf/telang_wattal.pdf

Turing, A (1936), "On computable numbers, with an application to the Entscheidungsproblem", Proceedings of the London Mathematical Society, Series 2, 42 pp 230–265

Weigelt, K. & Camerer, C. (1988) "Reputation and Corporate Strategy: A Review of Recent Theory and Applications" Strategic Management Journal, Vol. 9, No. 5 (Sep. - Oct., 1988), pp. 443-454, John Wiley & Sons