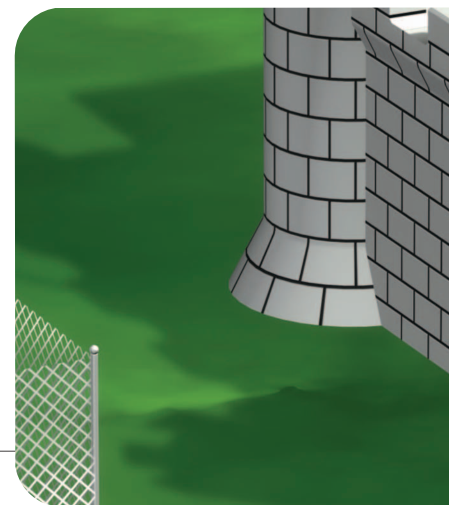
Is Finding Security Holes a Good Idea?

Despite the large amount of effort that goes toward finding and patching security holes, the available data does not show a clear improvement in software quality as a result.



Eric Rescorla
RTFM

Security professionals expend an enormous amount of effort every year on finding, publishing, and fixing security vulnerabilities in software products. For example, the ICAT (http://icat.nist.gov) vulnerability metabase added 1,307 vulnerabilities in 2002, and Microsoft Internet Explorer 5.5 alone had 39 published vulnerabilities that year. In September 2003, the Full Disclosure mailing list (http://lists.netsys.com/mailman/listinfo/full-disclosure), dedicated to the discussion of security holes, had more than 1,600 postings. Given all of this effort, we should expect to see a clearly useful and measurable result.

The basic value proposition of vulnerability finding is simple: *It's better for good guys to find and fix vulnerabilities than for bad guys to find and exploit them.* If good guys find a vulnerability and make a fix available, the number of intrusions—and hence the cost of intrusions—resulting from that vulnerability is less than if bad guys had discovered it. Moreover, fewer vulnerabilities will be available for bad guys to find.

This article aims to measure the effect of vulnerability finding. Any attempt to measure this kind of effect is inherently rough, depending as it does on imperfect data and several simplifying assumptions. Because I'm looking for evidence of usefulness, where possible, I bias such assumptions in favor of a positive result—explicitly calling out those assumptions biased in the opposite direction. Thus, the analysis in this article represents the best-case scenario, consistent with the data and my ability to analyze it, for the vulnerability finding's usefulness.

The life cycle of a vulnerability

To assess vulnerability finding's value, it's necessary to examine the events surrounding discovery and disclosure. Several authors, including Hilary Browne and colleagues[1] and Bruce Schneier[2] have considered a vulnerability's life cycle. In this article I use the following model, which is rather similar to Browne's.

- *Introduction*—the vulnerability is released as part of the software.
- *Discovery*—someone finds the vulnerability.
- *Private exploitation*—the discoverer or a small group known to the discoverer exploits the vulnerability.
- *Disclosure*—the vulnerability's description is published.
- *Public exploitation*—the general community of blackhats exploits the vulnerability.
- *Fix release*—the vendor releases a patch or upgrade that closes the vulnerability.

These events don't necessarily occur in this order. In particular, disclosure and fix release often occur together, especially when the manufacturer discovers a vulnerability and releases the announcement along with a patch. I'm most interested in two potential scenarios: *whitehat discovery* (WHD) and *blackhat discovery* (BHD).

Whitehat discovery

In the WHD scenario, the vulnerability is discovered by a researcher with no interest in exploiting it. The researcher notifies the vendor (often the discoverer is the vendor's employee) and the vendor releases an advisory along with a fix. Of course, it's possible to release the advisory before the fix, but this is no longer common practice. In the rest of this article, I'll assume that fixes and

public disclosures occur at the same time. In this scenario, the entire world (with the exception of the discoverer and vendor) finds out about the vulnerability at the same time. There is no private exploitation phase. Public exploitation begins at the time of disclosure.

The solid red line in Figure 1 shows the cost of attacks in the WHD case. Because attackers don't know of the vulnerability before the vendor discloses it, no intrusions occur up to disclosure time. At disclosure time the public exploitation phase begins and we start to see intrusions. The intrusion rate increases as attackers learn how to exploit the vulnerability. Eventually, people fix their machines or attackers lose interest in the vulnerability—perhaps due to decreasing numbers of vulnerable machines—and the number of intrusions decreases.

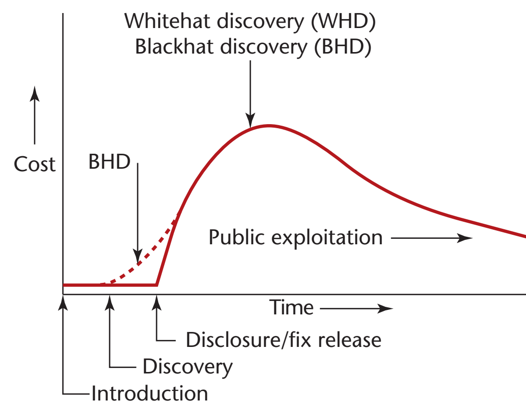


Figure 1. Vulnerability disclosure cost curve. The primary difference between the whitehat and blackhat scenarios illustrated here is the rate of exploitation between discovery and disclosure.

Blackhat discovery

In the BHD scenario, the vulnerability is first discovered by someone with an interest in exploiting it. The discoverer exploits the vulnerability and potentially tells some associates, who also exploit it. Thus, during the private disclosure period, a limited pool of in-the-know attackers can exploit the vulnerability, but the population at large cannot and the vendor and users are unaware of it.

Some time after the initial discovery, someone in the public community discovers the vulnerability. A person might find it independently, but more likely an aware operator will find it when an attacker uses the vulnerability to exploit the operator's system. At this point, the finder notifies the vendor and the process described in the previous section begins.

The red dotted line in Figure 1 shows the likely outcome of this scenario. The primary difference between it and the WHD scenario is the nonzero rate of exploitation in the period between discovery and disclosure. How large that rate is remains an open question. In Figure 1, it's shown as quite small compared to the public exploitation phase. This seems likely to be the case because many whitehat security researchers are connected to the blackhat community; any large-scale exploitation would likely be discovered quickly. No good data on this topic exists, but some observers have estimated that the average time from discovery to leak/disclosure is around a month (as Michal Zalewski noted in a 22 September 2003 posting to the Full-Disclosure mailing list, available at http://lists.netsys.com/pipermail/full-disclosure/2003-September/010673.html). After the bug is publicly disclosed, the WHD and BHD cost curves quickly join up as it becomes publicly known.

WHD vs. BHD

Figure 1 compares the costs due to intrusion in the two scenarios. In my model, the additional cost in the BHD scenario results from the private exploitation period, represented by the area between the dashed and solid lines in Figure 1. (I'm ignoring the effect of growth or shrinkage in the number of deployed copies of the package, as well as the effect of time-value discounting.)

It seems obvious that, given the choice, the WHD scenario would be preferable to the BHD scenario because it eliminates the private exploitation period. As a first approximation, we assume that except for this difference, the WHD and BHD scenarios are identical. Thus, the cost advantage of WHD over BHD is the cost incurred during the private exploitation phase. If we denote the cost of private exploitation as C_{priv} and the cost of public exploitation as C_{pub}, the cost of intrusions in the WHD scenario is $C_{WHD} = C_{pub}$, and the cost of intrusions in the BHD scenario is $C_{BHD} = C_{priv} + C_{pub}$. The advantage of WHD is $C_{BHD} - C_{WHD} = C_{priv}$.

Obviously, this approximation is imperfect and probably overestimates the cost difference. Administrators tend to be more diligent about patching if they know that a vulnerability is being actively exploited; thus, the total number of vulnerable systems will decline more quickly in the BHD scenario, and the peak rate of disclosure will be correspondingly lower. Similarly, some of the early exploiters immediately after disclosure are likely part of the private exploitation community, which means that disclosure probably will not produce as large a rise in initial exploitation in the BHD case as in the WHD. Conservatively, I ignore these effects.

Cost-benefit analysis of disclosure

Imagine you're a researcher and are the first person to discover a vulnerability in a widely used piece of software. You can keep quiet or disclose the vulnerability to the

Table 1. Disclose/don't disclose decision matrix.

DECISION	NOT REDISCOVERED ($1 - p_r$)	REDISCOVERED (p_r)
Disclose	C_{pub}	C_{pub}
Not disclose	0	$C_{pub} + C_{priv}$

vendor. If you notify the vendor, the WHD scenario will follow. If you don't notify the vendor, a blackhat might independently discover the vulnerability, thus initiating the BHD scenario. However, there's also a chance that no one will rediscover the vulnerability at all or that another whitehat will rediscover it. In the first case, the cost of disclosure will never be incurred. In the second, it will be incurred later.

To assess whether disclosure is a good thing, we can estimate the probability that a blackhat will rediscover the vulnerability. Consider a worst-case model: assume all potential rediscovery is by blackhats and denote the probability of rediscovery as p_r. Consistent with my practice, this simplifying assumption introduces a bias in favor of disclosure. The only way in which failure to disclose does harm is if a blackhat rediscovers the vulnerability. Thus, assuming that vulnerabilities are always rediscovered by blackhats overestimates the damage done by rediscovery and therefore the advantage of disclosure. Using standard decision theory,[3,4] we get the choice matrix in Table 1.

Working through the math, we find that choosing to disclose reduces the expected cost of intrusions only if $p_r (C_{priv} + C_{pub}) > C_{pub}$. To justify disclosing, then, the expected cost of excess intrusions in the case of BHD must be large enough to outweigh the known cost of intrusions incurred by disclosing in the first place.

From finding-rate to p_r

To attack this problem, I make one further simplifying assumption: that vulnerability discovery is a stochastic process. If a reasonable number of vulnerabilities exist in a piece of software, we don't expect that they'll be discovered in any particular order, but rather that any given extant vulnerability is equally likely to be discovered next. This assumption isn't favorable to the hypothesis that vulnerability finding is useful. If, for instance, vulnerabilities were always found in a given order, a vulnerability that isn't immediately disclosed would likely soon be found by another researcher. However, this simplification is probably approximately correct (because different researchers will probe different sections of any given program, the vulnerabilities they find should be mostly independent) and is necessary for our analysis. Using this assumption, we can use the overall rate of vulnerability discovery to estimate p_r.

Consider a piece of software containing V_{all} vulnerabilities. Over the software's lifespan, some subset of the vulnerabilities V_{found} will be discovered. Thus, the likelihood that any given vulnerability will be discovered during the software's life is

$$p_{discovery} = \frac{V_{found}}{V_{all}}$$

Similarly, if we choose a recently discovered vulnerability, the chance of its rediscovery has as its upper bound the chance of discovery:

$$p_r \leq p_{discovery} = \frac{V_{found}}{V_{all}}$$

Accordingly, if we know the total number of vulnerabilities in the software and the rate at which they're found, we can estimate the probability that someone will rediscover a vulnerability in a given time period. The problem therefore becomes to determine these two parameters.

Measuring the vulnerability discovery rate

I measure the rate of vulnerability discovery directly from the empirical data. Using that data and standard software reliability techniques I also derive an estimate for the number of vulnerabilities.

The procedure involves fitting a reliability model to the empirical data on vulnerability discovery rate, thus deriving the total number of vulnerabilities. The model also gives us the projected vulnerability-finding rate over time and, therefore, the probability of vulnerability discovery at any given time.

It's important to be specific about what I mean by a "piece of software." Software undergoes multiple releases in which vulnerabilities are fixed and other vulnerabilities are introduced. Our focus is individual software releases. For example, when FreeBSD 4.7 was shipped, it had a fixed number of vulnerabilities. During the software's life, some of those vulnerabilities were discovered and patched. If we assume that those patches never introduce new vulnerabilities—which, again, favors the argument for disclosure—the software's overall quality gradually increases. We're interested in the rate of that process.

In this context, we define system reliability as the number of failures (vulnerabilities) observed during a given time period. Thus, if a system is becoming more reliable, it's experiencing fewer failures. The literature on modeling software reliability is extensive and numerous models exist. For simplicity, most models assume that all failures are equally serious. We're primarily interested in models that show increasing reliability, because only those models predict a finite number of vulnerabilities. (If reliability doesn't increase, the projected number of vulnerabilities is effectively infinite

Table 2. Programs for analysis.

VENDOR	PROGRAM	VERSION	NO. OF VULNERABILITIES	RELEASE DATE
Microsoft	Windows NT	4.0	111	August 1996
Sun	Solaris	2.5.1	109	May 1996
FreeBSD	FreeBSD	4.0	43	March 2000
RedHat	Linux	6.2	58	March 2000

and the probability of rediscovery in any given time period must be low.)

The Goel-Okumoto Nonhomogenous Poisson Process model is the simplest such model.[5] The G-O model assumes that the number of vulnerabilities discovered in a single product per unit time $M(t)$ follows a Poisson process. The expected value of the Poisson process is proportional to the number of undiscovered vulnerabilities at t. The result is that the expected value curve follows an exponential decay curve of the form $rate = Nbe^{-bt}$, where N is the total number of vulnerabilities in the product and b is a rate constant. As more vulnerabilities are found, the product becomes progressively more reliable and the rate of discovery slows.

To measure the actual rate of vulnerability discovery, I used the ICAT vulnerability metabase. The US National Institute of Standards and Technologies makes the entire ICAT database available for public download and analysis, making it suitable for my purposes. I based my analysis on the 19 May 2003 edition of ICAT. I downloaded the database and processed it with a variety of Perl scripts. I performed all statistical analysis with R (see www.R-project.org for a description of R).

Do programs improve over time?

The obvious question is, "At what rate are vulnerabilities found in a given program?" For example, consider Microsoft Windows NT 4.0, released in August 1996. NT 4.0 had a fixed set of vulnerabilities—some already present in earlier revisions, but most introduced in that release. We can ask: How many of those vulnerabilities are found as a function of time? Because this question gives us a fixed starting point, this approach is susceptible to right censoring (bugs that persist past the end of the study period), but not left censoring (bugs found before the beginning of the study period).

However, the approach has two major problems:

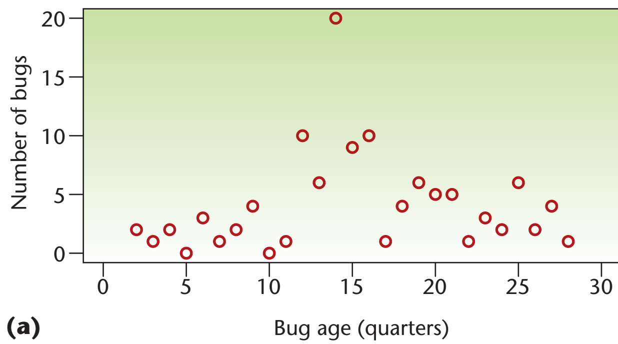
- Because the same vulnerabilities appear in multiple programs and multiple versions, it's impossible to analyze every program as if it were an independent unit.
- No individual program is likely to have a high number of vulnerabilities, which gives us low statistical power.

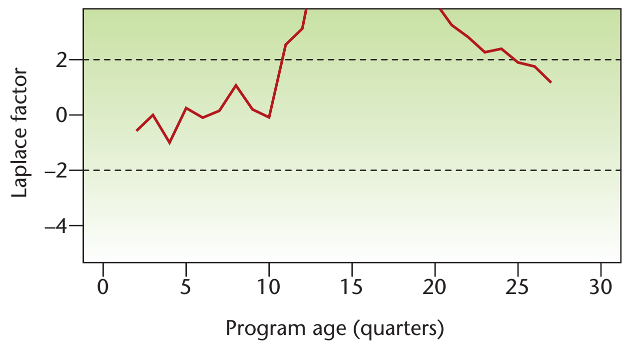
To keep the amount of interaction to a minimum, I focus on four program/version pairs (two open source and two closed source) as listed in Table 2. I chose these pairs to minimize interaction but to still provide enough data to analyze. For example, because they're closely related, I chose only one of the Windows and Internet Explorer group, despite the large number of vulnerabilities in both programs.

I measured age as the program's age, not the vulnerability's age. Thus, if a vulnerability was introduced in Solaris 2.5 but is still in Solaris 2.5.1, I'm concerned only with the time after the release of Solaris 2.5.1. If the bug-finding process is not memoryless, the results will be biased so that bug finding appears more effective than it actually is, because investigators have already had time to work on the bugs present in earlier versions. Conservatively, I ignore this effect.

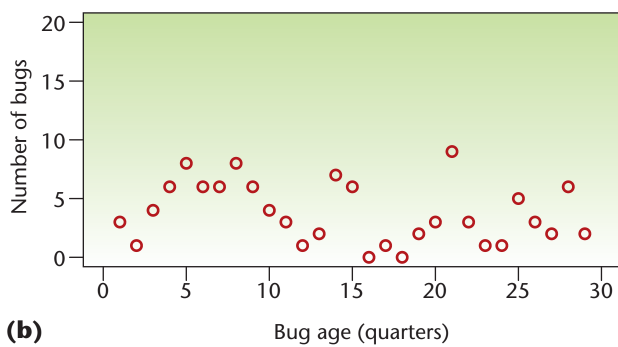
Figure 2 shows the vulnerability discovery rate for each program as a function of age. The left panels show the number of vulnerabilities found in any given period of a program's life (grouped by quarter). Visually, there are no apparent downward trends in finding rates for Windows NT 4.0 (the peak at quarter 14 appears to be the result of end-of-year cleanup at ICAT or Common Vulnerabilities and Exposures [CVE]), Solaris 2.5.1, and only a very weak one (if any) for FreeBSD. There is, however, a decided trend for Red Hat 6.2.

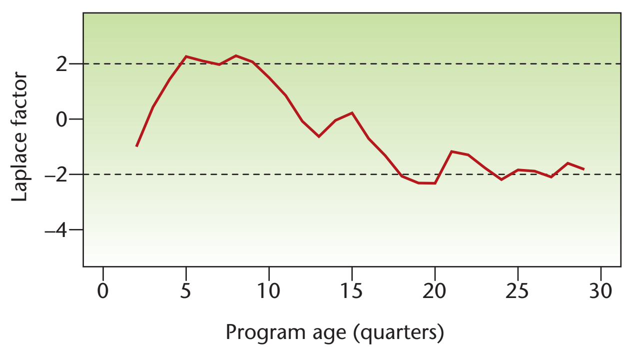
Moving beyond visual analysis, we can apply several statistical tests to look for trends. The simplest procedure is to attempt a linear fit to the data. Alternately, we can assume that the data fits a G-O model and fit an exponential using nonlinear least squares. Table 3 shows the results of these regressions. Neither fit reveals a significant trend for the first three programs—the data for Windows NT 4.0 is so irregular that the nonlinear least-squares fit for the exponential failed entirely with a singular gradient. The extremely large standard errors and p values indicate the lack of any clear trend. We see a significant result for both fits for Red Hat 6.2, which shows the type of exponential decay we would expect from a G-O process.



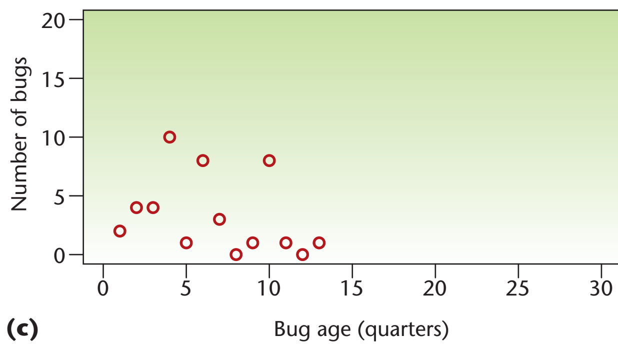


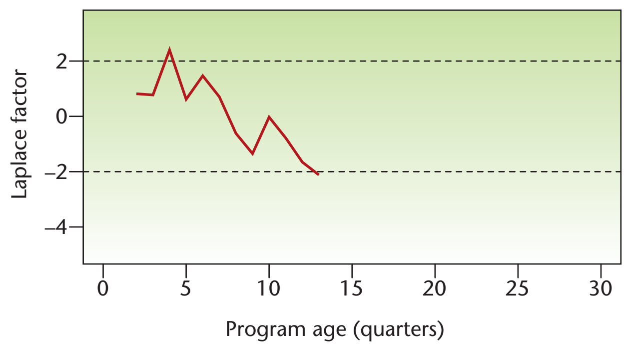
(a)



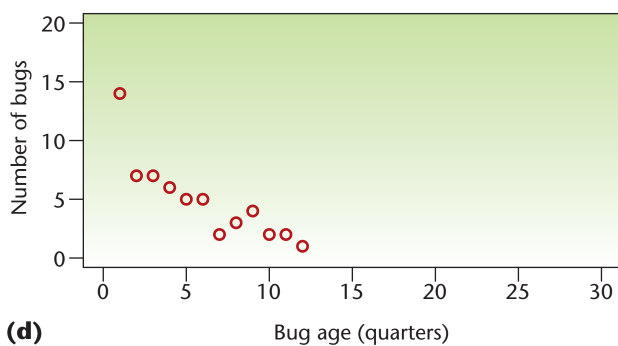


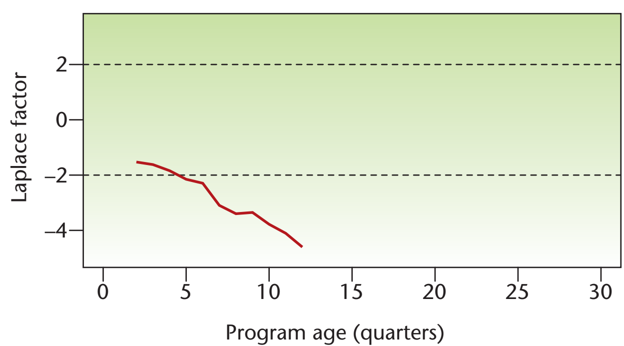
(b)





(c)





(d)

Figure 2. Vulnerability discovery rate by programs as a function of age: (a) Windows NT 4.0, (b) Solaris 2.5.1, (c) FreeBSD, and (d) Redhat 6.2.

Table 3. Regression results for program cohort data.

PROGRAM	LINEAR FIT SLOPE	LINEAR FIT STANDARD ERROR	LINEAR FIT p	EXPONENTIAL FIT Θ	EXPONENTIAL FIT STANDARD ERROR	EXPONENTIAL FIT p
Windows NT 4.0	.0586	.107	.589	n/a	n/a	n/a
Solaris 2.5.1	–.0783	.0565	.177	48.0	35.3	.185
FreeBSD 4.0	–.286	.245	.268	14.7	16.7	.399
RedHat 6.2	–.846	.157	<.001	4.72	.706	<.001

An alternative approach is to use the Laplace factor trend test,[6] shown in the right-hand panels of Figure 2. The top and bottom dotted lines indicate the 95-percent significance values for increasing and decreasing rates of vulnerability finding. The Laplace factor indicates only a statistically significant increase in reliability at the very end of the first three data sets. Because of the censoring at the end of the data set, we cannot consider this increase reliable. Again, there is a significant trend with Red Hat 6.2.

Out of the four programs shown, only one shows a significant trend. Though suggestive, this trend could also be due to confounding factors other than bug depletion. In particular, Red Hat 7.0 was released six months after Red Hat 6.2, potentially leading to reduced interest in finding bugs in Red Hat 6.2. The available data do not allow us to test this hypothesis. Based on this data, we cannot definitively reject the hypothesis that the rate of vulnerability finding in programs is constant over time, and we certainly cannot confirm that it decreases as the program ages.

Examining individual programs is only one way to examine the data. I've also attempted to analyze the data by examining the number of vulnerabilities that were introduced in a given time period, with similarly inconclusive results. More details, including detailed methodology, sources of error, and sensitivity analysis, are available in the full version of this article.[7]

I see several likely avenues for future research. First, obtaining more precise measurements for a larger group of vulnerabilities to confirm the rate measurements would be valuable. Second, it would be useful to start with a known but undisclosed group of security vulnerabilities and measure their rate of rediscovery, thus giving us a direct measurement of rediscovery rate. Andy Chou and colleagues[8] have done this for bugs in general. However, because vulnerabilities rarely manifest themselves in normal operating environments and therefore can be hard to find, their data is not directly usable in this context. Finally, having better measurements of the number and cost of intrusions due to undisclosed vulnerabilities would be useful. □

Acknowledgments

Thanks to Phil Beineke, Christopher Cramer, Theo de Raadt, Kevin Dick, Lisa Dusseault, Ed Felten, Eu-Jin Goh, Pete Lindstrom, Nagendra Modadugu, Vern Paxson, Stefan Savage, Kurt Seifried, Hovav Shacham, Scott Shenker, Adam Shostack, Dan Simon, Terence Spies, the members of the Bugtraq mailing list, and the anonymous reviewers at the Workshop for Economics and Information Security for their advice and comments on this work. Thanks also to Dirk Eddelbuettel and Spencer Graves for help with fitting the Weibull distribution using R. Finally, thanks to Steve Purpura for information about IE releases. The original version of this article appeared at the Third Workshop on Economics and Information Security.

References

1. H.K. Browne et al., *A Trend Analysis of Exploitations*, tech. report, Univ. of Maryland and Carnegie Mellon Univ., 2000.
2. B. Schneier, "Full Disclosure and the Window of Vulnerability," *Crypto-Gram*, 15 Sept. 2000, www.counterpane.com/cryptogram-0009.html#1.
3. R.C. Jeffrey, *The Logic of Decision*, Univ. of Chicago Press, 1965.
4. R.D. Luce and H. Raiffa, "Utility Theory," *Games and Decisions*, Dover, 1989, pp. 12–38.
5. A.L. Goel and K. Okumoto, "Time-Dependent Error Detection Rate Model for Software and Other Performance Measures," *IEEE Trans. Reliability*, vol. 28, no. 3, Aug. 1979, pp. 206–211.
6. D.R. Cox and P.A.W. Lewis, *The Statistical Analysis of a Series of Events*, Chapman and Hall, 1966.
7. E. Rescorla, "Is Finding Security Holes a Good Idea?" *Bugrate*, Sept. 2004, www.rtfm.com/bugrate.html.
8. A. Chou et al., "An Empirical Study of Operating Systems Errors," *Proc. Symp. Operating Systems Principles*, ACM Press, 2001, pp. 73–88.

Eric Rescorla *is principal engineer of RTFM, a security consulting company. His research interests include communications security protocols, anonymous communications, and evidence-based security. Rescorla has a BS in chemistry from Yale. He is a member of the ACM, the American Association for the Advancement of Science, and has served on the Internet Architecture Board since 2002. Contact him at ekr@rtfm.com.*