# Is finding security holes a good idea?

Eric Rescorla
RTFM, Inc.
ekr@rtfm.com

## Abstract

*A large amount of effort is expended every year on finding and patching security holes. The underlying rationale for this activity is that it increases welfare by decreasing the number of vulnerabilities available for discovery and exploitation by bad guys, thus reducing the total cost of intrusions. Given the amount of effort expended, we would expect to see noticeable results in terms of improved software quality. However, our investigation does not support a substantial quality improvement—the data does not allow us to exclude the possibility that the rate of vulnerability finding in any given piece of software is constant over long periods of time. If there is little or no quality improvement, then we have no reason to believe that that the disclosure of vulnerabilities reduces the overall cost of intrusions.*

## 1 Introduction

An enormous amount of effort is expended every year on finding, publishing, and fixing security vulnerabilities in software products.

- The Full Disclosure [1] mailing list, dedicated to the discussion of security holes, had over 1600 postings during the month of September alone.
- The ICAT [2] vulnerability metabase added 1307 vulnerabilities in 2002.
- Microsoft Internet Explorer 5.5 alone had 39 published vulnerabilities in 2002.

Clearly, many talented security professionals are expending large amounts of time on the project of finding and fixing vulnerabilities. This effort is not free. It comes at a cost to the companies funding the effort as well as a significant opportunity cost since these researchers could be doing other security work instead of finding vulnerabilities. Given all of this effort, we should expect to see some clearly useful and measurable result. The purpose of this paper is to measure that result.

The basic value proposition of vulnerability finding is simple: **It is better for vulnerabilities to be found and fixed by good guys than for them to be found and exploited by bad guys.** If a vulnerability is found by good guys and a fix is made available, then the number of intrusions—and hence the cost of intrusions—resulting from that vulnerability is less than if it were discovered by bad guys. Moreover, there will be fewer vulnerabilities available for bad guys to find. Thus, the project is to find the vulnerabilities before the bad guys do. [1]

In this paper we attempt to determine whether vulnerability finding is having a measurable effect. The value proposition just stated consists of two assertions:

1. It is better for vulnerabilities to be found by good guys than bad guys.
2. Vulnerability finding increases total software quality.

In Sections 3 and 4, we consider the first assertion. In Sections 5 and 6 we consider the second assertion. In Section 7 we use the results of the previous sections to address the question of whether vulnerability finding is doing any measurable good.

Any attempt to measure this kind of effect is inherently rough, depending as it does on imperfect data and a number of simplifying assumptions. Since we are looking for evidence of usefulness, where possible we bias such assumptions in favor of a positive result—and explicitly call out assumptions that bias in the opposite direction. Thus, the analysis in this paper represents the best case scenario that we were able to make for the usefulness of vulnerability finding that is consistent with the data and our ability to analyze it.

## 2 Previous Work

The impact of vulnerability finding has been the topic of endless discussion on mailing lists, newsgroups, and at security conferences. In general, actual data on exploitations as a result of disclosure has been rare. The major exception is Browne et al.'s [3] work on the rate of exploitation. Browne et al. found that the total number of exploits increases roughly as the square root of time since disclosure.

There is an extensive literature on software reliability but it is primarily targeted towards large scale

1. Another rationale typically cited for vulnerability disclosure is that embarrassment pressures software vendors into producing more secure software. There is no good empirical data for such an effect and some anecdotal data that vendors are unresponsive to such embarrassment. In this paper we focus only on the immediate costs and benefits of vulnerability discovery and disclosure.

industrial software. The literature on software reliability mostly focuses on large fault-tolerant systems, not on personal computer systems. Moreover, such studies typically focus on all faults, not on security vulnerabilities.

Chou et al. [4] measured the rate of bug finding and fixing in the Linux and BSD kernels but did not distinguish between vulnerabilities and other bugs. They did not attempt to fit a parametric model, but instead used Kaplan-Meier estimation and did not attempt to compute rediscovery probabilities. Their estimate of bug lifetime (mean=1.8 years) is somewhat shorter than ours, but they see a generally similar curve. Unfortunately, it is not possible to directly compare Chou et al.'s results with ours because we have extensive and unmeasurable censoring (i.e., if there are vulnerabilities that persist past the study period, our technique does not know about them at all). However, for the two programs for which we have long time baselines, thus partially ameliorating the censoring (NT 4.0 and Solaris 2.5.1), we find a much slower decay curve than found by Chou et al.

The theoretical work on bug finding is also sparse. In [5] and [6] Anderson presents a theoretical argument using reliability modeling that suggests that a large number of low probability vulnerabilities favors the attacker rather than the defender because it is easier to find a single bug than to find all of them. Thus, the defender needs to work much harder than a dedicated attacker in order to prevent a single penetration. In a related paper, Brady et al. [7] argue that reducing bugs through testing quickly runs into diminishing returns in large systems once the most obvious bugs (and hence vulnerabilities) are removed.

Anderson et al. do not, however, address the question of disclosure or whether attempting to find vulnerabilities is worthwhile. Answering these questions requires addressing empirical data as we do in this paper. As far as we know, we are the first to do so.

## 3 The life cycle of a vulnerability

In order to assess the value of vulnerability finding, we must examine the events surrounding discovery and disclosure. Several authors, including Browne et al. [3], and Schneier [8] have considered the life cycle of a vulnerability. In this paper, we use the following model, which is rather similar to that described by Browne.

- Introduction—the vulnerability is first released as part of the software.
- Discovery—the vulnerability is found.
- Private Exploitation—the vulnerability is exploited by the discoverer or a small group known to him.
- Disclosure—a description of the vulnerability is published.
- Public Exploitation—the vulnerability is exploited by the general community of black hats.
- Fix Release—a patch or upgrade is released that closes the vulnerability.

These events do not necessarily occur strictly in this order. In particular, Disclosure and Fix Release often occur together, especially when a manufacturer discovers a vulnerability and releases the announcement along with a patch. We are most interested in two potential scenarios, which we term *White Hat Discovery* (WHD) and *Black Hat Discovery* (BHD).

### 3.1 White Hat Discovery

In the White Hat Discovery scenario, the vulnerability is discovered by a researcher with no interest in exploiting it. The researcher then notifies the vendor—often he is an employee of the vendor—and the vendor releases an advisory along with some sort of fix. Note that it is of course possible for an advisory to be released prior to a fix but this is no longer common practice. During the rest of this paper, we will assume that fixes and public disclosures occur at the same time. In this scenario, Disclosure and Fix Release happen simultaneously, as 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.

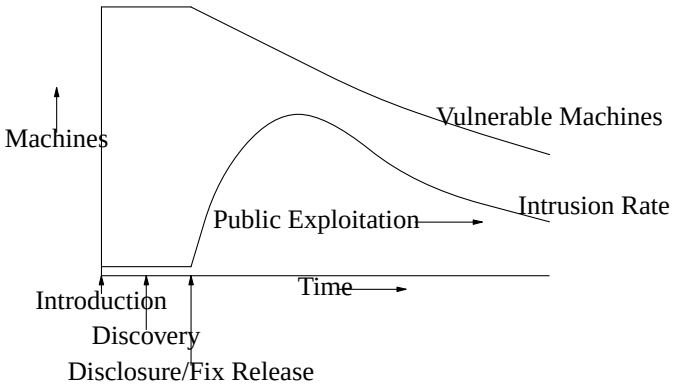


**Figure 1** White Hat Discovery process when disclosure and fix release occur together

Figure 1 shows the sort of process we expect to see. The bottom curve shows the number of intrusions as a function of time. As the vulnerability is not known to attackers prior to Disclosure, there are no intrusions up to this time. At Disclosure time the Public Exploitation phase begins and we start to see intrusions. The rate of

intrusions increases as the knowledge of how to exploit the vulnerability spreads. Eventually, people fix their machines and/or attackers lose interest in the vulnerability—perhaps due to decreasing numbers of vulnerable machines—and the number of intrusions goes down.

The top line shows the fraction of potentially vulnerable machines. We can model this as roughly constant up until the time of Disclosure. We are assuming that a fix is released at Disclosure time and therefore the number of vulnerable systems starts to decline at this point. In most situations, the time scale of this decline is very long, with substantial numbers of machines still vulnerable months after the disclosure of the vulnerability [9]. In environments where a vulnerability is very actively exploited (the Code Red worm, for instance), there is more selective pressure and fixing occurs more rapidly [10].

The rate of intrusion is the result of the interaction of two processes: the level of interest in exploiting the vulnerability and the number of vulnerable machines. The level of interest in exploiting the vulnerability is at least partially determined by the tools available. Browne et al. [3] report that when only the description of the vulnerability is available the rate of intrusion is relatively low but that it increases dramatically with the availability of tools to exploit the vulnerability. Whether the eventual decline in intrusions is a result of a decreasing number of vulnerable systems or of attackers simply moving on to newer and more exciting vulnerabilities is unknown. It seems likely that both effects play a part.

## 3.2 Black Hat Discovery

In the Black Hat Discovery scenario, the vulnerability is first discovered by someone with an interest in exploiting it. Instead of notifying the vendor, he exploits the vulnerability himself and potentially tells some of his associates, who also exploit the vulnerability. The information about the vulnerability circulates in the Black Hat community. Thus, during this period of Private Disclosure, some 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.

At some time after the initial Discovery, someone in the public community will discover the vulnerability. This might happen independently but seems more likely to happen when an attacker uses the vulnerability to exploit some system owned by an aware operator. At this point, the finder notifies the vendor and the process described in the previous section begins (assuming that the announcement of the vulnerability and the release of the fix happen at more or less the same time.)

Figure 2 shows what we expect to see. The primary difference from Figure 1 is that there is a nonzero rate of exploitation in the period between Discovery and Disclosure. It is an open question just how large that rate is. It is almost certainly less than the peak rate after disclosure since the Private Exploitation community is a subset of the total number of attackers. In Figure 2 we have shown it as quite a bit smaller. This seems likely to be the case, as many White Hat security researchers are connected to the Black Hat community and so large scale exploitation would likely be discovered quickly. There is no good data on this topic, but some observers have estimated that the average time from discovery to leak/disclosure is on the order of a month [11].

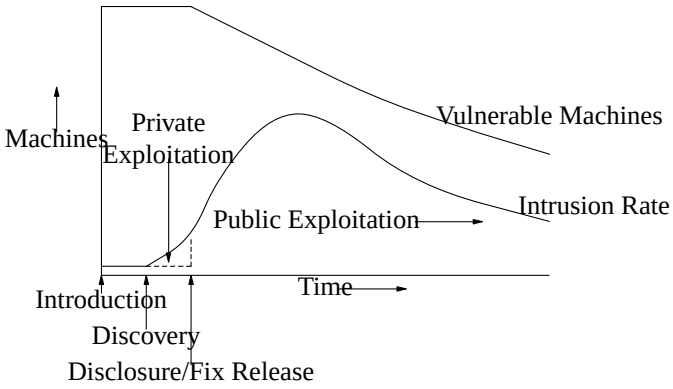


**Figure 2** Black Hat Discovery Process

## 3.3 The cost of a vulnerability

The Discovery of a vulnerability imposes two primary types of costs on users and administrators: fixing and intrusions. If an administrator opts to apply whatever fix is available, this imposes costs both in terms of administrator personnel time and potential system downtime. If an administrator opts not to apply a fix then there is some risk that their systems will suffer an intrusion, which entails emergency response, cleanup, system downtime, and potential destruction, corruption or theft of data.

In Figures 1 and 2, the cost of fixing is represented by the difference between the starting and ending values of the number of vulnerable machines—in other words the number of fixes that are applied, since the cost of fixing is very roughly proportional to the number of machines that must be fixed. For the purposes of this paper, we are principally concerned here with the cost of intrusion. If we assume that fixing rate is roughly the same in both scenarios, then the cost due to fixing will also be similar. The cost of intrusion is related to, but not totally controlled by the area under

the under the intrusion rate curve. Clearly, some machines are more valuable than others and therefore their compromise will have a greater cost. We expect that Black Hats will preferentially attack high value targets and that those high value targets will be fixed relatively quickly. Therefore, we might expect that the machines compromised during the private exploitation period will be more valuable in general than those compromised during the public exploitation period. Overall, this is probably true, however, even high value targets often take days or weeks to fix and therefore there will be substantial exposure right after Disclosure. Because most of the Black Hat community likely does not know about a given vulnerability, the period just after disclosure (when the rest of the Black Hats find out as well) will have plenty of new opportunities for attacks on high value targets. In addition, when a worm is available it will generally not discriminate between high and low value targets.

### 3.4 WHD vs. BHD

It seems intuitively obvious that if one has to choose between the BHD and WHD scenarios, one should prefer WHD, as WHD eliminates the period of Private Exploitation. 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 $C_{pub}$, then the cost of intrusions in the WHD scenario is given by:

$$C_{WHD} = C_{pub} \tag{1}$$

and the cost of intrusions in the BHD scenario is:

$$C_{BHD} = C_{priv} + C_{pub} \tag{2}$$

The advantage of WHD is

$$C_{BHD} - C_{WHD} = C_{priv} \tag{3}$$

Obviously, this approximation is imperfect and probably overestimates the cost difference. First, administrators are likely 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 and therefore Disclosure will likely not produce as large a rise in initial exploitation in the BHD case as in the WHD. However, the simple and conservative approach is to ignore these effects.

## 4 Cost-Benefit Analysis of Disclosure

Imagine that you are a researcher who is the first person anywhere to discover a vulnerability in a widely used piece of software. You have the option of keeping quiet or disclosing the vulnerability to the vendor. If you notify the vendor the WHD scenario of Section 3.1 will follow. If you do not notify the vendor, a Black Hat may independently discover the vulnerability, thus initiating the BHD scenario. However, there is also some chance that the vulnerability will never be rediscovered at all or that it will be rediscovered by another White Hat. In the first case, the cost of disclosure will never be incurred. In the second, it will be incurred later. Either outcome is superior to immediate disclosure.

Consequently, in order to assess whether disclosure is a good thing or not we need to estimate the probability of the following three outcomes:

1. The vulnerability is never rediscovered ($p_{null}$)
2. The vulnerability is rediscovered by a White Hat ($p_{whd}$)
3. The vulnerability is rediscovered by a Black Hat ($p_{bhd}$)

We consider a "worst-case" model: assume that all potential rediscovery is by Black Hats and denote the probability of rediscovery as $p_r$. Consistent with our practice, this simplifying assumption introduces a bias in favor of disclosure. The only way in which failure to disclose does harm is if the vulnerability is rediscovered by a Black Hat. Thus, assuming that vulnerabilities are always rediscovered by Black Hats overestimates the damage done by rediscovery and therefore the advantage of disclosure. Using standard decision theory (see, for instance [12, 13]) we get the choice matrix of Figure 3.

| | Not Rediscovered ($p_{null}$) | Rediscovered ($p_r$) |
|---|---|---|
| Disclose | $C_{pub}$ | $C_{pub}$ |
| Not Disclose | 0 | $C_{pub} + C_{priv}$ |

**Figure 3** Disclose/not disclose decision matrix

Working through the math, we find that the choice to disclose only reduces the expected cost of intrusions if:

$$p_r(C_{priv} + C_{pub}) > C_{pub} \tag{4}$$

In order to justify disclosing, then, the expected cost of excess intrusions in the case of BHD has to be large enough to outweigh the known cost of intrusions incurred by disclosing in the first place. The rest of this paper is concerned with this question.

## 5 From finding rate to $p_r$

In order to attack this problem, we make one further simplifying assumption: that vulnerability discovery is a stochastic process. If there are a reasonable number of vulnerabilities in a piece of software, we don't expect them to be discovered in any particular order, but rather that any given extant vulnerability is equally likely to be discovered next. Note that this assumption is *not* favorable to the hypothesis that vulnerability finding is useful. If, for instance, vulnerabilities were always found in a given order, then we would expect that a vulnerability which is not immediately disclosed will shortly be found by another researcher. However, this simplification is probably approximately correct—since 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 $S$ containing $V_{all}$ vulnerabilities. Over the lifespan of the software, some subset of them $V_{found}$ will be discovered. Thus, the likelihood that any given vulnerability will be discovered during the life of the software is given by:

$$p_{discovery} = \frac{V_{found}}{V_{all}} \tag{5}$$

Similarly, if we pick a vulnerability that has just been discovered, the chance of rediscovery has as its upper bound the chance of discovery:

$$p_r \leq p_{discovery} = \frac{V_{found}}{V_{all}} \tag{6}$$

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

## 6 The Rate of Vulnerability Discovery

We can measure the rate of vulnerability discovery directly from the empirical data. Using that data and standard software reliability techniques we can also derive an estimate for the number of vulnerabilities. The procedure is to fit 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.

For our purposes, it's important to be specific about what we mean by a "piece of software". Real software undergoes multiple releases in which vulnerabilities are fixed and other vulnerabilities are introduced. What we wish to consider here is rather individual releases of software. For instance, when FreeBSD 4.7 was shipped, it had a certain fixed number of vulnerabilities. During the life of the software, some of those vulnerabilities were discovered and patches were provided. If we assume that those patches never introduce new vulnerabilities—which, again, favors the argument for disclosure—then the overall quality of the software gradually increases. We are interested in the rate of that process.

### 6.1 Modeling Vulnerability Discovery

The literature on modeling software reliability is extremely extensive and a large number of models exist. No model has yet been found suitable for all purposes, but the dominant models for software reliability are stochastic models such as the Homogenous Poisson Process (HPP) models and Non-Homogenous Poisson Process models such as Goel-Okumoto (G-O) [14], Generalized Goel-Okumoto (GGO) [15], and S-shaped model [16].

In this context, reliability is defined as the number of failures (vulnerabilities) observed during a given time period. Thus, if a system is getting more reliable, that means that fewer failures are being observed. For simplicity, these models assume that all failures are equally serious. Roughly speaking, there are three classes of models depending on the overall trend of failures:

| Trend | Models |
|---|---|
| Reliability growth | G-O and Generalized G-O |
| Reliability decay followed by growth | Log-logistic/S-shaped |
| Stable reliability | Homogenous Poisson Process |

**Figure 4** Trend and corresponding models (after Gokhale and Trivedi[17])

We are primarily interested in models where reliability is increasing, since only those models predict a finite number of vulnerabilities. If reliability does not increase, then the projected number of vulnerabilities is effectively infinite and the probability of rediscovery in any given time period must be very low. The simplest such model is the Goel-Okumoto Non-Homogenous Poisson Process model, which we describe here.

In the G-O model, the number of vulnerabilities discovered in a single product per unit time $M(t)$ is assumed to follow 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 gets progressively more reliable and the rate of discovery slows down.

Chou et al.'s results [4] provide general confirmation that this is the right sort of model. Their studies of bugs in the Linux kernel found a slow decay curve with a mean bug lifetime of 1.8 years. It is widely believed that security vulnerabilities are more difficult to find than ordinary bugs because ordinary bugs often cause obvious failures in normal operation, whereas vulnerabilities are often difficult to exercise unintentionally and therefore must be found by auditing or other forms of direct inspection. Thus, while we cannot use their results directly, this correspondence provides an indication that this sort of model is correct.

Given the G-O model, the probability that a vulnerability will be discovered in a given time period is thus equal to the fraction of the area under the $p_r(t)$ curve during that time period. We can estimate $p_r(t)$ by the following procedure: First, we fit an exponential of the form $Ae^{-t/\theta}$ to the curve of vulnerability discovery. We can then easily find the total number of vulnerabilities by integrating:

$$N = \int_{t=0}^{\infty} \frac{A}{N} e^{-t/\theta} dt = A\theta \tag{7}$$

In order to compute the probability that a vulnerability will be found over any given time period $(t, t+\Delta t)$, we first normalize by dividing out $N$. This gives us:

$$p_r(t_0,\Delta t) = \int_{t=t_0}^{t_0+\Delta t} \frac{1}{\theta} e^{-t/\theta} dt = e^{-t/\theta} - e^{-(t+\Delta T)/\theta} \tag{8}$$

We are particularly interested in the probability at time $t$ that a vulnerability will be found in the next time period $\Delta t$. Since an exponential is memoryless, this probability is the same no matter what the age of the vulnerability and is given by equation (9).

$$p_r(\Delta t) = 1 - e^{-\Delta T/\theta} \tag{9}$$

Note that this assumes that all vulnerabilities will eventually be found. Again, this assumption is favorable to the argument for disclosure. If there are vulnerabilities in program X which remain unfound because interest in program X wanes, then this model will over-predict $p_r$ and therefore overestimate the left half of equation (4), making disclosure seem more desirable.

## 6.2 Measured Vulnerability Discovery Rates

In order to measure the actual rate of discovery of vulnerabilities, we used the ICAT vulnerability metabase [2]. ICAT is run by NIST, which describes it as follows:

> The ICAT Metabase is a searchable index of computer vulnerabilities. ICAT links users into a variety of publicly available vulnerability databases and patch sites, thus enabling one to find and fix the vulnerabilities existing on their systems. ICAT is not itself a vulnerability database, but is instead a searchable index leading one to vulnerability resources and patch information. ICAT allows one to search at a fine granularity, a feature unavailable with most vulnerability databases, by characterizing each vulnerability by over 40 attributes (including software name and version number). ICAT indexes the information available in CERT advisories, ISS X-Force, Security Focus, NT Bugtraq, Bugtraq, and a variety of vendor security and patch bulletins. ICAT does not compete with publicly available vulnerability databases but instead is a search engine that drives traffic to them. ICAT is maintained by the National Institute of Standards and Technology. ICAT is uses and is completely based on the CVE vulnerability naming standard (http://cve.mitre.org).

ICAT makes the entire database available for public download and analysis, which made it ideal for our purposes. Our analysis is based on the May 19, 2003 edition of ICAT. We downloaded the database and then processed it with a variety of Perl scripts. All statistical analysis was done with R [18]. [2]

### 6.2.1 Vulnerability Discovery Rate by Calendar Time

The simplest piece of information to extract is the rate of vulnerability disclosure over time. ICAT lists a "published before" date for each vulnerability, which represents an upper bound on publication. For vulnerabilities prior to 2001, the "published before" date is the earliest date that the ICAT maintainers could find. For vulnerabilities after 2001, the date is when the vulnerability was added to ICAT. The maintainers attempt to add vulnerabilities within a week of CVE or CAN ID assignment (candidate (CAN) assignment is generally fairly fast but vulnerability (CVE) assignment is often quite slow). In general, "the published before" date appears to be within a month or two of first publication. This introduces a modest amount amount of random noise to our data but otherwise does not substantially impact our analysis (see Section 6.8.2 for sensitivity analysis). Figure 5 shows the rate of vulnerability discovery by month.

2. Researchers interested in obtaining a copy of the raw or processed data should contact the author.

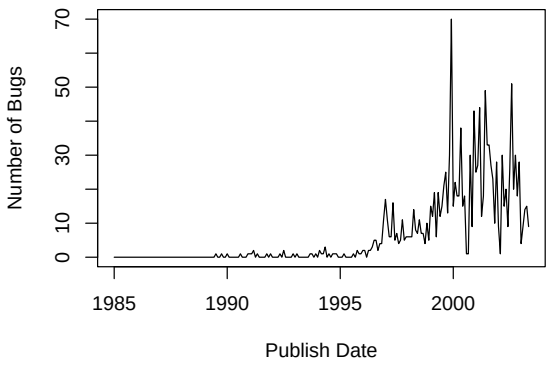


**Figure 5** The rate of vulnerability discovery

As is apparent from Figure 5, there is substantial variation in the rate of disclosure. Part of the variation appears to be due to the fact that batches of vulnerabilities are often found together. For instance, in month 211, a batch of 9 vulnerabilities (out of 51 that month) were found in Bugzilla. The peak at month 179 is less clear, but a large number of vulnerabilities in that month were inserted at 12/31/1999, so most likely this is an artifact of end-of-year cleanup. Cursory analysis yields no other obvious patterns in the variation.

Unfortunately, for our purposes, this representation of the data isn't that useful. Since each program was released on a separate date, the discovery rate by calendar time is a superposition of the discovery curves for a large number of programs. Each program has its own discovery curve with the clock starting at the time of release. To do better we need to take into account the time of introduction of the vulnerability into the code base.

### 6.2.2 Finding Affected Programs

The ICAT database does not list the time when a vulnerability was introduced into the code. However, it does provide the programs and version numbers which are affected by the vulnerability. We can use this as a proxy for the time when the vulnerability was introduced by using the release date of the earliest affected version.

The large number of programs in the ICAT database made finding release dates for all of them prohibitive. Instead, we selected the programs with the largest number of vulnerabilities. Somewhat arbitrarily, we chose 20 vulnerabilities as the cutoff point. This gave us the list of programs in Figure 6. [3] Note that the vulnerability counts here are purely informational because they are aggregated by program, regardless of version. Throughout the analysis we will be handling each version as separate. We were able to find release information for all of the listed programs except AIX, Oracle, Imail, Unixware, Firewall-1, and IOS. These programs were omitted from study.

| Vendor | Program | Vulnerability Count |
|---|---|---|
| * Oracle | Oracle9i Application Server | 20 |
| * Conectiva | Linux | 20 |
| Microsoft | Windows ME | 20 |
| * Ipswitch | Imail | 20 |
| Microsoft | Outlook | 21 |
| Microsoft | Outlook Express | 22 |
| Apple | MacOS X | 22 |
| * Oracle | Oracle9i | 23 |
| ISC | BIND | 25 |
| MIT | Kerberos 5 | 25 |
| * SCO | Unixware | 26 |
| Slackware | Linux | 27 |
| KDE | KDE | 27 |
| Netscape | Communicator | 27 |
| * Check Point Software | Firewall-1 | 29 |
| Mozilla | Bugzilla | 29 |
| * Oracle | Oracle8i | 29 |
| Apache Group | Apache | 32 |
| Caldera | OpenLinux | 35 |
| Microsoft | Windows XP | 36 |
| BSDI | BSD/OS | 37 |
| * Cisco | IOS | 38 |
| Microsoft | SQL Server | 42 |
| Microsoft | Windows 95 | 42 |
| SCO | Open Server | 46 |
| Microsoft | Windows 98 | 47 |
| MandrakeSoft | Linux | 51 |
| Linux | Linux kernel | 54 |
| S.u.S.E. | Linux | 65 |
| OpenBSD | OpenBSD | 68 |
| Sun | SunOS | 68 |
| NetBSD | NetBSD | 70 |
| Debian | Linux | 88 |
| Microsoft | IIS | 100 |
| * IBM | AIX | 122 |
| SGI | IRIX | 133 |
| Microsoft | Windows 2000 | 134 |
| Microsoft | Internet Explorer | 140 |
| HP | HP-UX | 142 |
| FreeBSD | FreeBSD | 152 |
| Microsoft | Windows NT | 171 |
| RedHat | Linux | 183 |
| Sun | Solaris | 192 |

* indicates release data not available.

**Figure 6** Programs under study

### Data Cleansing

Once we had identified the programs we wished to study, we eliminated all vulnerabilities which did not

3. Note, this table was generated before any data cleansing was done, so there may be small misalignments with the later analysis

affect one of these programs. This left us with a total of 1678 vulnerabilities. We manually went over each vulnerability in order to detect obvious errors in the ICAT database. We identified three types of error:

1. Obvious recording errors where the description of the vulnerability did not match the recorded version number or there was some obvious typographical mistake such as a nonexistent version number (n=16). We corrected these before further processing.
2. Vulnerabilities where the textual description read "version X *and earlier*" but the list of affected versions was incomplete (n=96). We tagged these vulnerabilities for future analysis but did not correct them. Note that this source of error makes vulnerabilities appear younger than they are. This creates a false appearance that vulnerability discovery rates decrease more over the age of the vulnerability thus overestimating the value of disclosure.
3. Vulnerabilities where we suspected that there had been a recording error (e.g. through external knowledge) but the database did not show it (n=31) Most of these were insufficently broad program assignments. For instance, CVE-2001-0235 [19] and CVE-1999-1048 [20] describe vulnerabilities that probably affect a broad number of UNIXes but were only assigned to a few. We tagged these vulnerabilities for future analysis but did not correct them.

The most serious problem with the data we found was that many programs were listed with "." as the affected version. This means that ICAT did not know which versions were affected. In some cases, "." was listed along with explicit version numbers, in which case we simply ignored the "." entry. In cases where the only affected version was "." that program was ignored (though the vulnerability was retained as long as some valid program could be found). To the extent to which this procedure introduces error it makes vulnerabilities appear younger than they in fact are and therefore biases the data in favor of the effectiveness of vulnerability finding. Section 6.8.1 contains the results of our attempts to compensate for this problem.

## 6.3 Estimating Model Parameters

Using the ICAT data, we attempted to derive the model parameters. There are a number of confounding factors that make this data difficult to analyze. First, many vulnerabilities appear in multiple programs and versions. Thus, it is difficult to talk about "vulnerability density" in a given program, since neither programs nor vulnerabilities are totally independent. Second, vulnerabilities were introduced both before and during the study period, and so we have both left and right censoring.

In order to provide robustness against these confounding factors, we analyze the data from two angles:

- The program's eye view in which we examine all the vulnerabilities in a given version of a specific program, regardless of when the vulnerability was introduced.
- A vulnerability's eye view in which we examine the time from vulnerability introduction to vulnerability discovery, regardless of which programs it was in.

We first consider data from the perspective of each affected program.

## 6.4 A Program's Eye View

The obvious question to ask is "What is the rate at which vulnerabilities are 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, most introduced in that release. We can then ask: how many of those vulnerabilities are found as a function of time. Because this gives us a fixed starting point, this approach is susceptible to right but not left censoring. However, it has two major problems:

1. Because the same vulnerabilities appear in multiple programs and multiple versions, it is not possible to analyze every program as if it were an independent unit.
2. Any individual program is not likely to have that many vulnerabilities, thus giving us low statistical power.

In order to keep the amount of interaction to a minimum, we focus on four program/version pairs, two open source and two closed source. These pairs were chosen to minimize interaction, while still allowing us to have a large enough data set to analyze. For instance, we chose only one of the Windows/IE group, despite there being large numbers of vulnerabilities in both Windows and IE, because the two sets of vulnerabilities are highly related.

Note that in this case the age being measured is the age of the program, not the age of the vulnerability. Thus, if a vulnerability was introduced in Solaris 2.5 but is still in Solaris 2.5.1, we're concerned with the time after the release of Solaris 2.5.1, not the time since first introduction in Solaris 2.5. Note that if the bug finding process is not memoryless, this biases the results so that bug finding appears more effective than it actually is, as investigators have already had time to work on the bugs that were present in earlier versions. Conservatively, we ignore this effect.

| Vendor | Program | Version | # Vulns | Release Month |
|---|---|---|---|---|
| Microsoft | Windows NT | 4.0 | 111 | August 1996 |
| Sun | Solaris | 2.5.1 | 106 | May 1996 |
| FreeBSD | FreeBSD | 4.0 | 39 | March 2000 |
| RedHat | Linux | 7.0 | 51 | August 2000 |

**Figure 7** Programs for analysis

Figure 8 shows the vulnerability discovery rate for each program as a function of age. For the moment, focus on the left panels, which show the number of vulnerabilities found in any given period of a program's life (grouped by quarter). Visually, there is no apparent downward trend in finding rate for Windows NT 4.0 and Solaris 2.5.1, and only a very weak one (if any) for FreeBSD. This visual impression is produced primarily by the peak in quarter 4. RedHat 7.0 has no visually apparent downward trend.

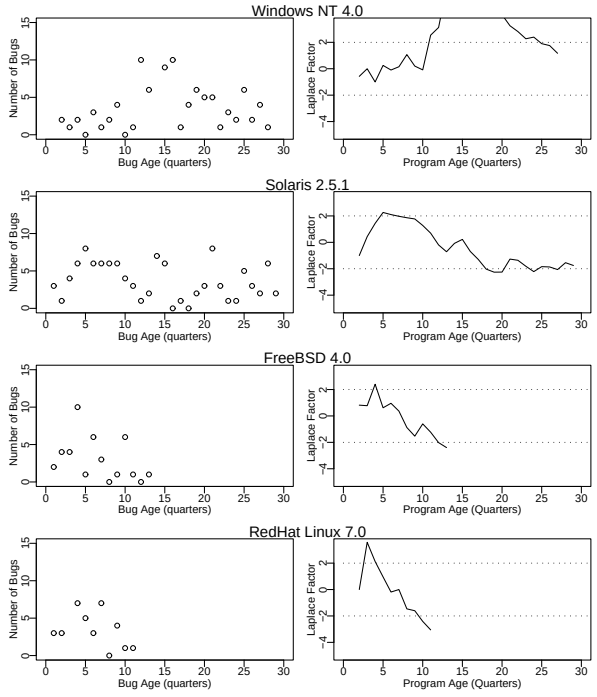


**Figure 8** Vulnerability discovery rate by programs

Moving beyond visual analysis, we can apply a number of 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 Goel-Okumoto model and fit an exponential using non-linear least-squares. Neither fit reveals any significant trend. In fact, the data for Windows NT 4.0 is so irregular that the non-linear least-squares fit for the exponential failed entirely with a singular gradient. The results of these regressions are shown in Figure 9. Note the extremely large standard errors and p values, indicating the lack of any clear trend.

| | Linear Fit | | | Exponential Fit | | |
|---|---|---|---|---|---|---|
| Program | Slope | Std. Err. | p | $\theta$ | Std. Err | p |
| Windows NT 4.0 | .0586 | .107 | .589 | - | - | - |
| Solaris 2.5.1 | -.0743 | .0529 | .171 | 48.5 | 34.8 | .174 |
| FreeBSD 4.0 | -.308 | .208 | .167 | 12.1 | 11.0 | .292 |
| RedHat 7.0 | -.627 | .423 | .172 | 9.72 | 9.35 | .325 |

**Figure 9** Regression results for program cohort data

An alternative approach is to use the Laplace factor trend test [21]. The Laplace factor test assumes that data is being produced by a Poisson process and checks for homogeneity of the process. Laplace factor values with greater than 1.96 (indicated by the top dotted lines) indicate significantly decreasing reliability (increased rates of vulnerability finding) at the 95% level (two-tailed). Values below -1.96 (the bottom dotted line) indicate significantly decreased rates of vulnerability finding. The results of the Laplace factor test are shown in the right hand set of panels. The Laplace factor only indicates a statistically significant increase in reliability at the very end of each data set. In view of the amount of censoring we are observing, this cannot be considered reliable.

Based on this data, we cannot reject the hypothesis that reliability of these programs is constant overall, and certainly cannot confirm that it is increasing. If anything, it appears that reliability decreases somewhat initially, perhaps as a result of increased program deployment and therefore exposure to vulnerability discoverers.

## 6.5 A Vulnerability's Eye View

The other option is to start from the point of vulnerability introduction and ask what the probability is that a vulnerability will be discovered at any time *t* after that. In order to analyze the data from this perspective, we first need to determine when a vulnerability was first introduced. We used the following procedure to determine the introduction date for each vulnerability:

1. Determine the first version of each program to which the vulnerability applies. We did this by numeric comparison of version numbers. Where affected versions were listed as "and earlier", "previous only" (applies to versions before the listed version), etc. we used the earliest known version.
2. Look up the release date for each such earliest version. Where release dates were not available, the program was ignored.
3. Select the program/version pair with the earliest release date (there may be multiples, for instance when a program appears in multiple Linuxes). If no such date was available, we ignored the

vulnerability. This problem occurred for approximately 110 vulnerabilities. An alternative procedure would be to look for a later version of the same package. Section explores this approach with essentially similar results to those presented here.

This procedure is susceptible to a number of forms of error. We briefly introduce them here, and will discuss them further in the course of our analysis. First, the ICAT database has errors. As noted previously, we corrected them where possible. However, there are almost certainly less obvious residual errors. In particular, vulnerability finders often seem to only check recent versions for a vulnerability. Thus, versions 1.0 through 4.0 may be affected but only versions 3.0-4.0 might be reported. We would expect this effect to make vulnerabilities look more recent than they in fact are. We ignore this effect, which makes vulnerability lifetime appear shorter and therefore the vulnerability depletion rate appear higher, thus favoring disclosure.

Second, it is not clear how to categorize vulnerabilities which appear in multiple programs. A vulnerability may affect more than one program for a number of reasons:

1. It may be present in a common ancestor of both programs, such as BSD 4.4 vulnerabilities which appear in both NetBSD or FreeBSD.
2. A package may be included in multiple operating systems. For instance, GCC is included in both Linux and *BSD.
3. Multiple programmers may have made the same mistake. For instance, both KDE and Internet Explorer failed to check the X.509 Basic Constraints extension and are listed in CAN-2002-0862 [22].

Situation 3 appears to happen quite rarely so we simply ignore it and treat it as part of our experimental error. In situation 1 it seems appropriate to treat the first appearance in *any* package as the date of vulnerability introduction. In situation 2, we might wish to treat programs which are packages but not part of the operating system separately. We have not currently done so but consider it a topic for future work (see Section 6.8.4).

We also detected 24 vulnerabilities where the earliest known publication date preceded the introduction date. In some cases, this is no doubt a result of vulnerabilities which are present in some version lumped under "unknown". In others, they are simply data errors. We discarded these vulnerabilities. At the end of this procedure, 1391 vulnerabilities remained.

Finally, in some situations we were unable to get precise release dates. The finest granularity we are concerned with is a month and so as long as we know the release month that is sufficiently precise. In <15 cases, all prior to 1997, we were able to get dates only to year resolution. We arbitrarily assigned them to the month of June, because that was the middle of the year. However, as our year cohorts begin in 1997, this should have no effect on the analysis of this section or of Section 6.4. Figure 10 shows the number of vulnerabilities by time of introduction. Note the two big peaks in mid 1996 and early 1998. These correspond to the release of Windows NT 4.0 and IIS 4.0 respectively.

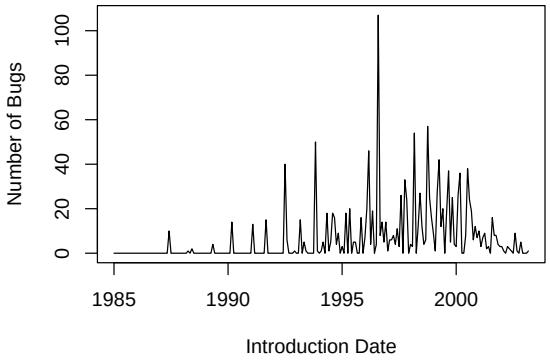


**Figure 10** Number of vulnerabilities by year of introduction

### 6.5.1 Vulnerability Production rate by Age

Figure 11 shows the number of vulnerabilities found by age of the vulnerability at time of discovery. There is a fairly clear downward trend, which might be indicative of depletion. However, note that this data is extremely subject to sampling bias, as our data is from the limited time window of 1997-2002. Because new programs are introduced during this period, and therefore cannot have vulnerabilities older then five or so years, we would expect to see fewer older vulnerabilities than newer vulnerabilities. In addition, if popularity of programs is a partial factor in how aggressively programs are audited, we would expect interest in programs to wane over time, thus producing the appearance of increasing reliability. Finally, as is evident from Figure 10, the rate of vulnerability introduction is highly variable over time. As all these factors tend to overestimate the depletion rate, fitting the data directly is problematic.

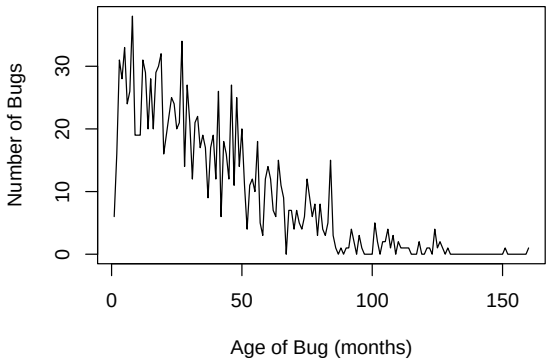


**Figure 11** Discovered vulnerabilities by age

### 6.5.2 Discovery rate by year cohort

One way to minimize the sampling bias mentioned above is to look only at vulnerabilities from fairly narrow age cohorts, such as a single year. Figure 12 shows the distribution of vulnerability ages for vulnerabilities introduced in the years 1997-2000, in the same format as Figure 8.

Once again, we can see that there is no obvious visual trend, except possibly in vulnerabilities introduced in 1999. Our regressions confirm this. Both linear regression and our exponential fit show no significant negative trend for any year but 1999. The results are shown in Figure 13. Note the very large standard errors and p values for every regression except than 1999 and the linear regression for 2000 (which is subject to a large amount of censorship).

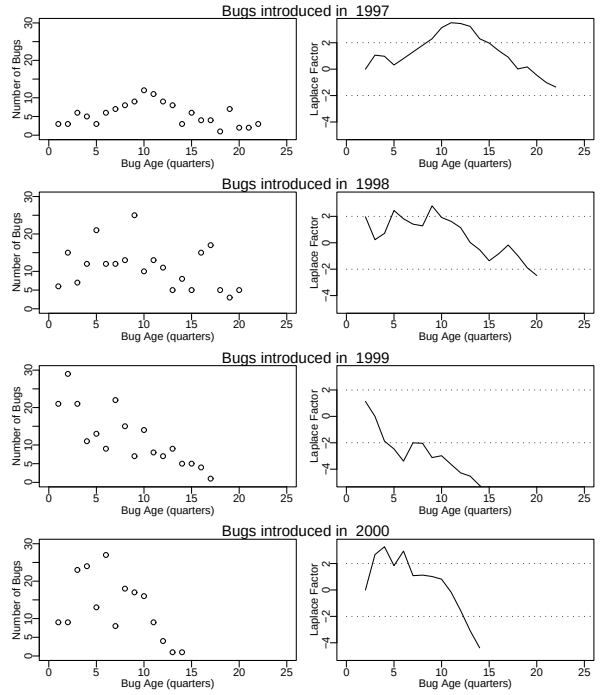


**Figure 12** Vulnerability discovery rate by age cohorts

| | Linear Fit | | | Exponential Fit | | |
|---|---|---|---|---|---|---|
| Year | Slope | Std. Err. | p | $\theta$ | Std. Err | p |
| 1997 | -.107 | .102 | .307 | 68.4 | 87.4 | .443 |
| 1998 | -.319 | .218 | .160 | 40.8 | 33.5 | .240 |
| 1999 | -1.25 | .218 | <.01 | 9.46 | 1.93 | <.01 |
| 2000 | -1.04 | .493 | .0565 | 16.7 | 11.6 | .175 |

**Figure 13** Regression results for age cohort data

Similarly, the Laplace Factor only shows significant increases in reliability for 1999 and the last 2 quarters of 1998 and 2000 (where it crosses the negative dotted confidence boundary). The last 2 quarters of the 1998 and 2000 data should be disregarded because the data from those quarters is extremely subject to the same kind of sampling bias. For instance, only programs which were published in the early part of 2000 could possibly have vulnerabilities that were as old as 36-42 months by the end of the study period.

The lack of a significant trend in the cohort data should make us quite skeptical of the claim that there is indeed a trend towards increasing reliability. The data does not allow us to discard the hypothesis that the vulnerability finding rate is essentially constant.

## 6.6 What if we ignore the bias?

In the previous sections, we analyzed cohort-sized subsets of the data in order to attempt to remove bias. However, this also had the effect of reducing the size of our data set and therefore the statistical power of our techniques. What happens if we ignore the bias and simply use the entire data set as-is? As we indicated previously, this overestimates the amount of vulnerability depletion, thus providing a result biased in favor of disclosure.

As Figure 11 shows a generally downward trend, we should either fit a Goel-Okumoto model or an S-shaped Weibull model (to account for the initial rise in the number of vulnerabilities discovered.) The G-O model was fit via least squares estimation and the Weibull model was fit using maximum-likelihood estimation. Figure 14 shows the result of fitting these two trend lines.

Both trend lines are superimposed on Figure 11. Figure 15 shows the exponential model parameters and Figure 16 shows the estimated model parameters for the Weibull model. Note that although the exponential fit is not perhaps as nice visually as we might like, we were able to validate it by working backwards to the implied number of original vulnerabilities and then using Monte Carlo estimation to simulate the vulnerability finding process. The results are fairly similar to our measured curve, indicating that the exponential is a reasonable model. Note that the larger data set here

allows us to aggregate the data into months instead of quarters, thus the exponential scale constants are a factor of three larger than with the cohort regressions of Figures 9 and 13.

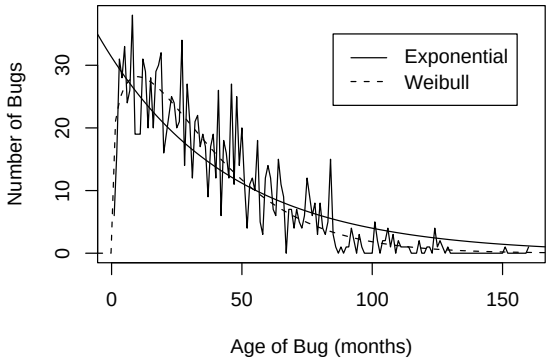


**Figure 14** Fitted overall vulnerability decay curves

| | |
|---|---|
| $A$ | 31.3 |
| $\theta$ | 48.6 |
| $N = A\theta$ | 1521 |

**Figure 15** Exponential fit parameters for vulnerability age at discovery time

| | |
|---|---|
| $\alpha(shape)$ | 1.25 |
| $\beta(scale)$ | 36.6 |

**Figure 16** Weibull fit parameters for vulnerability age at discovery time

Figure 17 shows the cumulative distribution functions for the probability that a vulnerability will be found by time $t$ given these fit parameters.

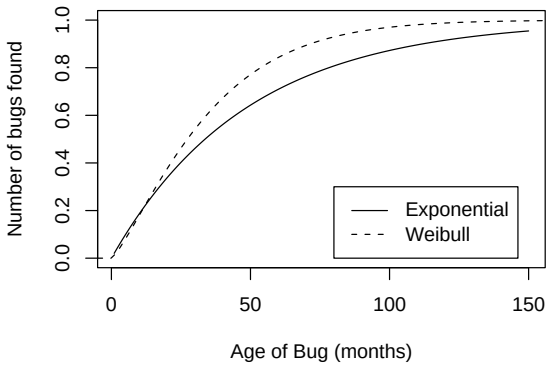


**Figure 17** Probability that a vulnerability will be found

As mentioned above, this estimate of $p_r$ is very likely to be an overestimate because of sampling bias.

## 6.7 Are we depleting the pool of vulnerabilities?

We are now in a position to come back to our basic question from Section 5: to what extent does vulnerability finding deplete the pool of vulnerabilities. The data from Sections 6.5 and 6.4 provides only very weak support for a depletion effect. Even under conditions of extreme bias, the highest depletion estimate we can obtain from Section 6.5.1, is that the half-life for vulnerabilities is approximately 2.5 years. However, no depletion whatsoever cannot be ruled out given this data. In that case, the probability of rediscovery $p_r$ would be vanishingly small.

The conclusion that there is fairly little depletion accords with anecdotal evidence. It's quite common to discover vulnerabilities that have been in programs for years, despite extensive audits of those programs. For instance, OpenSSH has recently had a number of vulnerabilities [23] that were in the original SSH source and survived audits by the OpenSSH team.

## 6.8 Sources of Error

In any analysis of this type there are a large number of potential sources of error. We discuss the known sources in this section.

### 6.8.1 Unknown Versions

As indicated in Section 6.2.2, a number of the program versions were listed as ".", meaning "unknown version". In approximately 15% of our data points time of first introduction was therefore somewhat indeterminate. We discarded these data points in our initial analysis. As a check for bias, we manually investigated all of these vulnerabilities and were able to determine version numbers for approximately 100 (8% of the total data set). We reran our regressions with largely similar results to the original data set. With this change, the 2000 cohort linear regression is now barely significant ($p = .0446$) instead of barely insignificant ($p = .0565$).

### 6.8.2 Bad Version Assignment

One problem with the introduction version algorithm described in Section 6.5 is that some programs do not issue version numbers in strict sequence. For instance, FreeBSD for some time maintained the 3.x and 4.x branches in parallel. Because we use version number as our primary sort, in some unusual cases this can make vulnerabilities appear younger than they in fact are, thus making disclosure look more attractive.

For instance, a vulnerability which appeared only in FreeBSD 4.1 and FreeBSD 3.5 would be recorded as "introduced" in 3.5, even though 4.1 was released

previously. Note, however, that a bug which also appeared in 3.4 would get the correct introduction date because 3.4 preceded 4.1. Handling this issue correctly is difficult because in some sense these code branches are different programs. In practice, there is no significant impact on the results because this misassignment occurs rarely. We double-checked our results by comparing the oldest **known** version to the assigned version and only found this form of misassignment in 10 cases. Rerunning our overall regressions using the earliest known introduction dates produced essentially equivalent results. Note that these assignment problems have no real impact on the analysis in Section 6.4.

### 6.8.3 Announcement Lag

From 2001 on, ICAT started using the date that entries were entered in ICAT as the "published before" date, replacing the previous standard of "earliest mention". This is a potential source of bias, making vulnerabilities appear older upon publication than they in fact are. In order to assess the magnitude of this potential bias, we subtracted 2 months (which seems to be the maximum common lag) from each vulnerability age and repeated the overall exponential regression. This produced a rate constant of $\theta = 44.2$, which is approximately 10% lower than our original estimate of the rate constant, but still within the confidence boundaries.

When we reanalyzed the age cohort data with this lag, year 2000 also becomes significant. The program cohorts still show no significance with this lag. Other manipulations of the data show slightly differing significance patterns. It's common to see significance under *some* manipulations of the data and this kind of instability to exact data set choice is generally a sign that what is being observed are artifacts of the analysis rather than real effects. Nevertheless, in the future we would like to determine exact publication dates for each vulnerability in order to confirm our results.

### 6.8.4 Vulnerability Severity

The severity of the vulnerabilities in ICAT varies dramatically. Some vulnerabilities are trivial and some are critical. It is possible that serious vulnerabilities are discovered quickly whereas non-serious ones leak out slowly. In order to test this hypothesis, we repeated our regressions using only the vulnerabilities that were ranked as having "High" severity in ICAT. This produced a slightly slower depletion rate ($\theta = 53.6$) and the individual age and program cohort regressions showed little evidence of depletion. With the exception of 1999, the linear trend lines are not significantly non-zero—and in some cases non-significantly positive. In addition, the Laplace factors are generally within confidence limits. Thus, if anything, severe vulnerabilities are depleted more slowly than ordinary vulnerabilities.

### 6.8.5 Operating System Effects

Some vulnerabilities in ICAT are listed as corresponding to a given operating system revision but actually correspond to a piece of software that runs on that version (e.g., Exchange on Windows NT 4.0), but are *not* listed under the actual program name as well. This produces a false introduction date corresponding to the introduction date of the operating system instead of the package. Inspection suggests that this is a relatively small fraction of the vulnerabilities and there is no good reason to believe that this would be a source of systematic bias rather than random error. However, we are currently considering ways of controlling for this sort of error. One approach we are considering is to manually go through the database and discover the exact status of each vulnerability. We have not done this yet, however.

A related problem is vulnerabilities which are listed both under a operating system and non-operating system packages. In many cases (e.g., OpenSSL), these programs are bundled with a given operating system. In such cases, as long as we have release dates for both the package and the operating system (which we do for most of the popular packages), then we are generally able to determine the correct vulnerability introduction date. In some cases, when the package is not bundled, however, this will yield an incorrect introduction date. Re-analyzing the data with data points listed under both an Operating Systems and a non-Operating System package ($n = 225$) removed yielded essentially the same results. [4]

### 6.8.6 Effort Variability

One possible problem with this analysis is that the amount of effort expended on any given program may vary throughout its lifetime, thus affecting the rate at which vulnerabilities are found. Unfortunately, the only metric we currently have for the amount of effort being expended is the number of vulnerabilities found, which is our measured variable. Thus, we cannot control for this effect. The overall level of bug finding, however, appears to have been fairly stable (though there is much inter-month variation) over the period 1999-2003, as shown in Figure 5.

4. This assignment is a little rough because we were not able to identify the nature of some rare packages and a few were clear errors (e.g., "IBM A/UX"). However, this should not significantly change the results.

### 6.8.7 Different Vulnerability Classes

Another source of error is the possibility that new vulnerability classes are being discovered. Thus, for instance, it may be that as soon as Foo Overflow errors are discovered, a rash of them are found in IE, but then all the easy ones are quickly found and no more Foo Overflow errors are found. This would be an instance of strong non-randomness in vulnerability discovery order. There doesn't seem to be enough data to repeat our analysis stratified by vulnerability category, but the overall ICAT statistics [24] suggest that the pattern of vulnerabilities found has been fairly constant over the period 2001-2003.

### 6.8.8 Data Errors

Aside from the other sources of error listed in the previous sections, there is the general problem of errors in the ICAT database leading to incorrect conclusions. We have attempted to identify all the sources of systematic error and believe that the manual procedure followed in Section 6.2.2 allows us to remove the obvious entry errors. However, ICAT is manually maintained and therefore we should expect that there will be errors that made their way into the analysis. We do not believe that this invalidates the basic conclusions. However, a larger data set with more precise data might yield evidence of effects which this study did not have sufficient power to resolve.

## 7 Is it worth disclosing vulnerabilities?

Before we address the question of whether vulnerability finding is worthwhile, we first address a more limited question: is disclosure worthwhile, even ignoring the cost of vulnerability finding? The combination of equation (4) and the analysis of Section 6 can be used to answer this question. In order to do this, we need to consider the two possibilities from Section 6.7:

- Vulnerabilities are not being depleted.
- Vulnerabilities are being slowly depleted, with a half-life of about three and half years.

We will examine these two cases separately.

### 7.1 No Depletion

If there is no vulnerability depletion, then there are effectively an infinite number of vulnerabilities and $p_r$ approaches zero. If this is correct, then the right half of equation (4) is always greater than the left half and disclosing vulnerabilities is always a bad idea, no matter what the relative sizes of $C_{priv}$ and $C_{pub}$. Thus, if there is no depletion, then disclosing vulnerabilities is always harmful, since it produces new intrusions (using the newly disclosed vulnerabilities) and there is no compensating reduction in intrusions from vulnerabilities which would have been discovered by black hats.

In practice, $p_r = 0$ is clearly unrealistic, since vulnerabilities *are* occasionally rediscovered. What is more likely is that the assumption of Section 5 that vulnerabilities are found in random order is not strictly correct. Rather, some vulnerabilities are sufficiently obvious that they are rediscovered but there is a large population of residual vulnerabilities which is not significantly depleted. In this case, the assumption $p_r$ would be non-homogenous but small or zero for most vulnerabilities.

#### 7.1.1 How many additional intrusions are created by disclosure?

If vulnerability disclosure increases the number of intrusions, than we would like to estimate the size of the increase. As shown in Figure 3, if we disclose we expect to incur cost $C_{pub}$. The expected value for the cost of intrusions if we don't disclose is $p_r(C_{priv} + C_{pub})$. If, as we have argued, $p_r$ is vanishingly small, then the additional cost of intrusions created by disclosure is $C_{pub}$—the entire cost of intrusions that resulted from our disclosure.

### 7.2 Slow Depletion

Consider the possibility that vulnerabilities are being slowly depleted. If we assume that all vulnerabilities will eventually be found, then $p_r = 1$ and equation (4) would seem to indicate that disclosure was a good idea. However, equation (4) ignores the effect of time. In general, if a vulnerability is not yet being exploited, most people would prefer that if a vulnerability must be disclosed it be disclosed in the future rather than now (see the end of this section for discussion of vulnerabilities that are already being exploited).

In welfare economics and risk analysis, this concept is captured by the discount rate $d$ [25]. The basic idea is that a dollar today is only worth $1 - d$ next year (to feed your intuition, think of the interest you could have earned if you had the dollar today). There is a lot of controversy about the exact correct discount rate, but standard values range from 3% to 12% annually. Given an annual discount rate $d$, we can compute the value any number of months in the future using a simple exponential function with rate constant $log(1 - d)/12$. Multiplying by $C_{BHD}$ we get equation (10).

$$C_{BHD} e^{-\frac{log(1-d)t}{12}} \tag{10}$$

In order to evaluate the effect of disclosure, we need to compute the expected value of the cost, which is simply the cost of a disclosure at time $t$ multiplied by

the probability of a disclosure at time $t$, integrated over all values of $t$. [5]

Using an exponential model, this gives us Equation (11).

$$\int_{t=0}^{t=\infty} C_{BHD}\, p_r(t) * e^{-\frac{\log(1-d)t}{12}}\, dt \qquad (11)$$

We can evaluate this by plugging in the fit parameters from Figure 15. Depending on the choice of $d$, this gives us values ranging from $.89C_{BHD}$ ($d = 3\%$) to $.66C_{BHD}$ ($d = 12\%$) with our exponential model. In other words, unless black hat disclosure is 12% worse ($1/.89 \approx 1.12$) than white hat disclosure (for $d = 3\%$) then the societal effect in terms of intrusions is actually worse than for white hat disclosure, even if we ignore the cost incurred in finding the vulnerabilities. The situation is more complicated with the Weibull model because it is not memoryless and therefore the shape of the $p_r$ curve depends on the age of the vulnerability. Table 18 shows some sample values depending on the age of the vulnerability using the fit parameters from Figure 16. Thus, for instance, the expected present cost of future disclosure of a vulnerability found at age 12 months is $.93C_{BHD}$ assuming a 3% discount rate.

| | Discounted Cost as fraction of $C_{BHD}$ | |
|---|---|---|
| Vulnerability Age (Months) | 3% | 12% |
| 0 | .92 | .72 |
| 12 | .93 | .75 |
| 24 | .93 | .77 |
| 36 | .94 | .78 |
| 48 | .94 | .79 |
| 60 | .94 | .79 |

**Figure 18** Expected values for disclosure costs with Weibull model

For two reasons, it seems unlikely that black hat disclosures are 10% worse than white hat disclosures (let alone 52% worse if we assume an exponential distribution and $d = 12$). Any significant number of exploitations should alert administrators to the existence of the vulnerability. This is particularly true in modern networking environments where network forensics systems are common. Moreover, wide dissemination of a vulnerability in the black hat community is likely to result in a leak to the white hat community. It has been claimed that the average time from discovery to disclosure is about one month [11]. Second, the process of patch deployment is very slow [9] and therefore vulnerabilities persist long past disclosure, increasing the total number of intrusions.

Note that if a vulnerability is already being exploited in the black hat community, then the cost/benefit analysis is somewhat different, since disclosure has an immediate benefit as well as an immediate cost. However, in the case where we are not aware of any such explanation, as the probability of rediscovery is low, the a priori probability that the vulnerability has already been discovered is correspondingly low. Therefore, in the absence of information that the vulnerability is previously known, we should behave as if we are the first discoverers.

## 7.3 The Bottom Line

If the effort we are currently investing in vulnerability finding is paying off, it should be yielding some measurable results in terms of decreasing defect count. In order to ensure that we were able to see any such effect, we have made a number of assumptions favorable to the usefulness of vulnerability finding:

1. All vulnerability rediscovery is by black hats (Section 4)
2. All vulnerabilities are eventually rediscovered (Section 6.1)
3. We ignore the fact that vulnerabilities in obsolete versions are often listed only for newer versions (Section 6.5)
4. We ignore the sampling bias introduced by our limited study period (Section 6.5.1)

Despite this built-in bias, we find little evidence that vulnerability disclosure is worthwhile, even if the cost of the vulnerability finding process itself is ignored. The "best case" scenario supported by our data and assumptions is that the process of vulnerability finding slightly increases reliability. However, even if there is such a increase, when we factor in discounting our model suggests that there is no net benefit in disclosing vulnerabilities. The bottom line, then, is that based on the evidence we cannot conclude that vulnerability

5. Note that we are assuming here that the real cost of some future intrusion is the same as that of a current intrusion, which implicitly assumes that the number of vulnerable machines is the same as well. This is of course not true, but then some programs become *less* popular instead of more over time, and for the overall argument we don't know what kind of program we are dealing with. In addition, the real cost of patching, which likely represents a substantial fraction of the cost of a published vulnerability and which we are ignoring here, goes up with the size of the installed base. It would be interesting to repeat this analyis for some hypothetical bug in a number of real commonly deployed pieces of software for which we know the popularity curve and the patching cost.

finding and disclosure provides an increase in software security sufficient to offset the effort being invested. [6]

## 8 Policy Implications

Given that the data does not support the usefulness of vulnerability finding and disclosure, how should we allocate our resources?

### 8.1 Deemphasize Vulnerability Finding

Clearly, we do not have enough evidence to definitively say that vulnerability finding is not a good idea. However, given the amount of effort being invested in it, not being able to find a significant effect is troublesome. At this point, it might pay to somewhat deemphasize the process of finding vulnerabilities and divert that effort into recovering from the vulnerabilities that are found, through user education and improved technology for response.

### 8.2 Improve Data Collection

We can only have limited confidence in these results because the data set we are working from is in quite bad shape. To a great extent this is the result of the somewhat informal nature of vulnerability reporting and database maintenance. If we are to have a definitive answer to the question of whether vulnerability finding is useful we will need better data. If we start recording data more carefully and formally now, in five years or so we will be in a much better position to answer this kind of question.

### 8.3 Improve Patching

A major reason why vulnerabilities are so dangerous even after patches are available is that the rate of patching is so slow [9]. If automatic patching were more widely used, then the $C_{pub}$ would decrease and disclosure would look more attractive. Conversely, cracker's ability to quickly develop and deploy malware based on disclosures or patches increases $C_{pub}$ and makes disclosure look less attractive. Even with automatic patching, a fast worm, such as Staniford et al.'s Warhol Worm [26], released shortly after disclosure would do a large amount of damage. Accordingly, any measures which improve the rate of patching and make fast development of malware more difficult are likely to pay off.

6. Note that if vulnerabilities can be fixed without disclosing them, then the cost/benefit equation changes and it's possible, though not certain, that vulnerability finding and fixing pays off. This kind of private fixing is generally impossible with Open Source systems but may be possible with Closed Source, for instance by releasing regular service releases with patches in them. It is unclear whether one could make such releases hard enough to reverse engineer that they did not leak information about the vulnerabilities they fixed. Rumors persistently circulate in the security community that black hats indeed do reverse engineer binary patches in order to discover the vulnerabilities they fix.

### 8.4 Limitations of This Analysis

Our analysis here has focused on vulnerability finding as a method of reducing the number of intrusions. However, it should be noted that it may be valuable in other contexts. In particular, research into new classes of vulnerability, as well as automatic discovery of vulnerabilities and defenses against attack may very well be worthwhile. Our argument here is primarily limited to the routine publication of vulnerabilities that are new instances of known classes of vulnerability.

In addition, even if vulnerability discovery and disclosure does not increase overall social welfare, it almost certainly advances the interests of certain constituencies. In particular, discoverers of new vulnerabilities may be able to profit from them, either via publicity or by sale of associated services such as vulnerability information. However, we need to consider the possibility that the interests of these constituencies are in opposition to those of society as a whole, making this a classic negative externality situation.

## 9 Conclusions and Future Work

If finding security defects is a useful security activity, then it should have some measurable effect on the software security defect rate. In this paper, we have looked for such an effect and only found very weak evidence of it. In the best case scenario we are able to make, the total defect count has a half life of approximately 2.5 years. However, our data is also consistent with there being no such effect at all. In either case, the evidence that the effort being spent on vulnerability finding is well spent is weak.

We see several likely avenues for future research. First, it would be good to attempt to obtain more precise measurements for a larger group of vulnerabilities in order to confirm our rate measurements. It would also be valuable to cross-check our results using another database of vulnerabilities, perhaps with better data on the date of publication. If better data were available, it would allow us to discriminate more finely between alternative reliability models. In particular, it would be interesting to fit to a BAB model [7]. Second, it would be useful to quantify the total decrease in welfare with better measurements of the number of and cost of intrusions which are due to undisclosed vulnerabilities. Finally, it would be useful to start with a known group of security vulnerabilities all present at the same time and measure the rate at which they are independently rediscovered, thus avoiding the left-censoring problems inherent in this work.

## Acknowledgements

The author would like to thank 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 to Dirk Eddelbuettel and Spencer Graves for help with fitting the Weibull distribution using R. Thanks to Steve Purpura for information about IE releases.

## References

[1] *Full Disclosure Mailing List.*
`http://lists.netsys.com/mailman/listinfo/full-disclosure`

[2] *ICAT Metabase.*
`http://icat.nist.gov/`

[3] Browne, H. K., Arbaugh, W. A., McHugh, J., and Fithen, W. L., "A Trend Analysis of Exploitations," *University of Maryland and CMU Technical Reports* (2000).

[4] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D., "An Empirical Study of Operating Systems Errors," *Symposium on Operating Systems Principles* (2001).

[5] Anderson, R.J., "Why Information Security is Hard — an Economic Perspective," *Proceedings of the Seventeenth Computer Security Applications Confernece*, pp. 358-365, IEEE Computer Security Press (2001).

[6] Anderson, R.J., *Security in Open versus Closed Systems — The Dance of Boltzmann, Coase and Moore.*
`http://www.ftp.cl.cam.ac.uk/ftp/users/rja14/toulouse.pdf`

[7] Brady, R.M., Anderson, R.J., and Ball, R.C., *Murphy's law, the fitness of evolving species, and the limits of software reliability.*
`http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-471.pdf`

[8] Schneier, B., "Full disclosure and the window of vulnerability," *Crypto-Gram* (September 15, 2000).
`http://www.counterpane.com/crypto-gram-0009.html#1`

[9] Rescorla, E., "Security Holes... Who cares?," *Proceedings of the 13th USENIX Security Symposium* (August 2003).

[10] Moore, D., Shannon, C.,, and Claffy, K., "Code-Red: a case study on the spread and victims of an Internet worm," *Internet Measurement Workshop* (2002).

[11] Zalewski, M., "[Full-Disclosure] Re: OpenSSH - is X-Force really behind this?," *Posting to Full-disclosure mailing list* (September 22, 2003).
`https://www1.ietf.org/mailman/options/saad/ekr%40rtfm.com`

[12] Jeffrey, R.C., *The Logic of Decision*, University of Chicago Press, Chicago (1965).

[13] Luce, R.D., and Raiffa, H., "Utility Theory" in *Games and Decisions*, pp. 12-38, Dover, New York (1989).

[14] Goel, A.L., and Okumoto, K., "Time-Dependent Error-Detection Rate Model for Software and Other Performance Measures," *IEEE Transactions on Reliability*, R-28, 3, pp. 206-211 (August 1979).

[15] Lyu, M.R. (ed.), *Handbook of Software Reliability*, McGraw Hill (1996).

[16] Yamada, S., Ohba, M., and Osaki, S., "S-Shaped Reliability Growth Modeling for Software Error Detection," *IEEE Transactions on Reliability*, R-32, 5, pp. 475-478 (December 1983).

[17] Gokhale, S., and Trivedi, K.S., *A Time/Structure Based Software Reliability Model*, 8, pp. 85-121 (1999).

[18] Ihaka, R., and Gentleman, R., "R: A Language for Data Analysis and Graphics," *Journal of Computational and Graphical Statistics.*

[19] *CVE-2001-0235.*
`http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0235`

[20] *CVE-1999-1048.*
`http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-1048`

[21] Cox, D.R., and Lewis, P.A.W.L., *The Statistical Analysis of a Series of Events*, Chapman and Hall (1966).

[22] Benham, M., "IE SSL Vulnerability," Bugtraq posting (August 5, 2002).

[23] OpenSSH, *OpenSSH Security Advisory: buffer.adv* (2003).
`http://www.openssh.com/txt/buffer.adv`

[24] *ICAT Bug statistics.*
`http://icat.nist.gov/icat.cfm?function=statistics`

[25] Kammen, D.M., and Hassenzahl, D.M., *Should We Risk It?*, Princeton University Press, Princeton, NJ (1999).

[26] Staniford, S., Paxson, V., and Weaver, N., "How to 0wn the Internet in Your Spare Time," *Proceedings of the 12th USENIX Security Symposium* (August 2002).