CODING BY CONTRACT: WHY THE FINE PRINT MATTERS

COLIN PERCIVAL

1. Introduction

If you ask most operating system developers what the role of their security team is, you will probably receive a rather simple answer — something along the lines of "find security problems, and fix them". As I have discovered during my tenure on the FreeBSD security team, while some problems are very straightforward, there are many other issues which raise difficult questions, both about how and where they should be fixed, and whether they should be fixed at all.

While these questions are worth considering — and, since they are not widely recognized, *documenting* — on their own merits, they also provide some insight into how software should be developed in a world where security is increasingly important and — thanks largely to the rise of "open source" licenses — code is often re-used for purposes unimagined by its original authors.

For clarity of terminology, in this paper a *problem* can refer to a software bug, design error, configuration, potentially dangerous default behaviour in a program, etc. — essentially, anything which one might wish to fix. Some *problems* are *security flaws* (or equivalently *security issues*), while most are fixed by developers without ever involving the a security team.

2. What is a security flaw?

The first decision which must be made concerning a problem reported to or discovered by a security team is whether it is — or rather, should be treated as — a security issue. Once it is clear under what conditions the problem can be exploited and what an attacker can accomplish by exploiting the problem, this becomes a question of policy.

The policy applied by the FreeBSD security team is that any privilege escalation, disclosure of potentially sensitive information, or denial of service will be treated as a security issue, with the exception of local denials of service. Denials of service which can only be exploited by authorized users of the system being attacked are excluded for two

major reasons; first, since a local user can always cause a performance degradation by consuming resources, it is hard to define when a user is causing "too much" of a performance impact on a system[1]; and second, since local users are typically easily identifiable, local denial of service attacks can usually be addressed via administrative or legal means. Nevertheless, particularly egregious local denials of service — for instance, where passing an incorrect value to a system call can result in a kernel panic — are usually corrected as "Errata", which are equivalent to security advisories aside from the distinction of receiving more testing since they are not required to be kept confidential.

Unlike FreeBSD, many Linux vendors do treat local denial of service problems as security issues. We believe that this difference is largely due to the nature of Linux as a kernel developed almost independently of the "userland" which surrounds it: While the concept of 'upgrading to a newer kernel' to fix a problem is foreign to FreeBSD users[2], it is widespread in Linux; since local denial of service problems almost always result from kernel bugs, correcting them in Linux involves nothing beyond advising users to upgrade to the latest kernel.

On the other hand, some operating systems do not treat any denial of service problems as security issues. To take a recent example, when a flaw was found in the pf firewall which allowed a sequence of three packets to cause a kernel panic in some firewall configurations, FreeBSD issued a security advisory [4], but while the problem was fixed in OpenBSD and the fix was merged to the `OPENBSD_3_7` and `OPENBSD_3_8` CVS branches corresponding to the previous two releases, no advisory was issued and no note was made of the issue on the 'release errata & patch list' web pages [5, 6]. From one perspective, it is correct to argue that denials of service are not security problems — after all, information can neither be obtained nor damaged by an attacker — but most companies which rely upon the Internet for their revenue would most likely disagree.

Finally, there are some people who even consider some potential privilege escalation problems to not be security issues. In the aftermath of the author's public announcement of a side channel exploitable in the shared cache of Intel processors with Hyper-Threading [7], an Intel spokesman remarked that 'in order for this particular exploit to be launched in a system, the system has to already have been compromised' [9] — ignoring the many systems which have untrusted local

[1]Note that any sufficiently advanced local denial of service attack is indistinguishable from a legitimate user who is trying to get a lot of work done.

[2]FreeBSD users are always encouraged to keep their userland and kernel synchronized.

users — while many argued that this problem should not be corrected since 'there are many far simpler attacks'[3].

3. Would anybody *REALLY* do that?

In some instances, even if a problem is serious enough to be considered a security issue, the conditions required for it to be exploited are sufficiently unlikely that the issue may reasonably be ignored. While this paper was being written, one such problem was reported to the FreeBSD Security Team by Jens Schweikhardt.

The problem in question concerned the handling of "here-documents" in the FreeBSD command interpreter `sh(1)`. When redirecting a program's input, rather than specifying a file from which to read the input, it is possible to place the input into the middle of a shell script; the interpreter reads this input, passes it to the program in question, and then continues executing on the line after said input ends. If desired, multiple here-documents can be provided and handed to the program being executed on different file descriptors. What Jens discovered was that if multiple here-documents were being used, and the first here-document used backtick-expansion, then the contents of subsequent here-documents would be executed rather than being correctly handled as data.

At first glance, this seems like an obvious and serious security issue: Data is being treated as code. However, the sequence of events required for an attacker to exploit this bug is bizarre in the extreme: A program must read data from an untrusted source (the attacker), use this data to construct a shell script — stopping to somehow check that the backtick expansions in the first here-document are safe[4] — and then execute that shell script. Since the FreeBSD Security Team was unable to imagine any situation where such an odd sequence of events would occur (if nothing else, one would expect the untrusted data to be written to a file, rather than passed as a here-document in a newly created script), the decision was made not to handle the problem as a security issue.

A similar, if perhaps more debatable, case concerns Daniel Bernstein's *qmail* mail transfer agent [1]. In May 2005, Georgi Guninski reported an integer overflow problem which permitted a remote attacker to obtain root privilege — but only by causing the *qmail* process being attacked to read over a gigabyte of data from the attacker [3]. Bernstein

[3]We note that this is a self-fulfilling prophecy: If local privilege escalation problems are not corrected, then there will indeed be many such attacks available.

[4]If the backtick expansions in the first here-document were not verified to be safe, then an attacker could execute his code using those, and the problem of accidental execution of the second here-document would not make the situation any worse.

responded to this by stating that 'Nobody gives gigabytes of memory to each qmail-smtpd process' [2] — a statement which, in our opinion, might be slightly over-optimistic in its assessment of the competency of most systems administrators.

Nevertheless, whether Bernstein's assessment of users' plausible behaviour is accurate or not, the point remains that in the absence of documentation stating under what conditions a program is to be used — or, put another way, indicating what sort of crazy things a user can do — the handling of security issues inevitably depends upon a judgement call made by the author or maintainer of the code in question.

4. Where is the security flaw?

When we move from considering a single application to considering a system involving multiple independently produced components, the questions become even harder: Now instead of wondering if we can rely upon the user not behaving in a strange manner, we must consider whether applications will use other applications, utilities, or libraries in strange ways.

Returning briefly to the aforementioned side channel attack against Intel processors with Hyper-Threading [7], we note that there are two entirely independent components, each of which could be argued to be at fault: first, the processors which allow information about the addresses of data and code accessed to "leak", and second, the cryptographic libraries and other programs which manipulate sensitive data in non-oblivious ways[5]. Given the lack of specifications concerning such matters — to the author's knowledge, prior to May 2005 no cryptographic library ever documented whether or not it was vulnerable to an attacker able to measure its memory access patterns, nor did any processor vendor document whether or not such information was leaked — the decision of which component was at fault became a judgement call as to whether it is reasonable to expect such memory access patterns to be disclosed[6].

While in the case of Hyper-Threading both components could be argued to be at fault, there can equally be circumstances where *neither*

[5]While the most obvious target for attackers is code where cryptographic keys are used and might be inadvertently disclosed, there is likely to be other sensitive information equally susceptible to disclosure; for example, the act of finding a node in a B-tree is likely to disclose information about the key being located.

[6]The author's personal opinion is that in light of the large body of pre-existing code which relies upon memory access patterns remaining secret for its security, it is more reasonable to find fault with the recent introduction of processors which violate this assumption; but this is an argument based on history rather than security.

component can be said to be insecure, yet a security flaw still exists in the system considered as a whole. Consider as a first component an application which downloads some files from a remote server (which the attacker is assumed to be able to compromise), verifies that they have an appropriate structure (say, that they contain lines of the form `key$value`, the `value` only contains letters and digits, and that no line is too long), performs some processing of these files using the standard UNIX text file manipulation utilities (sort, comm, join, ...) to produce a list containing a subset of the `value` fields, reads those values into shell variables, and uses them as the names of files to be created, modified, and/or deleted[7]. Given that the `value` fields were verified to be "safe", there should be no danger introduced by using them as file names.

Now consider as a hypothetical second component a version of the `sort(1)` utility which has the following bug: Any time it should output the line "`aaaaaaaaaaaaaaaaaaaa`", it instead outputs the line "`/etc/master.passwd`". A system which uses these two components together will clearly be insecure, since the attacker can modify or delete the system password file, causing at very least a denial of service; but where is the security flaw?

While the bug described in the hypothetical second component is certainly a very peculiar bug, the mere fact of producing incorrect output would not normally constitute a security flaw; since this hypothetical utility will never execute arbitrary code provided by an attacker, it is hard to justify claiming that it is insecure. On the other hand, the first component certainly doesn't contain a security flaw — it doesn't contain *any* flaw. Instead, the security flaw results from an "innocent bug" in one component together with how that component is used by an independent, and completely bug-free, second component.

5. Coding by contract

To see how these questions can be made somewhat less troublesome, consider the nature of an application or library interface: The fundamental essence of such an interface is a contract specifying the responsibilities of the caller and of the function(s) being called. At one extreme, a C header file providing function declarations offers a contract specifying the number and types of variables provided by the caller, the manner in which they are passed (which usually defaults to a standard calling convention) and the type of variable returned by the function;

[7]The author's Portsnap utility [8] for secure updating of the FreeBSD ports tree operates approximately in this manner.

more commonly there is also some vague and/or out of date documentation about how the code is to be used. As we have seen, such limited specifications create havoc for security officers: Without knowing how code is permitted to be used, it is impossible to determine except as a judgement call whether a problem exists in the code in question or instead in the unrealistic expectations of the person using it.

At the other extreme, formal specifications attempt to express the precise behaviour expected of a component into a rigid, and often machine-parsable, language. Formal specifications have a poor record of real-world usage — the vast majority of software developers either lack the expertise to use formal specification languages or find it too burdensome to update formal specifications every time code is modified or re-purposed — but even when used correctly, formal specifications provide at best proof that code is not operating as specified, not proof that the code in question is insecure. Just as when specifications are informal, or lacking altogether, the question of whether a problem should be considered to be a security issue can end up with a judgement call about how the code in question is likely to be used by an entirely independent component.

Rather than providing a single set of specifications — detailed or not, formal or informal — we believe that the needs of security dictate that two sets of specifications should be provided: First, the conventional specifications which state how code *should* behave, and second, a new set of specifications which state how code is *guaranteed* to behave.

6. The fine print

The concept of stating separate "expected" and "guaranteed" behaviour is not new to the field of computing; it has been used for decades in the analysis of algorithms, and as any undergraduate student should be able to tell you, running a naïve *quicksort* on data from an untrusted source is not a good idea if you want to get an answer quickly. In our case, however, we are not considering the range of possible running times of an algorithm over varying input; instead, we are considering varying *bugs*.

The key insight is that while the correct functioning of a system may depend upon every part of the system operating as expected, the security of a system usually depends only upon individual components not failing in particularly spectacular ways. To take the earlier example of a buggy `sort(1)` and a shell script using it, the correct functioning of the system relies upon `sort(1)` correctly sorting its lines of input; in contrast, the security (at least as far as the particular hypothetical

problem described earlier is concerned) relies solely upon the assumption that every line of output from `sort(1)` was a line of input to the utility.

Adding this fine print to code specifications has three benefits. First, by clearly stating what behaviour is guaranteed, and under what conditions it is guaranteed, the task of a security officer is made far simpler since it is immediately clear whether a problem needs to be handled as a security issue.

Second, this provides a mechanism for developers to telegraph to the users of their work an indication of which features, or under what conditions, the code is most reliable. Developers are often very much aware of the deficiencies in their work, but hesitate to publicly document such things as "I was drunk when I wrote this, and don't know how it works"; in such circumstances, simply not mentioning the code in question in the security guarantee would allow users to avoid relying upon that code where it might have security implications.

Finally, by separating features into two tiers — those which are believed to work, and those which are guaranteed to work — the task of debugging is immediately simpler. Fewer supported features means less code and fewer bugs; and given a fixed number of eyeballs, concentrating them on the code which really matters is going to make those bugs far more shallow than they would be otherwise.

References

1. D.J. Bernstein, *qmail*.
   `http://cr.yp.to/qmail.html`
2. D.J. Bernstein, *The qmail security guarantee*.
   `http://cr.yp.to/qmail/guarantee.html`
3. G. Guninski, *64 bit qmail fun*, May 2005.
   `http://www.guninski.com/where_do_you_want_billg_to_go_today_4.html`
4. FreeBSD Project, *Security Advisory FreeBSD-SA-06:07.pf*, January 2006.
   `http://security.freebsd.org/advisories/FreeBSD-SA-06:07.pf.asc`
5. OpenBSD Project, *OpenBSD 3.7 Errata*.
   `http://www.openbsd.org/errata37.html`
6. OpenBSD Project, *OpenBSD 3.8 Errata*.
   `http://www.openbsd.org/errata38.html`
7. C. Percival, *Cache missing for fun and profit*, BSDCan'05, May 2005.
   `http://www.daemonology.net/hyperthreading-considered-harmful/`
8. C. Percival, *Portsnap*, October 2004.
   `http://www.freebsd.org/cgi/cvsweb.cgi/src/usr.sbin/portsnap/`
9. J.G. Spooner, *Student Raises the Specter of an Attack on Intel Chips*, eWEEK.com, May 2005.
   `http://www.eweek.com/article2/0,1895,1815954,00.asp`

IRMACS Centre, Simon Fraser University, Burnaby, BC, Canada
E-mail address: `cperciva@freebsd.org`