Schneier on Security

← Attacking Online Poker Players Tor User Identified by FBI →

Security Vulnerabilities of Legacy Code

An interesting research paper documents a "honeymoon effect" when it comes to software and vulnerabilities: attackers are more likely to find vulnerabilities in older and more familiar code. It's a few years old, but I haven't seen it before now. The paper is by Sandy Clark, Stefan Frei, Matt Blaze, and Jonathan Smith: "Familiarity Breeds Contempt: The Honeymoon Effect and the Role of Legacy Code in Zero-Day Vulnerabilities," Annual Computer Security Applications Conference 2010.

Abstract: Work on security vulnerabilities in software has primarily focused on three points in the software life-cycle: (1) finding and removing software defects, (2) patching or hardening software after vulnerabilities have been discovered, and (3) measuring the rate of vulnerability exploitation. This paper examines an earlier period in the software vulnerability life-cycle, starting from the release date of a version through to the disclosure of the fourth vulnerability, with a particular focus on the time from release until the very first disclosed vulnerability.

Analysis of software vulnerability data, including up to a decade of data for several versions of the most popular operating systems, server applications and user applications (both open and closed source), shows that *properties extrinsic to the software play a much greater role in the rate of vulnerability discovery than do intrinsic properties such as software quality.* This leads us to the observation that (at least in the first phase of a product's existence), software vulnerabilities have different properties from software defects.

We show that the length of the period after the release of a software product (or version) and before the discovery of the first vulnerability (the 'Honeymoon' period) is primarily a function of familiarity with the system. In addition, we demonstrate that legacy code resulting from code re-use is a major contributor to both the rate of vulnerability discovery and the numbers of vulnerabilities found; this has significant implications for software engineering principles and practice.

Tags: academic papers, patching, security engineering, vulnerabilities, zero-day

Posted on December 17, 2013 at 7:10 AM • 16 Comments

Comments

David • **December 17, 2013 8:40 AM**

That's interesting and all, but I'm mostly just excited to see someone use "different ... from" correctly (instead of the commonly botched "different than", or the inexplicable "different to").

Nick P • **December 17, 2013 8:41 AM**

" In addition, we demonstrate that legacy code resulting from code re-use is a major contributor to both the rate of vulnerability discovery and the numbers of vulnerabilities found"

That particularly is well-known and often ignored by developers. They need the functionality quickly. However, they'd rather not understand all the security implications of how they use it. They more commonly just run some tests on it.

One could take the theme further and say that legacy is the software security problem. Ground up, layered, POLA architectures typically have few to no security problems during their application. The general purpose, complex systems architectures that most developers build solutions on lead to many vulnerabilities. These systems are inherently hard to secure, but familiar and offer plenty of existing 3rd party code/tools. Maybe the decision to use them is at the root of the problem as people take the productivity while accepting the risk.

Brian M. • **December 17, 2013 9:15 AM**

Familiarity breeds *expertise*, at least when it comes to hardware and software. So of course old software is more "vulnerable," because people have been working on cracking it for more time.

Clive Robinson • **December 17, 2013 9:31 AM**

I read this paper some time ago, what triggered the memory was this statement,

> *This leads us to the observation that (at east in the first phase of a product's existence), software vulnerabilities have different properties from software defects.*

I strongly disagreed with the way it was written.

Logicaly a vulnerability is a defect and thus are the same. What differs is not the defect but how the software is viewd by attackers.

We know that attackers --like defenders-- are resource limited, --with the exception of Gov emps-- the attacker is going to utilise their time and tools against what they percieve is the most worthwhile target. The metrics of what constitutes "worthwhile" are multi facited and thus may be difficult to predict. However what ever the metrics are they don't effect the existance of a defect,

just the likelyhood of it being detected and if possible exploited.

Clive Robinson • **December 17, 2013 9:57 AM**

@ Nick P,

> *That particularly is well-known and often ignored by developers. They need the functionality quickly. However, they'd rather not understand all the security implications of how they use it. They more commonly just run some tests on it.*

This is a major bugbear I noticed some years ago (prior to major movment of apps to web front ends). After looking into it what became clear was the following,

1, code was written to specification acceptance of "good data" and thus declaired "functional code".

2, Input error checking was not implicit in the functional code, it was moved "left" to the front of the program well away from the functional code.

3, As input errors were detected it was the error checking that was augmented to correct the problem, not the functional code.

4, Code re-use of the functional code frequently occured without the seperate error checking code.

On investigating further there appeared to be a disconnect in the programers thinking about functional code and error checking code. It could almost be seen as a legacy of "waterfall development" thinking wrapped up with the idea that errors belonged to the program not individual functional blocks within it. The resulting "up front" error checking code was thus a rats nest mixture of error checking that did belong to the program and error checking that belonged to the individual functional blocks and was thus difficult if impossible to seperate.

One major downside of this was when a program became "webified" the "front end" got chopped off and converted to javascript etc that ran in the browser. Along with it went a big chunk of input error checking. The result was just like code reuse all the functional code vulnerabilities on the server were exposed and in quite a few cases exploited...

You would have thought that those writing OO progs would be aware of the difference between error corection that belonged to the program and that which belonged to a functional block. But no all error correction still gets left shifted upto the "front end"...

Vinzent • **December 17, 2013 10:26 AM**

> Logicaly a vulnerability is a defect and thus are the same. What differs is not the defect but how the software is viewd by attackers.

Personally I tend to see it the same way as you, but from a purely technical point of view, defects and vulnerabilities are indeed different beasts.
A defect is where a system deviates from its requirements (e.g. "strings up to a length of 255 bytes must be accepted"), a vulnerability is something *missing* in the requirements, e.g. "Ok. Fine. And what shall happen if a string of more than 255 bytes is passed?".

And that is also a different view by the ultimate customer. The customer usually only tells you what functions it wants the system to perform, not what it shall *not* do, at best, the customer just expects that it doesn't.

It's like a pact with the devil:

Customer: "I want to be a famous rock-star."

*poof"

Devil: "Now you are John Lennon. - Date is December 8, 1980."

(Defects: None as the requirements are fulfilled. But at the same time being dead soon is probably not what you intended when expressing the requirement, so I'd count that as vulnerability.)

Another David • **December 17, 2013 11:51 AM**

@David
There is nothing incorrect about "different than". See http://stancarey.wordpress.com/2011/04/20/different-from-different-than-different-to (quoted approvingly by Mark Liberman at Language Log).

Jeremy • **December 17, 2013 2:32 PM**

Even if you argue that every vulnerability is a defect, not every defect is a vulnerability: many defects may impair a program's usefulness for legitimate users without offering anything useful for attackers, either.

Extreme example: a program that does absolutely nothing is entirely defective, but is not "vulnerable" in any meaningful sense.

So it could still be true that newly-minted code has more defects than older code without having significantly more vulnerabilities: that would imply that vulnerabilities are fixed at a lower rate than other kinds of defects (which I think is entirely plausible).

BP • **December 17, 2013 2:34 PM**

There are problems. I was going through a Linux package trying to make it work since I usually

don't know what I am doing, and found a lot of code that had borrowed from some package that had been used in the military in the late 1960s/early 1970s. It was obviously used to catalog items belonging to military personnel during that period to keep track of where items were shipped. Someone had not cleaned up that package well as some of it was downright humorous and personal with names that meant nothing and all kinds of code and details that hadn't been purged but must have been abandoned, but that detailed stuff that was lost by people moving. I just decrypted that part that was inside an archive of an archive of an archive ad infiniteum and ended up with the junk. It was just hidden in there until you ran the decrypt function normally to decrypt archives within archives, which lit it up. Too bad I didn't share it then but that is one package that needs cleaning up. Too bad I can't remember which OS and which package it was part of but I can't help but wonder if junk like that is also buried within other Linux packages.

BP • **December 17, 2013 2:37 PM**

Let me add. I should have used the word decompress instead of decrypt. The package wasn't encrypted, it was just compressed.

CallMeLateForSupper • **December 17, 2013 4:40 PM**

Oh come now, kids. We're all experienced enough to know, and old enough to remember, that no defect ever survives code release. It becomes a "feechur". ;-)

Max • **December 17, 2013 4:47 PM**

I did not read the study, but have an opinion anyway. :) Of course older stuff gets exploited more, duh! At the very least it has a chance to gain an install base, which makes it worthwhile to exploit.

PS. This reminds me of a statistical study where they used sophisticated data analysis to come to a mindblowing conclusion: beautiful people get laid more often than the ugly ones. :)

---

kashmarek • **December 18, 2013 5:59 AM**

@Jeremy:

"defect" versus "vulnerability"

Here is an example from the pre-Internet days, involved a "do nothing" program from IBM mainframe computing...

Programs on MFT, MVT, and MVS most notably ran in batch processing mode. One such "do nothing" program was IEFBR14. IEF is the code for a group of utility programs, the BR14 is a specific function, which is a single 2 byte instruction that indicate "branch register 14", which is typically the return address used by called batch programs to return control to the operating

system.

The problem was that return codes are passed back in register 15, and since IEFBR14 was only a single instruction program, it did not pass a "proper" return code in register 15 (the value there was the entry point address in memory of the IEFBR14 program).

Thus, job streams, which depend on conditional execution of JCL steps, test the return code, and thus would fail due to the value being unexpected. The "fix" was to make the IEFBR14 program two instructions, the first being SR 15,15 (to clear the return code register) followed by the now infamous BR 14 instruction (which goes back to the operating system).

The original intent of IEFBR14 was to provide a non-executing program step in JCL (job control language) for testing JCL statements. Of course, the originators of the program didn't plan for multiple step jobs using IEFBR14 to actually exist for the purpose of manipulating JCL for other steps in the job stream. That occurred as the operating system matured from the 1960s through the 1980s.

Aren't you glad it is so much simpler today?

Frank Wilhoit • **December 18, 2013 6:51 AM**

@Clive Robinson is talking about type safety! (As years on end may go by without hearing or reading anyone talking about type safety, it's a happy day.) Anyhow, if you start with real information modelling, the "input validation" is not, and intrinsically cannot be, separate from the "functional code". The type is the type. It has content and behavior. The rest is specializing the type through its lifecycle.

Mike the goat • **December 19, 2013 3:23 AM**

Clive: I agree with you. I was so annoyed at this excuse for a paper that I made a lengthy blog post on the issue which began as a critique and diverged into an essay on the difficulties encountered when attempting to migrate from legacy systems. I guess that's what happens when you have yet to adjust to the time zone and have been awake for over thirty hours.

My key point is that code reuse isn't necessarily a bad thing if the code that you are reusing is good and audited code. If we had everyone following these joker's advice regarding code reuse and writing their own SSL libraries rather than sticking with GNUTLS or OpenSSL (or whatever) then imagine all of the new vulnerabilities and bugs a coder who has no business coding crypto would introduce. Code reuse is good - and modular programming where routines are separated into easily used and clearly defined libraries makes this easy. Nothing wrong with reusing good code!

Wendy M. Grossman • **December 20, 2013 10:46 AM**

David: I believe "different to" is British. That may explain why it's inexplicable to you. :)

wg

Subscribe to comments on this entry

Leave a comment

Login

Name (required):

E-mail Address:

URL:

Remember personal info?

Fill in the blank: the name of this blog is Schneier on _____ (required):

Comments:

Allowed HTML: <a href="URL"> • <cite> <i> • <b> • <sub> <sup> •   • <blockquote> <pre>

Preview Submit