Faith and Hope Revisited

Rethinking methodologies for building trusted systems

John McHugh

3 April 2003

**Software Engineering Institute
and
Center for Computer and Communications Security
Carnegie Mellon University
Pittsburgh, PA 15213-3890**

Sponsored by the U.S. Department of Defense

Premise

While much of the software that we build is "pretty good", we lack the link between process and product that would permit us to predict accurately the quality of the resulting product.
As a result:

- **We cannot give prescriptive advice to developers.**
- **We cannot provide meaningful warranties**
- **We cannot predict the safety or reliability of systems that make extensive and critical use of software**

Is this really the case?

I believe that it is so.

I invite you, *implore you*, to convince me otherwise.

Let us consider some cases that matter.

- **Flight control systems**
- **Medical applications**
- **Weapons systems**
- **Commodity operating systems and applications**

The evidence is disheartening

Flight Control Systems

John Rushby of SRI has some wonderful case studies from an experimental military aircraft.

There is a lingering suspicion that most of the early A320 crashes have had software involvement.

I have collected enough anecdotes from pilots and flight crews to be leery of full authority systems.

- **Piedmont localizer**
- **UPS MD-11 fuel system**
- **ComAir RJ flaps**

Medical Applications

The Therac-25 is probably the best known example. It killed a number of people due to software errors.

Devices are not the only risk. Compromising patient care systems as DDoS zombies is recent.

We have seen instances of deliberate tampering with medical records, putting patients at risk.

Physicians are far from perfect, but the trend towards relying on automation as a cost control mechanism will probably not improve the quality of care.

Weapons Systems

We have been lucky with the bombs but not so with the decision making and control systems.

There are well documented software faults in the Patriot. Aegis design flaws led to the destruction of a civilian airliner with the loss of all on board.

Most large weapons systems today have a substantial software component. Most of the procurements attribute a large part of their cost overruns and delivery slips to software problems.

Commodity Software

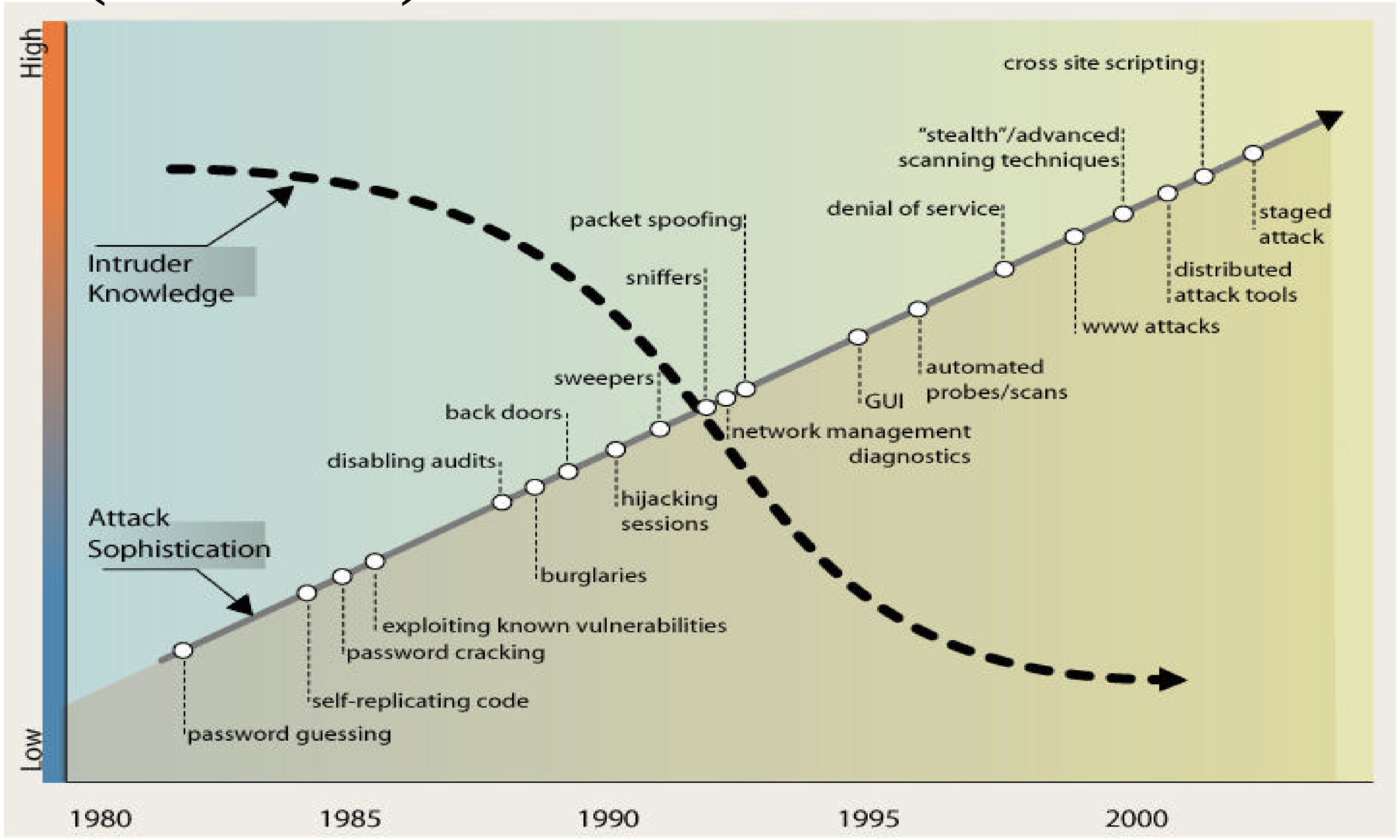
- **Commodity software (Windows, Linux, Word, etc.) is not really intended to be trustworthy, but does it have to be so bad?**
- **In addition to being accident prone, it is fairly easy to break as we will see.**
- **Recently, Microsoft declared that making its software secure was it's highest corporate priority. This is an encouraging sign, but we will wait for results.**

Now, throw in a little malice!

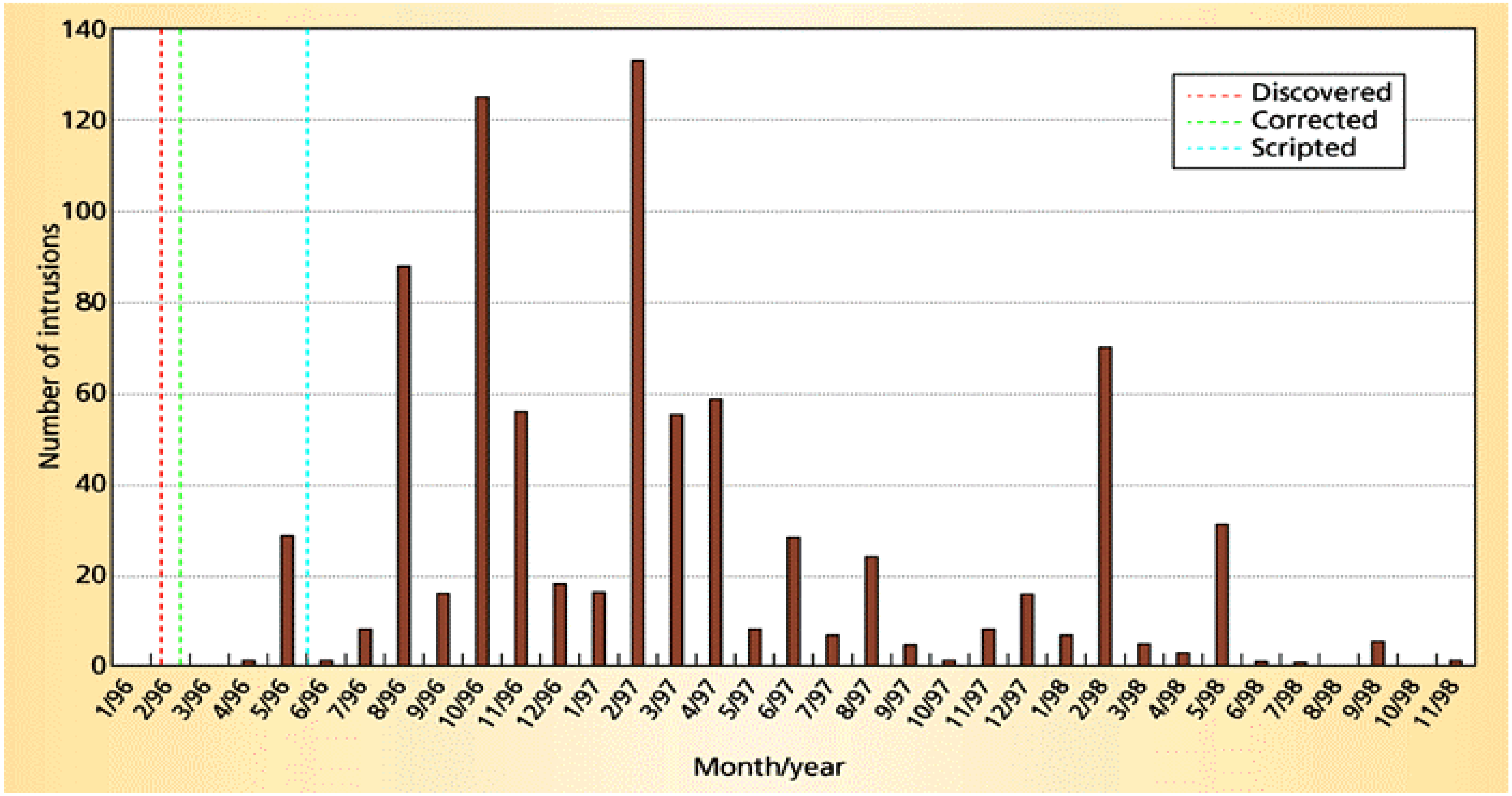
One problem that doesn't get addressed above is the notion that someone might actively try to break software, typically by using (abusing) it in ways that were not anticipated by its designer / implementer.

At the present time, we have a substantial number of people who are looking for flaws in commodity software. When they find them, they typically develop attacks or exploits that take advantage of the discovery.

Evolution (attacks), Devolution (attackers)



phf incident history – from Arbaugh’00



The evidence is disheartening

I first gave a version of this talk in 1994. It was an outgrowth of observations that date back to 1988, but were hardly original then. Let's look back at the motivating factors, then forward to today.

What happened in November of 1988 that that caused me to start this line of thought?

- **Some of you may remember – for others, this is ancient history.**
- **At the very least, history is good for the soul.**
- ***Don't all speak at once.***

The “Morris Worm”

On 2 November, 1988, the internet was subjected to it’s first widespread, self propagating attack. This was attributed to a graduate student at Cornell named Robert T. Morris, Jr.

- **Whether it was malicious is still debated**
- **It’s propagation was the result of a mix of**
  - **poor software engineering (2 cases) and**
  - **misplaced trust (1 general case). Poor security practice aggravated this case.**

Misplaced trust.

Unix systems have a notion of mutual trust.

- **globally in /etc/hosts.equiv**
- **per user in .rhosts**

It is possible to configure systems so that a user on one machine need not give a password to access a trusted peer.

By cracking weak passwords, the Morris worm was able to reach (and often infect) many machines using this mechanism, but, if it failed, ...

Software Engineering Failures

The worm became "aware" of other machines because they were mentioned in various files on the infected host. This allowed two other attacks

- **A buffer overflow exploit against *fingerd***
  A message was constructed that was too big for the array the program used to hold it. This caused code in the message to be executed.
- **A misconfiguration exploit against *sendmail***
  Commands sent to the attacked host were executed there.

Both allowed the worm a foothold on another host.

Avoidable? I claim so!

- The buffer overflow problem requires the programmer to reason about data sizes when they are known and to measure them when they are not.
- The problem can also be solved by checking data structure references for legality at run time or by using “type safe” languages such as Java rather than unsafe languages such as C
- In general, *Defensive Programming* covers this area.

Avoidable? I claim so!

The misconfiguration problem is more subtle. The sendmail program was so complex that trial and error was the rule. As Spafford noted

> **“Stories are often related about how system administrators will attempt to write new device drivers or otherwise modify the kernel of the OS, yet they will not willingly attempt to modify sendmail or its configuration files.”**

The failure to design programs so that they can be used easily, safely, and securly is a failure in the “human factors” part of software engineering.

Fast forward ... 14y, 5m later

One would think these problems would have been fixed, but consider these CERT advisories

- CA-2003-09: Buffer Overflow in Core Microsoft Windows DLL
- CA-2003-07 :Remote Buffer Overflow in Sendmail
- CA-2003-06 :Multiple vulnerabilities in implementations of the Session Initiation Protocol (SIP)
- CA-2003-05 :Multiple Vulnerabilities in Oracle Servers

and others in the past few months.

CA-2003-06 is like CA-2002-03

The associated vulnerability note, VU#854306, indicates that part of the SNMP problem is due to defects in message decoding. The messages are specified in a notation called ASN.1 and decoded using an ASN.1 library which suffers from buffer overflow and other problems.
ASN.1 is used for many things. X-509 certificates, SCADA systems, etc. The library is widely used. Many things may be vulnerable. It is hard to tell how many or to what extent. The problem has its origins many years ago.

Is This Necessary?

- We have seen similar phenomena in other areas. The histories of Navel architecture, bridge building, and steam engineering are replete with similar examples.
- As we move into new areas of development, we slowly learn that prior techniques are fraught with pitfalls.
- In early endeavors, the field is full of charlatans. These are brought under control, by societal pressure, either through market pressures, internal governance of the profession, or government regulation.

Some Nostrums

Lots of things have been suggested. Some may or may not work. We don't know.

- **Structured programming**
- **Formal Methods**
- **CMM and TDM**
- **Open vs. Closed source**
- **etc.**

• **Some we know don't work.**

- **N- Modular redundancy**
- **The “Contract Model” of programming**
- **etc.**

Is This Necessary?

- **AS LONG AS WE DON'T KNOW WHAT WE ARE DOING, A CERTAIN NUMBER OF ACCIDENTS ARE INEVITABLE!**
- **Failures can be a driving force for research and discovery.**
  - **Shortly after we learned about centers of mass, we stopped designing inherently unstable ships.**
  - **Failures in cast iron bridges and steam boilers led to fundamental work in materials science and structural engineering.**
- **The current situation in software engineering has similarities and differences**

Similarities

On the whole, we do pretty well.

- **The average individual is not often affected by software failures.**
- **Failures that affect lots of people are widely publicized, sometimes.**
- **We are in a state of denial about the inevitability of the problems, though we seldom attribute them to a deity.**

Differences

- Software is much more malleable
- We have many more small failures and are willing to tolerate them more often.
- We tend to anthropomorphize many failures and avoid looking for the real causes.
- We have not yet reached the threshold at which public pressure forces a change.
  - We may never.
    Commonalty is not recognized.
  - We aren't ready by a long shot.

Three Problems

- **Collectively and individually, we fail to learn from our past mistakes.**
- **We rush to impose standards without any evidence that the standard will improve the state of the practice.**
- **We don't understand what goes wrong.**
  - **This applies equally to product, process, and to the relationship between product and product.**

History

Those who cannot remember the past are condemned to repeat it. **-- Santayana**

- **Other engineering disciplines have overcome failures by collecting failure data and analyzing failures for commonalty that could lead to avoidance of that kind of failure in the future.**
- **Failure data in software is generally considered proprietary.**
  - **With few exceptions, failure data from product developments is not available for open research**

Failure Data

- At an NRC workshop in 1993, Win Royce of TRW urged the sharing of failure data. When pressed, he admitted that he was not in a position to offer TRW data.
- This is typical. Very little has appeared on attempts to use detailed failure data for product and process improvement. IBM had a program for some product lines. There is apparently a similar program in parts of HP.
- In regulated areas, the availability of product and process failure data for public scrutiny should be a matter of policy, possibly in trade for reduced liability exposure.

Standards

- **There is a rush towards premature standardization. MoD 0055 and 0056 are perhaps examples, though they were intended as driving forces.**
- **Standards such as MilStd 2167A, DO 178, etc., tend to be interpreted in a prescriptive fashion, even though there is no evidence that the prescribed activities are either effective or cost effective in meeting the goals of the standard.**
- **Traditional standards are codifications of long accepted practices. Software engineering standards tend to be more arbitrary.**

Standards Agenda

- **Standards that are not codifications of practices that have proven effective should have a validation component.**
- **This would require the collection of data that, used as case studies, would validate or refute the assumptions on which the standard was based.**

Where do errors come from?

The process of building software is a human activity.

- **The first generation of computer programmers came primarily from the physical sciences, engineering, and math.**
- **This kind of background is usually ill suited to careful consideration of the human factor.**
- **The social scientists who have come into computer science have generally looked at how people use computers, not how computers and programs are developed.**

We can only speculate about the latter.

Speculation

1) **Poor communications.**
   Customers and users don't speak the same language.
2. **Overwhelming complexity.**
   Programs offer complexity at many levels. Details get lost and resurface.
3) **Incompetence and over confidence.**
   Many programmers are amateurs and don't understand what they are up against.
4) **Individual differences**
   Why do different people write different programs from the same specifications? Why do they get it wrong differently but in the same place?

What can we do?

It is tempting to say, "I've identified the problem, the solution is up to you."
After all, you are going to be the folks in the trenches who have to live with it. But that wouldn't be right.

- **It may be that there is little or nothing we can do. What if the development process is chaotic?**
- **If we are going to solve the problems, we need information. This means systematically collecting and sharing failure data.**
- **Given enough data, we may be able to figure out what goes wrong and perhaps devise ways to fix or prevent the problems.**

Questions? Comments? Rants? Thanks!

If you want to discuss these issues further, I'll try to read and answer email in a timely fashion.

John M^{c}Hugh
jmchugh @ cert.org

Over the years, numerous colleagues, students, and friends have contributed to the ideas contained in this talk. I acknowledge their contributions with heartfelt thanks.