Death, Taxes, and Imperfect Software: *Surviving the Inevitable*

Crispin Cowan, PhD
Senior PM, Windows Security
Microsoft

Operating System Security Sucks

- OS security sucks
  - Every major OS from ships with latent vulnerabilities
  - Why can't they get it right?
- Hint: It isn't really the operating system
  - Bugs in the **kernel** are very rare
  - Bugs in the **huge** pile of programs that ship *with* the operating system are very common
- And then you load applications on the OS
  - And the security sinks to the level of the weakest application you are hosting

Security Just Generally Sucks

- Much more than other aspects of computing
  - Word processors process the words
  - Music players play the music
  - Web browsers browse the web
  - etc.
- But when you get a security system, you still aren't secure
- Computing is 65 years old
  - Ready for Medicaid but not ready for prime time?!
  - Why can't we get it right after all this time?

"The reason why you have people breaking into your software is because your software sucks."

Richard Clarke

Because it is Hard

- For all other kinds of computing, being correct for *normal* inputs is sufficient
  - Reliable software does what it is supposed to do
- But that is not enough for security
  - Secure software does what it is supposed to do, *and nothing else*
- Security is really simple: only use perfect software
  - ... but there is a supply side problem

Why Doesn't the Market Fix This?

- If security is important, throw money at the problem?
- This doesn't happen in practice because:
  - Developers are lazy, don't like to check return codes, etc.
  - Languages are unsafe
- Customers (and magazine product reviewers) react to shiny buttons more than quality:
  - You can see shiny buttons
  - Therefore managers won't give developers the time and tools to do software right
- Features. Quality. Ship date. Choose 2
  - Guess which two are the popular choices

So Really Good Vendors Should Be Delivering Secure Products ... ?

- Kinda :-(Diligence *helps* ...
  - Good coding practices
  - Peer review
  - QA, penetration testing, fuzz testing ...
- .. but benefits are limited
  - You can test for what **should** happen
  - You **cannot** test for what **shouldn't** happen in the presence of arbitrary input

Meet Alan Turing

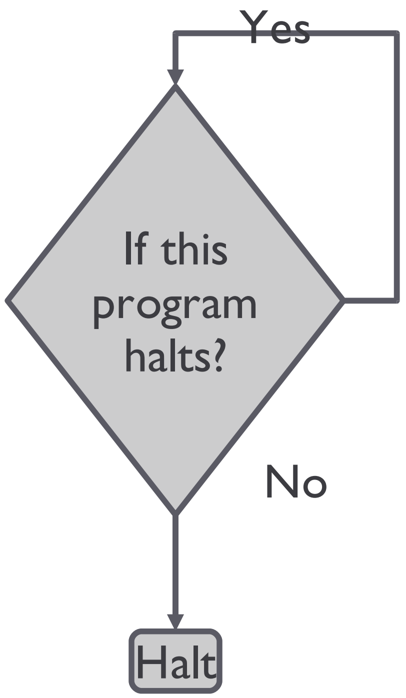
(CS grads can read some mail for a bit :-)

Alan Turing's Cute Theorem

- Gödel, 1931
  - A mathematical system complex enough to represent itself cannot be both **consistent** and **complete**
  - **Consistent:** all theorems are true
  - **Complete:** all true statements are provable
- Turing's lame corollary 1932
  - Imagine a machine that can compute states based on input
  - Give it an infinite tape drive
  - You cannot write a program that will analyze any other program + input and decide if it will halt or not
- Minor side effect: invented computers :-)

Proving Turing's Halting Problem: Diagonalization

- Consider some hypothetical program X that *can* solve Turing's Halting problem
  - Ask X to analyze program 1, 2, 3, …
  - When you ask X to analyze itself, program it to loop if X halts
  - So if it halts, it loops, and if it loops, it halts
  - Contradiction! -> X cannot exist
- Simplest form: “`This is a lie.`”



The Halting Problem Applied

- If you can't write an analyzer to determine halting, then you can't decide
  - If a program will or won't write to a given memory location
  - Will or won't overflow a buffer
  - Will or won't grant unintended access
- **Is or is not secure**

So We're Doomed?

- Not doomed ...
  - Security professionals have lifetime employment :-)
- But what to do?
  - Building secure programs is undecideable
  - Must instead build belt & suspenders protection layers that defend the system against vulnerable components
  - We used to call this "secure architecture"
  - Now we call it Intrusion Prevention

Meet John Boyd
(CS grads can wake up again :-)

Boyd's OODA Loop

- Boyd was an air force fighter pilot
- Invented OODA: a new way to think about air combat:
  - **Observe** your surroundings
  - **Orient** yourself to your context
  - **Decide** what to do
  - **Act** on that decision
- Air combat winners are those with the fastest *accurate* OODA loop
- Turns out this applies to computer security too

OODA and Intrusion Prevention

- Use OODA to classify IPS according to
  - **When**: Time in the software life cycle where IPS is inserted
    - Earlier is faster
    - Later is more precise
    - Design time, implementation time, run time
  - **Where**: Place in the network architecture where IPS is inserted
    - Closer to the incident is more precise
    - Farther out has broader impact, easier to deploy
    - Network or Host
  - **What**: Kind of mediation applied
    - Detection is easier if you don't have precision, but doesn't protect
    - Prevention requires precision to be tolerable

When

Design Time: Saltzer&Schroeder's 8 Principles of Secure Design

- These principles have held up well over time, but some more than others
  - Least privilege is a spectacular success
  - Least common mechanism not much used, with common mechanism that is carefully constructed fares better
- Unfortunately, these principles also turn out to be too expensive to apply
  - Easier to just ship crap :-)
- Users should demand “least privilege” operation as a feature!
  - If it isn't there, buy a different product

Implementation Time: Static Analysis

...that thing I said you couldn't do :-)

Source Code Checkers

- Syntax checkers: grep for bad stuff
  - gets, strcpy, printf(str, ...)
- Semantic checkers:
  - Type checking: use all your data consistently
  - Taint analysis: detect whether you filtered user input before depending on it
- Ask your vendor if they use these tools

Binary Checkers

- Similar analysis of binary programs
- Useful for enterprises
  - Large risk, so can afford to spend time on this
  - Dependent on proprietary applications that don't provide source code
- Use these tools if your vendor does not audit their own code

Implementation Time: Better Languages and Compilers

- Compiler Defenses:
  - StackGuard (USENIX 1998)
    - GCC and Microsoft /gs
  - FormatGuard
- Dynamic languages: Python, Ruby
  - Previously known as SmallTalk
  - Instead of vulnerability you get "uncaught exception"
  - but in the mean time, it lets you ship the broken code
- Static languages: Java, C#
  - Previously known as PL/1
  - Instead of vulnerability, you get "type error, program rejected" at compile time
- What about C++?
  - No: **not** type safe, because it still supports pointer arithmetic
  - C++: the safety of C, and the performance of SmallTalk :-)

Run Time: Library and Kernel Enhancements

- **Libsafe**: libc with smarter big-7 string functions
  - strcpy & friends introspect arguments, barf if the target is plausibly in the caller's stack frame
- **Open Wall Linux:** non-executable stack
  - Standard on classic CPUs, problematic on x86
  - Prevents instant shell code injection
- **PaX/ASLR:** non-executable heap
  - Standard on classic CPUs, **very** problematic on x86
  - Solution: fun with TLBs
- **NX/DEP:** x86 finally gets non-executable pages
- **RaceGuard:** blocks temp file race attacks

Where

Where: Network or Host

- **Host:** e.g. OS features
  - Up close
  - Gives you precise information on the intrusion, so your OODA loop is more accurate
  - Can respond quickly, so your OODA loop is tighter
  - Boyd would like this
- **Network:** e.g. firewalls
  - Farther out
  - Gives you a more global perspective, for better event correlation
  - Gives you more global impact for stronger mediation
  - ~~Generals~~ IT Managers like this

What

Detection or Prevention

- "Intrusion detection" is what you call it when your detector is too lame to prevent the attack
  - Too slow to prevent attack before it happens
  - Too inaccurate to allow it to automatically block
- Prevention (automatic blocking) requires speed and precision
  - Limits you to detection techniques that are fast and precise
  - Complex detection methods will come too late
  - Heuristics can be wrong, so can't let them automatically block

Presumed Innocent? Or Presumed Guilty?

- All those block **bad** behavior, and allow everything else
  - Misuse prevention
  - Default allow
  - Signature-driven security: AV, network IDS
  - What happens when attackers invent a new "bad" thing?
- **Anomaly prevention:**
  - Specify what is allowed, and block all else
  - Policy-driven security
- Which to use?
  - Misuse prevention easier to live with
  - Anomaly prevention more secure

Statistical Anomaly Detection

- Forrest et al: "Sense of Self" IEEE S&P 1996
  - Inspired by biological immune systems to distinguish "self" from "other"
  - Approach: "self" is applications whose syscall sequences match a pattern
  - Implementation: several MB of stats on rolling n-gram sequences of syscalls
  - Result: if you train it hard enough, it can detect intrusion and not disrupt legitimate actions
  - > Sana Security

Statistical Anomaly Detection and Mimicry Attacks

- Problem: Mimicry attacks
  - Attacker crafts attack so that its sequence of syscalls mimic the legitimate patterns
  - Use NOP syscalls to pad the attack sequence, e.g. open() on non-existent files or files that don't matter
- Improvement: measure more factors
  - Syscall parameters, address called from, time, etc.
- Response: more detailed mimicry
- Result: Arms race

Access Controls

- Instead of judging activities as “good” or “bad”, just decide definitively who can access what and how
- Design issues:
  - How to specify “who”
  - How to specify “what”
  - How to specify “how”
  - How to abstract all this because controlling every bit is too much

Network Access Controls

- **Firewall:** mediates access between networks
  - Based on source and destination IP address, port number, and protocol, i.e. stuff up to Layer 4
  - Rules are absolute: stuff gets through, or it doesn't
  - Default deny: everything blocked except what you allow
- **Network Intrusion Detection** and **Prevention:** also mediates access between networks
  - Based on packet content and context
  - Rules might be heuristic: gets through if it smells ok
  - Rules might be signature-based, i.e. **default allow**

So a NIDS is Just a Flaky Firewall?

- Well ... yes
- Network traffic is very regular up to layer 4
  - Can use strict, regular rules to regulate flow
- Network traffic is very *irregular* above layer 4
  - I.e. application content
  - Zillions of applications, new ones come along all the time
- You *can* build a default-deny NIDS
  - But you will hate it
  - It blocks everything it doesn't understand

Why Would I Want a Flaky Firewall?

- Signature-based NIDS can only block **known** vulnerabilities
  - NIDS is a kludge that you use when you can't patch your bugs
- Why would I want that?
  - Because sometimes you *can't* patch your bugs
    - Machine is in a mission-critical production mode and cannot be halted
    - Vendor hasn't issued a patch
    - Patch hasn't been QA'd yet
    - Patch just sucks
- Use NIDS to mitigate weakness in your patching strategy

Host Access Controls
De-perimeterization

- OS features to let you specify who can access what on the local machine
- **Discretionary** access control: he who creates the data can grant access to anyone else
- **Mandatory** access control: he who owns the *system* decides who can access a given resource, no matter who you are
    - Allows system manager to strive for the *principle of least privilege*

Access Control Lists vs. Capabilities

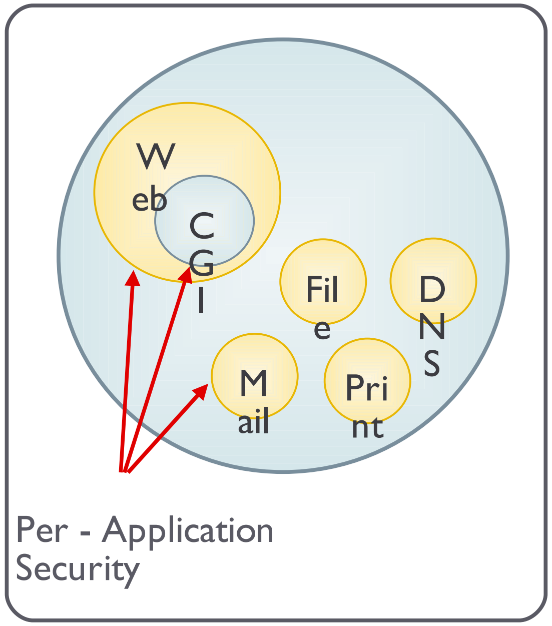
- **Access Control Lists:** security rules are associated with the object (file)
- **Capabilities:** security rules are associated with the subject (user or process)
- Classic UNIX mode bits are a *crude* ACL
  - List of length 1 for user mode and group mode access

Access Control Lists vs. Capabilities

- **Hard** to compute least privilege for a user or process with ACLs
  - Need to scan all objects in the system to determine what the subject can access
- To achieve approximate least privilege for intrusion prevention, want a Capability system
  - **First Class** capability system makes Capabilities be objects that programs can manipulate
  - **Ambient** capability system makes the capabilities external to the process
- Ambient better for confining legacy software

Least Privilege for Programs

- 1980s: most systems are timeshare
  - Need least privilege for users & groups
- 21st Century: most systems are
  - 1 user workstations
  - 0 user network servers
- Need least privilege for *programs*
  - Enforce that program does what it is supposed to do, *and nothing else*



Application Least Privilege

- For Linux
  - SELinux (Red Hat)
  - AppArmor (SUSE, Ubuntu, Mandriva)
- For BSD
  - Systrace
- For Windows
  - Okena Cisco CSA
  - Entercept McAfee Intrusion Prevention
  - Core/SDI Coreforce; quite similar to AppArmor

Summary

Summary:
Security is Harder Than it Looks

- Making a system secure is very hard
  - It is expensive
  - Customers **don't** demand it
  - "Is it secure?" is undecidable
- Therefore securing systems is a continuing process, not a condition
  - Supply belt *and* suspenders to defend your system against its inevitable latent vulnerabilities
  - We call this Intrusion Prevention

Summary: Intrusion Prevention

- **When:** Design time, Implementation time, Run time
- **Where:** network or host
- **What:**
  - **Detect or Prevent**
  - **Misuse or Anomaly**
  - **Statistical or Access Control**
- I'd draw a picture, but that is two nested 3-D cubes

Summary: The Art of Info War by ~~Sun Tzu~~ John Boyd

- OODA:
  - Observe, Orient, Decide, Act
- Winner:
  - The one with the tightest *accurate* OODA Loop
- Intrusion Prevention choices
  - Close to intrusion site will work better
  - Farther out will cover more ground with a single tool … at the cost of speed and accuracy
- As always, whether or not you get what you pay for, you definitely pay for what you get