Turing Around the Security Problem

Why Does Security *Still* Suck?

Crispin Cowan, PhD
Director of Software Engineering,
Security Architect, SUSE Linux

August 3, 2006

Security 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 Is Correctness More Important to Security?

Other fields are mission critical

- Aircraft fly-by-wire
- Nuclear reactor controllers, etc.

What makes security special?

Intelligent attackers:

- Other mission critical applications do not have to worry about improbable events
  - They are unlikely to happen :-)
- Security: attacker *looks* for poorly handled conditions and causes them to happen

The improbable becomes probable

So Correctness Matters a Lot: 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: Java and C# are the first really **popular** languages that are **type safe** since PL/1

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 (especially open source :-)
- 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

Goedel, 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 :-)

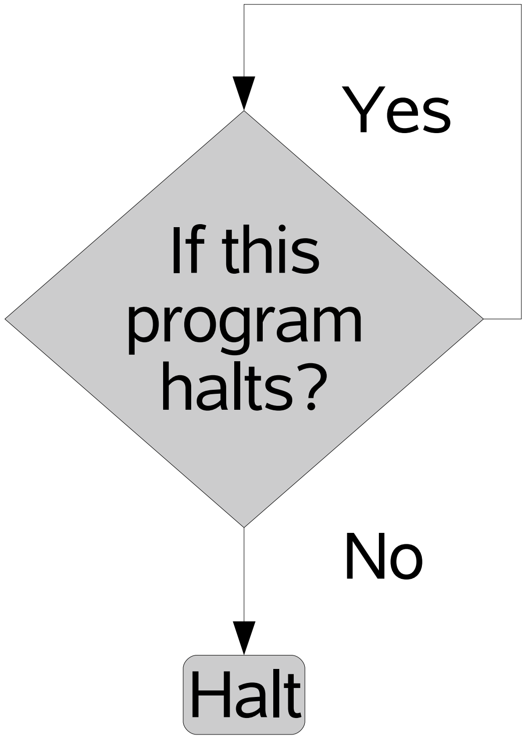
Proving Turing's Halting Problem: Diagonalization

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

What About Static Analyzers?

Heuristics:

- You can't analyze *arbitrary* programs, but you can prove that a *specific* program will halt ... or is secure
- You *can* encode this into provers that can say “safe”, “vulnerable”, or “don't know”
- Or be wrong :-)

What about type safe languages?

- “Type safe” is the subset of program behavior that *can* be statically proven
- **Note:** type safe languages quite often reject programs that actually are safe, they just can't be *proven* safe by the compiler

So We're Doomed?

Not doomed ...

- Security professionals have lifetime employment :-)

What to do?

- Building secure programs is undecidable
- 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
- Invended 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

1. Economy of mechanism: designs and implementations should be as small and simple as possible, to minimize opportunities for security faults, i.e. avoid bloat.
2. Fail-safe defaults: access decisions should default to deny unless explicitly specified, to prevent faults due to unanticipated cases.
3. Complete mediation: design such that all possible means of access to an object are mediated by security mechanisms.
4. Open design: the design should not be secret, and in particular, the design should not depend on secrecy for its security, i.e. no security through obscurity.

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

5. Separation of privilege: if human security decisions require more than one human to make them, then faults due to malfeasance are less likely.
6. Least privilege: each operation should be performed with the least amount of privilege necessary to do that operation, minimizing potential failures due to faults in that privileged process, i.e. don t do everything as root or administrator.
7. Least common mechanism: minimize the amount of mechanism common across components.
8. Psychological acceptability: security mechanisms must be comprehensible and acceptable to users, or they will be ignored and bypassed.

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 :-)

Implementation Time: Static Analysis

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

Syntax checkers: grep for bad stuff

- gets, strcpy
- printf(str, ...) instead of printf("format %s \n", str)
- etc.

Semantic checkers:

- Do deeper analysis of program to look for problems
- Type checking: use all your data consistently
- Taint analysis: detect whether you filtered user input before depending on it

Implementation Time: Better Static Languages

Safer language variants: e.g. Cyclone, CCured

- Produce a type-safe subset of C
- Then add back some stuff to make it usable

Static type safe languages: Java, C#

- Previously known as ML, Pascal, PL/1
- Instead of an exploitable buffer overflow, you get “type error, program rejected” at compile time

Implementation Time: Dynamic Language Techniques

Compiler Defenses:

- StackGuard (USENIX Security 1998)
  - Became GCC ProPolice and Microsoft /gs
- FormatGuard

Dynamic type safe languages: Python, Ruby

- Previously known as SmallTalk
- Instead of an exploitable buffer overflow, you get “uncaught exception”
- but in the mean time, it lets you ship the broken code

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: non-executable heap

- Standard on classic CPUs, **very** problematic on x86
- Solution: fun with TLBs

NX: 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 things 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

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 as 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

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*

Lampson's Access Control Matrix

- Enumerate every single subject (user) and object (file) in the system
- Populate a matrix with access modes

	Alice (sysadmin)	Bob (accounting)	Carol (engineering)
/var/spool/mail/alice	RW		
/usr/bin/login	RWX	X	X
/etc/motd	RW	R	R
/local/personnel/payroll		RW	
/local/eng/secret-plans			RW

Access Control Abstraction

Lampson's matrix lets you specify *exactly* least privilege

But the matrix is **huge**, so:

- Errors in the matrix are likely to occur
- Such a pain that most users unlikely to use it

Need more convenient abstractions to make specifying *approximate* least privilege feasible

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

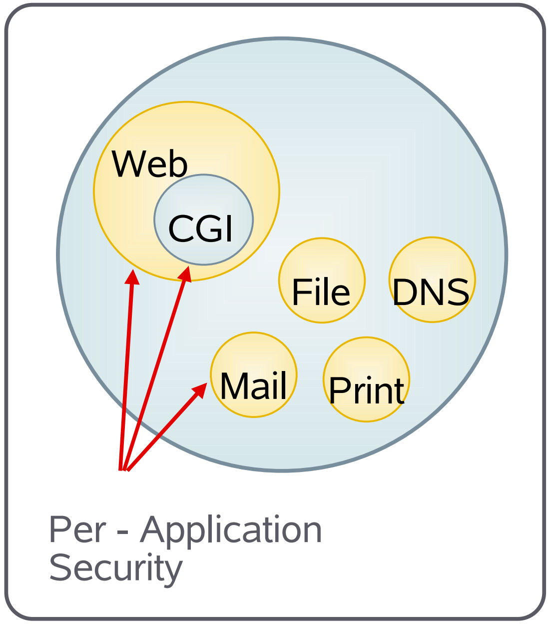
To achieve approximate least privilege for intrusion prevention, want a Capability system

- **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*



Danger! Product Pimping Ahead

(But it is all Open Source :-)

Application Least Privilege for Linux

SELinux

Type Enforcement

- Assign users or programs to Domains
- Label files with Types
- Write policy in terms of which Domains can access which Types

AppArmor

Pathnames

- Name a program by path
- When it runs, it can only access the files specified by pathname
- Generalize pathnames with shell syntax wild cards

Labels vs. Pathnames: Static vs. Dynamic

SELinux label scheme

- Half your policy is in the labeling scheme: labels applied to files
- Enables strong analyzability of your policy
- Forces you to specify label scheme ahead of time
- Re-labeling is **expensive**

AppArmor pathname scheme

- All of your policy is in the policy
- Enables late binding of policy to file names at the time they are accessed
- Trades away analyzability for flexibility in the presence of changing system configurations

Labels vs. Pathnames: Ambiguity

Pathnames

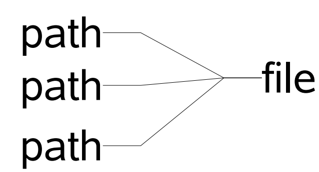
- A pathname is not the *only* name a file can have
- But a pathname does lead to only a single file, for a given namespace

Labels

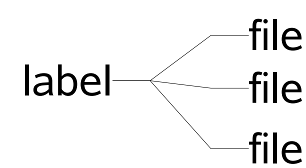
- A file can only have a single label
- But a label refers to many files

Which kind of ambiguity do you prefer?

AA



SELinux



Compare Policy: wuftp daemon

SELinux

```
##################################
#
# Rules for the ftpd_t domain
#
type ftp_port_t, port_type;
type ftp_data_port_t, port_type;
daemon_domain(ftpd, `, auth_chkpwd')
type etc_ftpd_t, file_type, sysadmfile;

can_network(ftpd_t)
can_ypbind(ftpd_t)
allow ftpd_t self:unix_dgram_socket create_sock
allow ftpd_t self:unix_stream_socket create_soc
allow ftpd_t self:process {getcap setcap};
allow ftpd_t self:fifo_file rw_file_perms;

allow ftpd_t bin_t:dir search;
can_exec(ftpd_t, bin_t)
allow ftpd_t { sysctl_t sysctl_kernel_t }:dir s
allow ftpd_t sysctl_kernel_t:file { getattr rea
allow ftpd_t urandom_device_t:chr_file { getatt

ifdef(`crond.te', `
system_crond_entry(ftpd_exec_t, ftpd_t)
can_exec(ftpd_t, { sbin_t shell_exec_t })
')

allow ftpd_t ftp_data_port_t:tcp_socket name_bi

ifdef(`ftpd_daemon', `
define(`ftpd_is_daemon', `')
') dnl end ftpd_daemon
ifdef(`ftpd_is_daemon', `
rw_dir_create_file(ftpd_t, var_lock_t)
allow ftpd_t ftp_port_t:tcp_socket name_bind;
allow ftpd_t self:unix_dgram_socket { sendto };
can_tcp_connect(userdomain, ftpd_t)
', `
ifdef(`inetd.te', `
domain_auto_trans(inetd_t, ftpd_exec_t, ftpd_t)
ifdef(`tcpd.te', `domain_auto_trans(tcpd_t, ftp

# Use sockets inherited from inetd.
allow ftpd_t inetd_t:fd use;
allow ftpd_t inetd_t:tcp_socket rw_stream_socke

# Send SIGCHLD to inetd on death.
allow ftpd_t inetd_t:process sigchld;
') dnl end inetd.te
')dnl end (else) ftp_is_daemon
ifdef(`ftp_shm', `
allow ftpd_t tmpfs_t:file { read write };
allow ftpd_t { tmpfs_t initrc_t }:shm { read wr
')

# Use capabilities.
allow ftpd_t ftpd_t:capability { net_bind_servi

# Append to /var/log/wtmp.
allow ftpd_t wtmp_t:file { getattr append };

# allow access to /home
allow ftpd_t home_root_t:dir { getattr search };

# Create and modify /var/log/xferlog.
type xferlog_t, file_type, sysadmfile, logfile;
file_type_auto_trans(ftpd_t, var_log_t, xferlog_t, file)
# Execute /bin/ls (can comment this out for proftpd)
# also may need rules to allow tar etc...
can_exec(ftpd_t, ls_exec_t)

allow { ftpd_t initrc_t } etc_ftpd_t:file r_file_perms;
allow ftpd_t { etc_t resolv_conf_t etc_runtime_t }:file { getattr read };
allow ftpd_t proc_t:file { getattr read };


')dnl end if ftp_home_dir
```

```
.
ifdef(`ftpd_daemon', `
define(`ftpd_is_daemon', `')
') dnl end ftpd_daemon
ifdef(`ftpd_is_daemon', `
rw_dir_create_file(ftpd_t, var_lock_t)
allow ftpd_t ftp_port_t:tcp_socket name_bind;
allow ftpd_t self:unix_dgram_socket { sendto };
can_tcp_connect(userdomain, ftpd_t)
', `
ifdef(`inetd.te', `
domain_auto_trans(inetd_t, ftpd_exec_t, ftpd_t)
ifdef(`tcpd.te', `domain_auto_trans(tcpd_t,
    ftpd_exec_t, ftpd_t)')

# Use sockets inherited from inetd.
allow ftpd_t inetd_t:fd use;
allow ftpd_t inetd_t:tcp_socket
    rw_stream_socket_perms;

# Send SIGCHLD to inetd on death.
allow ftpd_t inetd_t:process sigchld;
') dnl end inetd.te
')dnl end (else) ftp_is_daemon
ifdef(`ftp_shm', `
allow ftpd_t tmpfs_t:file { read write };
allow ftpd_t { tmpfs_t initrc_t }:shm { read write
    unix_read unix_write associate };
')
.
.
```

AppArmor


```
/usr/sbin/in.ftpd {
  #include <immunix-standard/base
  #include <immunix-standard/names
  #include <immunix-standard/authe
  #include <user-custom/ftpd>
  /
  /dev/urandom
  /etc/fstab
  /etc/ftpaccess
  /etc/ftpconversions
  /etc/ftphosts
  /etc/ftpusers
  /etc/shells
  /usr/sbin/in.ftpd
  /usr/share/ssl/certs/ca-bundle.c
  /usr/share/ssl/certs/ftpd-rsa.pe
  /usr/share/ssl/private/ftpd-rsa-
  /usr/share/ssl/.rnd
  /var/log/xferlog
  /var/run
  /var/run/ftp.{pids,rips}-all
}
```


```
/usr/sbin/in.ftpd {
  #include <immunix-standard/base>
  #include <immunix-standard/nameservice>
  #include <immunix-standard/authentication>
  #include <user-custom/ftpd>
  /                                      r,
  /dev/urandom                           r,
  /etc/fstab                             r,
  /etc/ftpaccess                         r,
  /etc/ftpconversions                    r,
  /etc/ftphosts                          r,
  /etc/ftpusers                          r,
  /etc/shells                            r,
  /usr/sbin/in.ftpd                      r,
  /usr/share/ssl/certs/ca-bundle.crt        r,
  /usr/share/ssl/certs/ftpd-rsa.pem         r,
  /usr/share/ssl/private/ftpd-rsa-key.pem   r,
  /usr/share/ssl/.rnd                    w,
  /var/log/xferlog                       w,
  /var/run                               wr,
  /var/run/ftp.{pids,rips}-all           wr,
}
```

SELinux uses a custom programming language to specify hard-to-manage rules

Classical Linux syntax with read/write/execute permissions:

No new jargon

Summary

Summary: Security is Harder Than it Looks

Making a system secure is very hard

- “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

Plug: NDSS Conference

Network and Distributed System Security

- Pragmatic security conference, similar to USENIX Security
- Papers due September 10
- Notification October 23
- Conference February 28-March 2 in San Diego
- PC Chairs: Me, and Bill Arbaugh

http://www.cs.umd.edu/~waa/ndss07.htmld

Novell®