

Programming Secure Applications for Unix-like Systems

David A. Wheeler

dwheeler@dwheeler.com

<http://www.dwheeler.com/secure-programs>

September 25, 2002

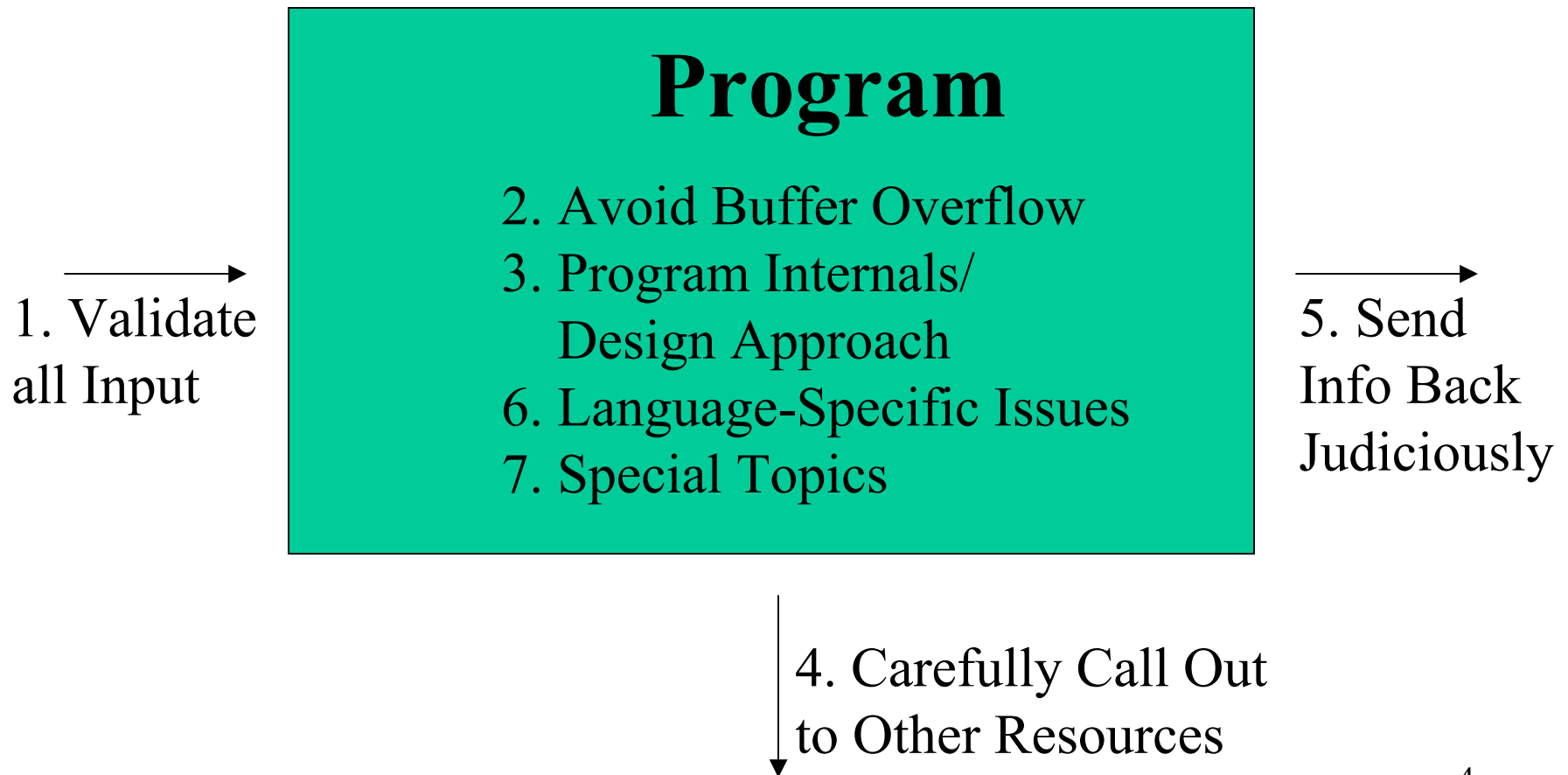
Introduction

- Contents: Lessons learned on how to write secure applications, based on past exploits (lots of detail)
 - Not how to break into software
 - Not how to configure existing software/systems
- Secure applications have inputs from untrusted users (setuid/setgid, daemon, web app, viewer,...)
 - Some recommendations don't apply to some app types
- My goal: Make software *secure* from attackers
 - Open Source Software *not* immune (sendmail, wu-ftpd)
 - People can't do it if they don't know how
 - Please, use this material and teach others!

First: What are Your Security Requirements?

- What is your security environment?
 - What threats, and how severe? Who's not trusted? What assumptions? What environment (platforms, network)? What organizational policies? What assets?
- What are your product's security objectives?
 - Confidentiality (“can't read”)
 - Integrity (“can't change”)
 - Availability (“works continuously”)
 - Others: Privacy (“doesn't reveal”), Audit, ...
- What functions and assurance measures are needed?
- Common Criteria useful checklist of requirements

Abstract View of a Program



Validate All Input: General

- Validate *all* input from untrusted sources
- Determine what's legal, reject non-matches
 - Don't do the reverse (check for just illegal values); “there's always another illegal value”
 - Use known illegal values to test validators
- Limit maximum character length
- Next: Various data types & input sources

Validate All Input: Strings and Numbers

- Watch out for special characters
 - Control characters, including linefeed, ASCII NUL
 - Shell metacharacters (e.g., *, ?, \, ...)
 - Internal storage delimiters (e.g., tab, comma, <, :)
 - Make sure encodings (e.g., UTF-8, URL encoding) are legal & decoded results are legal
 - Don't over-decode (i.e., don't decode more than once “unnecessarily”)
- Numbers: check min & max; min often 0

Validate All Input: War Story (Check Minimums!)

- Sendmail debug flags: *-dflag,value*
 - Sendmail `-d8,100` sets flag #8 to value 100
 - Name of config file (`/etc/sendmail.cf`) stored in data segment before flag array; that file gives `/bin/mail` path
 - Sendmail checked for max but **not** min flag numbers, since input format doesn't allow negative numbers
 - `int >= 231` considered negative by C on 32-bit hosts
 - Sendmail `-d4294967269,117 -d4294967270,110 -d4294967271,113` changed “etc” to “tmp”
 - Attacker creates `/tmp/sendmail.cf` which claims local mailer is `/bin/sh`; debug call gives root shell to attacker

Validate All Input: Other Data Types

- Email addresses: Complex, see RFC 822
- Filenames:
 - If possible, omit “/”, newline, leading “.”.
 - Omit “../” from legal pattern
 - Where possible, don’t glob (*, ?, [], maybe { })
- Cookies: Check if domain is correct
- HTML: Prevent cross-site malicious posting, takeover of format (limit tags & attributes)
- URIs/URLs: Validate first; will it be cross-posted?
- Locale: `[A-Za-z][A-Za-z0-9_+@\\-\\.=]*`

Validate All Input: Consider All Data Sources

- Command line:
 - Don't trust any value of command line if attacker can set them – including argv[0]
- Environment Variables:
 - Environment variables inherited; could they be from an attacker, even indirectly?
 - Local attacker can set *ANYTHING*, even undocumented variables with effects on the shell or other programs
 - Some variables may be set more than once; this may circumvent checking
 - Only solution: Extract and erase at trust boundary

Validate All Input: Consider All Data Sources

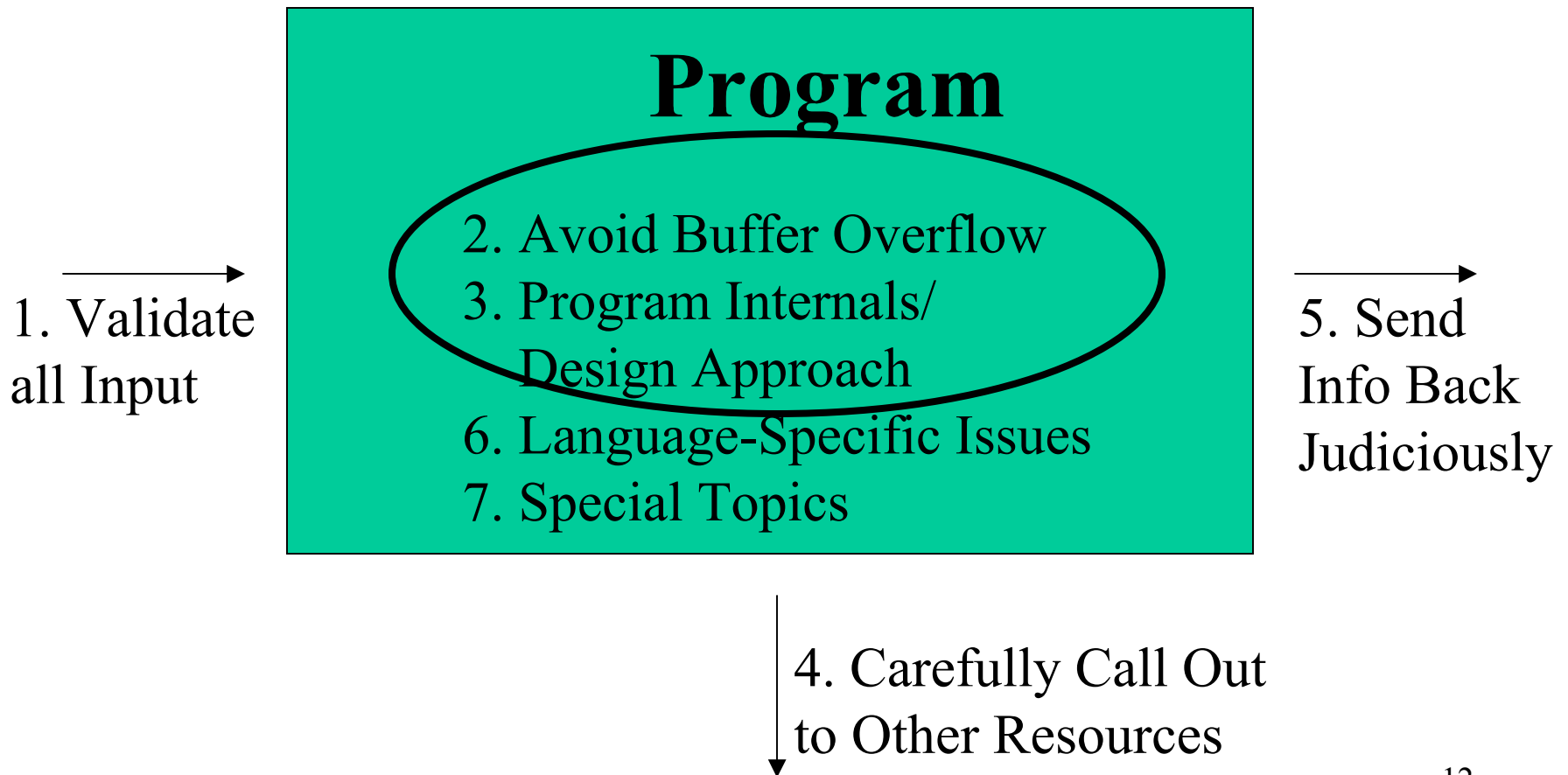
- File Descriptors:
 - (setuid/setgid) Don't assume stdin/stdout/stderr are open!
- File Contents:
 - Don't trust files that can be controlled by untrusted users (e.g., configuration files)
- Cookies & HTML form data:
 - Users can set them to arbitrary values; if you care, include authenticators and check them
- Other input: current directory, signals, memory maps, System V IPC, the umask, filesystem

Validate All Input:

Miscellaneous

- Web applications: Limit GET commands
 - Ignore/verify GET commands if it's not just a simple query (e.g., changing data, transferring money, signing up/committing something)
 - It may be a maliciously created cross-posted link, possibly on your own site
- Limit Valid Input Time/Load Level

Abstract View of a Program



Avoid Buffer Overflow: The Problem

- Buffer Overflow
 - Occurs when an attacker can cause data (usually characters) to be written outside a buffer's boundaries (usually past its end), overwriting previous values
 - If buffer is on the stack, also called “stack overflow” or “smashing the stack”; can change the return address and provide code you'd like it to return to and run
 - Possible because C/C++/asm don't autocheck bounds
 - Often allows attackers to modify data and/or force arbitrary code to run
 - Common : More than half of all CERT advisories 1998-1999; 2/3 said leading cause in 1999 Bugtraq survey

Avoid Buffer Overflow: War Story

- Wu-ftp realpath vulnerability (<2.4.2)
 - Realpath() canonicalizes pathname (eliminating “/./”..)
 - Realpath() implementation internally used fixed-length buffer and didn't prevent length from being exceeded
 - Attacker with ftp write access could create arbitrarily long path (e.g., mkdir AAA...; cd AAA...; then repeat)
 - At end of path, attacker created filename with return address and machine code to run (e.g., “run shell”)
 - When ftpd called realpath() to find real path, instead of returning, the function ran arbitrary code supplied by the attacker (e.g. root shell)

Avoid Buffer Overflow: The Solution

- Avoid or carefully use risky functions
 - `gets()`, `strcpy()`, `strcat()`, `*sprintf()`, `*scanf(%s)..`
- Alternatives: fixed-length vs. dynamic
- Choose an approach, e.g.:
 - Standard C fixed-length: `strncpy()`, `strncat()`, `snprintf()`
 - Standard C dynamic length: `malloc()`, ...
 - `Strncpy/strlcat` (fixed): easier to use than `strncpy`
 - `Libmib` (dynamic, separate library, rename if modify)
 - C++ `std::string` (not when converted to `char*`)

Program Internals/ Design Approach (1 of 6)

- Secure the Interface (“can’t circumvent it”)
 - Simple, narrow, non-bypassable; avoid macro langs
- Minimize privileges
 - Minimize privileges granted (setgid not setuid, run as special user/group not root, restrictive file permissions, limit/remove debug requests, limit writers)
 - Permanently give up privilege as soon as possible (e.g., open TCP/IP port, then drop completely)
 - Minimize time privilege active
 - Minimize the modules given the privilege: break program up to do so
 - Consider using FSUID, chroot, resource limiting

Program Internals / Design Approach (2 of 6)

- Use safe defaults
 - Install as secure, then let users weaken security if necessary after initial installation
 - *Never* install a working “default” password
 - Install programs owned by root and non-writeable by others (inhibits viruses)
- Load initialization values safely (e.g., /etc)
- “Fail safe”: stop processing the request if surprising errors or input problems occur

Program Internals / Design Approach (3 of 6)

- Avoid race conditions
 - Occur when multiple processes interfere with each other; an attacker may be able to exploit it
 - Races can be between secure program processes, or with an attacker's process
 - Don't use `access()` to check if it's okay and then `open()`; after the `access()` things may change!

Program Internals / Design Approach (4 of 6)

- Watch out for temporary files in shared directories (common race condition)
 - /tmp and /var/tmp are shared by all; attackers can often exploit this, e.g., by adding symlinks or their files
 - If possible, move to unshared locations (e.g., ~)
 - Shared directories must be sticky: test first
 - Repeatedly (1) create “random” filename, (2) open using (O_CREAT|O_EXCL) and minimal privileges, (3) stop on success; NFSv2 requires more magic
 - Or, create directory with restrictive permissions
 - tmpfile(3) unsafe on some, tmpnam(3) often unsafe

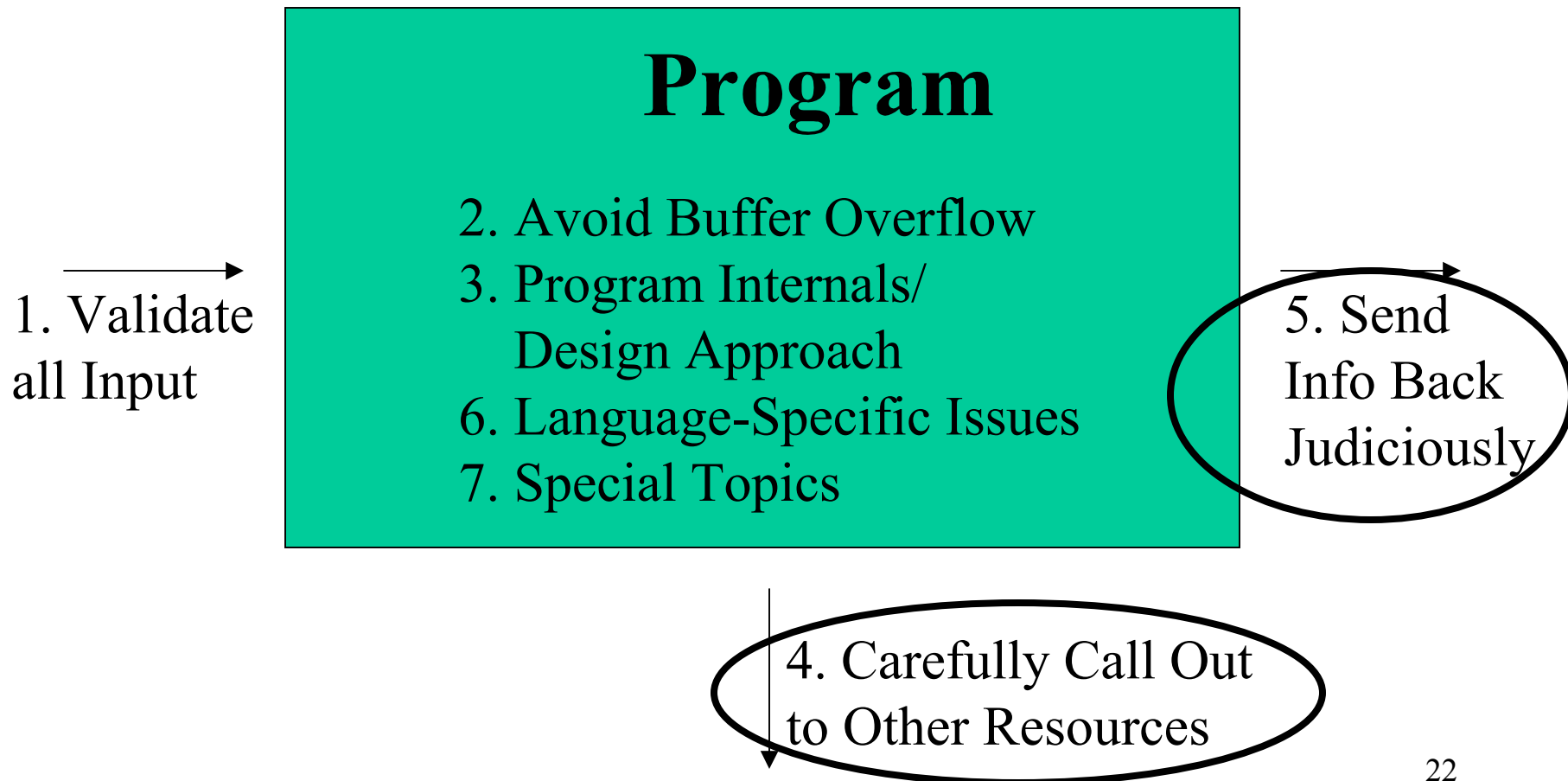
Program Internals / Design Approach (5 of 6)

- Trust only trustworthy channels
 - “From” IP addresses & email sources can be forged
 - DNS entries come from external entities
- Prevent Cross-site Malicious Content
 - Filter, or encode
- Counter Semantic Attacks
 - <http://www.bloomberg.com@badguy.com>
 - Confirm oddities, give more visual cues

Program Internals / Design Approach (6 of 6)

- Follow good security principles (S&S), e.g.:
 - Keep it simple
 - Open design: Encourage others to review it!
 - Complete mediation: Check every access. If it's client/server, server has to re-check everything
 - Fail-safe defaults: Deny by default
 - Make it easy/acceptable to use: “no urine tests”

Abstract View of a Program



Calling Out to Other Resources

- Call only safe library routines
 - If they're not portably safe, write your own
- Limit call parameters to valid values
- Escape/forbid shell metacharacters before calling shell; indeed, avoid calling the shell!
 - `& ; ` ' \ " | * ? ~ < > ^ () [] { } $ \n \r`
 - Whitespace are parameter separators – problem?
 - Other possible problems include: `#, !, -, ASCII NUL`
 - Shell often called indirectly (`popen, system, exec[lv]p`)
- Escape/forbid metacharacters of other tools used

Calling Out to Other Resources

- Call only interfaces intended for programs
 - Avoid calling mail, mailx, ed, vi, emacs; they all have exotic interactive escape mechanisms (~, :, !)
 - If you *do* use them, learn their escape mechanisms first and prevent them
- Check all system & library call returns
- Encrypt sensitive information
 - E.G., use SSL/TLS for private data over Internet
 - Encrypt data on disk if it's especially critical

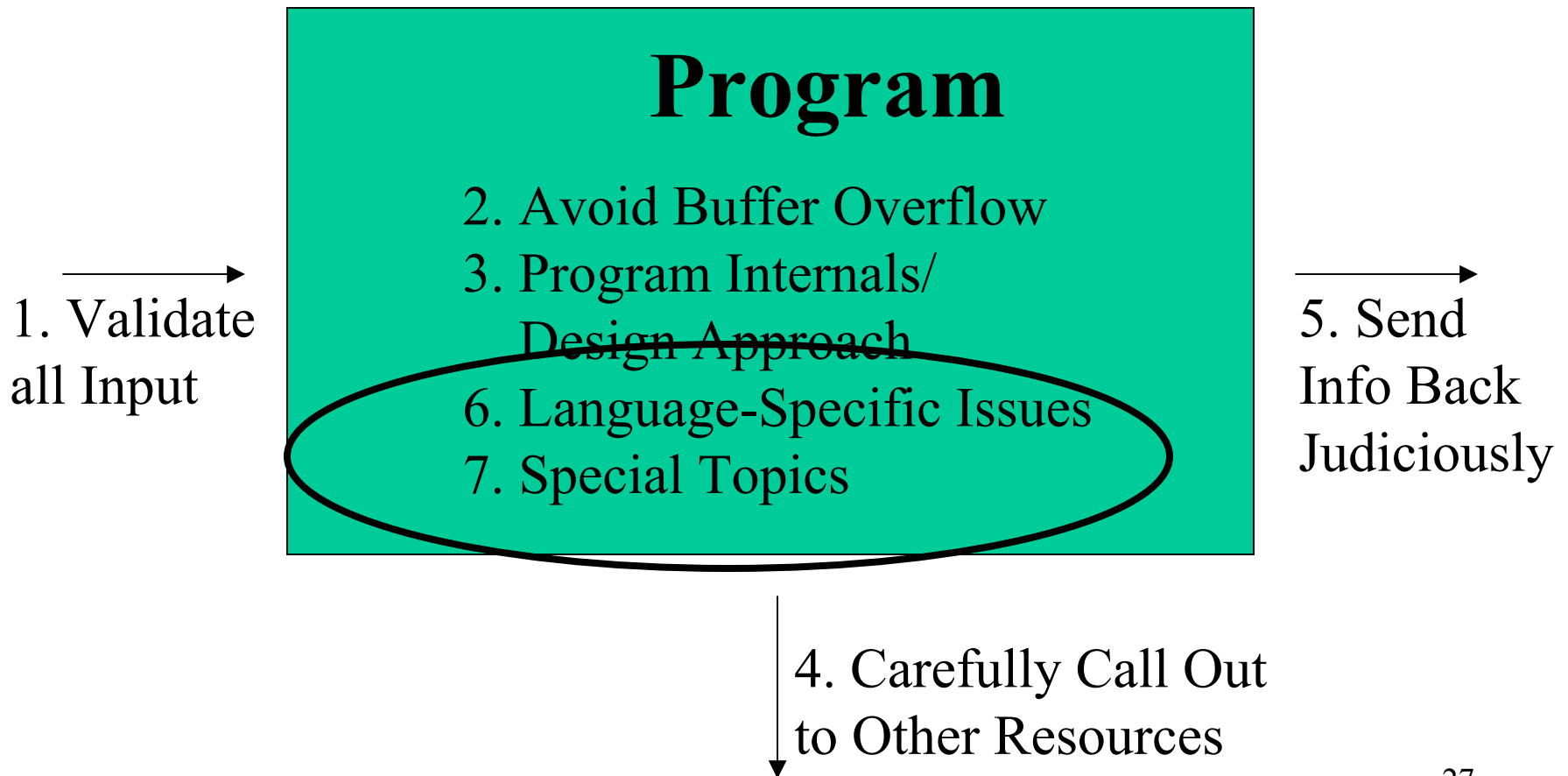
Output Judiciously

- Minimize feedback
 - Log failures - don't explain them to untrusted users
 - Don't send program version numbers
- Handle disk full/unresponsive recipient
- Control data formatting (“format strings”)
 - WRONG: `printf(stringFromUntrustedUser);`
 - RIGHT: `printf(“%s”, stringFromUntrustedUser);`
 - Attacker may use `%n` (writes *into* variables), select “parameters” to output arbitrary stack values, etc.
 - Currently a *major* problem

Output Judiciously: War Story

- PHP < 4.0.3 error logging format string:
 - If error logging enabled, `php_syslog` function called with user-provided data
 - `Php_syslog` called `printf`, using that data as the format string (!)
 - Attacker could cause process to overwrite its stack variables with arbitrary data
 - Allowed remote attacker to “take over” PHP process (usually with web server’s privileges)

Abstract View of a Program



Language-Specific Comments

- Perl:
 - Enable `-w` (warn) and `-T` (taint) options
 - Use 3-parameter `open()` to disable excessive magic (man `perlopentut` for more)
 - “use strict”
- Python:
 - Check uses of `exec`, `eval`, `execfile`, `compile`
 - Function input is very dangerous
 - Don’t use it for untrusted input; use e.g., `raw_input`

Language-Specific Comments

- Shell (sh, csh)
 - Don't use them for setuid/setgid; nonportable
 - Avoid using for secure programs unless heavily protected; too many ways to exploit
 - Filenames with whitespace, control chars, beginning with “-”
 - Magic environment variables (e.g., IFS, ENV)
 - Trusted programs okay if *all* input from trusted sources
- PHP
 - Set register_globals to “off”
 - Use PHP 4.1.0+ and use \$_REQUEST for external data
 - Filter data used by fopen()

Language-Specific Comments

- C/C++
 - Make types as strict as possible
 - Use enum, unsigned where appropriate
 - Watch out for char; signedness varies
 - Turn on all warnings, and resolve them
 - Use gcc `__attribute__` extension to mark functions that use format strings

Special Topics

- Random Numbers: use `/dev/(u?)random`
- Don't send passwords "in the clear" over Internet
- Web Authentication of Users
 - For intranets, use intranet authentication system (e.g., Kerberos)
 - Web basic authentication is in the clear – avoid it
 - Currently client-side certificates are poorly supported, so for many, use "Fu's approach" to authenticate web users (see document for details). Uses passwords over encrypted link, returns a temp cookie used for authentication. Not ideal, but it's practical for most sites

Special Topics

- Protect Secrets (passwords, keys) in user memory
 - Disable core dumps via ulimit; perhaps mmap to prevent swapping out the data; don't use immutable strings to store passwords; erase quickly once used
- Use existing *unpatented* crypto algorithms and protocols; don't invent your own
 - SSL/TLS, SSH, IPSec, GnuPG, Kerberos
 - AES or Triple-DES (*not* in ECB mode-use CBC). RSA
 - For hashing, move from MD5 to SHA-1
 - For integrity checking or MAC, use HMAC-SHA-1
- Have “development” branch (gives time to audit)

Tools

- Source Code Scanners
 - Flawfinder, RATS, LCLint, cqual
- Run random tests to try to crash
 - BFBTester

Conclusions

- Do it right! Avoid well-known problems:
 - Validate all input: Is it all legal?
 - Avoid buffer overflow
 - Structure program: Minimize privileges, avoid race conditions
 - Carefully call out: Shell metacharacters, check all system call return values
 - Reply judiciously: Minimize feedback, format strings
- You'll avoid >95% of reported vulnerabilities
- Be paranoid. They really *are* trying to get you
- See: <http://www.dwheeler.com/secure-programs>

Backup Material

Why Do Programmers Write Insecure Programs?

- “How to write secure programs” is almost *never* taught in schools, even though it’s critical
 - This is criminal! This should be a CS requirement
 - Teach at college & to developers in high school too
- Few books on the topic
- Unnecessarily hard to write secure code in C
- Consumers don’t select products based on their real security-so real security isn’t provided
- Security costs more (in \$, time, installation effort)

What's Open Source Software/Free Software?

- Software licensed in a way giving the freedom to:
 - (0) run the program, for any purpose
 - (1) study how the program works, and adapt it to your needs (requires access to the source code)
 - (2) redistribute copies so you can help your neighbor
 - (3) improve the program & release your improvements to the public, so that the whole community benefits
- “Open Source Software” often emphasizes belief in better results (e.g., higher reliability & security)
- “Free Software” emphasizes freedom for users
- See http://www.dwheeler.com/oss_fs_refs.html

Is Open Source/Free Software Good for Security?

- Some claim OS/FS gives more info to crackers
 - But crackers can disassemble & don't need source code to attack. Transparency helps the “good guys” more
- OS/FS *can* be better over time
 - After “good guys” have found/fixed problems
- But many caveats:
 - People have to actually review the code
 - Reviewers must know how to find insecure code
 - Problems found must be fixed, distributed, applied

Hacker, Cracker, Attacker: These Words Have Meanings

- Hacker: One who enjoys exploring the details of programmable systems & stretching their abilities; enjoys programming; (or) an expert or enthusiast*
- Cracker: One who breaks security on a system*
- Attacker: One who attacks a system
- Note the distinctions:
 - **Not all hackers are crackers** (e.g., white hats)
 - Not all crackers are hackers (e.g., script kiddies)
 - Not all attackers are crackers (e.g., DoS attacks)
- The media still (generally) don't get it

* The New Hacker's Dictionary (The Jargon File), ed. Eric S. Raymond