THE UNIVERSITY OF BRITISH COLUMBIA

Developing Secure Software

Security Flaws

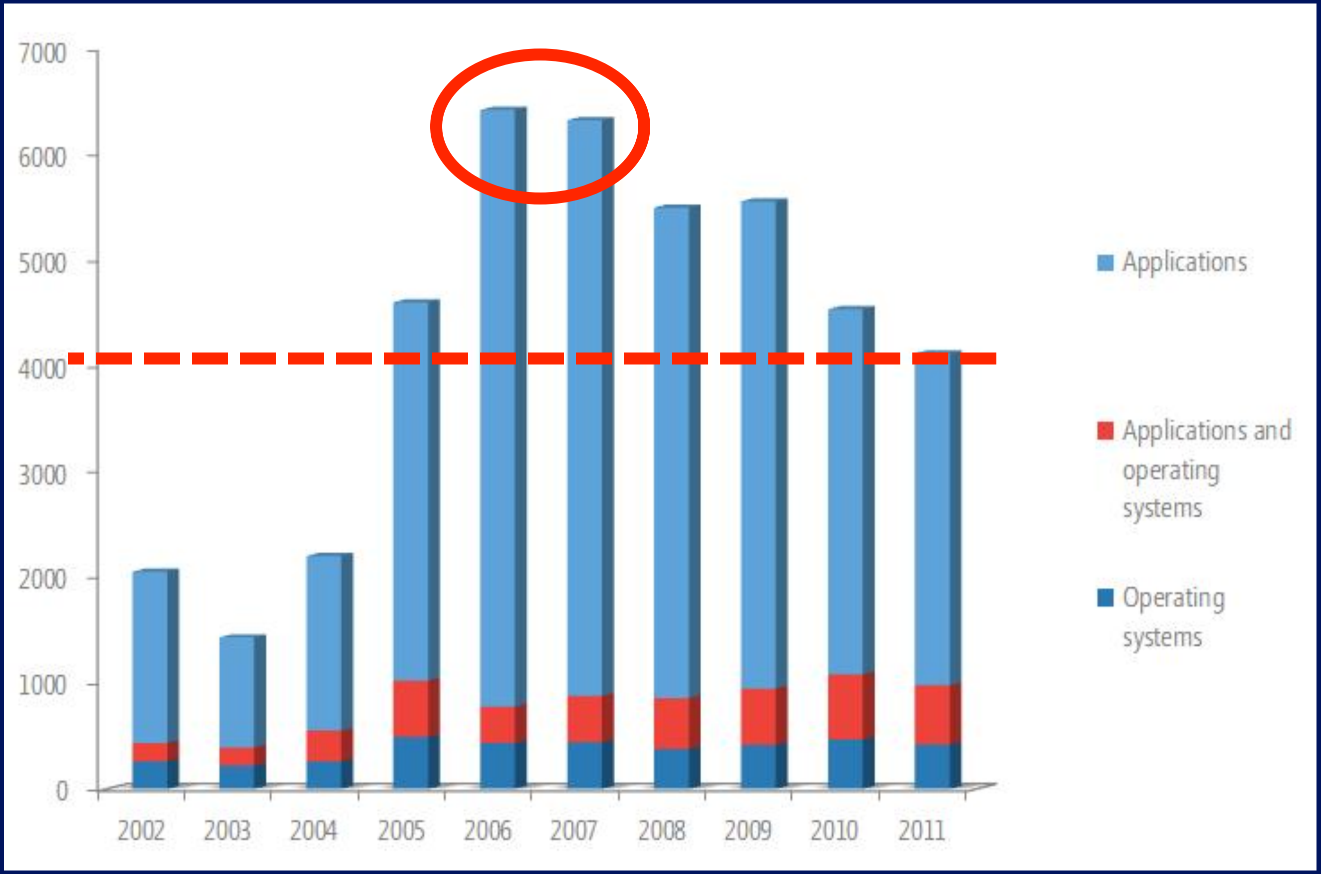
Last lecture

- Malicious logic
  - Malware
    - Stuxnet worm
  - Protection and Detection Techniques
    - Limitation of malware detection
    - Possible solutions
- Non-malicious program errors

Today lecture

Vulnerability Report Statistics

Application and operating system vulnerability disclosures



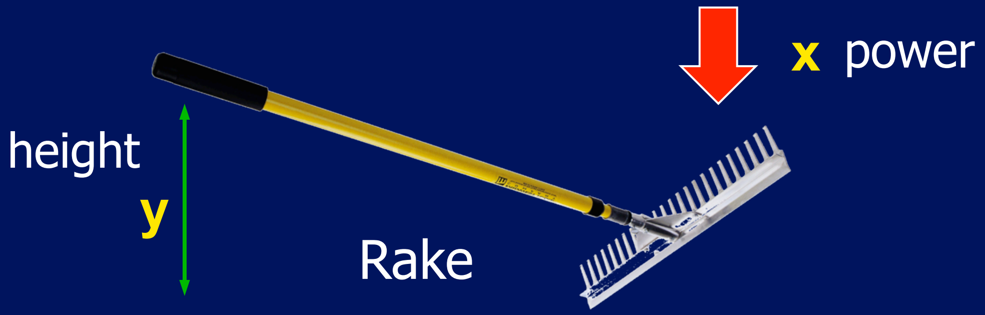
(MS Security Intelligence Report, http://www.microsoft.com/ security/sir/story/default.aspx#!10year_vulnerabilities)

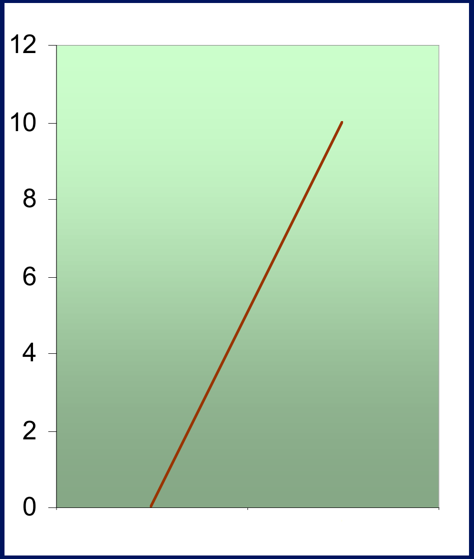
Outline

- Why developing secure software is hard?
- How are security bugs different?
- How does buffer overflow work?
- Guidelines for developing secure software

Why are there so many vulnerabilities in software?

Given **x**, can we calculate **y**?



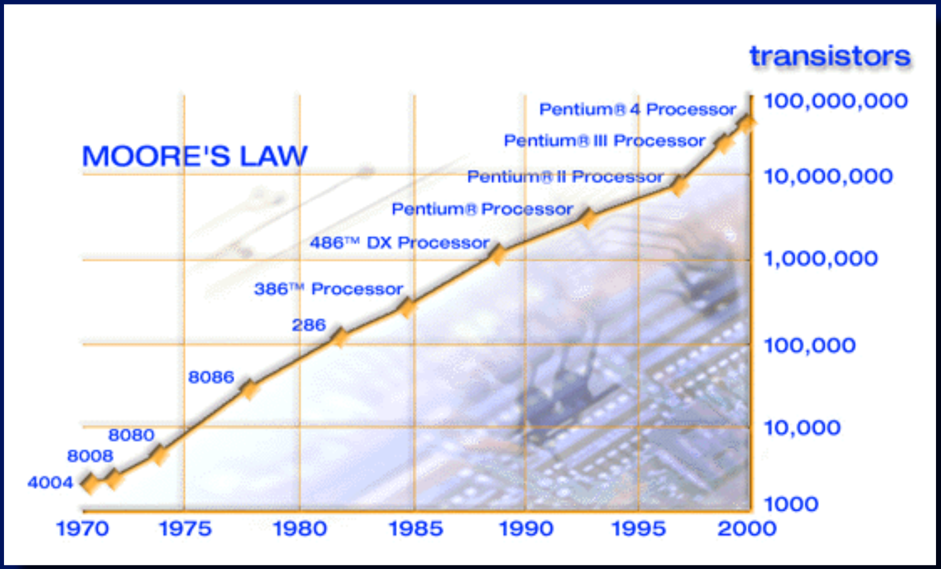


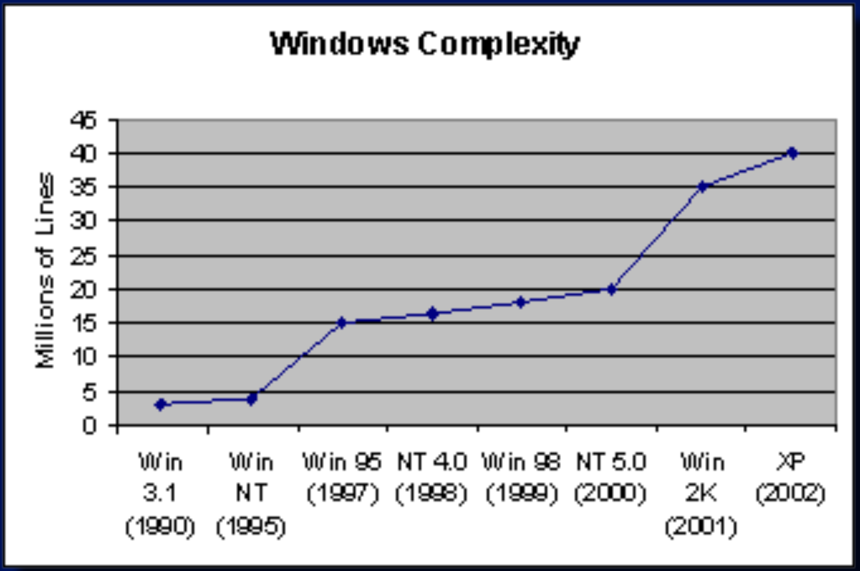
What makes this simple mechanical system predictable?

- Linearity (or, piecewise linearity)
- Continuity (or, piecewise continuity)
- Small, low-dimensional state spaces

Systems with these properties are
(1) easier to analyze, and (2) easier to test.

Increasing complexity of computers





- Computers enable highly complex systems
- Software is taking advantages of this
  - Highly non-linear behaviors; large, high-dim. state spaces

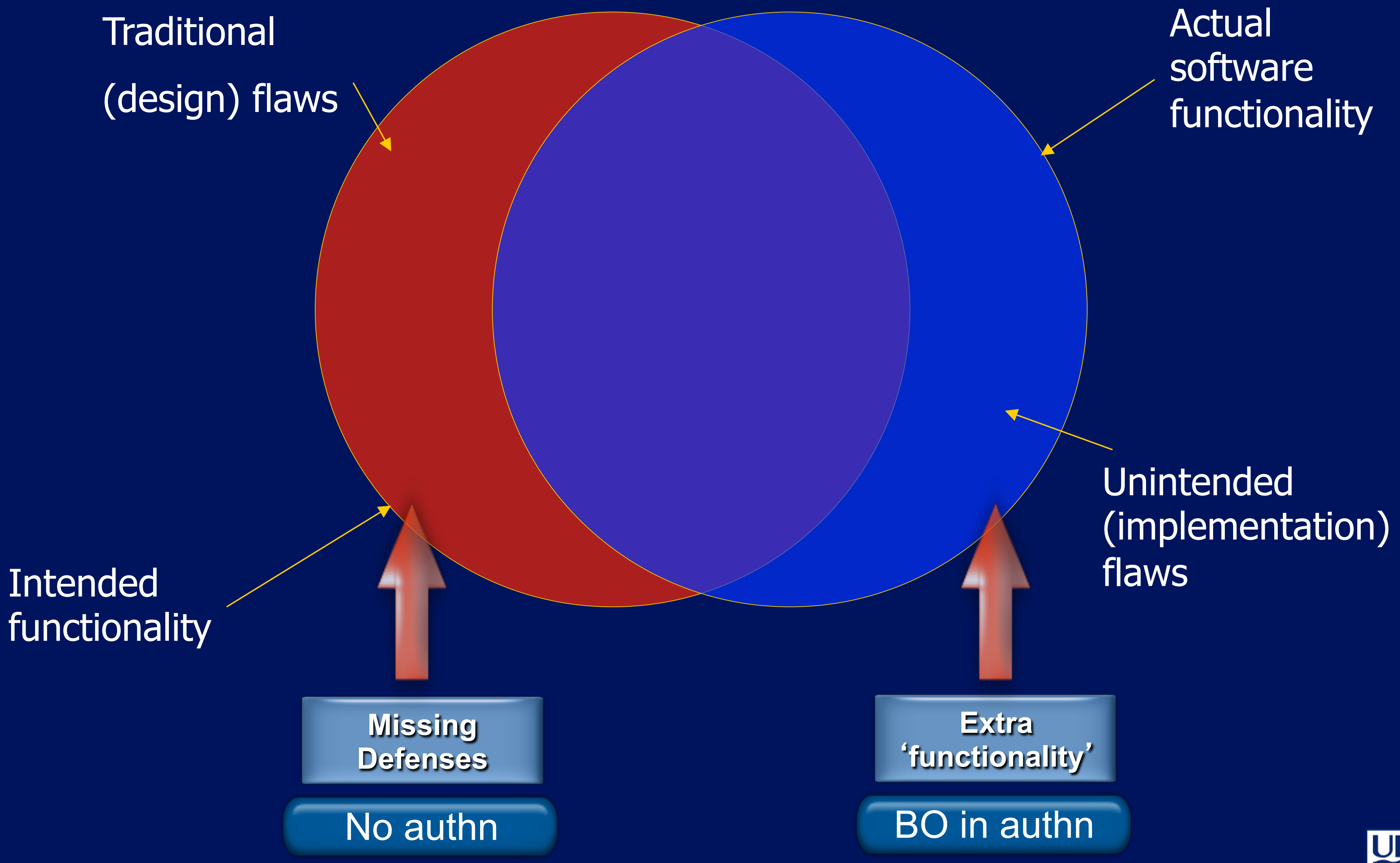
Other software properties make security more difficult

The Trinity of Trouble

- **Connectivity**
  - The Internet is everywhere and most software is on it
- **Complexity**
  - A lot of functions: Networked, distributed, mobile, feature-full
- **Extensibility**
  - Systems can evolve in unexpected ways and be changed on the fly

How Are Security Bugs Different?

Intended vs. Implemented Behavior



Traditional

(design) flaws

Actual
software
functionality

Unintended
(implementation)
flaws

Intended
functionality

Missing Defenses

No authn

Extra 'functionality'

BO in authn

Design flaws

- Incorrect
  - Supposed to do A but did B instead

Weak Authentication

- Missing
  - Supposed to do A *and* B but did only A.

No Authentication

Security problems are complicated

Implementation Flaws

- Buffer overflow
  - String format
- Race conditions
  - TOCTOU (time of check to time of use)
- Unsafe environment variables
- Unsafe system calls
  - System()
- Untrusted input problems

Design Flaws

- Misuse of cryptography
- Compartmentalization problems in design
- Privileged block protection failure (DoPrivilege())
- Catastrophic security failure (fragility)
- Type safety confusion error
- Insecure auditing
- Broken or illogical access control
- Method overriding problems (subclass issues)

Which one is more frequent?

Buffer Overflow

- Can be done on the stack or on the heap.
- Can be used to overwrite the return address (to redirect the control flow of a program).

How Buffer Overflow Works

Adopted from the material by
Dave Hollinger

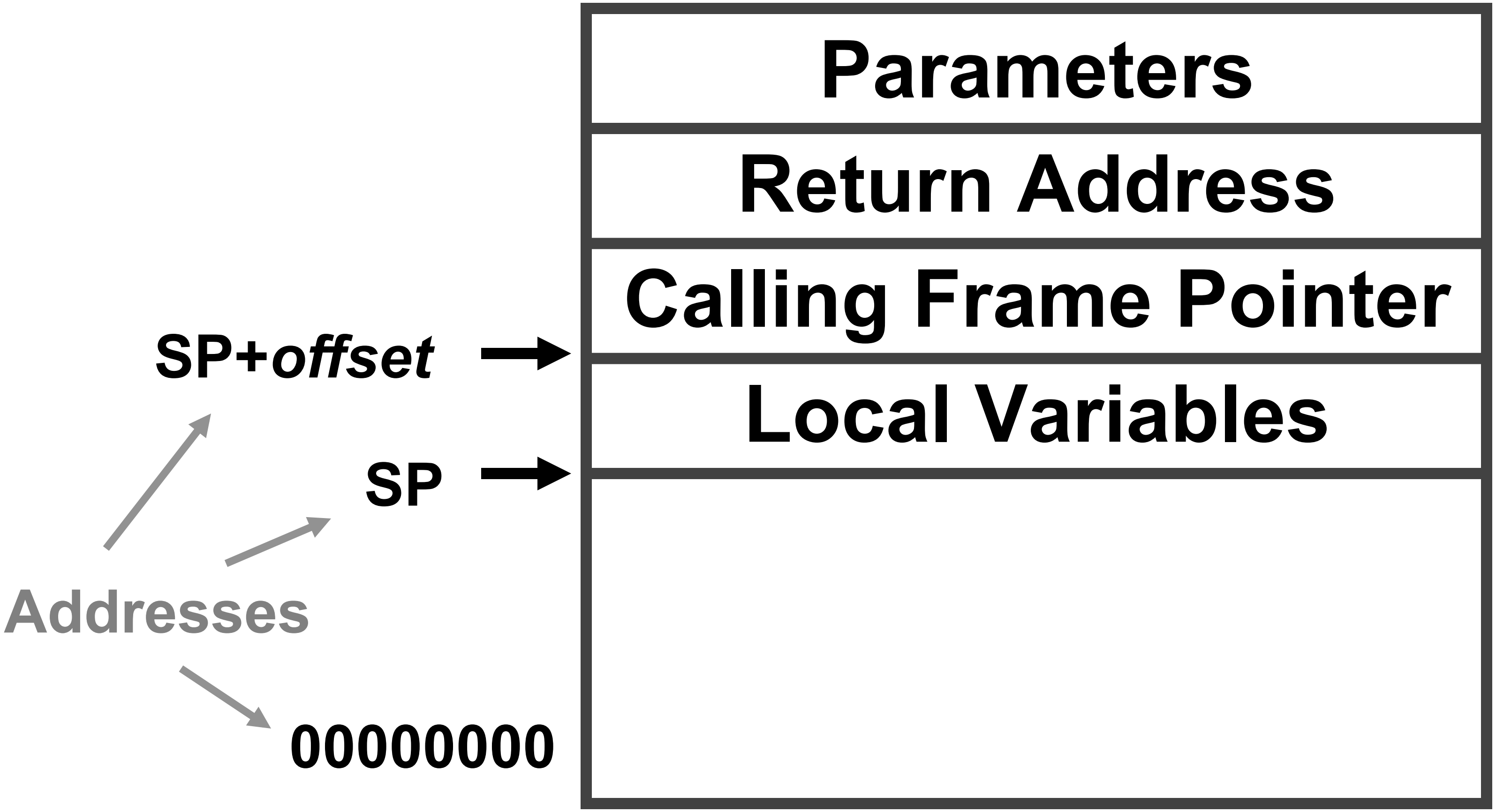
Overview of Buffer Overflow

- The general idea is to overflow a buffer so that it overwrites the return address.
- When the function is done it will jump to whatever address is on the stack.
- We put some code in the buffer and set the return address to point to it!

The Problem

```
void foo(char *s) {
  char buf[10];
  strcpy(buf,s);
  printf(“buf is %s\n”,s);
}
…
foo(“thisstringistolongforfoo”);
```

A Stack Frame



Parameters

Return Address

Calling Frame Pointer

Local Variables

SP+*offset*

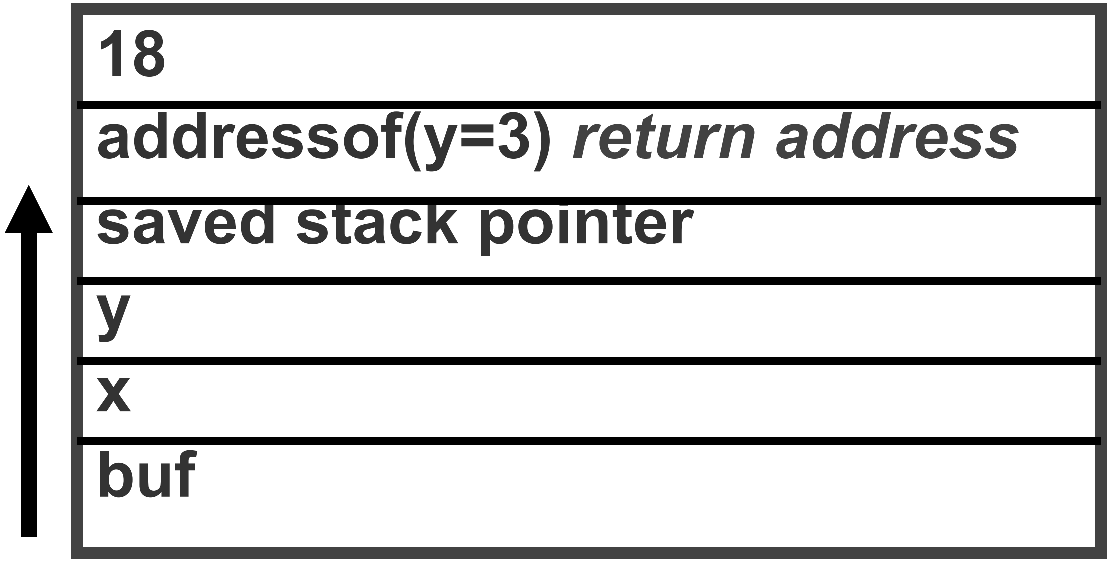
SP

Addresses

00000000

(SP: Stack Pointer)

Sample Stack



18

addressof(y=3) *return address*

saved stack pointer

y

x

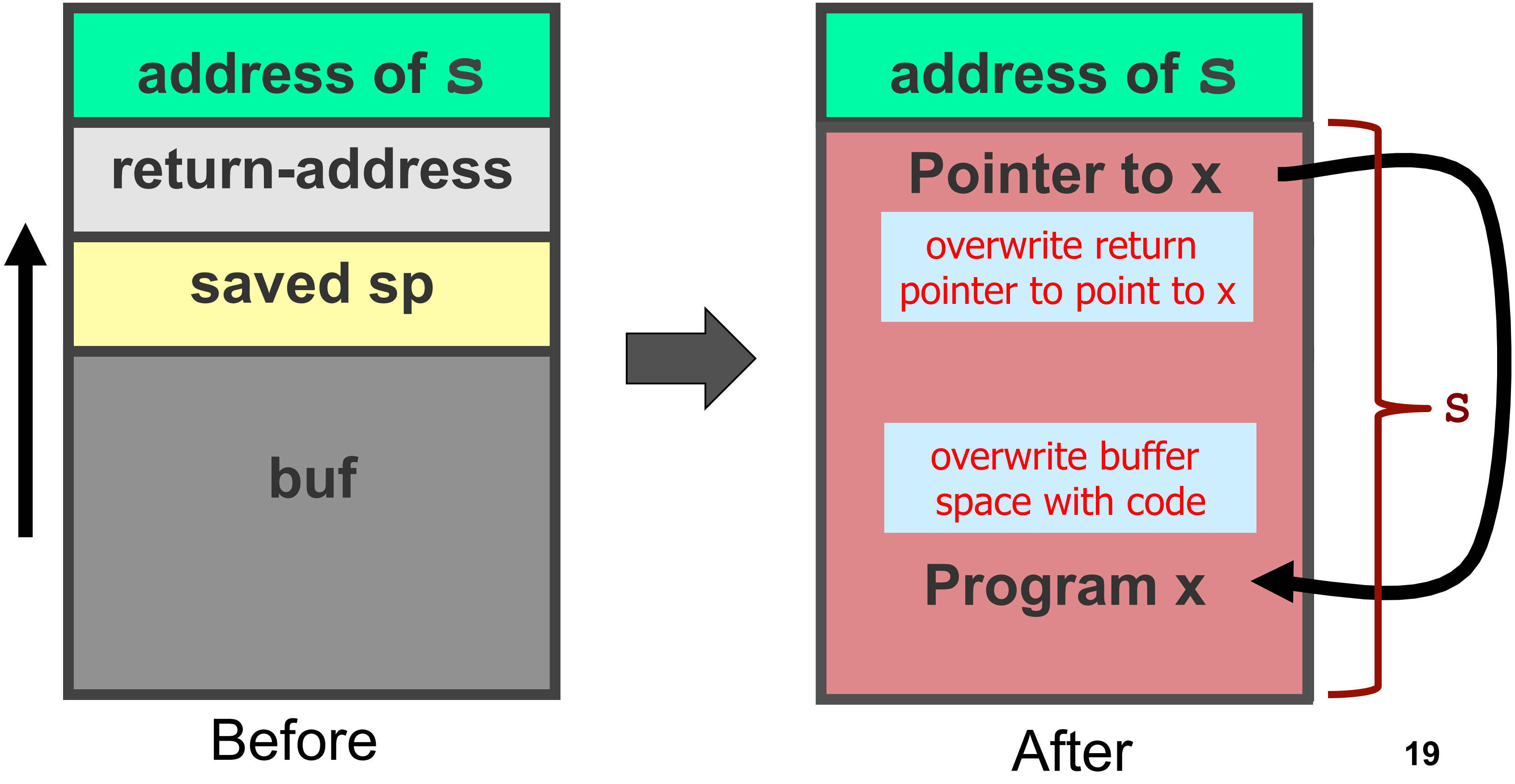
buf

```
x=2;
foo(18);
y=3;
```

```
void foo(int j) {
    int x,y;
   char buf[100];
    x=j;
     …
}
```

Before and After

```
void foo(char *s) {
        char buf[100];
        strcpy(buf,s);
        …
```



Two Issues

1. How do we know what value the pointer should have (the new “return address”).
   - It’s the address of the buffer, but how do we know what address this is?

2. How do we put the “program x” into the string “s”?

An Example of Program x

```
#include <stdio.h>

char *args[] = {"/bin/ls", NULL};

void main(void) {
  execv("/bin/ls",args);
  printf(“I’m not printed\n");
}
```

Generating a String

- You can take code like the previous slide, and generate machine language.
- Copy down the individual byte values and build a string.

A Sample Program/String

Does an execv() of /bin/ls:

```
unsigned char cde[] =
\xeb\x1f\x5e
  \x89\x76\x08\x31\xc0\x88\x46\x07\x89\x4
  6\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d
  \x56\x0c\xcd\x80\x31\xdb
  \x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff
  \xff/bin/ls
```

We use this string for buffer overflow!

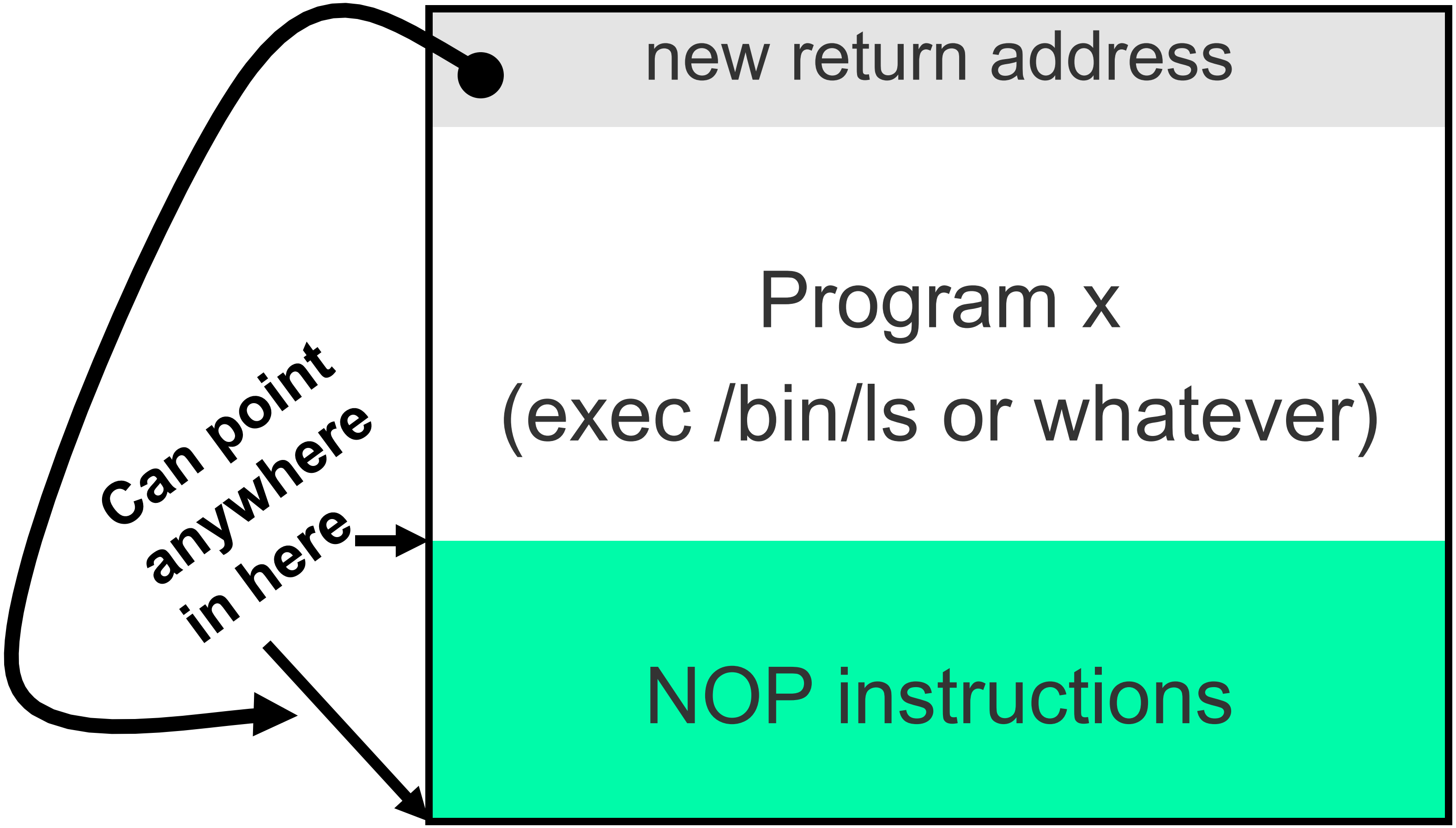
Sample Overflow Program

```
unsigned char cde[] = "\xeb\x1f\…

void foo(char *s) {
   char buf[10];
   strcpy(buf,s);
   printf(“buf is %s\n”,s);
}

int main(void) {
   printf("Running foo\n");
   foo(cde);
   printf("foo returned\n");
}
```

Guessing New Address

- We need to know the address of program x.

Using NOPs



new return address

Program x
(exec /bin/ls or whatever)

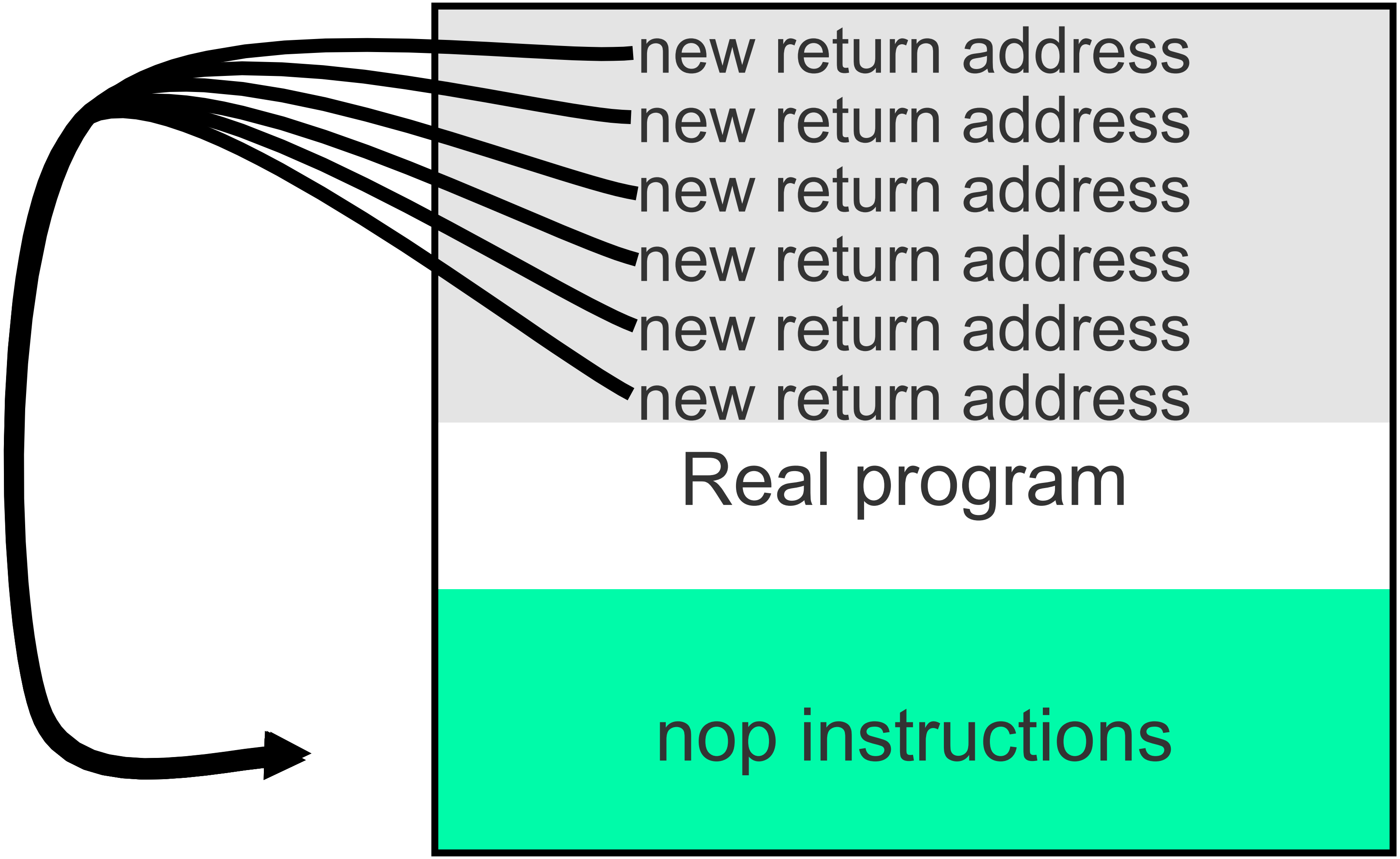
NOP instructions

As long as the new return-address points to a NOP we are OK

Estimating the stack size

- We can also guess at the location of the return address relative to the overflowed buffer.
- Put in a bunch of new return addresses!

Estimating the Location



Demo - Spock

- http://nsfsecurity.pr.erau.edu/bom/Spock.html

- How can you gain access as Dr. Bones when you don't know the correct password?

Techniques for Preventing Buffer Overflow Attacks

- Write or Execute, but not both
  - No program segment loaded into memory is both writable and executable

- Address Space Layout Randomization (ASLR)
  - Prevents an attacker from predicting information needed for correctly changing information flow towards the desirable computation

Pervasive C problems lead to bugs

- Calls to watch out for

Instead of:	Use:
gets(buf)	fgets(buf, size, stdin)
strcpy(dst, src)	strncpy(dst, src, n)
strcat(dst, src)	strncat(dst, src, n)
sprintf(buf, fmt, a1,…)	snprintf(buf, fmt, a1, n1,…) (where available)
*scanf(…)	Your own parsing

- Hundreds of such calls
- Use static analysis to find these problems
  - ITS4, SourceScope
- Careful code review is necessary

THE UNIVERSITY OF BRITISH COLUMBIA

How to Develop Secure Software?

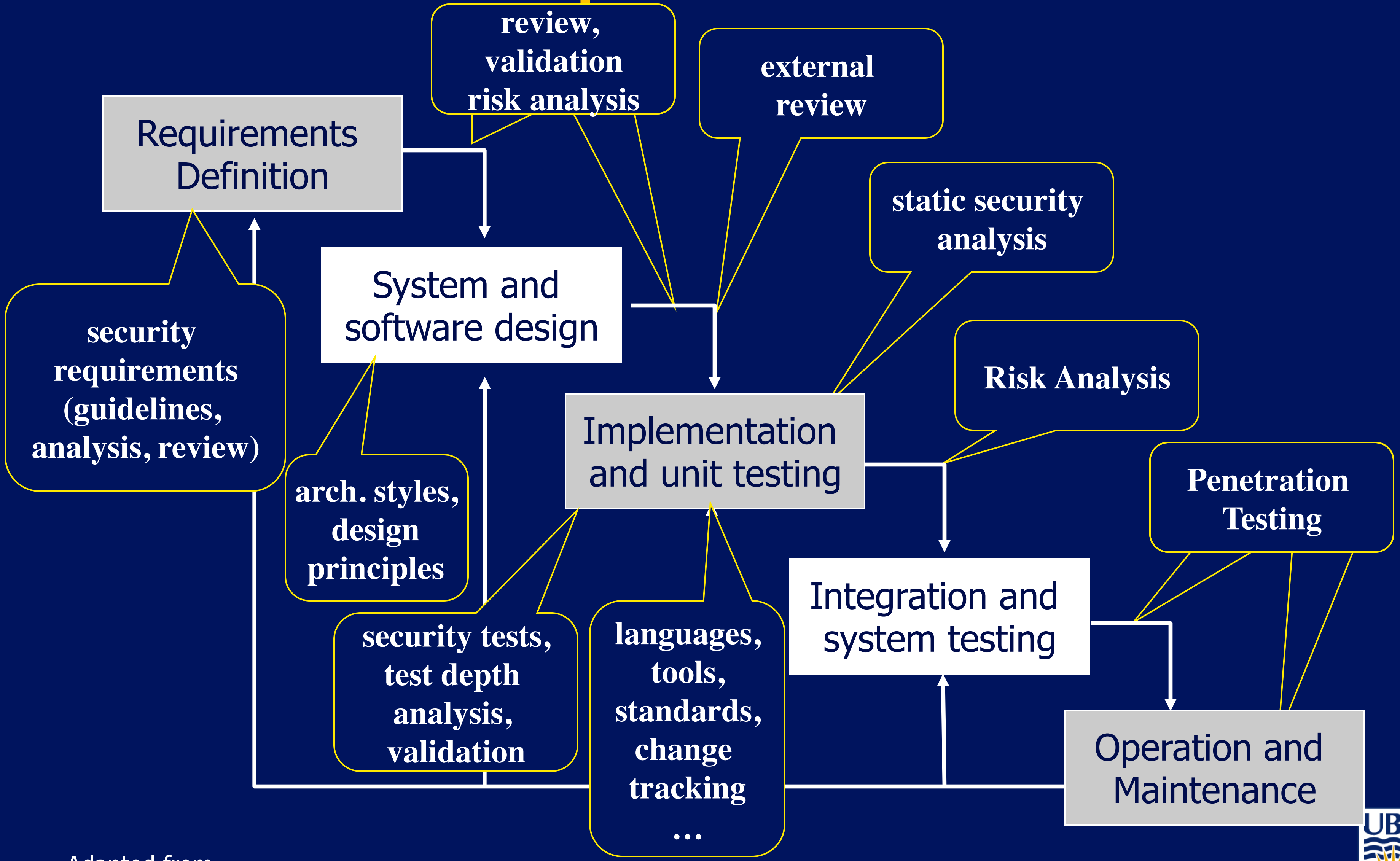
Guidelines

1. Produce quality software
2. Build security into development process
3. Practice principles of designing secure systems
4. Know how systems can be compromised
5. Develop and use guidelines and checklists
6. Choose safer languages, VMs, OSs, etc.
7. Provide tool support

1. Produce Quality Software

- Use well structured effective processes
  - e.g., Capability Maturity Model (CMM), *-CMM
- Use precise requirements and specifications

2. Build Security into Development Process



Requirements Definition

System and software design

Implementation and unit testing

Integration and system testing

Operation and Maintenance

review, validation risk analysis

external review

static security analysis

Risk Analysis

Penetration Testing

security requirements (guidelines, analysis, review)

arch. styles, design principles

security tests, test depth analysis, validation

languages, tools, standards, change tracking …

Adapted from
D. Verdon and G. McGraw, "Risk analysis in software design," *IEEE Security & Privacy*, vol. 2, no. 4, 2004, pp. 79-84.

Follow Best Practices

- These best practices should be applied throughout the lifecycle
- Tendency is to "start at the end" (penetration testing) and declare victory
  - Not cost effective
  - Hard to fix problems
- Start as early as possible

- Abuse cases
- Security requirements analysis
- Architectural risk analysis
- Risk analysis at design
- External review
- Test planning based on risks
- Security testing (malicious tests)
- Code review with static analysis tools

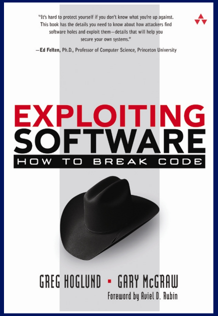
3. Practice principles of designing secure systems

Principles of Designing Secure Systems

1. Least Privilege
2. Fail-Safe Defaults
3. Economy of Mechanism
4. Complete Mediation
5. Open Design
6. Separation of Privilege
7. Least Common Mechanism
8. Psychological Acceptability
9. Defense in depth
10. Question assumptions

4. Know How Systems Can Be Compromised

1. Make the Client Invisible
2. Target Programs That Write to Privileged OS Resources
3. Use a User-Supplied Configuration File to Run Commands That Elevate Privilege
4. Make Use of Configuration File Search Paths
5. Direct Access to Executable Files
6. Embedding Scripts within Scripts
7. Leverage Executable Code in Nonexecutable Files
8. Argument Injection
9. Command Delimiters
10. Multiple Parsers and Double Escapes
11. User-Supplied Variable Passed to File System Calls
12. Postfix NULL Terminator
13. Postfix, Null Terminate, and Backslash
14. Relative Path Traversal
15. Client-Controlled Environment Variables
16. User-Supplied Global Variables (DEBUG=1, PHP Globals, and So Forth)
17. Session ID, Resource ID, and Blind Trust
18. Analog In-Band Switching Signals (aka “Blue Boxing”)
19. Attack Pattern Fragment: Manipulating Terminal Devices
20. Simple Script Injection
21. Embedding Script in Nonscript Elements
22. XSS in HTTP Headers
23. HTTP Query Strings
24. User-Controlled Filename
25. Passing Local Filenames to Functions That Expect a URL
26. Meta-characters in E-mail Header
27. File System Function Injection, Content Based
28. Client-side Injection, Buffer Overflow
29. Cause Web Server Misclassification
30. Alternate Encoding the Leading Ghost Characters
31. Using Slashes in Alternate Encoding
32. Using Escaped Slashes in Alternate Encoding
33. Unicode Encoding
34. UTF-8 Encoding
35. URL Encoding
36. Alternative IP Addresses
37. Slashes and URL Encoding Combined
38. Web Logs
39. Overflow Binary Resource File
40. Overflow Variables and Tags
41. Overflow Symbolic Links
42. MIME Conversion
43. HTTP Cookies
44. Filter Failure through Buffer Overflow
45. Buffer Overflow with Environment Variables
46. Buffer Overflow in an API Call
47. Buffer Overflow in Local Command-Line Utilities
48. Parameter Expansion
49. String Format Overflow in syslog()


“It’s hard to protect yourself if you don’t know what you’re up against. This book has the details you need to know about how attackers find software holes and exploit them—details that will help you secure your own systems.”
—Ed Felten, Ph.D., Professor of Computer Science, Princeton University
EXPLOITING
SOFTWARE
HOW TO BREAK CODE
GREG HOGLUND · GARY McGRAW
Foreword by Aviel D. Rubin

5. Develop Guidelines and Checklists

Example from Open Web Application Security Project (www.owasp.org):

- Validate Input and Output
- Fail Securely (Closed)
- Keep it Simple
- Use and Reuse Trusted Components
- Defense in Depth
- Security By Obscurity Won't Work
- Least Privilege: provide only the privileges absolutely required
- Compartmentalization (Separation of Privileges)
- No homegrown encryption algorithms
- Encryption of all communication must be possible
- No transmission of passwords in plain text
- Secure default configuration
- Secure delivery
- No back doors

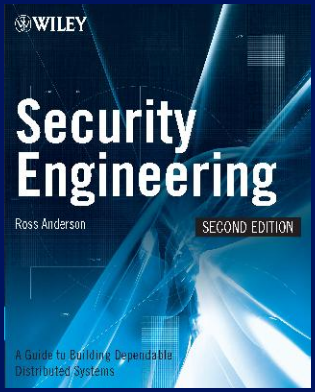
6. Choose Safer Languages, VMs, OSs, etc.

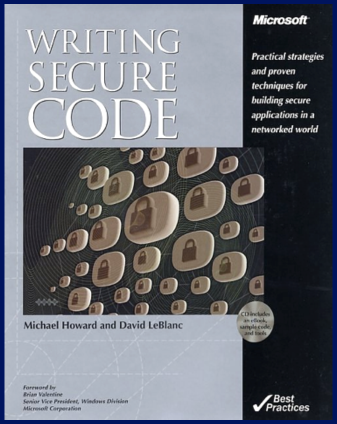
- C or C++?
- Java or C++?
- Managed C++ or vanilla C++?
- .NET CLR or JVM?
- Windows XP or Windows 2003?
- Linux/MacOS/Solaris or Windows?

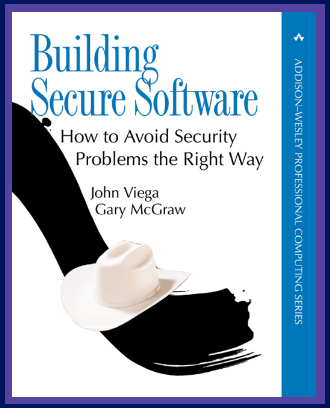
7. Use Good Tools

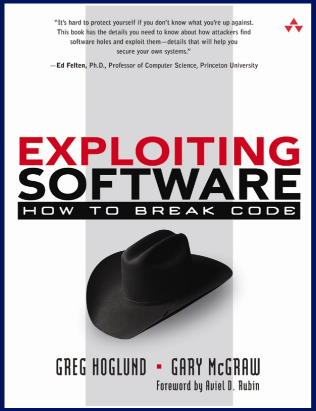
- automated tools for formal methods
  - http://www.comlab.ox.ac.uk/archive/formal-methods.html
- code analysis tools
  - RATS http://www.securesw.com/rats
  - Flawfinder http://www.dwheeler.com/flawfinder
  - ITS4 http://www.cigital.com/its4
  - ESC/Java http://www.niii.kun.nl/ita/sos/projects/escframe.html
  - PREfast, PREfix, SLAM www.research.microsoft.com
  - Fluid http://www.fluid.cmu.edu
  - JACKPOT research.sun.com/projects/jackpot
  - Many more …

Relevant Books




Microsoft
WRITING SECURE CODE
Practical strategies and proven techniques for building secure applications in a networked world
Michael Howard and David LeBlanc
CD includes an eBook, sample code, and tools
Foreword by Brian Valentine Senior Vice President, Windows Division Microsoft Corporation
Best Practices


Building Secure Software
How to Avoid Security Problems the Right Way
John Viega
Gary McGraw
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


"It's hard to protect yourself if you don't know what you're up against. This book has the details you need to know about how attackers find software holes and exploit them—details that will help you secure your own systems."
—Ed Felten, Ph.D., Professor of Computer Science, Princeton University
EXPLOITING SOFTWARE
HOW TO BREAK CODE
GREG HOGLUND
GARY McGRAW
Foreword by Aviel D. Rubin

and many more …

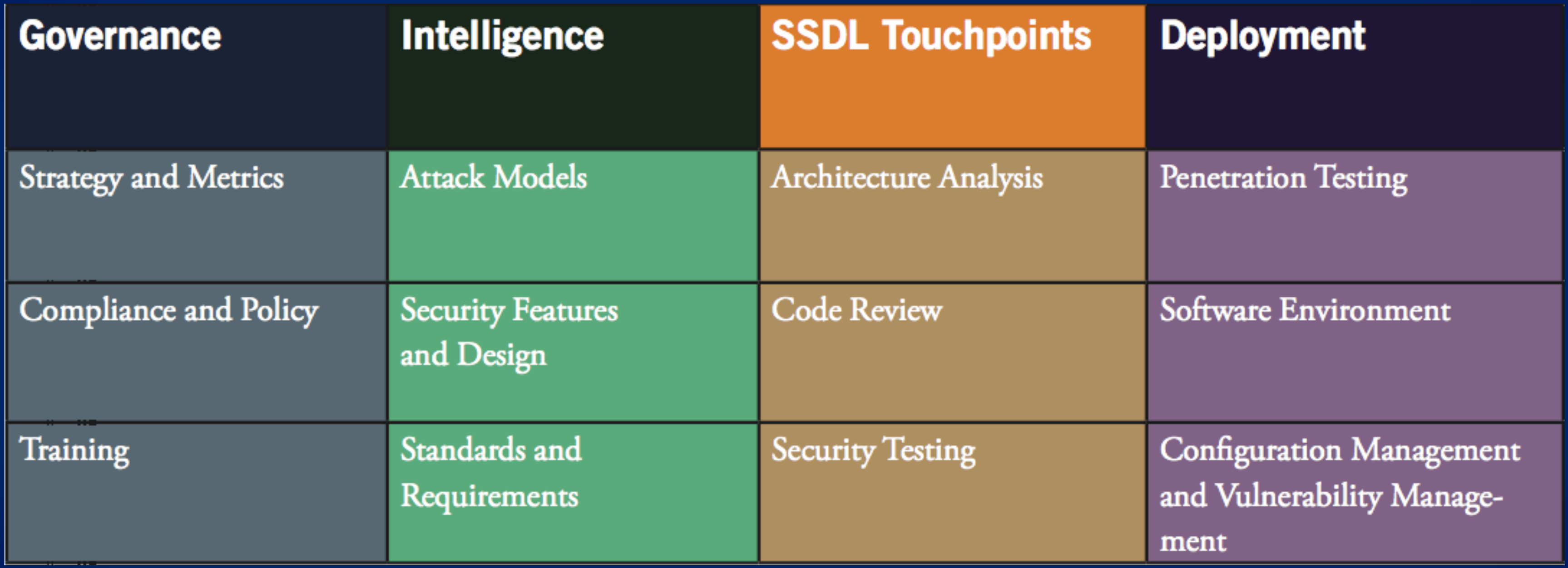
module summary

- developing secure software is hard because it’s
  - nonlinear, large, extensible, complex, has side-effects, networked
- security bugs are different because they are undocumented side-effects
- buffer overflow works through overriding return address and replacing data with code
- guidelines for developing secure software

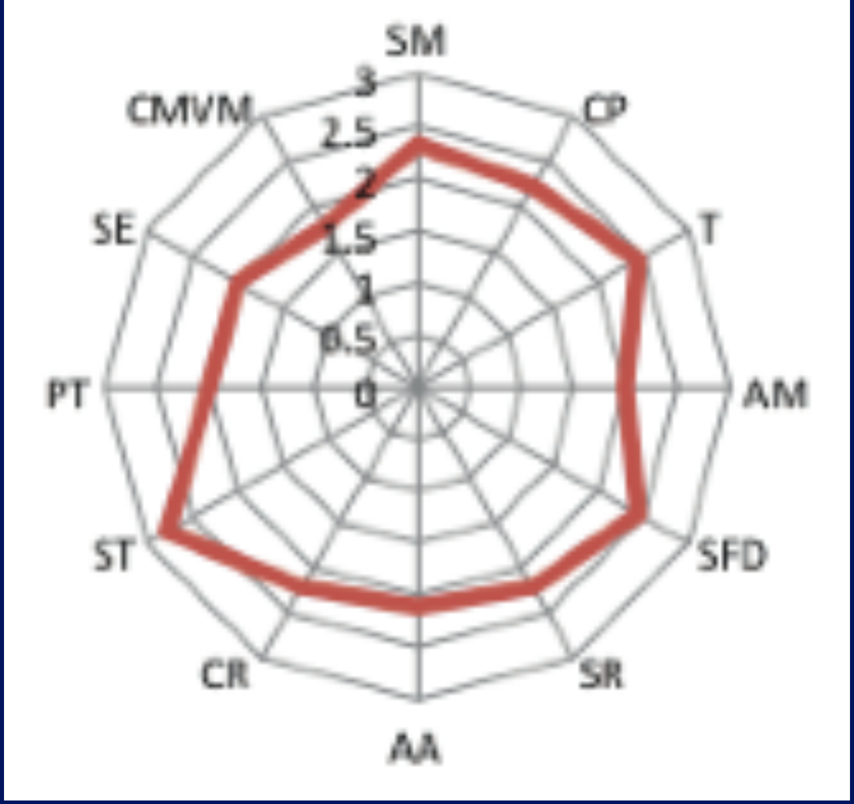
Case Study:
Build Security In Maturity Model

BSIMM

- Framework derived from SAMM Beta
- Based on collected data from 9 large firms



Governance	Intelligence	SSDL Touchpoints	Deployment
Strategy and Metrics	Attack Models	Architecture Analysis	Penetration Testing
Compliance and Policy	Security Features and Design	Code Review	Software Environment
Training	Standards and Requirements	Security Testing	Configuration Management and Vulnerability Management



Source: "Building Security In Maturity Model" by Gary McGraw, Brian Chess, Sammy Migues