LASIGE
Large-Scale Informatics Systems Laboratory

2010 Odisseia no Software

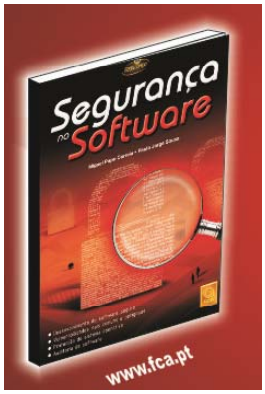
Miguel Pupo Correia

Universidade de Lisboa , Faculdade de Ciências, LASIGE

Confraria de Segurança, TB Store, Nov. 23, 2010

Motivations for giving this talk

- Há tempo que queria falar na Confraria ☺
- Segurança no Software
  - Miguel P. Correia, Paulo J. Sousa
  - FCA, Set. 2010
- http://segurancanosoftware.blogspot.com/



Motivation

- "We wouldn't have to spend so much time, money, and effort on *network security* if we didn't have such bad *software security*"
  - Viega & McGraw, Building Secure Software, Addison Wesley 200
- "the current state of security in commercial software is rather distasteful, marked by embarrassing public reports of vulnerabilities and actual attacks (…) and continual exhortations to customers to perform rudimentary checks and maintenance."
  - Jim Routh, Beautiful Security, O'Reilly, 2010

- "Software buyers are literally crash test dummies for an industry that is remarkably insulated against liability"
  - David Rice, Geekonomics: The Real Cost of Insecure Software, Addison-Wesley, 2007

The problem is Software: Stuxnet

- Malware for industrial control systems, probably those of "critical infrastructures" (power, gas, water,…)
  - Modifies programmable logic controllers (PLCs) that control these systems (weird but software too!)
- Some features:
  - Self-replicates through USB drives exploiting a vulnerability allowing auto-execution
  - Spreads in a LAN through a vulnerability in the Win.Print Spooler
  - Spreads through SMB by exploiting a Windows RPC vulnerabil.
  - Exploits another 2 unpatched privilege escalation vulnerabilities
  - Contains a Windows and a PLC rootkit
  - And many others…
  - *Source: Symantec W32.Stuxnet Dossier, Sep. 2010, version 1.0*

Stuxnet: possible impact

- CNN, Sept. 2007
- "Researchers who launched an experimental cyber attack caused a generator to self-destruct"
  - Financed by the Dep. Homeland Security
  - http://edition.cnn.com/2007/US/09/26/power.at.risk/

Only industry's fault?

- "We at Oracle have (...) determined that most developers we hire have not been adequately trained in basic secure coding principles (…)
- In the future, Oracle plans to give hiring preference to students who have received such training and can demonstrate competence in software security principles."
  - Mary Ann Davidson, Oracle's Chief Security Officer

Mary Ann Davidson Blog

ORACLE

Problem is in the software

The characteristics of current software:

- Complexity
  - Attacks exploit *bugs* called vulnerabilities
  - Estimated 5-50 bugs per Klines of code
  - Windows XP 40M
- Extensibility
  - What software is in your laptop? OS + production sw + patches + 3rd party DLLs + device drivers + plug-ins + …
- Connectivity
  - Internet (1 billion users) + control systems + PDAs + mobile phones + …

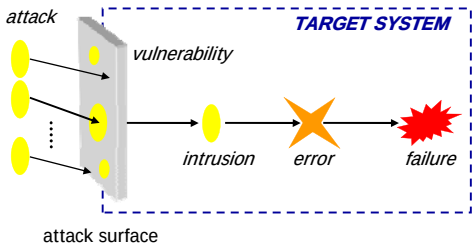
This talk

- Motivation
- The problem: Vulnerabilities
- The solution: Techniques and Tools
- Conclusions

The problem: Vulnerabilities

The problem

- Vulnerability + Attack → Intrusion → Security Failure
  - i.e., violation of confidentiality, integrity, availability



The problem

- From the software point of view, the problem are its defects, i.e., its *vulnerabilities*
  - **Design vulnerability**: inserted during the software design (e.g., lack of access control)
  - **Coding vulnerability**: a bug (e.g., missing end of buffer verification)
  - **Operational vulnerability**: caused by the environment in which the software is executed or its configuration (e.g., weak passwd)

- "the team leaders conveniently assumed that security vulnerabilities were not defects and could be deferred for future enhancements or projects" - Jim Routh, op. cit.

Coding vulnerabilities

There are many classes; we are going to see the top 3:

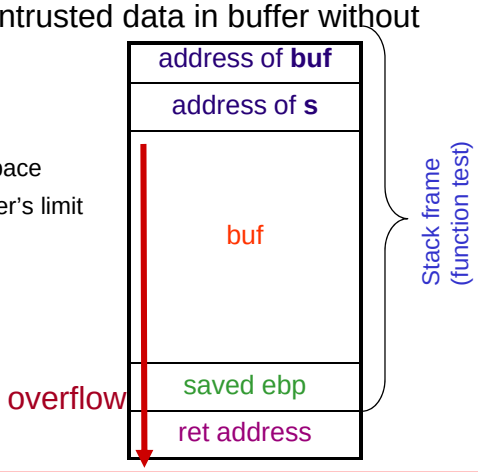
- Buffer overflows – traditionally most important (OSs, binary apps)
- SQL injection
- Cross site scripting

} more advisories than BOs since 2006 (web apps)

BO – Stack Smashing

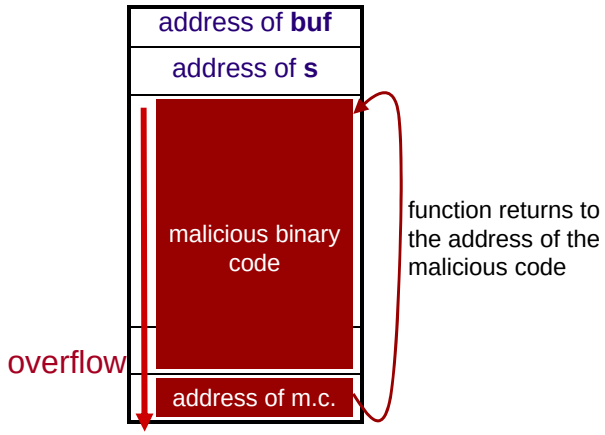
- Stack smashing is the “classical” *stack overflow* attack
- Vulnerable code (inserts untrusted data in buffer without checking the limits):

```
void test(char *s) { //s is untrusted
  char buf[10];       //gcc stores extra space
  strcpy(buf, s);     //doesn't check buffer's limit
}
```



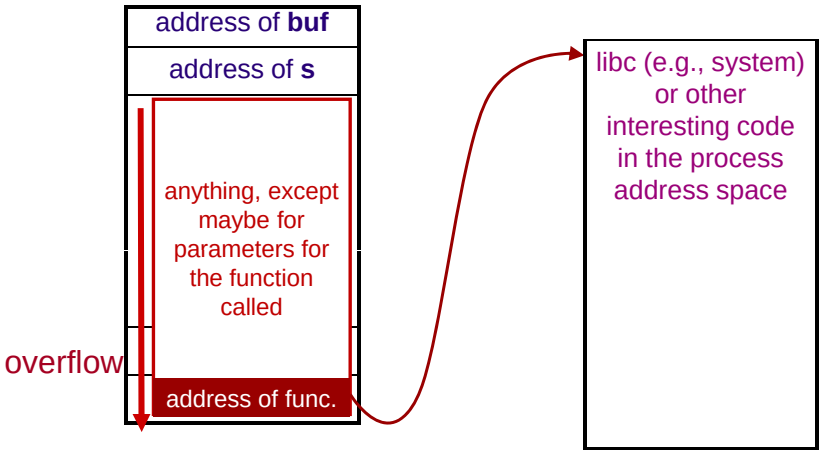
BO – Stack Smash. w/code injection

- Attacker executes arbitrary code in the victim’s machine:



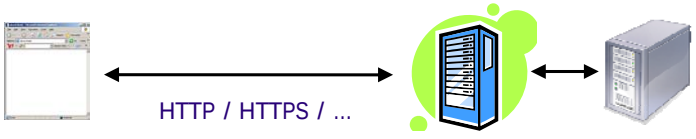
BO – Arc injection / return-to-libc

- Attacker forces jump to code somewhere else:



SQL Injection

- Totally different target: web applications



Client
(browser)
HTML, multimedia
JavaScript
...

Server
HTML
Server side scripting – PHP, ASP,...
...

DataBase

SQL Injection – basic

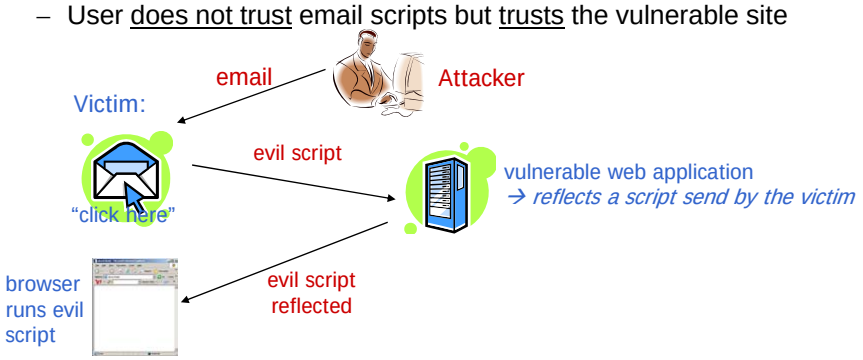
- The attack:
  - User provides inputs to the server
  - Inputs are inserted in queries to the DB
  - Client input with SQL metacharacters inserted in SQL queries
- Example – vulnerable PHP code in the server:

```
$order_id = $HTTP_POST_VARS ['order_id'];
$query = "SELECT * FROM orders WHERE id=" . $order_id;
$result = mysql_query($query);
```

- Good input: 123
  - SELECT * FROM orders WHERE id=**123**
- Attack input: 1 OR 1=1
  - SELECT * FROM orders WHERE id=**1 OR 1=1**

Cross Site Scripting (XSS)

- Also for webapps but the victim is the client/user
- Attack consists in running a malicious script in the browser of the victim (e.g. JavaScript)
- Example:
  - User does not trust email scripts but trusts the vulnerable site



Other vulnerabilities

- Race conditions
- Input validation – command injection, format string vulnerabilities
- Web – session management, direct reference to objects, cross site request forgery, …
- Malicious host – software piracy and tampering, fraud in online applications

- Besides many variants of those we just saw…

The solution: Techniques and Tools

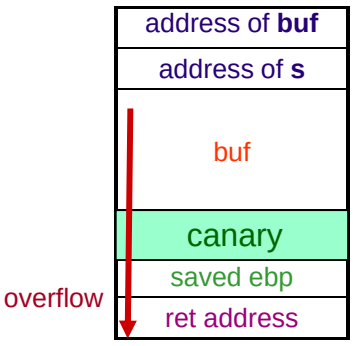
Solution 1 – Robust coding

- Buffer overflows
  - Simply check if there is enough space in the destination buffer
- SQL injection
  - Sanitize the inputs (it's easier to say than do)
- Cross Site Scripting
  - Sanitize the inputs, encode the outputs (but it's also easier…)
- but *errare humanum est,* code can be huge…

Solution 2 – Runtime protection

- Canaries / Stack cookies
  - Like canaries in coal mines
- Compiler introduces canaries and checks

```
void test(char *s) {
  push canary;
  char buf[10];
  strcpy(buf, s);
  …
  if (canary is changed) {log; exit;};
}
```



Solution 2 – Runtime protection

- Address space layout randomization
- The idea is to randomize the addresses where code and data are mapped in runtime
  - The memory layout tends to be the same for every execution
  - Does not prevent exploitation but usually makes it unreliable – what address shall be written over the return address?

Solution 3 – Static code analysis

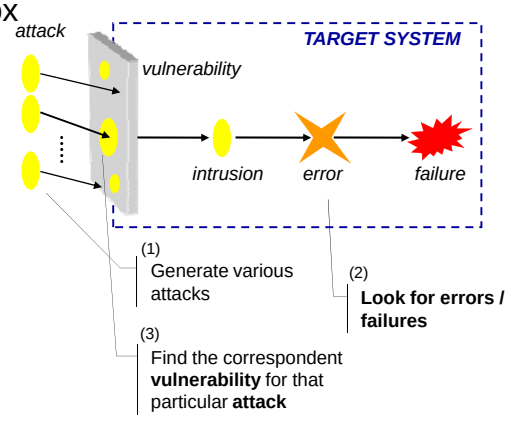
- Vulnerabilities are in the source code so a solution is… to look for them
  - But it's like finding a needle in the haystack
- Code analyzers do it automatically
  - "read" the (source) code and check if certain rules are satisfied (e.g., is memory free'd twice?)
- Commercial tools are available
  - Fortify, Coverity, Ounce Labs

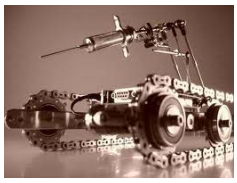
Solution 3 – Static code analysis

- Code analyzers work essentially in two phases
  - Generate an Abstract Syntax Tree – AST (like a compiler)
  - Search for vulnerabilities in the AST; several ways:
- **Syntactic analysis** – check if "dangerous" functions are called (e.g., *gets* almost always vulnerable)
- **Type checking** – check if data is manipulated according to its type (e.g., *unsigned int* = *int* is problematic)
- **Taint checking** – follow the data flow and check if input reaches dangerous functions (e.g., *strcpy*)
- **Control-flow analysis** – follow the control flow paths and do several checks (e.g., if there are double frees)

Solution 4 – Attack injection/fuzzing

- Look for vulnerabilities without delving into the complexity of the software, i.e., looking at it as a black box





Solution 4 – Attack injection/fuzzing

- Fuzzers
  - Late 80s/early 90s Miller/Fredrikse/So were studding the integrity of Unix command line utilities
  - During a thunderstorm one was attempting to use the utilities over a dial-up connection but the utilities were crashing
  - Data was being modified in the line due to noise
  - Thus they developed an utility called *fuzz* to generate random input and test the robustness of software
- Currently used to find vulnerabilities in software
  - Very successfully…

Solution 4 – Attack injection/fuzzing

- Recursive fuzzing
  - Iterating though all possible combinations of characters from an alphabet
  - Ex.: URL followed by 8 hexadecimal digits; try all possible combinations of the 8 digits
- Replacive fuzzing
  - Iterating though a set of predefined values, called fuzz vectors
  - Ex.: look for XSS vulnerabilities by providing the following inputs:
    - >"><script>alert("XSS")</script>&
    - ';!--"<XSS>=&{()}
- Attack injection (AJECT project)
  - Pick a state for the target and an input to inject; put the target in that state; inject; monitor; repeat

Other solutions

- Security-aware software development processes
- Software auditing
- Testing
- Validation and encoding
- Programming language security
- Virtualization
- Trusted computing

- Besides many variants of those we just saw…

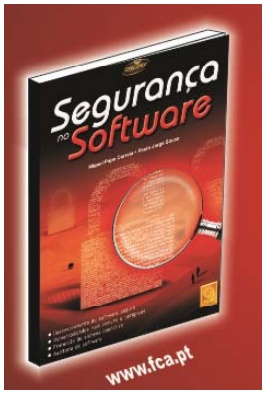
Conclusions

Conclusions

- Software security is important + interesting + difficult
  - New vulnerabilities every day
  - New types of vulnerabilities every year
  - New solutions every…
- Requires
  - Knowing current vulnerabilities
  - Know the new ones that appear (especially new types)
  - Know the solutions and use them
  - Run tools, run tools, run tools

Thank you. Questions?

- To probe further:



- Miguel Pupo Correia
  http://www.di.fc.ul.pt/~mpc/
  http://www.seguranca-informatica.net/