Desenvolvimento de Software Seguro - As Is Can’t Be

MIGUEL PUPO CORREIA

BE AWARE, BE SECURE.
www.ensia.europa.eu/cybersecmonth

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”

- “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

BE AWARE, BE SECURE.
www.ensia.europa.eu/cybersecmonth
2010 in numbers

- Symantec Internet Security Threat Report, Vol 16, April 2011

**286M+**

Threats

Polymorphism and new delivery mechanisms such as Web-attack toolkits continued to drive up the number of malware variants in common circulation. In 2010, Symantec encountered more than 286 million unique variants of malware.

**93%**

Increase in Web Attacks

A growing proliferation of Web-attack toolkits drove a 93% increase in the volume of Web-based attacks in 2010 over the volume observed in 2009. Shortened URLs appear to be playing a role here too. During a three-month observation period in 2010, 69% of the malicious URLs observed on social networks were shortened URLs.

**260,000**

Identities Exposed per Breach

This was the average number of identities exposed in each of the data breaches caused by hacking throughout the year.

**42%**

More Mobile Vulnerabilities

In a sign that the mobile space is starting to garner more attention from both security researchers and cybercriminals, there was a sharp rise in the number of reported new mobile operating system vulnerabilities—up to 16.3 from 11.5 in 2009.

---

2010 in numbers

**6,253**

New Vulnerabilities

Symantec recorded more vulnerabilities in 2010 than in any previous year since starting this report. Furthermore, the new vulnerabilities rose to 1,914, a 163% increase over the prior year.

**14**

New Zero-Day Vulnerabilities

The 14 zero-day vulnerabilities in 2010 were found in widely used applications such as Internet Explorer, Adobe Reader, and Adobe Flash Player. Industrial Control System software was also exploited. In a sign of its sophistication, Snort alert rules used four different zero-days.
The problem is Software: Stuxnet

- Invaded Iranian nuclear enrichment facility; damaged many centrifuges
  - Modified programmable logic controllers (PLCs) – 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 vulnerability
  - Exploits another 2 unpatched privilege escalation vulnerabilities
  - Contains a Windows and a PLC rootkit

The next Stuxnet?

- CNN, Sept. 2007 – “Researchers who launched an experimental cyber attack caused a generator to self-destruct”
  - Financed by the Dep. Homeland Security

- video
The 7 Coolest Hacks Of 2011

- 1. Remotely starting a car via text message.
- 2. Powering down the power plant – literally.
- 4. Insulin pumps go rogue.
- 5. 'Warflying': Hacking in midair.
- 6. When laptop batteries turn against you.

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

hospital
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 Vista 50M

- **Extensibility**
  - What software is in your laptop? OS + production sw + patches + 3rd party DLLs + device drivers + plug-ins + ...

- **Connectivity**
  - Internet (2.2 billion users) + control systems + PDAs + mobile phones + ...

Outline

- The problem: Vulnerabilities
- Solution part 1 - Prevention
- Solution part 2 – Runtime protection
- A taste of research
- Conclusions
The problem: Vulnerabilities

BE AWARE, BE SECURE.

www.enisa.europa.eu/cybersecmonth

The problem

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

TARGET SYSTEM

attack surface

vulnerability

error

failure

intrusion

attack
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 password)

- “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
BO – Stack Smashing

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

```c
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:

```
function returns to the address of the malicious code
```

BE AWARE, BE SECURE
www.enisa.europa.eu/cybersecmonth
**BO – Arc injection / return-to-libc**

- Attacker forces jump to code somewhere else:
  - Address of buf
  - Address of s
  - Anything, except maybe for parameters for the function called
  - Overflow
  - Address of func.
  - Libc (e.g., system) or other interesting code in the process address space

---

**SQL Injection**

- Totally different target: web applications

**Diagram:**
- Client (browser)
  - HTML, multimedia
  - JavaScript
- Server
  - HTML
  - Server side scripting
  - PHP, ASP
- Database

**Diagram Details:**
- HTTP / HTTPS / ...

---

**Source:**
- BE AWARE, BE SECURE
- www.enisa.europa.eu/cybersecuritymonth
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:
  ```php
  $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

Victim: email

```
"click here"
```

Attacker: vulnerable web application

message posted is a script that pops up window
Other vulnerabilities

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

- Besides many variants of those we just saw...

Solution part 1 - Prevention
Consider current laws (Sarbanes-Oxley Act, Health Insurance Portability and Accountability Act (HIPAA)), standards like ISO 17799, others like the Web Application Security Standards, etc.

- Waterfall model

Translation of generic requirements into specific software requirements. Can be done using misuse cases (using common vulnerability lists, resources used by the sw...)

Now considered part of design

Translation of requirements into mechanisms; avoid introducing design vulnerabilities by following design principles, doing risk analysis.
Solution 1 – Sec. development lifecycle

- Avoid introducing coding vulnerabilities using secure coding practices
- Collect information about security events, issue reports and patches
- Microsoft has been doing an excellent job disseminating its Security Development Lifecycle
  - Documentation and tools
Solution 2 – Risk analysis

- For software security, the idea is mainly to find and rank design vulnerabilities.
- Several approaches, one is Threat Modeling; steps:
  - Information gathering: from developers, documentation, code profiling
  - Application decomposition, in attack targets (data flow diagrams, UML)
  - Identify vulnerabilities: by analyzing each component and interaction using a vulnerability taxonomy (e.g., STRIDE)
  - Rank vulnerabilities: to prioritize which to correct first (e.g., with DREAD)

STRIDE taxonomy
- Spoofing identity
- Tampering with data
- Repudiation
- Information disclosure
- Denial of service
- Elevation of privilege

DREAD
- Damage potential
- Reproducibility
- Exploitability
- Affected users
- Discoverability

Solution 3 – Secure 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 4 – 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
  - HP-Fortify, Coverity, Ounce Labs, Veracode
  - Many open, free,...

Solution 4 – 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)
    - **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 5 – Attack injection/fuzzing

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

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 5 – 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

Solution part 2 – Runtime protection
Solution 6 – Runtime protections

- Canaries / Stack cookies
  - Like canaries in coal mines
- Compiler introduces canaries and checks
  ```c
  void test(char *s) {
    push canary;
    char buf[10];
    strcpy(buf, s);
    ...
    if (canary is changed) {log; exit;};
  }
  ```

Solution 6 – Runtime protections

- 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 7 – Language security

- Java’s (later .NET) challenge: running mobile (not trusted) code in a machine
- Solution/part 1: run code in a sandbox
  - Sandbox imposes a security policy to the code: it can only access the resources permitted by the sandbox
  - Sandbox administrator defines the policy
  - Policy depends on the code’s origin (URL) and/or signature
- Solution/part 2: secure the language conventions
  - Type safety – data always manipulated following its type
  - Memory safety – memory accesses restricted to object’s memory space
  - Control flow safety – jumps made only to valid places
- These invariants are enforced in 3 moments: compile time, load time, run time

Solution 8 – Trusted computing

- Trusted Computing Group – an industry consortium defining open specifications for “trusted computing”
- Main achievement is the Trusted Platform Module (TPM) – a chip now found on the mainboard of many PCs
- Two basic functions:
  - Storage of cryptographic keys – for keeping them secure
  - Storage of integrity measurements – to help detect software modifications
Solution 8 – Trusted computing

• What’s inside?

<table>
<thead>
<tr>
<th>Component classes</th>
<th>Subcomponents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional units</td>
<td>Random number generator Hash unit HMAC calculator RSA key generator RSA encryption/decryption/signing</td>
</tr>
<tr>
<td>Non-volatile memory</td>
<td>Endorsement key Storage root key Owner authorization secret key</td>
</tr>
<tr>
<td>Volatile memory</td>
<td>RSA key pairs Platform configuration registers Key handles Authorization session handles</td>
</tr>
</tbody>
</table>

TPM has at least 16 Platform Configuration Registers (PCR)

• A PCR stores (typically) a measurement of a software block, i.e., its cryptographic hash
  • During system boot, the i-th module to run stores the hash of the (i+1)th module in PCRi-1
  • Example: BIOS stores hash(boot loader) in PCR0, boot loader stores hash(hypervisor) in PCR1
  • A vector of PCR values gives a trusted measurement of the software configuration

• Can’t the 1st module provide a false hash of the 2nd?
  • We assume we can trust the 1st module, thus called the Static Root of Trust for Measurement (SRTM)

• Can’t a PCR be overwritten at any time?
  • No, there is no write operation, only extend
  • PCR_i ← H(PCR_i || h) (the 1st time, PCR_0=0)
Solution 8 – Trusted computing

- Remote attestation: computer gives to challenger a measurement of the software configuration, i.e., a vector of PCR values
  - Challenger has the Endorsement Key Certificate, signed by the TPM vendor (means it’s a real TPM!)

1- Request attestation
2- Request TPM vector of PCRs signed with EK
3- PCR vector (signed with EK)
4- Verify signature and if PCR values match a trusted configuration

Other usages for the TPM:
- Store cryptographic keys
  - Inside the TPM or outside of it but encrypted by it
- Bind release of cryptographic key to a certain software configuration
  - Used in Microsoft’s BitLocker Drive Encryption (but optional and typically disabled)
- Assign unique id to a blob of data
  - Using a TPM counter and its signature
A taste of research

BE AWARE, BE SECURE.
www.enisa.europa.eu/cybersecmonth

Static analysis with code correction

- Preliminary
- AWAP: static analysis of PHP code + automatic correction
  - When a vulnerability is found, a “patch” is inserted
  - Also, a detailed report is created for the programmer to learn with the mistake
- Example of vulnerable code:
  - $a = $_GET['user'];
  - $b = $_GET['pass'];
  - $query = "SELECT * FROM users WHERE u = \"\$a\" AND p = \"\$b\";"
  - $r = mysql_query($query);
- Corrected code:
  - $a = mysql_real_escape_string($_GET['user']);
  - $b = mysql_real_escape_string($_GET['pass']);
  - ...
Intrusion tolerance with replication

- Basic idea
  - Accept the inevitability of vulnerabilities and successful attacks
  - Replicate in several and guarantee that it works as long as no more than \( f\) are suffer intrusions
  - Example instantiation: a storage cloud-of-clouds

- Example instantiation:
  - Amazon S3
  - Nirvanix
  - Rackspace
  - Windows Azure

Benefits:
- Can tolerate data corruption, e.g., due to malicious insiders, successful attacks, accidental faults (e.g., due to bugs)
- Can tolerate datacenter and cloud outages
- No vendor lock-in
- Confidentiality (data is encrypted)

Costs
- \$ cost doubles
- Reads become faster
- Writes become slower
- (experiments with 437000+ reads/writes between Sep. 10th and Oct. 7th 2010, clients scattered through the world, from Brazil to Japan)
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

Much research going on
- THIS SOFTWARE IS PROVIDED “AS IS” is not acceptable
Thank you. Questions?

- To probe further:

- Miguel Pupo Correia
  http://homepages.gsd.inesc-id.pt/~mpc/
  http://www.seguranca-informatica.net/