

Principles and fundamentals of security  
methodologies of information systems - Secure  
coding good practices

M2SSIC-Metz

Pascal Steichen

# Contents

<b>1 Introduction</b>	<b>3</b>
<b>2 Architecture</b>	<b>3</b>
2.1 Architectural Document . . . . .	3
2.2 Principles of security architecture . . . . .	4
<b>3 Design</b>	<b>8</b>
3.1 Principles of security design . . . . .	9
3.2 Some special design issues . . . . .	11
<b>4 Implementation</b>	<b>12</b>
4.1 Principles of security implementation . . . . .	12
<b>5 Operation</b>	<b>14</b>
5.1 Principles of security operation . . . . .	15
<b>6 Testing</b>	<b>16</b>
6.1 Recent web application vulnerabilities . . . . .	17
<b>7 Bibliographic references</b>	<b>19</b>

# 1 Introduction

Secure coding is not simply getting behind it's keyboard and hack, it is real engineering.

Why do bridges support trains for hundreds of years?

How could an Eiffel Tower swing several meters sideways in the wind and still welcome thousands of visitors per day, without harm?

Well engineering is taken serious. First there is an **architectural** model, which will provide **design** plans (blueprints), only then construction (**implementation**) can start.

Why shouldn't we (IT people) adopt this procedure too?

Well we should!

# 2 Architecture

A *security architecture* is the process of selecting design elements and principles to match a defined security need. This implies to know how secure the program should become !

**MEMO:** job candidate test: how secure could you make our systems ?

A good security architecture can be applied many times, to many applications and should serve as a framework for secure design decisions. A good advice is to work with an *architectural document*, where the different aspects are laid down.

## 2.1 Architectural Document

- Program organization
- Change strategy
- Buy versus build decisions
- Major data structures

- Key algorithms
- Major objects
- General functionality
- Error processing (corrective or detective)
- Active or passive robustness
- Fault tolerance

## 2.2 Principles of security architecture

- Ask questions

About what ? Some basic topics, are:

- About the worries
- About the resources
- About the software itself
- About the goals

And get answers before continuing the architecture process.

- Focus before leaping

It is important to know the destination before stepping on the gas. Engineers are problem solvers by nature, so first the problem should be well defined as well as the goals to reach.

**MEMO:** building example (skyscraper or one family home)

- Define "just secure enough"

The idea is not to make an application *as secure as possible*, but *just secure enough*. Ressources (of any type) are expensive so a precise definition on how secure an application should be, that is appropriate to the (business) context (it's environment), is crucial.

- Do engineering not prototyping

Employing standard engineering/construction techniques for software is critical for a good development process. Good security requires good design and good design techniques. This can avoid factors like :

- lack of any design
- simple human weakness
- poor coding practices

**MEMO:** montgolfiere vs. wright example

- Identify assumptions

Good engineers have in common the ability to look objectively at elements like mental models, system resources and process interruptibility and suspension. This enables clear identification of assumptions to be handled correctly.

E.g. *The users of this application will be human beings.*

- Get security in from day one

Research reveals that fixing a security bug in the design phase costs 1/60th of the cost for the same bug to be patched after the release. Planning with this in mind does save costs, maintenance and as such security of the application.

- Design with the enemy in mind

This is the *adversary principle*. It's important to try to anticipate how an attacker might approach solving the security puzzle of a specific application. This implies knowing a little of the usage environment of the application, to identify potential enemies.

- Work with the chain of trust

The chain of trust must be understood and respected, it's not save to invoke untrusted programs from within trusted ones. Secure applications should always validate what is presented to it, should not pass tasks to less-trusted entities and only emit valid and safe information.

- Be stingy with privileges

Sometimes called *principle of least privilege*, the idea is to limit privileges to it's uther most needed.

**MEMO:** the 4As (authentication, autorisation, accounting, audit)

- Always test against policy

Every attempted action must be tested against policy to get a stringent security, this is also called the *principle of complete meditation*.

**MEMO:** file handle example (?)

- Build in fault tolerance

To build appropriate levels of fault tolerance, the 3 Rs, from the CERT-CC, come in the play:

- Resistance (deter attacks)
- Recognition (recognize attacks and potential impacts)

– Recovery (provide full service recovery after attacks)

- Address appropriate error-handling

This should indeed be considered at every stage of a software's lifecycle:

**Architect** Adopt a general plan for error handling, like stop on bizarre (unexpected) issues and log on all others.

**Designer** Devise rules on how to detect, discriminate and respond to errors.

**Coder** Capture the triggers and implement the design.

**Operator** Needs to check processes and if error handling is efficiently done.

- Degrade gracefully

Graceful degradation is the fact to continue operating in a restricted or degraded way, if something fishy happens, instead of simply stopping or failing.

**MEMO:** car crumple zones example and VMS (memory) example

- Fail safely

What should be the default fail behavior of a given application ?

**MEMO:** examples: firewall, lounge-machine

- Choose safe defaults

The "fail-safe" argument from before is somehow a sub-element of this: the need to provide safe default actions in general.

**MEMO:** nuclear plant cooling water example

- KISS (Keep It Simple Stupid)

Simple systems are easier to design, implement and test well, moreover, features that don't exist can't be a security risk, nor can have bugs.

"A good theory should be as simple as possible - but not simpler."  
Albert Einstein

- Modularize

Modularize thoroughly and fully.

**MEMO:** postfix example

- Don't rely on obfuscation
 

Security through obscurity doesn't work ! Concealing how something works (like encryption algorithms) is in no sense an advantage, but can, in contrary, be dangerous. Security should be intrinsic.

**MEMO:** especially useful in client side stuff (javascript, applets, ...)
- Seek statelessness
 

By state we think of the information a program retains while a transaction (or command) is executed. If a program retains minimal state, it's harder for it to get into a confused, disallowed state.
- Strive for practical measures and useability
 

In theory there should be no difference between theory and practice, but in practice, there is. That's why it's important to create a useable user environment (GUI, etc.) that makes it easy to do the right thing. This is also called the *principle of psychological acceptability*.
- Make accountability always possible
 

A good architecture must ensure that every action, as well as the responsibility of the assets, can be attribute to a specific individual.
- Limit resources consumption
 

A gentle resource usage contributes to the overall security of a system. Strange or seldom tested exceptions are often only detected when reaching the limits of resources exhaustion.

However, resource-consumption limitation, must also be combined with meaningful error recovery and handling to be really effective.
- Make event-reconstruction possible
 

It must be possible to track the exact sequence of events of key actions. This implies keeping audit-logs, it's the *principle of auditability*.
- Eliminate "weak links"
 

"A chain is only as strong as it's weakest link." Try to eliminate them. This implies planning the security as a whole, addressing the entire range of elements of a specific application.
- Use multiple layers of defense
 

In depth defense is the key, wear a belt *and* suspenders !

**MEMO:** passwd and perms example

- View things in it's holistic whole

This goes even farer than eliminating "weak links", it's not enough to create a secure application by it's own. All it's interactions with the outside world must also be considered and from design on.

**MEMO:** do developers think about remote admin access as an at-tacker way in ?

- Reuse secure code

Don't reinvent the wheel ! Make use of code, libraries or whole programs, that are known to be secure. It might even be interesting to have "code reuse" clauses in the policies.

- Don't rely on off-the-shelf software

However, despite the previous point, code reuse is not to be taken an easy task. For the sake of security issues off-the-shelf programs have to be treated very carefully, especially for mission critical operations. Make sure to assess the security aspects of any solution intended for in-house usage.

**MEMO:** prefer open-source

- Don't forget democratic principles

Security should stay security. Don't implement security measures that mistreat individual privacy rights. In most countries there is a legal framework for this, regard it.

**MEMO:** mainly in regard to the 4As (authentication, autorisation, accounting, audit)

- What did I forget ?

Always ask yourself this question before moving along, humans forget things !

### 3 Design

Good design is the basis for an efficient software development process, as it not only enables to build a good defensive basis into the software from the beginning, but provides safe foundations for future extensions and maintenance.

**MEMO:** efficiency = effectif and performant

Secure design has to be elaborated thoroughly, the following steps being typical efforts to perform.

### 3.1 Principles of security design

- Risk assessment

Before starting to protect, one has to know and understand what is to be protected and from whom. This process is commonly called a risk analysis, trying to determine the threats, the vulnerabilities and their impacts.

It probably sounds a little strange to perform a risk analysis for the sole purpose of developing an application, as such an analysis normally includes the whole organisational aspects as well. Well that's exactly what is sought. The aim is to define all the implications the new element will have on the whole and vice-versa.

Despite the specific method chosen, be it OCTAVE, NIST SP800-30, EBIOS, ..., the following essential points should be addressed:

- How does the organization adopt the application ? Are there existing directives ?  
**MEMO:** users ? recovery plan ? mission ?
- What influence will the application have on the organization ?  
**MEMO:** downtime ? importance ? impact ?
- What about the information manipulated by the application ?  
**MEMO:** importance ? flow ? type ? CID ?

- Risk mitigation

After the risks were assessed it is time to think about the management of those risks. Risk is good : *Having* risks is a sign that one's still in business, *managing* risks is a good way to stay in business.

Again dependant on the specific method chosen, this process might diverge slightly. Generally the options of risk management are:

- Risk assumption **MEMO:** grin and bear it
- Risk avoidance **MEMO:** eliminate vulnerability
- Risk limitation **MEMO:** minimize impact
- Risk planning **MEMO:** rely on recovery plan
- Risk acknowledgment and research **MEMO:** warn and solve
- Risk transference **MEMO:** use insurance

- Work with a mental model

This practice might somehow look strange, but it is a nice way to get a global picture of the design. Trying to build a metaphor of the future application, can already show some potential issues and suggest

solutions. Again, Albert Einstein, with his famous "Gedankenexperimente" can be taken as a prominent example of the effectiveness of such an approach.

- Define high-level techniques

Only now the more technical work can begin. Technical issues can be divided in 3 categories:

- those relating to the application's interaction with itself or other software ;

**MEMO:** platform/lang/DB ; 3 AAA (auth, authorize, audit) ; process comm ; safeguards

- those relating to network interaction:

**MEMO:** comm ; access ; topo (DMZ)

- the specific defenses against attack.

**MEMO:** DoS ; race cond. (atomicity) ; concealment ; user interaction

- Choose appropriate measures

It's not enough to cover, threats, vulnerabilities and attacks, implementation means more. Detailed and carefully chosen measures are the key for success. Various good practices exist, here some thoughts to consider.

- Background factors

Get the "big picture" of the environment and it's customs.

- \* User "culture"

- \* Technical traditions

- \* History of security issues (successful attacks, etc.)

- \* Potential targets

- Business issues **MEMO:** use auth mechanisms example all over

Address corporate culture and cost issues, important factors are:

- Maintenance

- Purchase cost

- Business efficiency

- Least noise technique

- False positives (negatives)

- Cost-benefit analysis

Implementing security measures is, of course, a crucial element, however, as with so many things in live, the cost/benefit ratio has to be coherent.

Pay attention to include all costs:

- training costs,
- maintenance costs,
- testing costs
- etc.

To determine benefits is harder, use the detailed risk analysis from before to get the picture.

- Methodologies

With large or long-term projects it is often difficult to keep all the information (collected until now) consistent. In that case a more stringent method can be used to rationalize the approach. This has the following advantages:

- Assumptions and decision become explicit
- Links between assumptions and decision become clear
- Changes can be automatically reevaluated
- Investment versus risk expectation becomes visible
- Intuitions become explicit

- Evaluate the process

Far less evident yet, is the *principle of repeatable results*. The defined processes for security design decisions should yield consistent, coherent and reproducible results.

### 3.2 Some special design issues

Security design is an important part of software engineering, and the above practices cover the global picture well, or ? Well what about software you didn't design, third-party libraries, existing applications, that have to be secured ? The following three issues are good design practices for those kind of special issues.

- Retrofitting

Without access to the source code one isn't powerless, several significant security techniques exist to "retrofit" such applications:

- Wrappers **MEMO**: move orig, create script to create "safe-heaven"
- Interposition **MEMO**: the proxy way (if no access to client nor server)

- Maintenance

Existing applications, for which access to source code is granted, needing a security maintenance to be up-to-date can be another source of vulnerabilities. Considering the following few advices can help to keep the global security safe.

- Handle with the same care and scrutiny as new code
- Understand the security design spirit in place and follow it
- Learn how the program works (don't trust the manuals)
- Don't introduce new trust relationships

- Compartmentalization

This concept approaches security issues the way that it places untrustworthy users, programs, objects in a kind of virtual box so that they can't do any harm. The three most compartmentalization techniques are:

- Jails **MEMO**: java jail, chroot
- Playpens **MEMO**: constrain and mislead attackers on-the-fly ; ACE example with mayan temples (nice metaphor example)
- Honey pots **MEMO**: 2 types: low- and high-interaction

## 4 Implementation

Unfortunately (already known since Morris's Internet Worm in early 1988) the most common implementation flaw is still the *buffer overflow*. Here some good implementation practices to fight them and others (of course).

### 4.1 Principles of security implementation

- Inform yourself

This might sound implicit, but it's always good to be reminded about.

- Follow vulnerability discussions
  - Read books and papers
  - Explore open source software
- Handle data with care
 

One of the central causes of those famous *buffer overflow* flaws is most probably lousy data input handling. The solution is simple: Verify carefully every piece of data input. Ways to do this includes practices like:

    - Data cleansing **MEMO:** parse for malicious intend; prefer safe-lists to blacklists
    - Bounds checking **MEMO:** essence of buffer overflows ; check length vs allocated space
    - Checking config files **MEMO:** is also an untrustworthy data input source (think bad file permissions)
    - Checking command-line arguments **MEMO:** never tust "user" input
    - Treating web content and urls carefully **MEMO:** urls are like cli arguments ; variables can be in hidden html fields
    - Checking cookies **MEMO:** this also can be easily altered by users
    - Checking environment variables **MEMO:** OS specific config is often forgotten ; example \$PATH, \$LD\_PATHs etc.
    - Checking all other data sources **MEMO:** examples are system ressources, signals, etc.
    - Setting valid initial values **MEMO:** initializes variables with safe content
    - Handling filename references carefully **MEMO:** direct vs indirect referencing
    - Storeing sensitive data appropriately **MEMO:** e.g passwd's, credit card numbers ; use multiple layers of defense
  - Reuse code
 

It always makes sense to reuse software or pieces of code that has been thoroughly reviewed and tested, and has withstood the tests of time and users.
  - Thoroughly review
 

Some common review techniques:

- Peer review **MEMO:** 4 eyes principle ; maintain a checklist
  - Independant Validation and Verification (IV&V) **MEMO:** external audit, e.g. Common Criteria
  - Use available security tools **MEMO:** automatic ; however caution only ue as starting point
- Use checklists
 

As stated before checklists are a good way to be sure to cover everything necessary (we're just humans). Some basic entries a good checklist should contain, are:

    - Use at least passwords for user access
    - User ID's are unique
    - Access control is role-based
    - Passwords are not to be transfered in cleartext over the network
    - Data transfert is encrypted between servers and clients
    - ...
  - Create maintainable code
 

In correspondence with the *Maintenance* section from the design chapter, where "we" were to maintain existing code. Here it's "our" job to be kind to the maintainers. For that task the following practices are useful:

    - Use standards **MEMO:** for var names ; documentation
    - Remove obsolete code **MEMO:** don't mess up with "standby" code, save in a code archive or something, but remove it from the app
    - Test changes **MEMO:** as thoroughly as in the first place

## 5 Operation

Traditionally in many companies, the development staff and the operational staff are sepearate and even sometimes thorough competitors. As should have become clear till know this is a very bad approach. Development and operations are two sides of the same coin.

Security is everybody's problem !

The operational level security measures can be seen as a layered system of practices.

## 5.1 Principles of security operation

- Harden the network

Security most of the time begins on the network level, but one should not stop here.

- Allow only used services **MEMO:** is a network design issue
- Use secure protocols **MEMO:** ssh instead of telnet ; admin stuff should be more secure than the standard
- Compartmentalize the network **MEMO:** data and admin should be separated ; is also a performance issue
- Monitor unauthorized traffic **MEMO:** via IDS systems for instance
- Deploy multiple layers of defense **MEMO:** defense in depth ; don't worry about redundancy most of the time it's useful
- Log **MEMO:** think *accountability principle*

- Secure the OS

The network and OS are the foundations on which the applications are build on, consider them deeply.

- Start with a secure installation **MEMO:** basic secure config ; with a repeatable method
- Use good file access controlling **MEMO:** remember the *least privilege principle* from chapter 2
- Allow only used services **MEMO:** again ? but this time on the OS level ; php mysql example
- Remove unused stuff **MEMO:** do you really need a compiler on a webserver ? called *bastion host*
- Patch **MEMO:** be rapid, uptodate and transparent !
- Log **MEMO:** again the *accountability principle*

- Deploy carefully

Now, with the network and the OS secured, comes the application.

- Consider file access controls **MEMO:** especially the config files
- If feasible, use compartmentalization **MEMO:** jails, playpens, from above
- Switch event logging on **MEMO:** again acc...

- Apply same standards to third-party code **MEMO**: here you can go further, via retrofitting (wrappers, etc.)
- Define sound operations practices
  - Manage privileges **MEMO**: think role-based
  - Conduct operations tasks securely **MEMO**: use secure tools, remember admin stuff should be even securer than the app itself
  - Manage configurations **MEMO**: app and OS ; keep a coherent and secure whole
  - Keep patches up to date **MEMO**: speaks for itself ;)
  - Manage users and accounts **MEMO**: a central user db concept can be very useful
  - Treat temporary staff appropriately **MEMO**: don't use group accounts (no accountability)
  - Test configurations **MEMO**: best is to have a testbed environment to harrass
  - Set up checks and balances **MEMO**: responsibilities separation ; e.g. mandatory in fin sector
  - Do backups, securely ! **MEMO**: securitas taking the tape box in the lobby
  - Keep incident response plan (ready) **MEMO**: first have one, second use it well => (protect, detect, react)
- Finally ...
  - ... keep in mind that security measures are a process not a one time work. Applications decay, by itself or via environment changes, maintain them !

## 6 Testing

About automation and testing, some useful automation tools to test the finalized applications.

- [Libsafe](#) (prevents BOs during execution)
- [AppArmor](#) (Unix discretionary access control (DAC) model by providing mandatory access control (MAC))
- [Runkit\\_Sandbox](#) (php sandboxing)

- [RATS](#) (scans C, C++, Perl, Python and PHP for common security flaws)
- [PHPUnit](#)
- [Splint](#) (scans C for vulnerabilities and program mistakes)
- [Top 20 web vulnerability scanners](#)
- [Top 125 network security tools](#)

## 6.1 Recent web application vulnerabilities

**Platform:** Web Application - Cross Site Scripting

**Title:** [epesi BIM Multiple Cross-Site Scripting Vulnerabilities](#)

**Description:** [epesi BIM](#) is a PHP-based application for creating dynamic Web applications. The application is exposed to multiple cross-site scripting issues because it fails to properly sanitize user-supplied input. [epesi BIM 1.2.0 rev 8154](#) is vulnerable; prior versions may also be affected.

**Ref:** [https://www.htbridge.ch/advisory/multiple\\_vulnerabilities\\_in\\_epesi\\_bim.html](https://www.htbridge.ch/advisory/multiple_vulnerabilities_in_epesi_bim.html)

**CVE:** [CVE-2011-3990](#)

**Platform:** Web Application - Cross Site Scripting

**Title:** [PukiWiki Plus! Cross-Site Scripting](#)

**Description:** [PukiWiki Plus!](#) is an application which provides wiki functionality to websites. [PukiWiki Plus!](#) is exposed to a cross-site scripting issue because it fails to properly sanitize web form entries. [PukiWiki Plus! 1.4.7plus-u2-i18n](#) and prior versions are affected.

**Ref:** <http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3990>  
<http://www.securityfocus.com/bid/51173/info>

**CVE:** CVE-2011-3839,CVE-2011-3838,CVE-2011-3837,CVE-2011-3836,CVE-2011-3835

**Platform:** Web Application

**Title:** Wuzly Multiple Security Vulnerabilities

**Description:** Wuzly is a PHP-based blog application. Wuzly is exposed to the multiple remote security issues. See reference for further details. Wuzly version 2.0 is affected; other versions may also be affected.

**Ref:** [https://secunia.com/secunia\\_research/2011-88/](https://secunia.com/secunia_research/2011-88/)  
[https://secunia.com/secunia\\_research/2011-89/](https://secunia.com/secunia_research/2011-89/)

**Platform:** Web Application

**Title:** OBM Multiple Remote Vulnerabilities

**Description:** OBM is a messaging and collaboration application. The application is exposed to multiple remote issues. 1) A local file-include issue affects the "module" parameter of the "exportcsv\_index.php" script. 2) Multiple SQL injection issues. 3) Multiple cross-site scripting issues. 4) An insecure file permissions issue occurs because "test.php" is stored with insecure permissions. OBM 2.4.0-rc13 is vulnerable; other versions may also be affected.

**Ref:** <http://www.securityfocus.com/archive/1/520986>

**CVE:** CVE-2011-4782

**Platform:** Web Application

**Title:** PhpMyAdmin "\$host" Variable HTML Injection

**Description:** phpMyAdmin is a web-based administration interface for MySQL databases; it is implemented in PHP. The application is exposed to an HTML injection issue because it fails to properly sanitize user-supplied input to the "\$host" variable. phpMyAdmin versions 3.4.x prior to 3.4.9 are affected.

**Ref:** [http://www.phpmyadmin.net/home\\_page/security/PMASA-2011-19.php](http://www.phpmyadmin.net/home_page/security/PMASA-2011-19.php)

## 7 Bibliographic references

- [Secure Coding - Principles & Practices](#), M. Graff & K. van Wyjk
- [Code Complete - second edition](#), S. McConnell
- [Secure Programming for Linux and Unix HOWTO](#), D. Wheeler
- [Security Attribute Evaluation Method: A Cost Benefit Approach \(SAEM\)](#)
- [CERT-CC Top 10 Secure Coding Practices](#)
- [SANS Top Cyber Security Risks](#)
- [OWASP 2010 Top 10 Application Security Risks](#)
- [OWASP PHP Project](#)
- [OWASP PHP Top 5](#)
- [BAD practices: Bastard Operator From Hell \(BOFH\)](#)