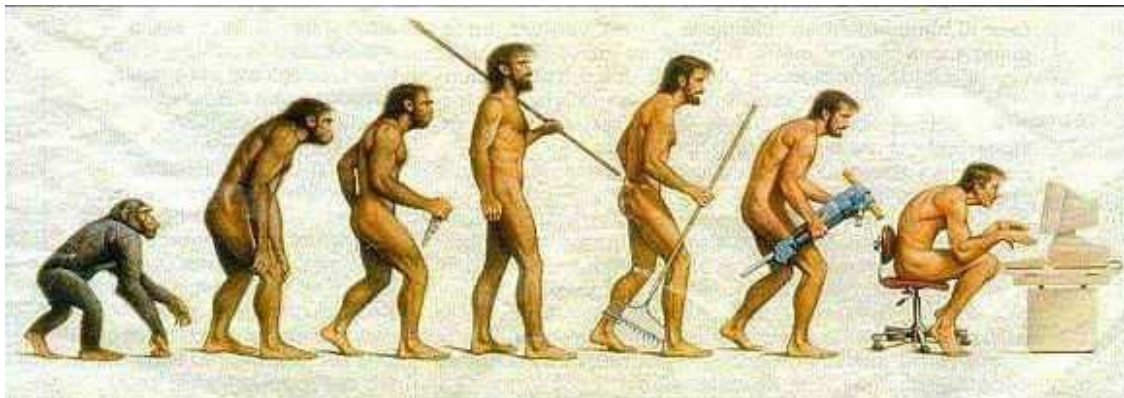




SERIOUS

DELIVERABLE

D3.4 - Catalogue of security/safety refactoring metrics

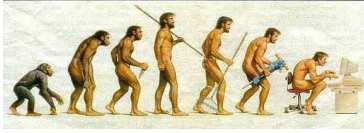


Project number: ITEA 04032
Document version no.: WP3 Deliverable 3.4 final version v1.0
Edited by: SURLOG "12-20-2006"

ITEA Roadmap domains:
Major: Services & software creation

ITEA Roadmap categories:
Major: Software engineering
Minor: System engineering

This document will be treated as public.



HISTORY

Document version #	Date	Remarks
V0.1	-	Starting version, template
V0.2	06-15-2006	Definition of ToC
V0.3	12-21-2006	Draft version, contributions by partners
V0.4	-	Updated draft
V0.5	-	Final draft to sign of by PCC members
V1.0	28-03-2008	Final Version (Approved by PCC)

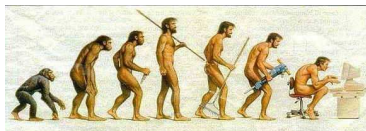
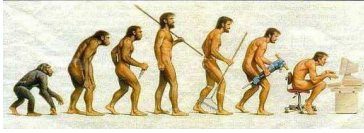
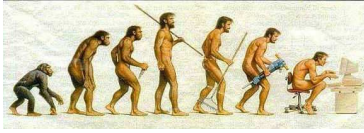


TABLE OF CONTENTS

I.	EXECUTIVE SUMMARY	5
I.1	On security, safety and dependability	5
II.	A SECURITY/SAFETY METRICS FRAMEWORK	7
II.1	Forewords	7
II.1.1	The “language-dependent” metrics and threats.....	7
II.1.2	The “programming paradigm-dependent” metrics and threats.....	7
II.1.3	The “universal” metrics and threats.....	9
II.2	Metrics, threats and rules	9
II.2.1	Any constructor subject to be called with one unique argument has to be declared as explicit.	9
II.2.2	Avoid functions with default arguments.....	10
II.2.3	Avoid functions with variable number of arguments.	10
II.2.4	Avoid recursivity.....	10
II.2.5	Any function parameter has to be used.....	11
II.2.6	Loops have to be bounded by a constant.....	11
II.2.7	No empty process in a generalized conditional's default case.....	11
II.2.8	Generalized conditional's case without end.....	12
II.2.9	Generalized conditional's without default case.....	12
II.2.10	Any variable has to be initialized prior usage	13
II.2.11	No compilation error or warning must remain.....	14
II.2.12	Dynamic memory and object allocation.....	14
II.2.13	Structured data initialized along its structure	14
II.2.14	Avoid the use of unsafe primitives.....	15
II.2.15	Size-dangerous cast	15
II.2.16	Sign-dangerous cast.....	16
II.2.17	Signed misuse	17
II.2.18	Drastic compilation options	17
II.2.19	Incomplete for construct.....	17
II.2.20	Avoid exceptions.....	18
II.2.21	Avoid explicit program termination	18
II.2.22	Avoid explicit failure constructs	18
II.2.23	Equality / inequality between floating point numbers	18
II.2.24	Avoid multiple inheritance	21
II.2.25	Avoid virtual functions	21
II.2.26	Preserve overloaded functions semantics.....	22
II.2.27	Conflicting calls.....	22
II.2.28	No assignments in function call parameters.....	22
II.2.29	No multiple exits of function	22
II.2.30	Test of function call returned value	22
II.2.31	Reduce explicit cast usages.....	23
II.2.32	No multiple enters in a block.....	24
II.2.33	No nested incrementations or decrementations	24
II.2.34	Assignment within test	25
II.2.35	Assembly inlines	25



II.2.36	Avoid deeply nested loops	25
II.2.37	Declaring for	26
II.2.38	Declaration out of block header.....	26
II.2.39	Avoid union usage	26
II.2.40	Arithmetic on pointer	28
II.2.41	Remove useless assignments.....	30
II.2.42	Remove useless comparisons	30
II.2.43	Avoid deeply nested pointers	30
II.2.44	Avoid Non-typed function return.....	30
II.2.45	Explicitly declared functions	31
II.2.46	Parameter name mismatch	31
II.2.47	No usage of operator implicit priority	32
II.2.48	Brace less statement	33
II.2.49	Empty block	34
II.2.50	Variable used as constant.....	34
II.2.51	Octal constant.....	34
II.2.52	Dead code	35
III.	REFERENCES	36



I. Executive Summary

The aim of the WP 3.4 task is to elaborate the framework basis to later (WP3.5) synthesize an automated tool extracting a security model from existing source code. This model, expressed in UML syntax, will represent the source code architecture. Moreover, it will be extended with safety / security properties and / or threats inferred from this source code.

In the present document, we will especially focus on these safety / security properties, hence building a catalogue of safety / security refactoring metrics.

It is obviously important to determine which kind of properties (metrics) are relevant in the context of safety / security and can be statically extracted from a source code. Based on this set of properties, an analyzer will be developed to automate the detection of possible threats in the software source.

This document covers three goals: to present the chosen metrics, to explain the risks they denote and to discuss the limitations of the future framework they will contribute to.

We proposed to start with a first section presenting a short analysis of the differences between the safety and security domains, and the threats they address. Then, the second section will be devoted to the proposed metrics presentation.

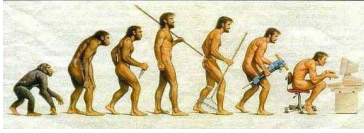
I.1 On security, safety and dependability

Although “*safety*” and “*security*” are commonly confused, they do not exactly denote the same property. “*Safety*” is the property ensuring that a system will not injure the human or material environment integrity. “*Security*” is more oriented toward the preservation of sensible information, defending against malicious activities.

In the early stages of computers usage, the main identified issue related to a running system quickly appeared to be the safety (one major notion of dependability -- [3]). Why safety instead of security? The probable reason is that such systems were used in industrial critical processes that could not suffer any dysfunction. Obviously, aside these processes, military applications also made use of computers. And security is a well-known property aimed by the military domain! But, because computers were not widespread (no Internet, no personal computers), risks of attacks were pretty reduced compared to what can be today.

Nowadays, computers and more generally programmable systems invade the finest space of the today life: your watch, your microwave oven, your cell phone, your car, the train you try to catch every morning, your debit card, your banking account management, your medical information, the nuclear plant that provides you electricity ... to run your computer! From this evolution, the notion of “*security*” (protection of data and secrets) became a real concern and specific techniques appeared ([2]).

The major issue while talking about “*security*” is that this term has been used in several contexts, with a few semantics differences. The first meaning nowadays coming in mind is the protection against intrusions. However, dependability previously mentioned “*security*”, in a less restrictive way. Let’s compare a quick summary of dependability and “*security*” main objectives. Dependability is mostly summarized by the RAMS acronym:



- **Reliability:** ability of a system to perform its mission in some defined usage conditions.
- **Availability:** ability of a system to be operational when its service is requested.
- **Maintainability:** ability of a system to be maintained and restarted.
- **Safety:** ability of a system to preserve its users and its environment.

On the other side, security “*intrusion-oriented*”, that should be more accurately defined as “*security of information*” tends more to preserve, restore and ensure the security of data and their hosting system. The following three axes may summarize the major principles:

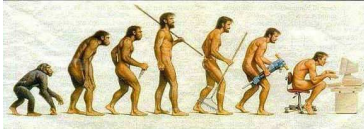
- **Integrity:** ability of a system to prevent (or at least detect) information destruction or modification.
- **Confidentiality:** ability of a system not to reveal information to non-authorized parties.
- **Availability:** as in dependability, ability of a system to be maintained and restarted.

However, the frontier between dependability and security remains extremely thin, and most of the studies and concepts elaborated from the dependability experience also apply in the domain of security. Defending against involuntary errors or problems remains the first step to efficiently go on malicious (voluntary) attacks.

The difference between the 3 widespread terms “*vulnerability*”, “*threat*” and “*risk*” is interestingly describe in [1]. “A *vulnerability* is as a weakness possibly leading to undesirable consequences [...]. A “*threat*” is the danger that a vulnerability can actually lead to undesirable consequences [...]. A “*risk*” is a potential problem with causes and effects [...] it is the harm that can result if a threat is actualized [...]” It shows the relationship and scale between involuntary unwished systems behaviors and the human-triggered ones.

One major problem with security requirements is that they are often very high level (e.g. “a key should be at least 128-bits wide”). For this reason, it is difficult to automate their detection in a software source code. Most security attacks are the result of a human reasoning, i.e. a complex research also based on experience and that can poorly reproduces by a tool. Indeed, the state of the art in security shows numerous schemes of attacks and defenses, but mostly based on combinations of system calls, operating system features and external complete software products. Schemes based on programming constructs or paradigms, which are features inherent to the inner architecture of a system, are really more rare. However, starting with a sane architecture is an important pillar to maximize chances to build stable systems.

Hence, we now understand that security and safety are closely linked and that security can poorly be reached without maintaining an important degree of safety and more generally of dependability.



II. A security/safety metrics framework

Since we wish to elaborate a catalogue of metrics that can be extracted from a source code, the quality approach addressed here obviously tends to be reactive approach. More details about the different quality approaches can be found in the chapter 3.2 “Partners’ perspectives and expectations” of the previous D3.1 “Shared Vision on Quality Models” document.

II.1 Forewords

According to the conclusion of the section I.1, we decide to make a global presentation of the metrics of this catalogue, merging the security and the safety concepts. If however a metric is less important for one or the other concept, we will mention this point in the metric’s discussion.

The following metrics and threats may have been elaborated for different reasons. One important idea under these metrics is to reject implicit features: security and safety require things to be explicitly said, hence thought! One can roughly classify them in three categories.

II.1.1 The “language-dependent” metrics and threats

They may be induced by the syntax of the programming language. Hence, they are only relevant for this language. For example, the well-known equality operator == of C, C++ and Java, is also well confused with the assignment operator =. A common error is to write code such:

```
p = MemAlloc (...) ;  
if (p = NULL) NotifyNoMem () ;  
else ...
```

instead of:

```
p = MemAlloc (...) ;  
if (p == NULL) NotifyNoMem () ;  
else ...
```

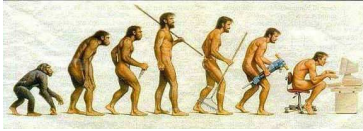
The major difference is that the second code tests if the memory allocation is successful and if not, reacts properly whereas the first one explicitly assign the NULL value to the pointer p, hence destroying the address returned by the allocation primitive.

This confusion is only due to the similarity of the operator in the language. Hence, such a metrics will not apply to other programming languages.

II.1.2 The “programming paradigm-dependent” metrics and threats

Depending on the kind of the used programming language, different design paradigms may form a program. Several languages may provide the same paradigm. Several paradigms may be cumulated in a same language. Hence metrics will be relevant on any language having this paradigm, even if it is not expressed with the same syntax, keywords or layout. One can mainly identify six kinds of programming.

- *Imperative programming.* In this programming model, the memory is viewed as a state that can be modified in place by side effects. Modifications are performed in sequence by instructions. This means that the order in which the expressions are



evaluated is important. Changing this order may result in getting a different result. Most of the programming languages provide this mechanism (C, C++, Pascal, ADA, Objective Caml, Perl...). For example, in C,

```
i = 0 ;  
printf ("%d %d\n", i++, i++) ;
```

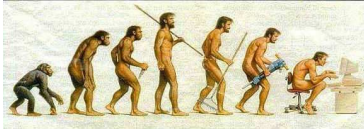
may lead to two different printings: "0 1" or "1 0" depending whether the function arguments are evaluated from left to right or conversely.

- *Functional programming.* Conversely the above model, pure functional programming does not allow side effects. The memory is no more an implicit "big array" that can be modified. It is only to bind values to identifiers but not to modify them later. Hence, the order of evaluation of expressions becomes non-relevant for the final results. ML is the corner stone of these languages. Languages providing this mechanism do often also provide imperative features (Scheme, Lisp, Objective Caml, SML...). For example, programs written with the functional style may look like the following well-known factorial function:

```
let rec fact n =  
  if n = 0 then 1 else n * fact (n - 1) ;;
```

We clearly see that the order in which n and $\text{fact}(n - 1)$ are evaluated before the multiplication is not relevant for the final result.

- *Modular programming.* Modules are a mean for abstraction and encapsulation. Note that other mechanisms also contribute to this aim (as classes we will describe below). A module is a collection of types, data and procedures that manipulate these types and data. Modules are composed of an implementation and an interface. The interface only makes visible a subset of the module internal whereas the implementation is the source code of the actual processes, types and data. Conversely to classes, modules do not provide inheritance. However, in elaborated module systems, it is possible to build "*functors*", i.e. functions taking as arguments modules and returning a fresh module. A module may have a body that is a part of code that gets executed as soon as the module is loaded or instantiated. For example, in such a configuration, this implies that the order in which modules are linked together may impact the order their bodies are executed. Modules are available in many languages like ADA, Modula, Objective Caml... Note that C and C++ provide a weaker module system thanks to the ".h" files, the static functions and extern data qualifier.
- *Logical programming.* Prolog is certainly the best example for these languages. In such a model, instead of writing instructions to solve a problem (i.e. instead of writing an explicit algorithm), the problem to solve is encoded under the form of a set of predicates (or "constraints"). The language runtime (constraints solver) will then run over this set in order to extract the result. This kind of programming radically changes compared to the usual procedural paradigm.
- *Object programming.* Object frameworks are mostly based on the notions of classes and object (instances of a class). Some more complex frameworks also allow meta-classes (from which classes are themselves instances). Like the modules previously presented, classes contribute to abstraction and encapsulation. In addition to modules, classes allow inheritance, i.e. the property for a child to have the same properties than its parent(s) unless it redefines them.



Children may or may not be allowed to call their parents procedures (called methods) depending on the kind of inheritance (public or private). Among object languages, let's mention Smalltalk, C++, Java, Objective Caml...

- *Lazy programming.* Most of the programming languages perform function call “per value” by evaluating the value of the arguments before getting in the body of the called function. This means that even if the value of an argument is not finally needed to compute the result of the call, this value will be computed by evaluating the argument expression. In lazy languages, these arguments are evaluated later, only if and when the body of the called function really needs their values. As a consequence, the runtime behavior can be drastically different from a conventional language. For instance, let examine the following program.

```
void print (int x) {  
    printf (“Foo\n”) ;  
}  
  
int non_ending_loop () {  
while (1) ; /* Do nothing. */  
return (0) ;  
}  
  
void strange () {  
    print (non_ending_loop ()) ;  
}
```

In a non-lazy language, because the argument of the call to `print` in the function `strange` is always evaluated before the call, such a program would loop forever. In a lazy framework, because the parameter `x` of the function `print` is never used, it would not be evaluated, leading the program to terminate normally. Let's cite Haskell as an example of such languages.

II.1.3 The “universal” metrics and threats

Aside the previous kind of risks and threats, some others are possible whatever the used language. For example, pointers (i.e. variables whose type is the address of one data) are available in roughly any programming languages. Numerous caveats are due to pointers (NULL pointer assignment, pointers designing illegal address, wrong type of pointed data...) Recursivity may lead to stack overflow or non-termination. Such risks are mostly due to the mechanism of programming rather than to a particular construct or a particular language.

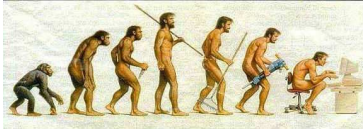
II.2 Metrics, threats and rules

II.2.1 Any constructor subject to be called with one unique argument has to be declared as explicit.

Language: C++

Paradigm: Object

Description: A constructor with one unique argument defines by default an implicit conversion. A constructor declared as `explicit` will only be called if explicitly requested. For example,



```
String s = 'a' ;
```

defines `s` as a string with `int('a')` elements. This is pretty confusing, and can easily be confused with the creation of a string with the only character “a”. By declaring the constructor explicit, the code `String s = 'a'` will be rejected whereas `String s = String('a')` or `String s = String(10)` will be accepted. Hence, being forced to explicitly call a constructor when required impose a voluntary action of the developer, forcing him to wonder if it is really what he wants to do.

II.2.2 Avoid functions with default arguments

Language: Any

Paradigm: Any

Description: Various languages (C++, Objective Caml...) allow specifying default values for function arguments (ex: `void f(int x, char y = 5)`). If they are not provided during a call, then these values are used instead of reporting an arity mismatch. Although this feature seems handy to avoid writing several times a value that gets “nearly always used”, this has two major drawbacks. First, it loses the ability to pinpoint arity mismatches (wrong number of arguments passed to a function, especially forgotten arguments). Secondly, it prevents the developer from explicitly wondering which value has to be actually provided to the call and whether the implicit default one is actually the right one. This rule follows the principle stating that it is imperative that any treatment must be explicit in order to clearly reveal the possible omissions.

II.2.3 Avoid functions with variable number of arguments.

Language: C, C++

Paradigm: Any

Description: Although functions usually have a finite signature (i.e. arity, or number of arguments), C and C++ allow defining functions with a non-specified number of arguments (the ... type, also called ellipsis). However, the number of actual arguments remains statically fixed for each call written to the function (no dynamic changing arity). Internally, this can be seen as passing a NULL-ended array as argument, or an array and the number of actual arguments. Well-known functions of this kind are `printf`, `scanf` and their derivations. The difficulty is then to ensure that the number of provided arguments is correct and to set a robustness process in such functions. Moreover, ellipsis in the scope of the `printf`, `scanf`, the usage of `format` and ellipsis has been reported as the major threat in security [1][5].

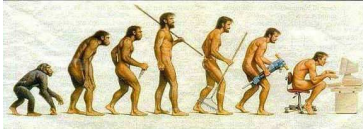
II.2.4 Avoid recursivity

Language: Any

Paradigm: Any

Description: Recursive calls present two major risks. As any “*looping*” construct, they may lead in non-termination (kind of deny of service). One could then argue that in this case loops (for, while...) should also be avoid.

The point is that recursivity is first more complex to figure out and that it also implies nested calls that may result in a stack overflow. In effect, even if the recursivity is structurally bounded, the amount of stack to push each context call may be superior to the stack space available on the running target. In embedded architectures, memory constraints are often severe, and the system stack is often reduced. If the number of recursive calls depends on the runtime context, the stack may become non sufficient. In this case and if recursivity has to be maintained, to prevent any runtime



crash, a static bound test on the number of calls has to be set up in the recursive function(s).

II.2.5 Any function parameter has to be used.

Language: Any

Paradigm: Any

Description: For each function parameter, at least one execution path must use it. If not, the question is why is this parameter present in the function. This may denote a missing process in the function.

```
float LogBase (float x, float n) {  
    float res ;  
    res = log (x) ;  
res /= log (n) ; // Erased, for example by an unwished "CTRL-X  
(cut)"  
    return (res) ;  
}
```

On the above example, due to a sniped line of code, one line previously using the second argument, the function LogBase will not anymore compute the logarithm in base n, but the natural logarithm (in base e, i.e 2.71828...).

Although it may seem odd having lines of code "magically" disappearing, this really occurs in the real life... Just consider changing your favorite text editor: under *GNUEmacs*, *CTR-X-CTRL-S* saves the buffer, under Windows tools, *CTRL-X* cuts the selection, then *CTRL-S* saves the buffer. As a result, applying *GNUEmacs* bindings under Windows will destroy the selected line before saving, and if this line was the `res /= log (n)`, here is how it may have disappeared!

Another way is to have a selected text area; depending of the text editor, typing will delete the selected part and replace it by the new text; or it won't care about the highlighted area and continue entering text at the current cursor position (*GNUEmacs*, for example).

Finally, the missing process may simply be a forgotten part of code that was intended to be written but that was not for any reason...

II.2.6 Loops have to be bounded by a constant

Language: Any

Paradigm: Imperative

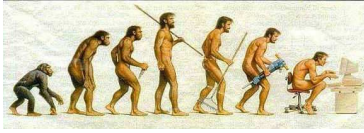
Description: As we previously mentioned about recursivity, "looping" constructs may lead in non-termination (kind of deny of service). The same kind of robustness process has to be set up around loops. This implies either to have a constant number of iterations (in this case, the loop statically ends), or to add static bound test on the number of iterations.

II.2.7 No empty process in a generalized conditional's default case

Language: Any

Paradigm: Any

Description: Generalized conditionals (or multi-branches conditionals) allow testing a value against several constants, choosing the process related to the constant that is equal to the matched value. For example, C proposes the `switch` construct as:



```
int v ;  
...  
switch (v) {  
case (-2): ... break ;  
case (0) : ... break ;  
...  
case (42): ... break ;  
default: ... break ;  
}
```

If the value v is equal to -2 , then the first case will be executed. If it is equal to 0 , then the second case will be and so on. If v 's value doesn't match any of the enumerated cases of constant, then the default case will be selected. If none, then nothing will be done. Two situations are possible. Either the default case should never happen; in this case a robustness process has to be installed. Or the default case really implies nothing to be done; in this case at least a comment has to be written in order to justify the empty process.

II.2.8 Generalized conditional's case without end

Language: Any

Paradigm: Imperative

Description: Any case of a generalized conditional, (switch instruction in C, C++), each case must be ended by an end marker (break instruction in C, C++). Absence of this marker will make the program executing the process dedicated to the next case. When reading such a construct, the obvious arising question is to know if this sequence is really wished or if the end marker has been forgotten. At least, to disambiguate between an implicit continuation and an error, if cases really must be cumulative, an extra documentation (comment) has to be written.

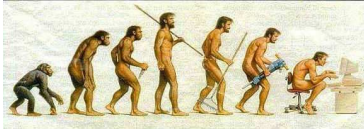
II.2.9 Generalized conditional's without default case

Language: Any

Paradigm: Any

Description: Generalized conditionals aim to discriminate a value among several possible constants. Whenever the tested variable matches one of the enumerated constant, the related block of process gets executed. If no match appears, then either a default case exists and then the related process is executed, or no default case is found and no process is executed.

```
void ProcessCubeDiceFace (char face) {  
switch (face) {  
case 1: ... break ;  
case 2: ... break ;  
case 3: ... break ;  
case 4: ... break ;  
case 5: ... break ;  
case 6: ... break ;  
}  
score + = face ;  
...  
}
```



}

Hence, without any default process, robustness cannot be ensured because no process will notify a value out of the functional range. Moreover, such a construct may injure integrity, preventing from discovering that data has been corrupted. On the above example, the integer value range from 0 to 255 but only 6 values are considered. Obviously the programmer assumes a dice value may only range between 1 and 6. But any other integer value may flow until the test. In this case, the inconsistent value will be silently processed, propagating the result inconsistent behavior in the next execution steps of the software without any error notification.

II.2.10 Any variable has to be initialized prior usage

Language: Any

Paradigm: Any

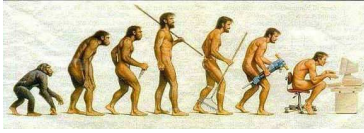
Description: Non-initialized variables represent a non-ending list of possible threats, most of them pretty hardly reproducible in a deterministic way. The resulting bugs get hence difficult to detect, locate and correct.

1) First of all, a non-initialized variable is a lack of defensive programming that is a strong basis of safety and security. Defensive programming is a model in which instead of assuming that a process will correctly run and possibly detect faults, the opposite philosophy is taken. One considers that “everything is wrong” and then one elaborates the proof that the process correctly ran. In other words, instead of destroying the proof that all went right, one builds the proof that nothing went wrong. Let's understand the importance of the difference between these concepts on the following example.

<i>Non defensive</i>	<i>Defensive</i>
<pre>bool status_ok ; void Ensure () { if (Check₁ ()) { status_ok = true ; if (Check₂ ()) status_ok = true ; ; else status_ok = false ; } else status_ok = false ; }</pre>	<pre>bool status_ok = false ; void Ensure () { bool b₁ = false, b₂ = false ; b₁ = Check₁ () ; b₂ = Check₂ () ; status_ok = b₁ && b₂ ; }</pre>

If the function Ensure is not executed completely, with the non-defensive program, if something should report an error in the Check tests, then the abnormal event remains hidden and no emergency measure will be taken. Conversely, with the defensive program, the pessimistic option being assumed by default, if a feared event got hidden, then the emergency process will anyway get launched.

2) Forgetting to initialize a variable is a target for external malicious data injection. In effect, for example, if the memory area is write-enabled before being used, it is then possible to artificially assign it a value chosen by the intruder in order to modify the behavior of the hosting component.



3) Non-initialized variables also constitute covert channels, an important threat in the security domain. In effect, values remaining in the variables from prior usage may leak sensible information. For example, let's imagine one gets the memory for an array at the same address where was stored a PIN or a password. If the array is not initialized, then old data remain reachable, revealing the secret information that should have been confined.

4) Finally, a variable non-initialized on an execution path may contain a spurious random binary configuration leading to a value out of the functional range, hence leading to non-deterministic failures.

By extension of the initialization issue, in object-oriented languages, any class attributes have to be initialized in the constructors of this class. In effect, such attributes are in fact variables bound to each instance (or common to all of them, depending on their qualification) of the class.

A special attention is held for global variables: they need to be initialized while defined. Because of possible parallel processes and sharing, it gets statically impossible to ensure that such a variable will be actually initialized along all the execution paths of all parallel traces if the variable does not get a value soon as it is created.

II.2.11 No compilation error or warning must remain

Language: Any

Paradigm: Any

Description: Compilers are now smart enough to provide relevant information about possible problems. An important pre-requisite for safe software is at least to listen to one's compiler. Any error, even warning must be eliminated by the proper program modification. Obviously, it will not be safe to suppress a warning by relaxing the compilation options.

II.2.12 Dynamic memory and object allocation

Language: Any / C++ for objects

Paradigm: Any / C++ for objects

Description: Dynamic memory allocation presents an important risk of runtime failures. An allocation may fail if no more memory is available in the system. For this reason, any allocation should be tested to ensure the resource was available. Moreover, if no memory was allocated, the program must plan a robustness process that will ensure the system will either continue to run or properly go in fallback position without crashing. Forgetting to check the allocated memory will first hide the lack of resources and trigger usage (reads/writes) of illegal memory areas. Memory allocations must be paired with the corresponding memory releases to prevent exhausting the system's resources and a shortcoming deny of service.

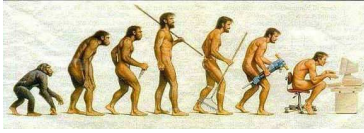
In addition to the "object-oriented" behavior of objects, these ones also fall in the scope of the current threat because they may be dynamically allocated too.

II.2.13 Structured data initialized along its structure

Language: Any

Paradigm: Any

Description: In addition to scalar (basic) values, programming languages allow structured datatypes (or aggregates) hence grouping scalar and/or recursively aggregates inside one new datatype. C hence provides union and struct. C++



provides the `class` (`struct` and `union` are in fact particular classes with weaker constraints). Like basic-typed variables have to be initialized, structured ones must also be.

Moreover, to explicit the fact that the initialization matches the datatype, each field of the aggregate must be initialized with a value of the convenient type, and at the convenient nesting level (using the structural nesting token of the language). Such a constraint limits (unfortunately not avoids) initializations of structured variables of one type with data making-up a value of another structured type. Let's enumerate a few examples:

<code>int t[3] = { 0, 0 };</code>	Wrong init: not enough values (3 required).
<code>int t[3] = { 0, 0, 0, 0 };</code>	Wrong init: too many values (3 required).
<code>int t[3] = { 0, 0, 0 };</code>	Right init.
<code>struct s { int a ; int b };</code>	
<code>struct s a[2] = { 0, 0, 0, 0 };</code>	Wrong init: structural nesting not respected.
<code>struct s a[2] = { { 0, 0}, { 0, 0 } };</code>	Right init: structural nesting respected and right number of values.

II.2.14 Avoid the use of unsafe primitives

Language: Any

Paradigm: Any

Description: Some primitive of the standard library are referenced as unsafe for various reasons. The state of the art in safety and security [6] established lists of them, sometimes advising the use of alternative functions. As long as alternatives are available, they must be preferred.

II.2.15 Size-dangerous cast

Language: Any

Paradigm: Any

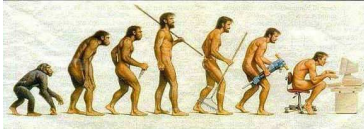
Description: Casting (implicitly or explicitly) a data in a smaller sized one must not be used otherwise, most significant bits will be lost, leading to a value modulo ^{2^{size in bits}} of the receiving data.

```

short int i ;
char c ;
i = 254 ;
c = (char) i ;
i = 256 ;
c = (char) i ;

```

The above example shows a 1-bit value cast into a 8-bits value. The first assignment tries fitting 254 in the byte. According to the following scheme, this will not cause any issue.



i

0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

c

x	x	x	x	x	x	x	x	x
---	---	---	---	---	---	---	---	---

c

1	1	1	1	1	1	1	1	0
---	---	---	---	---	---	---	---	---

In the second assignment, the value 256 holds on 9 significant bits. Because a char is only 8 bits wide, bits 8 to 15 (numbering from 0) will be discarded, resulting in the variable c getting the value ... 0.

i

0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

c

x	x	x	x	x	x	x	x	x
---	---	---	---	---	---	---	---	---

c

0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---

II.2.16 Sign-dangerous cast

Language: Any

Paradigm: Any

Description: For signed integers, the most significant bit (*MSB*) represents the sign whereas for unsigned number this bit has no special meaning. In signed a configuration, if the *MSB* is 1 then the number is negative, if it is 0 then the number is positive.

That's not the only needed difference to represent a signed number otherwise one would write 2 as 00000010 and -2 as 10000010. This would be a first issue because 0 could be represented twice: 00000000 and 10000000 would be respectively equal to 0 and -0.

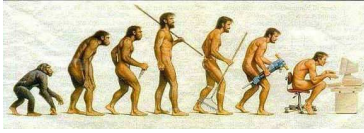
Moreover, arithmetic operations between positive and negative numbers would lead to wrong results: 00000011 + 10000100 = 10000111, that is 3 + (-4) = (-7) instead of (-1).

Hence, the 2-complement representation is used to solve these problems. Positive numbers are regularly encoded as powers of 2. Negative numbers are created from their positive opposite by logically inverting the bits (1-complement) then add 1 to the result. This way, the *MSB* is automatically set to 1 by the process. For example, -4 is encoded as follows:

- Opposite of -4 = 4
- 4 in binary is 00001000
- Logically invert bits: 11110111
- Add 1 : 11110111 + 1 = 11111000

To recover the positive opposite of a negative value, the same process applies. For instance, let's get the opposite of -5:

- -5 is represented as: 1111 1011
- Logically invert bits: 00000100
- Add 1 : 00000100 + 1 = 00000101



From these explanations, casting a signed integer into an unsigned integer must not happen. In effect, as long as the converted value is positive, no error occurs. When the value is negative, the semantics of the sign bit gets lost. The unsigned value obtained is then interpreted as a regular binary configuration (not anymore as 2-complemented) leading to $2^{\text{size of data in bits}} - |\text{original signed value}|$. Hence, a 16 bits wide signed value of -1 will appear as a 65535 16 bits wide unsigned value as shown in the following example. A 8 bits signed value of -5 will appear as a the 8 bits unsigned value 251.

On the other side, casting an unsigned integer into a signed one is valid only if the size of the signed data is strictly greater to the size of the unsigned one. In this case, there will be room for the sign bit that will put at 0 to get the signed positive value equal to the original unsigned (hence positive) value. If the size of the receiving data is equal to the size of the converted data, the semantics on the most significant bit gets toggled as a sign bit. Then if the original value is greater or equal to $2^{\text{sizeof data in bits}} / 2$, then the obtained signed value gets negative, with a value of original unsigned value $- (2^{\text{sizeof data in bits}} - \text{original value})$. Hence a 16 bits wide unsigned value of 32800 will then appear as a bits wide signed value of -32736 . Such behaviors, reported to be used in order to access out (below) of arrays bounds, represent real threats in the security domain.

II.2.17 Signed misuse

Language: Any

Paradigm: Any

Description: Usage of signed arrays indices is strongly inadvisable since it allows to access array out of their bounds (using a negative displacement). For example, a common attack consists in using a negative index through local data in order to reach the data in various previous stack frames, or the address where to return from a call.

Moreover, depending of the location of the accessed data (global or local), allocation will be either on the stack or in the head, depending of the compilation policy, negative indices may cause overflow or underflow accesses in the array.

II.2.18 Drastic compilation options

Language: Any

Paradigm: Any

Description: Options used to control the compiler's warnings and error must be set to the maximal level of severity. Combined with the previous rule requiring that no error or warning must remain after the compilation, this reduce the cases of forgotten problems that could have been detected at compile-time. It is even advised to configure the compiler to handle warnings like errors, to ensure that even warnings will be properly treated.

II.2.19 Incomplete for construct

Language: C, C++

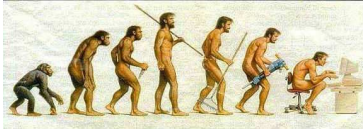
Paradigm: Imperative

Description: The for construct of C and C++ is shaped as follows:

```
for (expr1; expr2; expr3) stmt ;
```

It is basically compiled as a regular while loop according to the following pattern:

```
expr1 ;  
while (expr2) {
```



```
stmt ;  
expr3 ;  
}
```

Hence, it intuitively groups a prelude ($expr_1$), a condition ($expr_2$), a postlude ($expr_3$) and a body ($stmt$) to be executed as long as the condition holds. The syntax allows omitting any of the 3 expressions. In such a case, this means that the power of expression of a `for` is not needed and a simpler `while` should be used. Applying this policy prevents from mixing “correctly” incomplete `for` and loops where a treatment has in fact been forgotten.

II.2.20 Avoid exceptions

Language: C++

Paradigm: Any

Description: Exceptions are a mechanism allowing abruptly aborting a computation. They are commonly used to report error, the impossibility to go further in a process, or non-value results (for instance division by 0). Two actions are implied by exception manipulation:

- exception raising : one asks for the computation to be interrupted,
- exception handling : one catch exceptions that have been raised and execute a process to react.

Handling an exception is done installing a “*handler*” surrounding the computation that may raise this exception. When an exception is raised, the current computation is stopped and the exception walks back the dynamic calls tree until it find a handler design to catch it.

Although exceptions present an elegant and powerful mean to federate errors handling, they constitute a severe runtime danger since if no handler is found, then the exception goes back up to the toplevel and ends the whole program execution. Moreover, installing handlers is a tricky process since they must be set as close as possible from the raising location to prevent exceptions from propagating too far and hence abort wider areas of computations.

II.2.21 Avoid explicit program termination

Language: Any

Paradigm: Any

Description: An explicit termination implies for sure the end of the execution of the software. For numerous systems (monitoring, control-command, security...) this behavior is not wished for availability concerns, unless external protection systems are set up (watchdogs) that will overcome the end of the system’s execution.

II.2.22 Avoid explicit failure constructs

Language: Any

Paradigm: Any

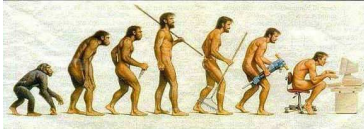
Description: In the same philosophy than the explicit program termination, using constructs that will explicitly lead to failure injuries the availability of a system and must then be avoid.

II.2.23 Equality / inequality between floating point numbers

Language: Any

Paradigm: Any

Description: Floating-point numbers use an encoding format (mantissa, exponent) allowing the representation of a wider range of numbers than the physical number of



bits would do in the regular binary format (powers of 2). As a drawback, rounding and precision issues arise during computations. For this reason,

- Operations on floats with different exponents may result in wrong results according to the normal mathematical expected properties.
- Due to rounding issues during computations, two values that should be mathematically equal may practically differ from an infinitively small factor. Then absolute inequality/ equality must be replaced by testing the belonging or not to a small interval surrounding the tested value. For example, instead of testing if x is equal to 5.0, one tests whether $5.0 - \varepsilon < x < 5.0 + \varepsilon$ for a ε chosen small enough according to the required precision.

To get an effective view of these problems, let's consult the following piece of code. Here float stands for standard IEEE-754 [7][8] floating point numbers in simple precision.

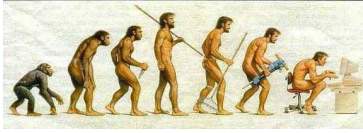
```
int main () {
    float f11 = 48431.1231 ;
    float f12 = 48431.1239 ;
    float f13 = 48431.1250 ;

    printf ("f11: %f\nf12: %f\nf13: %f\n", f11, f12, f13) ;
    printf ("f11 == f12 ? %d\n", (f11 == f12)) ;
    printf ("f12 == f13 ? %d\n", (f12 == f13)) ;
    printf ("f11 == f13 ? %d\n", (f11 == f13)) ;
    f11 = f11 + 0.0001 ;
    printf ("f11 + 0.0001: %f\n", f11) ;
    return (0) ;
}
```

Against the mathematical expected result, one gets the following output:

```
f11: 48431.125000
f12: 48431.125000
f13: 48431.125000
f11 == f12 ? 1
f12 == f13 ? 1
f11 == f13 ? 1
f11 + 0.0001:
48431.125000
```

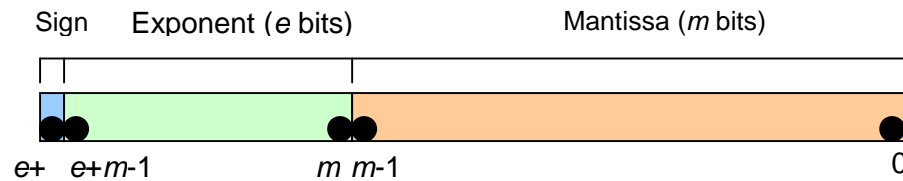
What happened is that the encoding format of the floating number didn't have enough precision to differentiate the 3 values we provided. This may seem odd according to the fact that we can represent accurately tons of other larger and smaller values! Hence, all these values are considered equal. Moreover, this behavior strongly



depends on the values: if we respectively change f11 and f12 values by 15000.1231 and 15000.1239, then we would get that f11, f12 and f13 are all different.

```
f11: 15000.123047
f12: 15000.124023
f13: 15000.125000
f11 == f12 ? 0
f12 == f13 ? 0
f11 == f13 ? 0
f11 + 0.0001:
15000.123047
```

As a short explanation, floating points numbers are encoded using 3 different areas inside the memory word: a sign, an exponent and a mantissa. Depending on the float precision, exponent and mantissa may vary in size, while the sign always remains on 1 bit. For instance, in simple precision, exponent is 8 bits and mantissa is 23 bits.

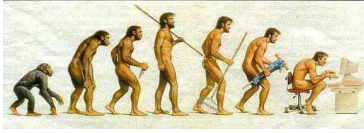


The exponent is biased by $2^{e-1} - 1$. Biasing is done because exponents have to be signed values in order to be able to represent both tiny and huge values, but two's complement, the usual representation for signed values, would make comparison harder. To solve this, the exponent is biased before being stored, by adjusting its value to put it within an unsigned range suitable for comparison. A slight variation exists on the bias (also known as normalized and denormalized numbers) but this is mostly out of the scope of this simple explanation and will only consider normalized floats, i.e. with bias. The mantissa represents the fractional part of the number in term of powers of 2. Hence, for such a float, its value is $v = s \times 2^{\text{exp}} \times m$ where:

- $s = +1$ (positive numbers) when the sign bit is 0
- $s = -1$ (negative numbers) when the sign bit is 1
- $\text{exp} = \text{"Biased exponent"} - 127$ (in effect, the exponent is stored with 127 added to it, also called "biased with 127" because we assume a 8 bits exponent in our example)
- $m = \text{"1.Fraction in binary"}$ (that is, the binary number 1 followed by the radix point followed by the binary bits of the fractional part (written in binary) of the original number). Hence, $1 \leq m < 2$.

Let us encode the decimal number -118.625 using the IEEE 754 system.

1. First we need to get the sign, the exponent and the fraction. Because it is a negative number, the sign is 1.
2. Now, we write the number (without the sign) using binary notation. The result is 1110110.101.
3. Next, let's move the radix point left, leaving only a 1 at its left: $1110110.101 \rightarrow 1.110110101 \times 2^6$. This is a normalized floating-point number. The mantissa is



the part at the right of the radix point, filled with 0 on the right until we get all 23 bits. That is 1101101010000000000000.

4. The exponent is 6, but we need to convert it to binary and bias it (so the most negative exponent is 0, and all exponents are non-negative binary numbers). For the 32-bit IEEE 754 format, the bias is 127 and so $6 + 127 = 133$. In binary, this is written as 1000101.

Then as a result we get the following binary configuration:

1 1 0 0 0 0 1 0 1 1 1 0 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Obviously, it can better be understood that if the fractional part cannot fit on the 23 bits according to the required exponent, then several values will be physically equal although they should not. Because the fractional part depends on the exponent, one understand why the rounding behavior is different for numbers however belonging to a close range of value (in our previous example, 15000.123047 and 48431.125000 are mathematically in the same order of size conversely to 10^{-10} and 10^5).

II.2.24 Avoid multiple inheritance

Language: C++

Paradigm: Object

Description: To avoid method selection ambiguity if several parents provide a same method and to ease readability and maintainability, any class must inherits from at most one parent per inheritance level (i.e. must have at most one direct parent).

Moreover, inheritance induces coupling between two classes, and coupling is not appreciated when designing robust and modular systems.

II.2.25 Avoid virtual functions

Language: C++

Paradigm: Object

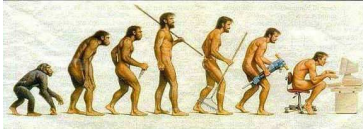
Description: A virtual function or virtual method is a function whose behavior, is determined by the definition of a function with the same signature deeper in the inheritance hierarchy of the instantiated object on which it is called.

When a derived class inherits from a base class, an object of the derived class may be referred to (or cast) as either being the base class type or the derived class type. If there are base class functions overridden by the derived class, a problem then arises when a derived object has been cast as the base class type. When a derived object is referred to as being of the base's type, the desired function call behavior is ambiguous.

The distinction between virtual and not virtual is provided to solve this issue. If the function is designated "virtual" then the derived class's function will be called (if it exists). If it is not "virtual", the base class's function will be called.

Virtual methods are pretty difficult to properly design. They require two, independent, sets of semantics: one for callers and another for implementers. Designing a class to be robust and secure against malicious or naive overriding is not a trivial process.

In addition to the already seen inheritance aspect, virtual methods make coupling stronger between classes, because one class replaces methods in the other. It's harder to change the implementation of a class that has lots of virtual methods,



because descendants are more closely coupled to the ancestor. This obviously increases the modularity, robustness and maintainability issues.

Finally because virtual methods allow dynamic function resolution (i.e. the effective called function will be determined at runtime), it prevents compilers and static analyzers from helping in the development process.

II.2.26 Preserve overloaded functions semantics

Language: C++

Paradigm: Object

Description: All instances of an overloaded function must own the same functional semantics. Although this requirement can obviously not be statically checked, it is important for maintainability. This avoids having functions with a same name but performing completely different processes. Such a design pattern would obviously lead to a confuse software architecture and increase the probability to add bugs by misunderstanding in further evolutions of the system. Just imagine the operator minus overloaded for (`float`, `float`), performing the addition of its two arguments!

II.2.27 Conflicting calls

Language: Any

Paradigm: Any

Description: Usage, in a same expression, of several functions with one of them modifying inputs of the others is rejected because of portability issues (order of evaluation of the expressions). For instance, `a = f(x) + g(x)` with `x` (or any global variable shared between `f` and `g`) modified by `f` will lead to unpredictable results, depending on the compiler, the optimizations, the context around the expression and so on.

II.2.28 No assignments in function call parameters

Language: Any

Paradigm: Imperative

Description: In the same spirit than for the conflicting call metric, performing assignments inside expressions provided as argument for a function call may result in unpredictable results due to the dependence on the order of evaluation. For instance, `f(++x, g(x))` makes not clear whether `x` will be incremented before being provided to `g`. Such an effect can be obtained in a simpler way: `printf("%d %d\n", a++, a++)` will display different values depending on the order chosen by the compiler. Assuming `a` is originally 0, then one may obtain "0 1" or "1 0" (with `a` always equal to 2 at the end of the instruction).

II.2.29 No multiple exits of function

Language: Any

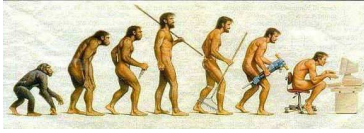
Paradigm: Any

Description: Functions where it is possible to get out in many places denote a non-structured programming in terms of control flow. Code understanding is deteriorated. Moreover, "cleanup" processes (for instance interrupt re-enabling in an interrupt handler) that may be performed at the exit must be duplicated at every exit point, hence duplicating ... the bugs. As a consequence, testing gets more difficult because of the multiple exit configurations that have to be tested to ensure the correct ending of the function in all the cases.

II.2.30 Test of function call returned value

Language: Any

Paradigm: Any



Description: The return value of any non-void function (i.e. of real function, not procedure for languages making the difference) has to be tested. In effect, a returned value indicating a success or error must be checked in order to set a robustness handler in the calling component. If a status code is returned by a component, it means that this component may fail in its process. Then not using this code may hide an incorrect behavior of the called function, preventing to detect a dysfunctional event in time.

II.2.31 Reduce explicit cast usages

Language: Any

Paradigm: Any

Description: Transgressing a data structure by applying an explicit cast may indicate a design error (architecture) or an implementation error. It can result in handling this structure incorrectly. Obviously, casts can be required in some particular cases. Hence, this construct is acceptable if the component manages low-level entities (ports, screens...) that require a particular representation of data because of their hardware constraints.

Memory allocation functions also often return non-typed pointers (`void*`) that need to be cast for proper usage afterwards. Aside these special cases, functions should be typed correctly to ensure that their arguments and returns always take and provide conveniently typed data structures, hence preventing the need for extra explicit casts. In the same idea, it is not wished to explicitly cast data in the type it already has. For example:

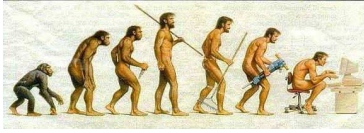
```
int a = 5 ;  
...  
a = (int) a + 42 ;
```

Even though this form will not add any extra code and will not change the program semantics compared to:

```
int a = 5 ;  
...  
a = a + 42 ;
```

the point is that systematically using cast will definitely hide errors caused by incompatible type mixing that would have been discovered if no cast were applied. For example:

```
struct tree {  
    struct node n ;  
    struct tree *children ;  
};  
struct node *n ;  
struct tree *t ;  
...  
n = getNode (...) ;  
t->children = (struct tree*) n ;  
...
```



The explicit cast of `n` in `struct tree*` will cause a type error to be hidden. Such errors commonly arise when handling complex data structures, such as recursive ones, where several types are mutually imbricated.

II.2.32 No multiple enters in a block

Language: Any

Paradigm: Imperative

Description: The ability to get into a block in many places denotes a non-structured programming in terms of control flow. Code understanding is deteriorated. Moreover, initializations performed at the regular entry point of the block may not be effective if enter is done elsewhere (for instance, interrupt disabling).

This obviously includes enters performed thanks to `goto` instruction, but also with the `switch` generalized conditional. Let's consider the pathological piece of code, left indented as it was automatically by a development text editor:

```
int main (void) {
    int j = 24 ;

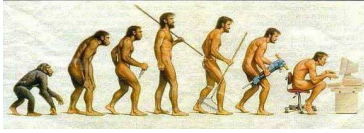
    switch (j) {
    case 23 : {
        int t = 42 ;
        printf ("Case 23\n") ;
        while (j != 23) {
            default : {
                printf ("Case default, %d\n", t) ;
            }
        }
    } break ;
    }
    return (0) ;
}
```

An extra entry indeed exists in the `while (j != 23)`'s block because of the default label of the `switch` construct. Because it is now possible to enter the block, then the loop, without entering first by the test whether `j` equals 23, the loop that was at a first glance dead code becomes an infinite loop for our value 24 of `j`. Moreover, the initialization of the variable `t` will never be performed, even though it is read by the `printf` instruction and now compilation warning will be issued! As a result, one will get a desperate infinite loop printing a garbage value for `t`.

```
Case default, 1628825324
Case default, 1628825324
Case default, 1628825324
...
```

II.2.33 No nested incrementations or decrementations

Language: Any



Paradigm: Imperative

Description: Multiple combinations of incrementation or decrementation operators inside a same expression first strongly reduce the understandability of this expression. One must take care of the order the side effects that are performed to ensure the final result. For instance, expressions like `t[--i]--` may not appear as pretty clear.

Moreover, depending on the evaluation order, side effects may overlap leading to unpredictable results. One must remind that the compiler is free to choose this order, and moreover, may introduce optimizations (common sub-expressions sharing, for instance) that can cause parts of computation to be evaluated once instead of several time, leading to an unexpected number of side effects. This is strongly true when specifying high levels of optimization, where compilers do not pretend to exactly preserve the original semantics of programs! Hence, expressions like:

```
int t[5][5][5] ;
int *u ;
...
v = t[--i][t[0][i][0]][i++]-- ;
v = (u++)[(--u)[--i]]++ ;
...
```

should obviously be definitely avoided.

II.2.34 Assignment within test

Language: C, C++

Paradigm: Object

Description: As describe in section II.1.1, usage of the assignment operator `=` in tests is rejected to prevent hiding a confusion between the test operator `==` and the assignment operator. Although this policy may seem simple, errors due to the confusion of these two operators (or more simply to an typing error on the keyboard) is really common. Even if the C idiom `if (ptr = f ())` is often used, to both assign and check a pointer, it has to be split in two separate instructions.

By extension, one advises not to perform any assignment in a test because such an expression is designed to semantically return a truth-value, not to perform any other action.

II.2.35 Assembly inlines

Language: Any

Paradigm: Any

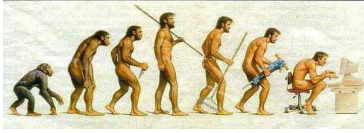
Description: Related assembly lines must be enclosed inside a same component. This helps to concentrate portions of the code that contains such constructs instead of spreading them all around the source. Because writing assembly code is a tricky process, it is valuable to be able to quickly identify sections where such inlines appear.

Moreover, because assembly code depends on the target architecture, portability issues obviously arise. If assembly code is clearly localized, then it will be easier to identify which parts would have to be rewritten if one migrates to another platform.

II.2.36 Avoid deeply nested loops

Language: Any

Paradigm: Imperative



Description: Loop nesting complicates the algorithmic understanding. We can see there a similarity with the reasoning in terms of dimensions: the human one generally has difficulties of representation beyond 3 dimensions. A nesting depth beyond 5 appears to be really too complex to understand and especially maintain.

II.2.37 Declaring for

Language: (extensions of C), C++

Paradigm: Imperative

Description: It is possible to define variables in the prelude of a for loop, for example in order to use them as loop indices. For instance:

```
for (int i = 0; i < ...; i++) {  
    ...  
}
```

in which a new local variable *i* is defined. Disseminating variable declarations make difficult traceability and maintainability. Moreover, the scope of this variable is not clearly identified by a block: it ranges from its definition until the end of the current block.

II.2.38 Declaration out of block header

Language: C++

Paradigm: Imperative

Description: Any local variables must be declared and initialized at the beginning of a block. Hence declaration in the running code, as allowed by C++ is disallowed. Conversely to C, C++ allows to mix instructions and declarations, like in the following example.

```
{  
    printf ("Foo\n") ;  
    int i = 0 ;  
    for (;;) {  
        i++ ;  
        int j ;  
        j = i + 1 ;  
    }  
}
```

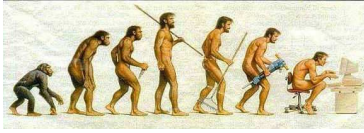
It is preferable to create a single block for all the local variables in order to clearly limit their scope. Moreover, this prevents from redefining variables with same name inside a same block just because definitions span all along the code instead of being declared only in a single and restricted area (hence making the definitions easier to check).

II.2.39 Avoid union usage

Language: C, C++

Paradigm: Imperative

Description: Use of union allows interpreting, according to different types, the same data. This construction is often based on the data representation in material architecture supporting the software. The most common problem is the portability of code using unions between computers with different endianness. Consider the hexadecimal value 0x4A3B2C1D.



In *big endian* representation (used by MC680x0, SPARC, POWERPC), most significant bytes will have lower addresses.

addresses		100	101	102	103	
bytes	...	4A	3B	2C	1D	...

In *little endian* representation (used mostly by Ix86, Pentium), addresses increase with byte significance.

addresses		100	101	102	103	
bytes	...	1D	2C	3B	4A	...

Moreover, *little endian* representation may imply a different representation for words longer than 8 bits, thanks to the notion of "*atomic element*". For instance, having atomic elements 16 bits wide, a 32 bits value would be represented as:

addresses		100		101		
atomic elements	...	2C	1D	4A	3B	...

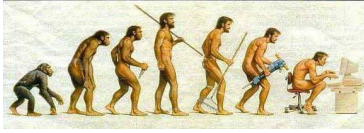
Hence, consider reading 32 bits at once then accessing them via a union split into bytes and 16 bits words:

```
union harm {
    unsigned int as_32 ;
    unsigned short as_16[2] ;
    unsigned char as_8[4] ;
};
```

The following program will print each view of a same data from this union.

```
int main ()
{
    unsigned int v = 0x4A3B2C1D ;
    union harm h ;

    h.as_32 = v ;
    printf ("as_32: 0x%lx\n", h.as_32) ;
    printf ("as_16: 0x%x 0x%x\n", h.as_16[0], h.as_16[1]) ;
    printf ("as_8: 0x%x %0xx 0x%x 0x%x\n",
        h.as_8[0], h.as_8[1], h.as_8[2], h.as_8[3]) ;
    return (0) ;
}
```



As a result, an Intel x86 (*little endian*) will display:

```
as_32: 0x4a3b2c1d  
as_16: 0x2c1d 0x4a3b  
as_8: 0x1d 0x2c 0x3b 0x4a
```

although a MC68000 (*big endian*) will display:

```
as_32: 0x4a3b2c1d  
as_16: 0x4a3b 0x2c1d  
as_8: 0x4a 0x3b 0x2c 0x4a
```

As another effect, union introduces an aliasing on a storage area. Indeed, the content of this memory can be seen in several different ways. The programmer has the responsibility for the correct way at one given moment “to look at” the zone, since the compiler cannot help on it. Hence, a robustness treatment is needed to ensure that the data is used in accordance with its format.

II.2.40 Arithmetic on pointer

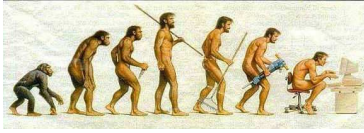
Language: Any

Paradigm: Imperative

Description: Performing arithmetic computation on pointers is both a weakness for portability and robustness. Because the size of machine's words can vary according to material architecture, handling pointers as integers can lead to addressing defaults when porting onto another architecture. Moreover, because data structure length may be specially handled by the compiler, assuming a structure is exactly as long as the sum of its elements length, then using this length to vary pointers may lead to invalid memory accesses. Such errors are usually pretty difficult to identify and to reproduce.

Finally, manually playing with the addresses is harmful, especially when dealing with local variables and parameters because, it relies on the knowledge of the way the compiler allocates them on the stack (and whether it allocates them really on the stack!).

```
unsigned int t[3] = {  
    0x10111213,  
    0x20212223,  
    0x30313233  
};  
  
int main ()  
{  
    int v ;  
  
    v = (unsigned int) t ;
```



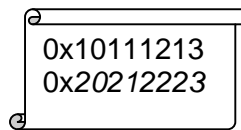
```

printf ("0x%x\n", *((int*) v)) ;
v++ ;
printf ("0x%x\n", *((int*) v)) ;
return (0) ;
}

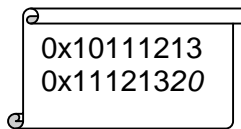
```

For the previous program at least two intuitive behaviors immediately come in mind to guess the second printed integer.

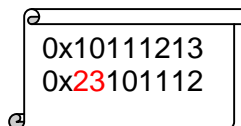
For a naïve programmer, who does not know about pointers, the result would probably be to display, as second integer, the value second of the second cell of the array (thinking that the address displacement will be done taking into account the scaling required by the size of one pointed element). That is:



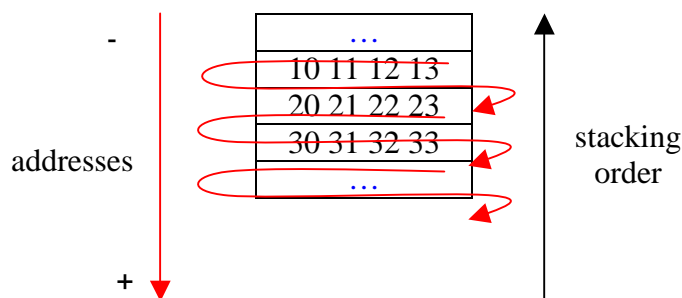
For a more skillful programmer, knowing that no scaling will then occur, the expected result may be to print the 32 bits integer value starting at the byte whose address is array's one plus one. That is:



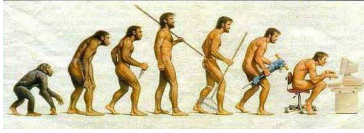
In fact none of these results may be the right one. For instance, on one compiler (this meaning that results may differ on an other!), the result is:



What does it mean? It means that the program will in effect display the integer value starting at address one byte (and not one int) **plus** the address of the beginning of the array. But as one can see, this address is not the intuitive "next byte in the array"! What happened? In fact, from this example, it is clear how the compiler performed the layout of the data. Let's just represent the stack with our local data:



The first cell of the array is at the lowest address. More accurately, assuming we are on a *little endian* architecture, the least significant byte of first cell is at the lowest address. Hence it is also the starting address of the array. Because the stack increases toward decreasing addresses, adding 1 to our v variable effectively points



on the byte 12 and not 11 as one could intuitively think. From this address, because one wants to print an `int`, it is assumed to be 32 bits wide, the compiler takes care of the endianness and display the bytes in reverse order, that's to say: "23 10 11 12". Voila! This illustrates the difficulty to be sure of the behavior of a program using arithmetic on pointers.

Nowadays, compilers allow managing pointers displacement according to the pointed data (scaling in term of address increment). Hence, it is safer to let the compiler performing the computation.

II.2.41 Remove useless assignments

Language: Any

Paradigm: Any

Description: An assignment is useless if the written variable is never read again or if it is never read before being rewritten on any execution path. If this assignment is useless it represents dead code that must be removed. Such an assignment can indicate a programming error due to the fact that the treatment using this variable was not implemented.

Some rare exceptions can be found when a variable is in fact mapped onto a hardware register acting as a latch. In effect; such a register has the very particular property that reading triggers a physical action in the hardware, whatever the read value obtained. In this case, the variable must be properly documented to clearly pinpoint its special behavior.

II.2.42 Remove useless comparisons

Language: Any

Paradigm: Any

Description: In the same spirit than II.2.41, a useless comparison represents some dead code that must be removed. Moreover, in languages like C and C++, because equality and assignment operators are really close (`==` versus `=`), useless comparisons like:

```
a == 5 ;
```

will probably denote a misspelling of an assignment.

II.2.43 Avoid deeply nested pointers

Language: Any

Paradigm: Imperative

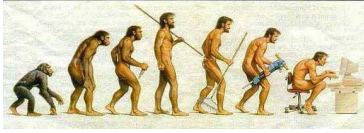
Description: Pointers represent levels of indirection in the memory. Nesting pointers (e.g: `int ***x`) cumulates indirections and makes a delicate algorithmic understanding. Beyond a nesting level of 4 pointers, data structures appear to be hardly understandable and maintainable. Moreover, C and C++ not providing *IN-OUT* and *OUT* modes for parameters, using a pointer on an argument is the only mean to simulate this mechanism. Nesting the modes by nesting pointers is another reason for such a construct to appear. Allowing such mode nesting multiplies the levels of mutability, hence making difficult to figure out which memory areas get modified.

II.2.44 Avoid Non-typed function return

Language: C, C++

Paradigm: Any

Description: In C/C++ a function whose return type is not explicitly defined implicitly returns `int` and not `void` as the syntax wrongly shows. Implicit cases often cause



lacks of functional of understanding, degrading maintainability. The return of such a function must be explicitly typed.

II.2.45 Explicitly declared functions

Language: Any

Paradigm: Any

Description: Any used function must be declared prior its usage. This ensures the compiler will be able to properly check the consistence of calls in term of parameters and return value. Without prior declaration, C and C++ compiler assume that a function return `int` (c.f. II.2.44). Moreover, the declaration enables the compiler to perform arguments conversions if needed, instead of considering that effective arguments keep their type before the call. Let's consider two separate source files that get linked together:

```
#include <stdio.h>
void callme (int v)
{
    printf ("%d\n", v) ;
}
```

and

```
int main ()
{
    double pi = 3.14159 ;
    callme (pi) ;
    return (0) ;
}
```

Because no declaration of `callme` is visible when compiling the function `main`, the compiler will not implicitly convert the argument `pi` of the call. Then the floating value will be passed as is and interpreted in `callme` as a regular integer. Then instead of getting 3 displayed, the result will be:

-266631570

Adding the extra line

```
void callme (int v) ;
```

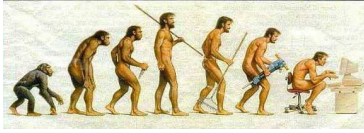
in the first source file (either directly in the file, or *#include-ing* a header file containing this declaration) would have provided the correct display, i.e. 3 (3.14159 rounded as an integer, that is truncated, only keeping the integer part of the number).

II.2.46 Parameter name mismatch

Language: Any

Paradigm: Any

Description: Providing names to the formal parameters of a function, when declaring it, is not an obligation. If they are mentioned, it is advised to respect these names in the implementation of the function. Change of the names can hide the fact that the significance of the parameters was changed or misunderstood between when the



function was declared (design) and when it was developed (implementation). Moreover, failing to this rule degrades maintainability, exposing different “non-formal” description of the function. By “non-formal”, we mean that even if parameters names are not relevant in term of computation (any program remains equivalent after identifier renaming), most often, humans heavily use parameters names to get an intuitive understanding of a function prototype. If a function is declared as:

```
float Divide (float numerator, float denominator) ;
```

obviously any user wishing to compute $\frac{3}{4}$ will tend to call the function with 3 as first and 4 as second argument! Implementing the function by:

```
float Divide (float a, float b) {  
    return (b / a) ;  
}
```

would really be both a stupid and confusing choice, even if this function would correctly compute $\frac{3}{4}$ when called by `Divide (4, 3)`! Without applying the present rule, a dishonest implementer could then argue that his function requires the numerator as second argument, “as written in the code”, and no matter the prototype, the truth is in the implementation! Then having to read the implementation of any function prior its usage would dramatically make development harder!

II.2.47 No usage of operator implicit priority

Language: Any

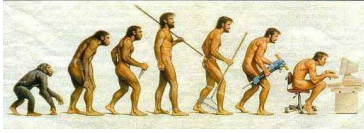
Paradigm: Any

Description: Except for mathematical operators having obvious priority and associativity relationship implying a unique evaluation order, parentheses have to be used in order to enforce the wished evaluation order. Failing to this rule may lead to unpredictable result depending on the compiler order choice. This is both portability and maintainability issue. Portability is obvious: changing the compiler may give a different result. Maintainability is also impacted because, during software evolutions (or corrections) instructions may vary around the expression, causing the compiler to change its code generation strategy (possibly for optimization purpose), hence changing the order operators are applied.

The following simple example illustrates the importance of specifying explicit priority. Although multiplication and division are mathematically commutative, depending on the execution order, two equivalent arithmetical expressions may lead to different results:

```
int main () {  
    int v = 65536 ;  
    int u = 65536 ;  
    printf ("%d\n", u * v / 65536) ;  
    printf ("%d\n", u * (v / 65536)) ;  
}
```

- 1) 0
- 2) 65536



II.2.48 Brace less statement

Language: Any

Paradigm: Imperative

Description: Instructions belonging to a block must be enclosed between delimiters (braces in C, C++), even in the degenerated case where the block only contains one instruction. This makes explicit the fact the block really needs one instruction. Due to the different development tools, the indentation policy may differ. A common error is to only check at the indentation to get a quick (and approximate) idea of the code's structure. When looking at the following piece of code:

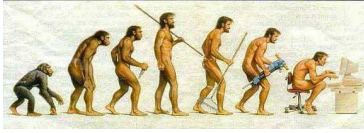
```
...  
if (p == 0)  
p++ ;  
use(p) ;  
dosomething () ;  
p = 42 ;  
...
```

obviously the question whether the use and dosomething instructions should be in the same block or not (here they are not according to the C syntax)!

Maintainability may also suffer such constructs because it is pretty easy to add instructions in a if or else clause that must be executed with the previously existing. If no block already exists, one must not forget to create one otherwise the freshly added instructions will not be executed as wished.

Finally, one must admit that block delimiters are pretty convenient to quickly figure out which code section is related to which test, especially when the development editor proposes highlight of matching braces. As an example, finding the conditional nesting in the following code in order to add one instruction when one of the conditions is satisfied in the following code is pretty painful, especially when the function spans on several pages!

Shorter but harder	Longer but easier
<pre>void Headache () { if (...) if (...) ... ; else { if (...) { for (;;) if (...) { if (...) ... ; else ... ; } if (...) ... ; } else ... ; } }</pre>	<pre>void LessHeadache () { if (...) { if (...) { } } else { if (...) { for (;;) { if (...) { if (...) { } } } } } }</pre>



<pre>else if (...) { if (...) ... ; } else ... ; } else if (...) { if (...) ... ; } else ... ; }</pre>	<pre>if (...) { } else { } else { if (...) { if (...) { } } else { } } }</pre>
--	--

II.2.49 Empty block

Language: Any

Paradigm: Imperative

Description: In term of maintainability, empty blocks are confusing. This poses the problem of ambiguity between empty and forgotten process. It is advised to prefer using the empty instruction (the semi character “;” in C) to explicitly denote a wished empty process. In effect, the main leading question is why a beginning and ending of block have been written without any code inside when it would have been easier not to write anything!

II.2.50 Variable used as constant

Language: Any

Paradigm: Any

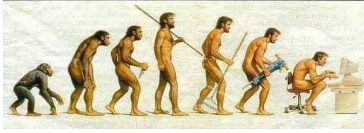
Description: Variables (global or local) used as constant must explicitly be declared “constant” (const in C, C++) or replaced by a macro-definition (#define in C, C++). This makes explicit the fact that the variable is no more assigned after its definition, removing the doubt of a missing process on it.

Then, this allows the compiler to store them in an appropriate data section. This section has indeed vocation to be put in non-volatile memory as well as the code program, thus the constant's value will be protected in case of variable trashing by the software or in case of failure of the material supporting the software. Moreover, on some constrained architecture, this will allow to save RAM (that is available in lower amount), storing these variables in ROM. In case where the variable is replaced by a macro-definition, the code obtained will usually be smaller and faster, leading in an encoding of the constant directly inside the assembly opcode. This last point may be interesting in the scope of constrained architectures and timings.

II.2.51 Octal constant

Language: C, C++

Paradigm: Any



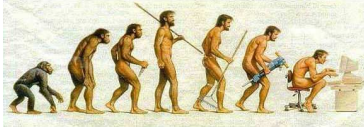
Description: In C and C++, octal constants are written prefixed by a **0** character. Hence, 010 is an octal constant whose decimal value is 8. An inattentive reading or an ignorance of octal C/C++ syntax can easily lead to a bad interpretation of the constant.

II.2.52 Dead code

Language: Any

Paradigm: Any

Description: Dead code is some unreachable code. If actually useless, it holds memory space unnecessarily and should be removed. Presence of such a code harms algorithmic understanding because it pollutes the really performed treatments. It may also indicate a programming error that turns this code unreachable whereas it should have been reachable.



III. References

- [1] "Computer-Related Risks", Peter G. Neumann. Addison-Wesley ISBN 0-201-55805-X.
- [2] "Ensuring Software Reliability", Ann Marie Neufelder. Dekker inc ISBN 0-8247-8762-5.
- [3] "Sûreté de fonctionnement des systèmes industriels", Alain Villemeur 1988. Éditions Eyrolles. ISSN 0399-4198
- [4] "Analysis of format string bugs", Andreas Thuemmel Feb-2001.
- [5] "Buffer Overflow and Format String Overflow Vulnerabilities", Kyung-Suk Lhee and Steve J. Chapin in Software Practice and Experience 2002.
- [6] Flaw finder - Secure Programming for Linux and Unix HOWTO, David A. Wheeler.
- [7] IEEE Working Group WEB site <http://grouper.ieee.org/groups/754/>
- [8] March 1981 issue of IEEE Computer (vol. 14, no. 3)
 - "A Proposed Standard for Binary Floating-Point Arithmetic", David Stevenson, et al.
 - "Analysis of Proposals for the Floating-Point Standard", William J. Cody.
 - "Underflow and the Denormalized Numbers", Jerome T. Coonen.
 - "Applications of the Proposed IEEE-754 Standard for Floating Point Arithmetic", David Hough.