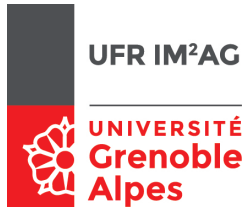

Software security, secure programming (and computer forensics)

Lecture 2: How (un)-secure is a programming language ?

Master on Cybersecurity – Master MoSiG (HECS & AISSE)

Academic Year 2016 - 2017

Overview

Sotware and cathedrals are very much the same -
first we build them, then we pray . . . *[S. Redwine]*

Overview

Sotware and cathedrals are very much the same -
first we build them, then we pray . . . *[S. Redwine]*

Unsecure softwares are everywhere . . . but:

- How much programming languages are responsibles ?
- Is there “language features” more (or less !) “secure” than others ?
- How to evaluate the “dangerousness” of a language ?
- How to recognize (and avoid) unsecure features ?
- How to enforce SW security at the programming level ?
  (even with an unsecure language)

Overview

Sotware and cathedrals are very much the same -
first we build them, then we pray . . . *[S. Redwine]*

Unsecure softwares are everywhere . . . but:

- How much programming languages are responsibles ?
- Is there “language features” more (or less !) “secure” than others ?
- How to evaluate the “dangerousness” of a language ?
- How to recognize (and avoid) unsecure features ?
- How to enforce SW security at the programming level ?
  (even with an unsecure language)

→ Let’s try to address these questions:

- in a partial way (i.e., through some example)
- without any “best language” hierarchy in mind . . .

Defining a programming language

An unreliable programming language generating un- reliable programs constitutes a far greater risk to our environment and to our society than unsafe cars, toxic pesticides, or accidents at nuclear power stations. Be vigilant to reduce that risk, not to increase it. *[C.A.R. Hoare]*

Defining a programming language

An unreliable programming language generating un- reliable programs constitutes a far greater risk to our environment and to our society than unsafe cars, toxic pesticides, or accidents at nuclear power stations. Be vigilant to reduce that risk, not to increase it. *[C.A.R. Hoare]*

language = syntax + type system + (dynamic) semantics

What is the influence of each of these elements w.r.t. security ?

A first concern is to reduce the discrepancies between:

- what the programmer has in mind
- what the compiler/interpreter understands
- how the executable code may behave . . .

Outline

Language syntax

concrete syntax = the (infinite) set of **"well-formed"** programs
(i.e., not immediately rejected by the compiler . . .)
$\rightarrow$ usually specified as an an-ambiguous **context-free grammar**

an-ambigous $\Rightarrow$ a **unique** derivation tree per program
$\Rightarrow$ a **unique** Abstract Syntax Tree per program
$\Rightarrow$ This grammar can be used inside a language "reference manual"

So, no possible programmer/compiler mis-understood, everything is fine ?

Language syntax

concrete syntax = the (infinite) set of **"well-formed"** programs
(i.e., not immediately rejected by the compiler . . .)
$\rightarrow$ usually specified as an an-ambiguous **context-free grammar**

an-ambigous $\Rightarrow$ a **unique** derivation tree per program
$\Rightarrow$ a **unique** Abstract Syntax Tree per program
$\Rightarrow$ This grammar can be used inside a language "reference manual"

So, no possible programmer/compiler mis-understood, everything is fine ?

However:

$\exists$ many examples of (very) bad syntactic choices those effects are

- to confuse the programmer
- to confuse the code reviewers . . .

$\Rightarrow$ opens the way to potential vulnerabilities . . .

Exemple 1: assignemnts in C

In the C langage:

- assignment operator is noted $=$
- an assignment is an **expression** (it returns a value)
- no booleans, integer value 0 interpreted as “false”

$\rightarrow$ a (well-known) trap for C beginners . . .

Exemple 1: assignemnts in C

In the C langage:

- assignment operator is noted $=$
- an assignment is an **expression** (it returns a value)
- no booleans, integer value 0 interpreted as “false”

$\rightarrow$ a (well-known) trap for C beginners . . .

A backdoor (?) in previous Linux kernel versions

```
if ((options==(__WCLONE|__WALL)) && (current->uid=0)
   retval = -EINVAL ;
 /* uid is 0 for root */
```

Exemple 2: macros and pre-processing in C

In the C langage:

$\exists$ a notion of **macros re-written** before compilation:

`#define M 42` $\rightsquigarrow$ `M` replaced by `42`

`#define F(X) (X=X+1)` $\rightsquigarrow$ `F(foo)` replaced by `(foo=foo+1)`

$\Rightarrow$ the effect is not always easy to predict . . .

Exemple 2: macros and pre-processing in C

In the C langage:

$\exists$ a notion of **macros re-written** before compilation:

`#define M 42` $\rightsquigarrow$ `M` replaced by `42`

`#define F(X) (X=X+1)` $\rightsquigarrow$ `F(foo)` replaced by `(foo=foo+1)`

$\Rightarrow$ the effect is not always easy to predict . . .

Example

Is there a difference between these two definitions ?

```
#define abs(X) (X)>=0?(X):(-X)
```

and

```
int abs (int x) {return x>=0?x:-x;}
```

Exemple 2: macros and pre-processing in C

In the C langage:

$\exists$ a notion of **macros re-written** before compilation:

`#define M 42` $\rightsquigarrow$ `M` replaced by `42`

`#define F(X) (X=X+1)` $\rightsquigarrow$ `F(foo)` replaced by `(foo=foo+1)`

$\Rightarrow$ the effect is not always easy to predict . . .

Example

Is there a difference between these two definitions ?

```
#define abs(X) (X)>=0?(X):(-X)
```

and

```
int abs (int x) {return x>=0?x:-x;}
```

Answer : compute `abs(x++)` ...

Exemple 3: comments in CAML

CAML **comments** are delimited by `(*` and `*)`, and they can be nested

Example

Is the example below syntaxtically correct ?
If yes, what is the final value of `a` ?

```
let a=true ;;
    (* this is inside a comment ...
    (*   msg="true ... *)
    let a=false ;;
    (*   msg="false ...*)
    ...
```

Exemple 3: comments in CAML

CAML **comments** are delimited by `(*` and `*)`, and they can be nested

Example

Is the example below syntaxtically correct ?
If yes, what is the final value of `a` ?

```
let a=true ;;
    (* this is inside a comment ...
    (*   msg="true ... *)
    let a=false ;;
    (*   msg="false ...*)
    ...
```

Indication: a constant string can be openned within a comment

Rk: a similar phenomenon may occur in C . . .

Outline

Types and type systems

What is a type?

- It defines the set of **values** an expression can take at run-time.
- It defines the set of **operations** that can be applied to an identifier
- It defines the **signature** of these operations
- It defines how variables should be **declared**, **initialized**, etc.

$\rightarrow$ allows to (safely) reject meaningless syntactically correct pgms !
Types and “typing rules" can be formalized using a type system . . .

Types and type systems

What is a type?

- It defines the set of **values** an expression can take at run-time.
- It defines the set of **operations** that can be applied to an identifier
- It defines the **signature** of these operations
- It defines how variables should be **declared**, **initialized**, etc.

$\rightarrow$ allows to (safely) reject meaningless syntactically correct pgms !
Types and “typing rules" can be formalized using a type system . . .

Type systems

A proof system on the (abstract) language syntax

- “judgements” + axioms + inference/deduction rules
- allows to prove whether a pgm is correctly typed (or not)
- allows to (fully) specify/implement the type-checking algorithm
- allows to reason on languages typing rules
- etc.

Question: but, what are types useful for in a programming language ?

What are Types Useful for?

At least three possible arguments . . .

Program correctness

```
var x : kilometers ;
var y : miles ;
x := x + y ; -- typing error
```

What are Types Useful for?

At least three possible arguments . . .

Program correctness

```
var x : kilometers ;
var y : miles ;
x := x + y ; -- typing error
```

Program readability

```
var e : energy := ... ; -- partition over the variables
var m : mass := ... ;
var v : speed := ... ;
e := 0.5 * (m*v*v) ;
```

What are Types Useful for?

At least three possible arguments . . .

Program correctness

```
var x : kilometers ;
var y : miles ;
x := x + y ; -- typing error
```

Program readability

```
var e : energy := ... ; -- partition over the variables
var m : mass := ... ;
var v : speed := ... ;
e := 0.5 * (m*v*v) ;
```

Program optimization

```
var x, y, z : integer ; -- and not real
x := y + z ; -- integer operations are used
```

Types as a safeguard ?

“Well-typed programs never go wrong . . . ” *[Robin Milner]*

safe language: no **untrapped** errors at runtime

→ So, let’s *well-type* our codes and everything will be fine ?

Types as a safeguard ?

“Well-typed programs never go wrong . . . ” *[Robin Milner]*

safe language: no **untrapped** errors at runtime

$\rightarrow$ So, let’s *well-type* our codes and everything will be fine ?

Unfortunately:

- This assertion holds only for a few programming languages . . .
  - needs an expressive and well-defined **type system**
  - this type system should be proved **correct** and **complete** . . .
    (**when ?** before excution ? at runtime ??)

Types as a safeguard ?

"Well-typed programs never go wrong . . . " *[Robin Milner]*

safe language: no **untrapped** errors at runtime

→ So, let's *well-type* our codes and everything will be fine ?

Unfortunately:

- This assertion holds only for a few programming languages . . .
  - needs an expressive and well-defined **type system**
  - this type system should be proved **correct** and **complete** . . .
    (**when ?** before excution ? at runtime ??)
- the whole language should be concerned (not only a small kernel)
  possible problems:
  - interactions with the OS (libraries, input-output, etc.)
  - interactions with other languages
  - specific optimizations
  - etc.

Types as a safeguard ?

"Well-typed programs never go wrong . . . " *[Robin Milner]*

safe language: no **untrapped** errors at runtime

→ So, let's *well-type* our codes and everything will be fine ?

Unfortunately:

- This assertion holds only for a few programming languages . . .
  - needs an expressive and well-defined **type system**
  - this type system should be proved **correct** and **complete** . . .
    (**when ?** before excution ? at runtime ??)
- the whole language should be concerned (not only a small kernel)
  possible problems:
  - interactions with the OS (libraries, input-output, etc.)
  - interactions with other languages
  - specific optimizations
  - etc.
- the programmer should understand the type system . . .

Types as a safeguard ?

"Well-typed programs never go wrong . . . " *[Robin Milner]*

safe language: no **untrapped** errors at runtime

→ So, let's *well-type* our codes and everything will be fine ?

Unfortunately:

- This assertion holds only for a few programming languages . . .
  - needs an expressive and well-defined **type system**
  - this type system should be proved **correct** and **complete** . . .
    (**when ?** before excution ? at runtime ??)
- the whole language should be concerned (not only a small kernel)
  possible problems:
  - interactions with the OS (libraries, input-output, etc.)
  - interactions with other languages
  - specific optimizations
  - etc.
- the programmer should understand the type system . . .
- compiler/interpreter + runtime environment "correct" as well ?

Still many typed/untyped but unsecure programming languages . . .

Types and type constructions

Basic types

integers, boolean, characters, etc.

Types and type constructions

Basic types

integers, boolean, characters, etc.

Type constructions

- cartesian product (structure)
- disjoint union
- arrays
- functions
- pointers
- recursive types
- ...

Types and type constructions

Basic types

integers, boolean, characters, etc.

Type constructions

- cartesian product (structure)
- disjoint union
- arrays
- functions
- pointers
- recursive types
- ...

But also:
subtyping, polymorphism, overloading, inheritance, coercion, overriding, etc.
[see `http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf`]

A bit of terminology . . .

Typed language

A dedicated type is associated to each identifier & expression

Ex: Java, Ada, C, Pascal, CAML, etc.

strongly typed vs weakly typed languages

$\rightarrow$ **explicit** (progmer aware) vs **implicit** (compiler aware) type conversions

A bit of terminology . . .

Typed language

A dedicated type is associated to each identifier & expression

Ex: Java, Ada, C, Pascal, CAML, etc.

strongly typed vs weakly typed languages

$\rightarrow$ **explicit** (progmer aware) vs **implicit** (compiler aware) type conversions

Untyped languages

A single (universal) type is associated to each identifier & expression

Ex: Assembly language, shell-script, Lisp, etc.

A bit of terminology . . .

Typed language

A dedicated type is associated to each identifier & expression

Ex: Java, Ada, C, Pascal, CAML, etc.

strongly typed vs weakly typed languages

$\rightarrow$ **explicit** (progmer aware) vs **implicit** (compiler aware) type conversions

Untyped languages

A single (universal) type is associated to each identifier & expression

Ex: Assembly language, shell-script, Lisp, etc.

Type checking

Check if “type annotations” are used in a consistent way throughout the pgm

A bit of terminology ...

Typed language

A dedicated type is associated to each identifier & expression

Ex: Java, Ada, C, Pascal, CAML, etc.

strongly typed vs weakly typed languages

$\rightarrow$ **explicit** (progmer aware) vs **implicit** (compiler aware) type conversions

Untyped languages

A single (universal) type is associated to each identifier & expression

Ex: Assembly language, shell-script, Lisp, etc.

Type checking

Check if "type annotations" are used in a consistent way throughout the pgm

Type inference

Compute a **consistent** type for each program fragments.

Remark

- in general both type checking and type inference are used
- in some languages (e.g., Haskel, CAML), type annotations are not mandatory (all types are/can be infered).

Subtyping

Subtyping is a **preorder relation** $\leq_T$ between types.

It defines a notion of substitutability:

If $T_1 \leq_T T_2$,
then elements of type T_2 may be replaced with elements of type T_1.

Subtyping

Subtyping is a **preorder relation** $\leq_T$ between types.

It defines a notion of substitutability:

If $T_1 \leq_T T_2$,
then elements of type T_2 may be replaced with elements of type T_1.

Example

- class inheritance in OO languages ;
- Integer $\leq_T$ Real (in several languages) ;
- Ada :

```
type Month is Integer range 1..12 ;
-- Month is a subtype of Integer
```

Static vs Dynamic type checking/inference

Static

All the type check/inference operations performed at *compile-time*

- all the information should be available . . .
- may induce some over-approximations of the pgm behavior (and reject **correct** pgms . . .) but allows to reject **incorrect** pgms

Static vs Dynamic type checking/inference

Static

All the type check/inference operations performed at *compile-time*

- all the information should be available . . .
- may induce some over-approximations of the pgm behavior (and reject **correct** pgms . . .) but allows to reject **incorrect** pgms

Dynamic

Some check/inference operations performed at *runtime*
$\rightarrow$ necessary to correctly handle:

- dynamic binding for variables or procedures
- polymorphism
- array bounds
- subtyping
- etc.

Leads to **trappped runtime errors** (i.e., through *exceptions*)

$\Rightarrow$ For most programming languages, both kinds of checks are used. . .

Some security problems raised by a bad understanding of typing rules

Weakly typed languages:

- implicit type cast/conversions
  integer $\rightsquigarrow$ float, string $\rightsquigarrow$ integer, etc.
- operator overloading
  - $+$ for addition between integers and/or floats
  - $+$ for string concatenation
  - etc.

$\Rightarrow$ weaken type checking and may confuse the programmer . . .

Some security problems raised by a bad understanding of typing rules

Weakly typed languages:

- implicit type cast/conversions
  integer $\rightsquigarrow$ float, string $\rightsquigarrow$ integer, etc.
- operator overloading
  - $+$ for addition between integers and/or floats
  - $+$ for string concatenation
  - etc.

$\Rightarrow$ weaken type checking and may confuse the programmer . . .

In practice:

- happens in many widely used programming languages . . .
  (C, Java, PHP. JavaScript, etc.)
- may depend on compiler options / decisions
  (e.g., size of an enumerated type in C)
- often exacerbated by a lack of clear and un-ambigous documentation . . .

Possible problems with type conversions [C]

Example 1

```
int x=3;
int y=4;
float z=x/y;
```

Is it correct, what's the value of `z` ?

Possible problems with type conversions [C]

Example 1

```
int x=3;
int y=4;
float z=x/y;
```

Is it correct, what's the value of `z` ?

Example 2

```
unsigned char x=128;
unsigned char y=2;
unsigned char z=(x*y);
unsigned char t=(x*y)/y;
```

Is it correct, what's the value of `z`, of `t` ?

Possible problems with type conversions [JavaScript]

Example 1: what is the ouptut produced ? why ?

```
if (0=='0') write("Equal"); else write ("Different");
switch (0) {
       case '0': write("Equal");
       default: write("Different");
}
```

Possible problems with type conversions [JavaScript]

Example 1: what is the ouptut produced ? why ?

```
if (0=='0') write("Equal"); else write ("Different");
switch (0) {
       case '0': write("Equal");
       default: write("Different");
}
```

Example 2: what is the ouptut produced ? why ?

```
write('0'==0) ; write(0=='0.0'); write('0'=='0.0');
```

Possible problems with type conversions [JavaScript]

Example 1: what is the ouptut produced ? why ?

```
if (0=='0') write("Equal"); else write ("Different");
switch (0) {
       case '0': write("Equal");
       default: write("Different");
}
```

Example 2: what is the ouptut produced ? why ?

```
write('0'==0) ; write(0=='0.0'); write('0'=='0.0');
```

Example 3: what is the ouptut produced ? why ?

```
a=1; b=2; c='Foo';
write(a+b+c);
write(c+a+b); write(c+(a+b);
```

Possible problems with type conversions [PHP]

Example 1: what is the ouptut produced ? why ?

Incrementing `"2d8"` returns `"2d9"` ...

```
$x="2d8";
print (++$x."\n"); print (++$x."\n"); print (++$x."\n");
```

Possible problems with type conversions [PHP]

Example 1: what is the ouptut produced ? why ?

Incrementing `"2d8"` returns `"2d9"` ...

```
$x="2d8";
print (++$x."\n"); print (++$x."\n"); print (++$x."\n");
```

Example 2: what is the ouptut produced ? why ?

```
if "oxf9" == "249e0"
    print ("True);
else
    print ("False);
```

Possible problems with type conversions [bash]

```
PIN_CODE=1234
echo -n "4-digits pin code for autentication: "
read -s INPUT_CODE; echo

if [ $PIN_CODE" -ne $INPUT_CODE" ]; then
   echo "Invalid Pin code"; exit 1
else
   echo "Authentication OK"; exit 0
fi
```

There is a very simple way to pawn this authentication procedure . . .

What about strongly typed languages ?

Examples : Java, Ada, ML, etc.

In principle:

strong and consistent type annotations
(programmer provided and/or automatically infered)

+

semantic preserving type-checking algorithm

$\Rightarrow$ **safe *and secure* codes (no untrapped errors . . .) ?**

What about strongly typed languages ?

Examples : Java, Ada, ML, etc.

In principle:

strong and consistent type annotations
(programmer provided and/or automatically infered)

+

semantic preserving type-checking algorithm

$\Rightarrow$ **safe *and secure* codes (no untrapped errors . . .) ?**

However:

- how reliable is the type-checking algorithm/implementation ?
- beware of *unsafe* constructions of these languages
  (often used for “performance” or “compatibility” reasons)
- beware of code integration from other languages . . .

$\hookrightarrow \exists$ security problems with Java code as well . . .

Outline

Programming language (dynamic) semantics

What is the meaning of a program ? How is it defined ?

Programming language (dynamic) semantics

What is the meaning of a program ? How is it defined ?

A possible answer:

- meaning of program = its **runtime behaviour**
  = the infinite set of all its possible execution sequences
  (including the “unforeseen ones” !)
- defined by the programming language (dynamic) semantics
  $\rightarrow$ defines the behavior of each language construct

Programming language (dynamic) semantics

What is the meaning of a program ? How is it defined ?

A possible answer:

- meaning of program = its **runtime behaviour**
  = the infinite set of all its possible execution sequences
  (including the “unforeseen ones” !)
- defined by the programming language (dynamic) semantics
  $\rightarrow$ defines the behavior of each language construct

Several ways to define a programming language semantics

- axiomatic semantic:
  how a pgm transforms a set of **assertions** (on its variables)
- denotational semantics:
  what is the **function** a pgm define ($\neq$ *how* it is computed)
- operational semantics:
  defines **how** an interpreter would execute the pgm

Programming language (dynamic) semantics

What is the meaning of a program ? How is it defined ?

A possible answer:

- meaning of program = its **runtime behaviour**
  = the infinite set of all its possible execution sequences
  (including the “unforeseen ones” !)
- defined by the programming language (dynamic) semantics
  $\rightarrow$ defines the behavior of each language construct

Several ways to define a programming language semantics

- axiomatic semantic:
  how a pgm transforms a set of **assertions** (on its variables)
- denotational semantics:
  what is the **function** a pgm define ($\neq$ *how* it is computed)
- operational semantics:
  defines **how** an interpreter would execute the pgm

Language semantic definition in practice:

informal text + compiler behvavior . . .

Possible issues of the language semantics w.r.t security ?

Some general issues:

- semantics should be known and understandable
- "unexpectable" side effects should be avoided (see examples later)
- undefined behaviors are (large !) security holes
  $\rightarrow$ the compiler can silently optimize the code ...
- the real program semantics is defined at the binary level
  what you see is not what you execute !
- pgm execution = mix of language semantics and OS runtime support
  (memory management, garbage collection, low-level library code, etc.)
- the compiler/interpreter should correctly implement the semantics ...
- etc.

Possible issues of the language semantics w.r.t security ?

Some general issues:

- semantics should be known and understandable
- "unexpectable" side effects should be avoided (see examples later)
- undefined behaviors are (large !) security holes
  $\rightarrow$ the compiler can silently optimize the code ...
- the real program semantics is defined at the binary level
  what you see is not what you execute !
- pgm execution = mix of language semantics and OS runtime support
  (memory management, garbage collection, low-level library code, etc.)
- the compiler/interpreter should correctly implement the semantics ...
- etc.

Other possible issues:

- *evaluators* inside the langage (PHP, JavaScipt, C, etc.)
  $\rightarrow$ allows to dynamically produce & execute code ...
- compiler-defined and machine-dependent behaviors
- etc.

Possible problems with side effects

With C

```
{int c=0; printf("%d %d\n",c++,c++); }
{int c=0; printf("%d %d\n",++c,++c); }
{int c=0; printf("%d %d\n",c=1,c=2); }
```

What is the output ? What is the final value of `c` ?

Possible problems with side effects

With C

```
{int c=0; printf("%d %d\n",c++,c++); }
{int c=0; printf("%d %d\n",++c,++c); }
{int c=0; printf("%d %d\n",c=1,c=2); }
```

What is the output ? What is the final value of `c` ?

With CAML

CAML is not a “pure” functionnal language . . .

```
lat alert = function true -> "T" | false -> "F";;
(alert false).[0] <- 'T';;
alert false;;
```

What is the result of the 2nd call to `alert` ?
This side effect can occur with CAML library functions as well . . .

Possible problems with C undefined behaviors

Dereferencing a null pointer is undefined

From CVE-2009-1987:

```
struct my_struct *s = f();
int t = s-> f ;   // s is dereferenced
if (!s)
return ERROR;
...
```

The `return ERROR` instruction may never execute . . . Why ?

Possible problems with C undefined behaviors

Dereferencing a null pointer is undefined

From CVE-2009-1987:

```
struct my_struct *s = f();
int t = s-> f ;   // s is dereferenced
if (!s)
return ERROR;
...
```

The `return ERROR` instruction may never execute . . . Why ?

Signed integer overflows are undefined

→ for a C compiler no need to check if `x + 100 < x` may overflow . . . !
A more concrete example:

```
int offset, len ;  // signed integers
...
if (offset < 0 || len <= 0)
    return -EINVAL;  // either offset or len is negative
if ((offset + len > MAXSIZE) || (offset + len < 0)
    return -EFBIG // offset + len does overflow
```

The `return -EFBIG` instruction may never execute . . . Why ?

Undefined behaviors (cont'd)

Many other undefined behaviours in C . . .

- **oversized shifts** (shifting more than n times an n-bits value)
- **division by zero**
- **out-of-bound pointers:**
  (pointer + offset) should not go beyond object boundaries
- **strict pointer aliasing:**
  pointers of different types should not be aliases
  comparison between pointers to $\neq$ objects is undefined

Undefined behaviors (cont'd)

Many other undefined behaviours in C ...

- **oversized shifts** (shifting more than n times an n-bits value)
- **division by zero**
- **out-of-bound pointers:**
  (pointer + offset) should not go beyond object boundaries
- **strict pointer aliasing:**
  pointers of different types should not be aliases
  comparison between pointers to $\neq$ objects is undefined

Compilers:

- may assume that undefined behaviors never happen
- have no "semantic obligation" in case of undefined behavior

$\Rightarrow$ **dangerous gaps** between pgmers intention and code produced ...

Rk: $\exists$ undefined behaviors in some C library functions (`memcpy`, `malloc`)

As a (temporary !) conclusion . . .

Some prog. language features lead to unsecure code . . .

- how do you choose a programming language ?
  mix from performance, efficiency, knowledge, existing code, etc.
  $\hookrightarrow$ what about security ???
- no “perfect language” yet . . .

As a (temporary !) conclusion . . .

Some prog. language features lead to unsecure code . . .

- how do you choose a programming language ?
  mix from performance, efficiency, knowledge, existing code, etc.
  $\hookrightarrow$ what about security ???
- no “perfect language” yet . . .

What can we do ?

- several **dangerous patterns** are now (well-)known . . .
  ex: buffer overflows with `strcpy` in C, SQL injection, integer overflows, `eval` function of JavaScript, etc.
  $\rightarrow$ use secure coding patterns instead . . .
- $\exists$ compiler options and (lightweight) code analysis tools
  $\rightarrow$ detect / restrict “borderline” pgm constructs
- security should become a (much) more important coding concern . . .

Credits and references

- “Mind your Language(s)” [Security & Privacy 2012]
  (E. Jaeger, O. Levillain, P. Chifflier - ANSSI)

- “Undefined Behavior: What Happened to My Code?” [APSys 2012]
  (X. Wang, H. Chen, A. Cheung, Z. Jia, M. Frans Kaashoek)