UFR IM²AG

UNIVERSITÉ Grenoble Alpes

Grenoble INP Ensimag

Software security, secure programming

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

Master M2 Cybersecurity

Academic Year 2023 - 2024

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]*

How to reduce this risk ?

language = syntax + (static) semantics (type system) + (dynamic) semantics

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

→ avoid discrepancies between:

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

→ avoid program undefinedness and run-time errors . . .

→ provide well-defined **abstractions** of execution plateform, security mechanisms (access control, authentication, etc.), . . .

Reminder: compilation vs interpretation (Several ways to execute a program . . .)

1. (full) Compilation [C, C++, Ada, Rust, . . .]

   $\hookrightarrow$ generation of an executable code from a source code by a compiler

   - efficient executable code, static code checking
   - portability issues . . .

2. (full) Interpretation [JavaScript, Perl, Ruby, . . .]

   source code level execution by an interpreter

   - portability, dynamic code checking $\leadsto$ remote/dynamic code execution
   - efficiency issues . . .

3. Hybrid approaches [Java, Python, JavaScript . . .]

   byte-code interpretation, JIT (Just-In-Time) compilation

   - portability vs efficiency trade-off
   - byte-code verification facilities

$\Rightarrow$ Consequences on the security ?

Outline

Security issues at the syntactic level

Types as a security safeguard ?

Security issues at runtime

Language syntax

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

unambigous ⇒ a **unique** derivation tree per program
⇒ a **unique** Abstract Syntax Tree per program

⇒ This grammar can be found inside a language "reference manual"

So, no possible programmer/compiler mis-understood, everything looks fine
. . .

However:

∃ many examples of (very) bad syntactic choices those effects are

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

⇒ 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”

→ 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` $\leadsto$ `M` replaced by `42`

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

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

Example: function inlining

Replace

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

by

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

Is it always safe ?

Try to compute `abs(x++)` ...

Outline

Security issues at the syntactic level

Types as a security safeguard ?

Security issues at runtime

Types

Type as data abstraction mechanisms

- 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.
- (formal) type systems to specify/implement type-checking algorithms

→ allows to (safely) reject some meaningless syntactically correct pgms

Types in programming languages

- **(strongly/weakly)** typed vs **untyped** languages
- type **checking** and/or type **inference**
- **static** and/or **dynamic** type checking/inference

Types as a security safeguard ? (1)

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

Type safety

type safe language ⇒ **NO** meaningless well-typed programs

↪ no “out of semantics” program execution, no **untrapped** run-time errors, no undefined behaviors, . . .

According to this definition:

- C, C++ are **not** type safe
- ML, Rust are type safe
- Java, C#, Python, OCaml are “considered as” type safe

Remarks about type safe languages:

- well-typedness is **preserved** at execution
  (bit strings) values are processed according to their (pgm level) types
- (meaningless) ill-typed programs can be rejected
  either at **compile time** or at **execution time**
- “type safe” type systems are usually **incomplete**
  ⇒ may also reject **meaningful** pgms (expressivity issue)

Types as a security safeguard ? (2)

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.
- pointer arithmetic
- etc.

$\Rightarrow$ weaken type checking and may confuse the programmer ...
(runtime type may not match with the intended operation performed)

In practice:

- happens in many widely used programming languages ...
  (C, C++, PHP, JavaScript, etc.)
- may depend on compiler options / decisions
- often exacerbated by a lack of clear and un-ambiguous documentation

Implicit type conversions [C]

Example 1 [C]

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

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

Example 2 [Java]

```
    short x = Short.MAX_VALUE;
    System.out.println(x+1);
short z = x+1;
    System.out.println(z);
```

Is it correct, what is the printed value ?

Implicit type conversions [JavaScript, PHP] (2)

Example 1 [JS]: 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 [JS]: what is the ouptut produced ? why ?

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

Example 3 [PHP]: what is the ouptut produced ? why ?

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

Implicit type conversions [JavaScript] (3)

Array slicing with JavaScript

```
var a=[] ;
// fill array a with 100 values from 0.123 to 99.123
for (var i=0; i<100; i++) a.push(i + 0.123) ;
// fill array b with the 10 first values of a
var b = a.slice(0, 10);
```

$\rightsquigarrow$ b = [0.123, 1.123, 2.123, ..., 9.123]

Implicit conversion and object values

```
var c = a.slice(0, {valueOf:function (){return 10;}});
```

$\rightsquigarrow$ c = [0.123, 1.123, 2.123, ..., 9.123]

Now with an (un-detected) side effect . . .

```
var d = a.slice(0,valueOf:function(){a.length=0;return 10;}});
```

$\rightsquigarrow$ d = [0.123, 1.123, 2.1219959146e-313, 0, 0, ...]

$\rightarrow$ out-of-bounds read, memory leakage [CVE-2016-4622 in JavaScriptCore]

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** and **type safe** languages ?

Examples : Java, Ada, Rust, 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 may arise as well . . . !

Outline

Security issues at the syntactic level

Types as a security safeguard ?

Security issues at runtime

Programming language (dynamic) semantics

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

A possible answer:

- meaning of a 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

However, **language semantic definition in practice:**

informal text + compiler behavior ...

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

- 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 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 . . .

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

What is the result of the 2nd call to `alert`[1] ?

[1] no longer the case with recent CAML versions . . .

Possible problems with C undefined behaviors

Out-of-bounds buffer accesses are undefined

```
char i=0;
char t[10] ;
t[10]=42;
printf("%d\n", i) ;
```

What is the printed value ? Why ?

Signed integer overflows are undefined

```
int a, b ;  // signed integers
...
if ( a <= 0 || b <= 0)
    return ERROR1;  // either a or b is negative
// from here both a and b are assumed strictly positive
if (a + b < 0)
    return ERROR2; // a + b does overflow
...
```

The `return ERROR2` instruction may never execute . . . Why ?

Undefined behaviors (cont'd)

Many undefined behaviours in C ...

- **out-of-bounds** buffer accesses
- **arithmetic overflows** on signed integers
- **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
- **etc**

Compilers:

- may assume that undefined behaviors never happen
- have no "semantic obligation" in case of undefined behavior
  $\rightsquigarrow$ aggressive optimizations ... able to **suppress security checks!**

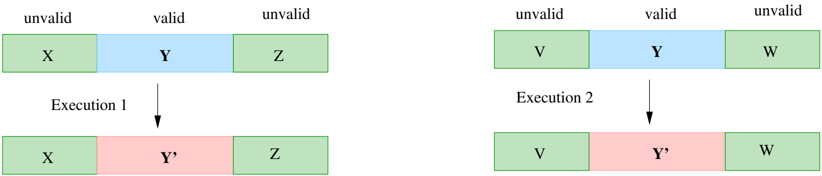
$\Rightarrow$ **dangerous gaps** between pgmers intention and code produced ...

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

Memory safety

A (highly !) **desired runtime property**:

only valid memory locations should interfere with
– or be interfered with – the pgm behavior



valid memory locations ?

- of correct type and size $\leadsto$ no <u>spatial</u> memory violation
- properly allocated and initialized, "freshness" (no re-use)
  $\leadsto$ no <u>temporal</u> memory violation
- no memory leakage, etc.

Memory safety in practice ?

- some consensus: "C (and C++) are **not** memory-safe", "Java (and C#) are considered memory-safe", "Rust is designed to be memory-safe"

- ∃ **language/compiler extensions** to **partially enforce** memory safety

- real world context (finite memory space, unsafe language constructs) **weaken** memory safety in practice

the root cause of approximately 70% of security vulnerabilities that Microsoft fixes and assigns a CVE are due to memory safety issues

Remark

Memory safety requires type safety . . .

Calling external code

Software applications may rely on "external code" (OS primitives and/or specific libraries), sometimes written in $\neq$ programming languages : file and resource management, data bases, GUI, crypto, access control, etc.

$\Rightarrow$ Two main advices:

Correctly use the provided APIs

- beware of types and type conversions . . .
  ($\neq$ typing rules and data representation from one language to another)
- respect the "programming guides" (e.g., in crypto: long enough keys, initialization, default modes, etc.)

Check what you transmit & receive

- input and output control and sanitization
  (see CWEs on **command injection**, **code injection**, **argument injection/modifification**, **improper input neutralization**, etc.
- use dedicated APIs (when available)
  e.g., use JavaMail*TM* than `Runtime.exec()` to send a mail in Java

As a (temporary !) conclusion . . . (1)

Some important programming language features:

- type safety: the actual (runtime) type **matches** with the expected one
  $\rightarrow$ memory operations are compatible with the source-level abstraction
  (may forbid the use of un-initialized variables)

- memory safety: no **unintended/invalid** memory access

- thread safety: no **unintended** operations between threads
  $\rightarrow$ no race conditions, safe synchronization facilities, etc.

- no undefined behaviors ($\sim$ "time bombs")
  - no need for the compiler to detect or mitigate them !
  - $\rightsquigarrow$ aggressive optimizations, able to suppress security checks !

- control-flow integrity: preserves intended control-flow
  method call/returns (e.g, Java), valid paths in the control-flow graph, etc.

- data-flow integrity: preserves intended use-def variable relation relations

- etc.

As a (temporary !) conclusion . . . (2)

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

- how do you choose a programming language ?
  mix from performance, efficiency, knowledge, existing code, etc.
  ↪ what about security ???
  (have a look at IEEE 2022 programming language ranking)
- no "perfect language" yet . . . but some languages are improperly used !

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.
  → use secure coding patterns instead . . . **[see next week !]**
- ∃ compiler options and (lightweight) code analysis tools
  → 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)

- "The Programming Languages Enthusiast" (Michael Hicks) blog
  - Software security is a programming language issue
  - what is type safety ?
  - what is memory safety ?