Programming Language Concepts
Language-Based Security

Janyl Jumadinova
20 April, 2017
Overview

Unsecure software are everywhere, but:

- How much programming languages are responsible for?
- Are 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)
An unreliable programming language generating unreliable 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]
Imagine...

- Tossing together 100,000,000 lines of code
- From 1,000s of people at 100s of places
- And running 10,000,000s of computers holding data of value to someone
- And any 1 line could have arbitrary effect

All while supporting the principle of least privilege?!
"Give each entity the least authority necessary to accomplish each task"
“Give each entity the least authority necessary to accomplish each task”

versus

- Buffer overruns (read/write any memory)
- Code injection (execute any memory)
- Coarse library access (system available by default)
Existing mechanisms to enforce SW security

At the programming level:

- disclosed vulnerabilities → language weaknesses databases → secure coding patterns and libraries;
Existing mechanisms to enforce SW security

At the programming level:
- disclosed vulnerabilities $\rightarrow$ language weaknesses databases $\rightarrow$ secure coding patterns and libraries;
- aggressive compiler options $+$ code instrumentation $\rightarrow$ early detection of unsecure code, ...
Existing mechanisms to enforce SW security

At the programming level:
- disclosed vulnerabilities $\rightarrow$ language weaknesses databases $\rightarrow$ secure coding patterns and libraries;
- aggressive compiler options $+$ code instrumentation $\rightarrow$ early detection of unsecure code, ...

At the OS level:
- sandboxing
- address space randomization
- non executable memory zones, ...
Existing mechanisms to enforce SW security

At the programming level:
- disclosed vulnerabilities $\rightarrow$ language weaknesses databases $\rightarrow$ secure coding patterns and libraries;
- aggressive compiler options $+$ code instrumentation $\rightarrow$ early detection of unsecure code, ...

At the OS level:
- sandboxing
- address space randomization
- non executable memory zones, ...

At the hardware level:
- Trusted Platform Modules (TPM)
- secure crypto-processor
- CPU tracking mechanisms (e.g., Intel Processor Trace), ...
CERT Secure Coding Standards

CERT C Secure Coding Standard
- Version 1.0 (C99) published in 2009
- Version 2.0 (C11) published in 2011
- ISO/IEC TS 17961 C Secure Coding Rules Technical Specification
- Conformance Test Suite

CERT C++ Secure Coding Standard
- Not completed/not funded

CERT Oracle Secure Coding Standard for Java
- Version 1.0 (Java 7) published in 2011
- Java Secure Coding Guidelines
- Identified Java rules applicable to Android development
- Planned: Android-specific version designed for the Android SDK

The CERT Perl Secure Coding Standard
- Version 1.0 under development
What is the influence of PL 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
Security issues at the syntactic level

- **concrete syntax** = the (infinite) set of “well-formed” programs (i.e., not immediately rejected by the compiler ...)
  - usually specified as an **un-ambiguous context-free grammar**
Security issues at the syntactic level

- **concrete syntax** = the (infinite) set of “well-formed” programs (i.e., not immediately rejected by the compiler ...)
  - usually specified as an un-ambiguous context-free grammar
- un-ambiguous
  - a **unique** derivation tree per program
  - a **unique** Abstract Syntax Tree per program → This grammar can be used inside a language “reference manual”
Security issues at the syntactic level

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

- un-ambiguous
  - a **unique** derivation tree per program
  - a **unique** Abstract Syntax Tree per program → This grammar can be used inside a language “reference manual”

- So, no possible programmer/compiler misunderstanding, everything is fine?
Language Syntax

- Many examples of (very) bad syntactic choices those effects are
  - to confuse the programmer
  - to confuse the code reviewers
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
Example 1: assignments in C

In the C language:

- assignment operator is noted =
- an assignment is an expression (it returns a value)
- no booleans, integer value 0 interpreted as “false”
Example 1: assignments in C

In the C language:

- 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 . . .

Ex.: backdoor (?) in previous Linux kernel versions
if ((options==(__WCLONE|__WALL)) && (current->uid=0)
    retval = -EINVAL ;
/* uid is 0 for root */
Example 1: assignments in C

In the C language:

- 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 . . .

Ex.: backdoor (?) in previous Linux kernel versions

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

Example 2: macros and pre-processing in C

In the C language:

- there is a notion of macros re-written before compilation:
  
  ```c
  #define M 42  ->  M replaced by 42
  #define F(X) (X=X+1)  ->  F(foo) replaced by (foo=foo+1)
  ```
Example 2: macros and pre-processing in C

In the C language:

- there is a notion of macros re-written before compilation:
  - `#define M 42` -> M replaced by 42
  - `#define F(X) (X=X+1)` -> F(foo) replaced by (foo=foo+1)
- the effect is not always easy to predict ...
Example 2: macros and pre-processing in C

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;}
Example 2: macros and pre-processing in C

Is there a difference between these two definitions?

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

and

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

Compute abs(x++)...
Types as a security safeguard?

Types and “typing rules” can be formalized using a type system:

- A proof system on the (abstract) language syntax
- "judgements" + axioms + inference/deduction rules
- Allows to prove whether a program is correctly typed (or not)
- Allows to (fully) specify/implement the type-checking algorithm
- Allows to reason on languages typing rules
Types and “typing rules” can be formalized using a **type system**

Type system: a proof system on the (abstract) language syntax

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

```plaintext
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 optimization

```plaintext
var x, y, z : integer ; – and not real
x := y + z ; – integer operations are used
```
Types as a safeguard?

**safe language:**

no untrapped errors at runtime
Types as a safeguard?

*safe language:*

- no untrapped errors at runtime

Well-type our codes and everything will be fine? But, ...
Types as a safeguard?

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

Well-type our codes and everything will be fine? But, ...
- This assertion holds only for a few programming languages ...
- The whole language should be concerned (not only a small kernel) possible problems
- the programmer should understand the type system
- compiler/interpreter + runtime environment “correct” as well?
Well-type our codes and everything will be fine? But, ...

- This assertion holds only for a few programming languages ...
- The whole language should be concerned (not only a small kernel) possible problems
- the programmer should understand the type system
- compiler/interpreter + runtime environment “correct” as well?

Still many typed/untyped but unsecure programming languages
Typed vs. Untyped languages

Typed language:
- A dedicated type is associated to each identifier and expression
  - Ex: Java, Ada, C, Pascal, etc.
- Strongly typed vs. weakly typed languages
  - explicit (programmer aware) vs. implicit (compiler aware)
  type conversions
Typed vs. Untyped languages

**Typed language:**
- A **dedicated** type is associated to each identifier and expression
  - Ex: Java, Ada, C, Pascal, etc.
- **Strongly** typed vs. **weakly** typed languages
  - **explicit** (programmer aware) vs. **implicit** (compiler aware)
  - *type conversions*

**Untyped language:**
- A single (universal) type is associated with each identifier and expression
  - **Ex:** Assembly language, shell-script, Lisp, etc.
Type checking: check if “type annotations” are used in a consistent way throughout the program.
Types

- **Type checking**: check if “type annotations” are used in a consistent way throughout the program
- **Type inference**: compute a consistent type for each program fragments
Types

- **Type checking**: check if “type annotations” are used in a consistent way throughout the program
- **Type inference**: compute a consistent type for each program fragments
- 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).
Security problems raised by a bad understanding of typing rules

**Weakly typed languages:**

- implicit type cast/conversions
  - integer → float, string → integer, etc.

- operator overloading
  - + for addition between integers and/or floats
  - + for string concatenation
Security problems raised by a bad understanding of typing rules

**Weakly typed languages:**

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

Weaken type checking and may confuse the programmer
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 program behavior (and reject correct programs), but allows to reject incorrect programs
Static vs. Dynamic type checking/inference

Dynamic: Some check/inference operations performed at runtime → necessary to correctly handle:

- dynamic binding for variables or procedures
- polymorphism
- array bounds
- subtyping
Static vs. Dynamic type checking/inference

*Dynamic*: Some check/inference operations performed at runtime → necessary to correctly handle:
  - dynamic binding for variables or procedures
  - polymorphism
  - array bounds
  - subtyping

Leads to trapped runtime errors (i.e., through exceptions)
Possible problems with type conversions (C)

Example 1:

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

Is it correct, what is the value of z?
Possible problems with type conversions (C)

Example 1:

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

Is it correct, what is the value of \( z \)?

Example 2:

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

Is it correct, what is the value of \( z, t \)?
Type conversion problems (JavaScript)

Example 1: What is the output produced? why?

```javascript
if (0==‘0’) write(‘Equal’); else write (‘Different’);
switch (0) {
    case ‘0’: write(‘Equal’);
    default: write(‘Different’);
}
```

Example 2:
```javascript
write(0==0) ; write(0==’0.0’); write(0==’0.0’);
```

Example 3:
```javascript
a=1; b=2; c=‘Foo’;
write(a+b+c);
write(c+a+b); write(c+(a+b));
```
Type conversion problems (JavaScript)

Example 1: What is the output produced? why?

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

Example 2:

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

Example 3:

```javascript
a=1; b=2; c='Foo';
write(a+b+c);
write(c+a+b); write(c+(a+b);
```
Type conversion problems (JavaScript)

Example 1: What is the output produced? why?

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

Example 2:

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

Example 3:

```javascript
a=1; b=2; c='Foo';
write(a+b+c);
write(c+a+b); write(c+(a+b);
```
What about strongly typed languages?

*Examples:* Java, Ada, ML, etc.

**In principle:**

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

+ semantic preserving type-checking algorithm

→ safe and secure codes (no untrapped errors ... )?

▶ 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 ...
What about strongly typed languages?

*Examples: Java, Ada, ML, etc.*

**In principle:**

- strong and consistent type annotations (programmer provided and/or automatically inferred)
- semantic preserving type-checking algorithm
- → 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 ...
Security issues at runtime

Programming language (dynamic) semantics
What is the meaning of a program? How is it defined?
Security issues at runtime

Programming language (dynamic) semantics
What is the meaning of a program? How is it defined?

Possibly,

- meaning of program = its runtime behaviour = the infinite set of all its possible execution sequences (including the unforeseen ones)
Security issues at runtime

Programming language (dynamic) semantics
What is the meaning of a program? How is it defined?

Possibly,

- 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 -> defines the behavior of each language construct
Possible issues of the language semantics w.r.t security?

- semantics should be *known* and *understandable*
Possible issues of the language semantics w.r.t security?

- semantics should be *known* and *understandable*
- “unexpectable” *side-effects* should be avoided
Possible issues of the language semantics w.r.t security?

- semantics should be *known* and *understandable*
- “unexpectable” *side-effects* should be avoided
- *undefined* behaviors are (large!) *security holes* (the compiler can silently optimize the code)
Possible issues of the language semantics w.r.t. security?

- semantics should be *known* and *understandable*
- “unexpected” *side-effects* should be avoided
- *undefined* behaviors are (large!) *security holes* (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)
Possible issues of the language semantics w.r.t security?

- semantics should be known and understandable
- “unexpected” side-effects should be avoided
- undefined behaviors are (large!) security holes (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)
- program execution = mix of language semantics and OS runtime support (memory management, garbage collection, low-level library code, etc.)
Possible issues of the language semantics w.r.t security?

- semantics should be *known* and *understandable*
- “unexpectable” *side-effects* should be avoided
- *undefined* behaviors are (large!) *security holes* (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)
- program 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
Possible issues of the language semantics w.r.t security?

- semantics should be *known* and *understandable*
- “unexpectable” *side-effects* should be avoided
- *undefined* behaviors are (large!) *security holes* (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)
- program 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
Possible issues of the language semantics w.r.t security?

- semantics should be *known* and *understandable*
- “unexpectable” *side-effects* should be avoided
- *undefined* behaviors are (large!) *security holes* (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)
- program 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
- compiler-defined and machine-dependent behaviors
Possible problems with side effects

```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); }
```
Possible problems with side effects

```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 C undefined behaviors

*Dereferencing a null pointer is undefined*
From CVE-2009-1987:

```c
struct my_struct *s = f();
int t = s-> f ; // s is dereferenced
if (!s)
    return ERROR;
...
```
Possible problems with C undefined behaviors

*Dereferencing a null pointer is undefined*
From CVE-2009-1987:

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

The return ERROR instruction may never execute
Signed integer overflows are undefined: for a C compiler no need to check if $x + 100 < x$ may overflow.

```c
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 -EFAULT // offset + len does overflow
```
Possible problems with C undefined behaviors

Signed integer overflows are undefined: for a C compiler no need to check if \( x + 100 < x \) may overflow.

```c
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
Evolution of a tackling software security

- first, do nothing
  some problems may happen and then you patch
Evolution of a tackling software security

- first, do nothing
  some problems may happen and then you patch
- then, implement support for regular patching
Evolution of a tackling software security

- first, do nothing
  some problems may happen and then you patch
- then, implement support for regular patching
- then, pre-emptively have products pen-tested
  eg. hire pen-testers, set up bug bounty program, ...
Evolution of a tackling software security

- first, do nothing
  some problems may happen and then you patch
- then, implement support for regular patching
- then, pre-emptively have products pen-tested
  eg. hire pen-testers, set up bug bounty program, ...
- then, use static analysis tools when coding
Evolution of a tackling software security

- first, do nothing
  some problems may happen and then you patch
- then, implement support for regular patching
- then, pre-emptively have products pen-tested
  eg. hire pen-testers, set up bug bounty program, ...
- then, use static analysis tools when coding
- then, train your programmers to know about common problems
Evolution of a tackling software security

- first, do nothing
  some problems may happen and then you patch
- then, implement support for regular patching
- then, pre-emptively have products pen-tested
  eg. hire pen-testers, set up bug bounty program, ...
- then, use static analysis tools when coding
- then, train your programmers to know about common problems
- then, think of abuse cases, and develop security tests for them
Evolution of a tackling software security

- first, do nothing
  some problems may happen and then you patch
- then, implement support for regular patching
- then, pre-emptively have products pen-tested
  eg. hire pen-testers, set up bug bounty program, ...
- then, use static analysis tools when coding
- then, train your programmers to know about common problems
- then, think of abuse cases, and develop security tests for them
- then, start thinking about security before you even start development
Some programming language features lead to unsecure code

- how do you choose a programming language?
  - mix from performance, efficiency, knowledge, existing code, etc.
- what about security?
Some programming language features lead to unsecure code

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

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

▷ compiler options and (lightweight) code analysis tools
  – detect / restrict “borderline” program constructs

▷ security should become a (much) more important coding concern