Programming Language Concepts
Overview of Language-Based Security

Janyl Jumadinova

17–19 April, 2023

What is Software Security?

Not the same as security software, such as:

- Firewalls, intrusion detection, encryption.
- Protection of the environment within which the software operates.

What is Software Security?

Not the same as security software, such as:

- Firewalls, intrusion detection, encryption.
- Protection of the environment within which the software operates.

Goal: Software security is the idea of engineering software so that it continues to function correctly under malicious attack.

Software Security

We focus on software security, but don't forget that security is about, in no particular order, people (users, employees, sys-admins, programmers,...), access control, passwords, biometrics, cryptology, protocols, policies and their enforcement, monitoring, auditing, legislation, persecution, liability, risk management, incompetence, confusion, lethargy, stupidity, mistakes, complexity, software, bugs, verification, hackers, viruses, hardware, operating systems, networks, databases, public relations, public perception, conventions, standards, ..., physical protection, data protection,...

Sources of Software Insecurity

- Complexity, inadequacy, and change.
- Incorrect or changing assumptions (capabilities, inputs, outputs).
- Flawed specifications and designs.
- Poor implementation of software interfaces (input validation, error and exception handling).
- Inadequate knowledge of secure coding practices.

Sources of Software Insecurity

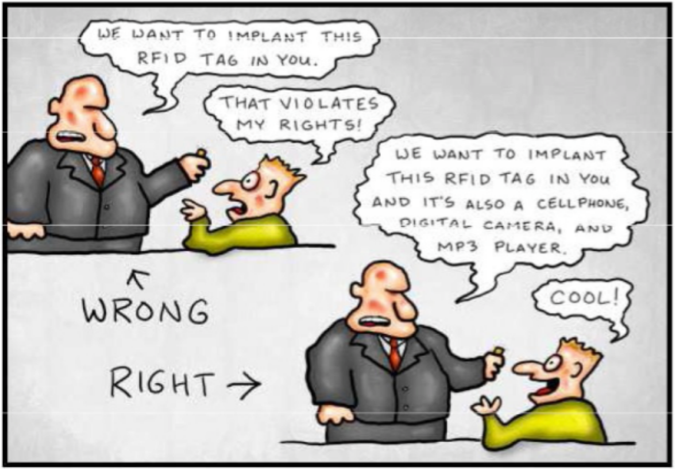
- Unintended, unexpected interactions
  - with other components
  - with the software's execution environment
- Absent or minimal consideration of security during all lifecycle phases
- Not thinking like an attacker

Sources of Software Insecurity

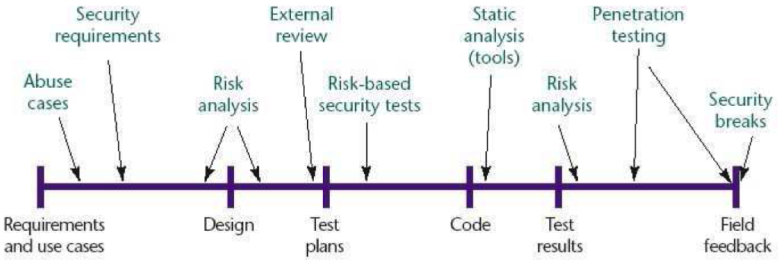
- Unintended, unexpected interactions
  - with other components
  - with the software's execution environment
- Absent or minimal consideration of security during all lifecycle phases
- Not thinking like an attacker

Most of the vulnerabilities in the National Vulnerability Database (`https://nvd.nist.gov/`) are due to programming errors

Security is always a secondary concern


WE WANT TO IMPLANT THIS RFID TAG IN YOU.
THAT VIOLATES MY RIGHTS!
WRONG
WE WANT TO IMPLANT THIS RFID TAG IN YOU AND IT'S ALSO A CELLPHONE, DIGITAL CAMERA, AND MP3 PLAYER.
COOL!
RIGHT

Security Concepts



Software Security

Different types of software vulnerabilities:

- **bugs** aka implementation flaws or code-level defects.
  - vulnerability in the software introduced when implementing a system.
- **design flaws** vulnerability in the design.

Roughly speaking, bugs and design flaws are equally common.

Software Security Class Activity

Find and discuss three recent vulnerabilities.

- `https://nvd.nist.gov/`
- `http://www.us-cert.gov/ncas/bulletins`

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 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)

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?!

Least Privilege

"Give each entity the least authority necessary to accomplish each task"

Least Privilege

> “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 → language weaknesses databases → secure coding patterns and libraries;
- aggressive compiler options + code instrumentation → early detection of unsecure code, ...

Existing mechanisms to enforce SW security

At the programming level:

- disclosed vulnerabilities → language weaknesses databases → secure coding patterns and libraries;
- aggressive compiler options + code instrumentation → 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 → language weaknesses databases → secure coding patterns and libraries;
- aggressive compiler options + code instrumentation → early detection of unsecure code, ...

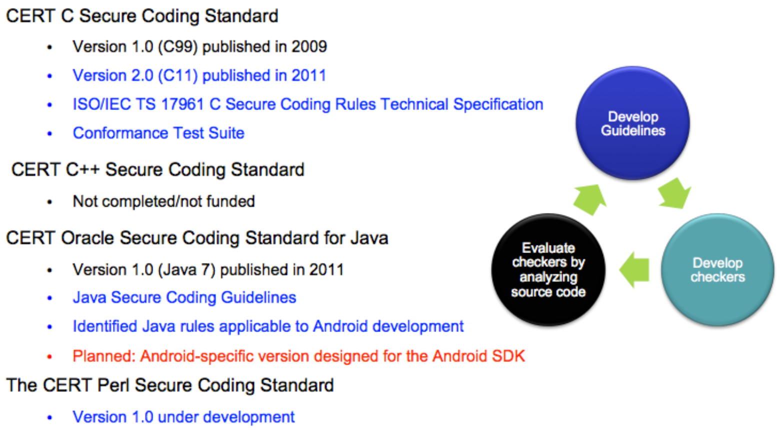
At the OS level:

- 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**
- Bad syntactic choices can:
  – to confuse the programmer
  – to confuse the code reviewers

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**
- Bad syntactic choices can:
  – 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 . . .

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 */
```

https://freedom-to-tinker.com/2013/10/09/the-linux-backdoor-attempt-of-2003/

Types as a security safeguard?

Types as a security safeguard?

Types and "typing rules" can be formalized using a **type system**

Types as a security safeguard?

Types and "typing rules" can be formalized using a **type system**

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

- 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

```
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

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 ?

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 ?

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.

Types

- **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 $->$ float, string $->$ 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:

```
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:

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

Is it correct, what is 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 is the value of z, t?

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 ...)?

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*
- "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)

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.)

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

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

Summary

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?

Summary

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