Language Based Security: Overview of Types

Lujo Bauer

18-732
Spring 2015
(Slides credit: Deepak Garg)
Course topics

0. Attacks
1. Software Security Architectures
2. Security Analysis of Software
3. Language-based Security
4. Run-time Security Enforcement

Goal: Become familiar with (language-based) approached to build software that is safe by construction
Language-based security

- **Objective**
  - Design security into the language
  - Compiler rules out insecure programs
  - Compiler does not run the program (no testing)

- **What’s insecure?**
  - Buffer overflows
  - Information-flow leaks
  - Violations of access rights
  - ...

- **One approach is based on types**

Today: overview of types
Next few lectures:
- Non-interference
- Security-typed languages
- Typed assembly language
Types: Overview

- A type is a *specification* of data or code in a program

Examples from C:
- Basic types
  - int, char, float, double, void
  - int x;  --- variable x will store an integer
- Function types
  - int -> double, ...
  - int factorial(int);  --- factorial is a function that takes an integer as an argument and returns an integer (type int -> int)
Types: Overview

- **Examples from C (cont’d):**
  - Structures types
    - `struct pair {int x; int y;};`  --- pair is a type
    - `pair p;`  --- variable `p` will store a pair
  - Array types
    - `int[n], char[n][m], ...`
    - `int arr[8];`  --- variable `arr` will store a sequence of 8 integers
  - Pointer types
    - `int*, int**, ...`
    - `int* p;`  --- variable `p` stores the address of a variable that stores an integer
    - `int** p;`  --- variable `p` stores the address of a variable that stores the address of a variable that stores an integer
Type-correct program

```c
int readint(); // reading: void -> int
void writeint(int); // writeint: int -> void

int factorial(int n){
    int f = 1; // f: int
    int c = 1; // c: int
    while (c <= n) {
        f = f * c; // *: (int,int) -> int
        c++; // ++: int -> int
    }
    return f; // f: int
}

void main() {
    int v; // v: int
    v = readint(); // readint(): int
    writeint(factorial(v)); // factorial(v): int
}
```

Types are specifications of program behavior
Type-correct program

```c
int readint(); // reading: void -> int
void writeint(int); // writeint: int -> void

int factorial(int n) { // factorial: int -> int
    int f = 1; // f: int
    int c = 1; // c: int
    while (c <= n) {
        f = f * c; // *: (int,int) -> int
        c++; // ++: int -> int
    }
    return f; // f: int
}

void main() { // main: void -> void
    string s; // s: string
    s = readint(); // readint(): int - Error
    writeint(factorial(s)); // factorial(s): No type
}
```

`s` stores strings, but `readint()` returns int.

`s` is a string, but `factorial()` expects int.
Why types (1)? Compilation

- Types are *necessary* to compile source code
  - What is the binary representation of variables?
    - `int x;` --- `x` is 4 bytes
    - `char x;` --- `x` is 1 byte
    - `int arr[8];` --- `arr` is 32 bytes
  - How to compute in assembly?
    - `int x; int y; x + y → add r1,r2`
    - `float x; float y; x + y → fadd r1,r2`
    - Difference is based on types

Types are used to compile code, but (usually) don’t exist in the compiler’s output
Why types (2)? Reject bad programs

- Type checking: A compiler can ensure that data and code are used only as specified by types without running the program
- Types can be used to reject at compile time programs that:
  - Use strings as integers
  - Use integers as pointers
  - Cause null-pointer exceptions
  - Cause array overflows
  - Leak secret information
  - …
Type-safety definitions

- A *program* is called *safe* if it is not stuck and has not crashed
- A *language* is called *type-safe* if well-typed programs always remain safe

- Presence of types does not imply type-safety
  - E.g., C has types but is not type-safe
Buffer overflow with scanf in C

```c
void readstring(char str[]) {
    scanf("%s", str);
}

void main() {
    char buf[8];
    readstring(buf);
    ...
}
```

This program is well-typed, but can crash

*C is not type-safe*

User could provide input longer than 7 bytes, causing a buffer overflow
void readstring(char str[]) {
    scanf("%s",str);
}

void main() {
    char buf[8]; readstring(buf);
    ...
}

length of array is unbounded (type char[])

buf has length 8 (type char[8])

char[] != char[8]
This program should be rejected! However, C allows unsafe type-casting and accepts the program
Unsafe type-casting in C

- Type-casting: convert the type of a variable to another
- The C compiler will convert types (e.g., char[8] to char[])
- Some conversions violate type specifications
- This makes C type-unsafe
## Safe and unsafe type casts in C

<table>
<thead>
<tr>
<th>Type-cast</th>
<th>Safe in C?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char * x; int * y = (int *) x</code></td>
<td>No</td>
</tr>
<tr>
<td><code>int * x; void * y = (void *) x</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>void * x; int * y = (int *) x</code></td>
<td>No</td>
</tr>
<tr>
<td><code>int x; char * y = (char *) x</code></td>
<td>No</td>
</tr>
<tr>
<td><code>char * x; int y = (int) x</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>int x; float y = (float) x</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>float x; int y = (int) x</code></td>
<td>Yes</td>
</tr>
</tbody>
</table>
Absent, weak, and strong typing

- **Untyped languages don't have types**
  - E.g., Bash, Perl, Ruby, …
  - Usually interpreted; difficult to compile without types
  - Program safety is programmer's responsibility: programs difficult to debug (really!)

- **Weakly typed languages use types only for compilation; no type-safety**
  - E.g., C, C++, etc.
  - Often allow unsafe casts, e.g., char[8] to char[] and int to char*
  - Program safety is programmer's responsibility: buffer overflows and segmentation faults are common in programs

- **Strongly typed languages use types for compilation and guarantee type-safety**
  - E.g., BASIC, Pascal, Cyclone, Haskell, SML, Java, etc.
  - No unsafe casts, e.g., an integer cannot be cast to a pointer, an array of length 8 is not an unbounded array, etc.
  - Safety is guaranteed but believed unsuitable for some low-level programs (debatable)
Summary of types

- Types are **specifications** of data and code
- Compiler may check well-typedness without executing the program
- Existence of type specifications may imply program safety (**type-safety**)
- Not all languages with types are type-safe
  - E.g., C is not type-safe

Rest of the lecture:
Take a simple type-safe language and understand type-safety more formally
Mini-C: A strongly typed language

- Mini-C is a small part of C

Types 
\[ T ::= \text{int} \mid \text{bool} \]

Variables 
\[ x, y \]

Declarations 
\[ \Delta ::= x_1 : T_1; \ldots; x_n : T_n \]

Integers 
\[ n ::= \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \]

Expressions 
\[ e ::= x \mid \text{true} \mid \text{false} \mid n \mid e_1 + e_2 \mid e_1 * e_2 \mid e_1 <= e_2 \mid e_1 == e_2 \]

Commands 
\[ c ::= \text{noop} \mid x = e \mid e_1; e_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c \]

Programs 
\[ P ::= \text{decl } \Delta \text{ begin } c \text{ end} \]

Values 
\[ v ::= \text{true} \mid \text{false} \mid n \]

Stores 
\[ \sigma ::= x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \]

No functions
No structures
No pointers
No casts
Mini-C factorial program

decl
  f : int; c : int; m : int
begin
  m = 10;   // Compute 10!
  f = 1;
  c = 1;
  while c <= m do
    f = f * c;
    c = c + 1
  // Output f here
end

\[\Delta (\text{Declarations})\]

\[\mathcal{C} (\text{Commands})\]

\[\mathcal{E} (\text{Expressions})\]
Type-safety through semantics

- Unsafe programs are those that cannot compute
- Semantics define safe program computation

- **Type-safety**: A correctly typed program is never stuck

- **Game-plan (next few slides)**:
  - Describe semantics for Mini-C
  - Describe typing for Mini-C
  - Prove that typing $\Rightarrow$ Not stuck (i.e., Type-safety)
Semantics of expressions

Expressions \( e \) ::\( x \mid \text{true} \mid \text{false} \mid n \mid e_1 + e_2 \mid e_1 \times e_2 \mid e_1 \leq e_2 \mid e_1 == e_2 \)

Values \( v \) ::\( \text{true} \mid \text{false} \mid n \)

- Expressions **compute** by evaluation to values

- Formalized by a **reduction relation**:

  \[
  \sigma \triangleright e \leftrightarrow e'
  \]

  - Reads: In memory \( \sigma \), \( e \) reduces to \( e' \)

Stores \( \sigma \) ::\( x_1 \leftrightarrow v_1, \ldots, x_n \leftrightarrow v_n \)
Semantics of expressions (cont’d)

- To evaluate a variable, read its value from memory

\[
\frac{(x \mapsto v) \in \sigma}{\sigma \triangleright x \mapsto v}
\]

- To evaluate \((e_1+e_2)\), first evaluate \(e_1\), then evaluate \(e_2\), and then add them

\[
\begin{align*}
\sigma \triangleright e_1 & \mapsto e_1' \\
\sigma \triangleright e_1 + e_2 & \mapsto e_1' + e_2
\end{align*}
\]

\[
\begin{align*}
\sigma \triangleright e_2 & \mapsto e_2' \\
\sigma \triangleright v_1 + e_2 & \mapsto v_1 + e_2'
\end{align*}
\]

\[
\frac{\text{add}(n_1, n_2) = n}{\sigma \triangleright n_1 + n_2 \mapsto n}
\]
Semantics of expressions (cont’d)

- Similar rules for other expressions

\[
\begin{align*}
\sigma \triangleright e_1 &\leftrightarrow e'_1 \\
\sigma \triangleright e_1 \ast e_2 &\leftrightarrow e'_1 \ast e_2
\end{align*}
\]

\[
\begin{align*}
\sigma \triangleright e_2 &\leftrightarrow e'_2 \\
\sigma \triangleright v_1 \ast e_2 &\leftrightarrow v_1 \ast e'_2
\end{align*}
\]

\[
\begin{align*}
\sigma \triangleright e_1 &\leftrightarrow e'_1 \\
\sigma \triangleright e_1 \leq e_2 &\leftrightarrow e'_1 \leq e_2
\end{align*}
\]

\[
\begin{align*}
\sigma \triangleright e_2 &\leftrightarrow e'_2 \\
\sigma \triangleright v_1 \leq e_2 &\leftrightarrow v_1 \leq e'_2
\end{align*}
\]

\[
\begin{align*}
\sigma \triangleright e_1 &\leftrightarrow e'_1 \\
\sigma \triangleright e_1 \equiv e_2 &\leftrightarrow e'_1 \equiv e_2
\end{align*}
\]

\[
\begin{align*}
\sigma \triangleright e_2 &\leftrightarrow e'_2 \\
\sigma \triangleright v_1 \equiv e_2 &\leftrightarrow v_1 \equiv e'_2
\end{align*}
\]

\[
\begin{align*}
\text{mult}(n_1, n_2) &= n \\
\sigma \triangleright n_1 \ast n_2 &\leftrightarrow n
\end{align*}
\]

\[
\begin{align*}
\text{leq}(n_1, n_2) &= b \\
\sigma \triangleright n_1 \leq n_2 &\leftrightarrow b
\end{align*}
\]

\[
\begin{align*}
\text{eq}(n_1, n_2) &= b \\
\sigma \triangleright n_1 \equiv n_2 &\leftrightarrow b
\end{align*}
\]

Important:
No rules to evaluate unsafe expressions like (true * 7)
Semantics of commands

Commands $c ::= \text{noop} \mid x = e \mid c_1; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c$

- Commands compute to noop, but change memory
- Formalized by a reduction relation

$$\sigma ; c \rightarrow \sigma' ; c'$$

- Reads: Starting from memory $\sigma$, command $c$ reduces to command $c'$ and updates memory to $\sigma'$
Semantics of commands (cont’d)

- To evaluate \((x = e)\), first evaluate expression \(e\) to a value \(v\). Then update memory by putting the value \(v\) in \(x\)

\[
\sigma \triangleright e \leftrightarrow^* v \\
\sigma ; (x = e) \longrightarrow \sigma [x \leftarrow v]; \text{noop}
\]

- To evaluate \((c_1; c_2)\), first evaluate \(c_1\), then evaluate \(c_2\)

\[
\begin{align*}
\sigma ; c_1 &\longrightarrow \sigma' ; c'_1 \\
\sigma ; (c_1 ; c_2) &\longrightarrow \sigma' ; (c'_1 ; c_2)
\end{align*}
\]

\[
\sigma ; (\text{noop}; c_2) \longrightarrow \sigma ; c_2
\]
Semantics of commands (cont’d)

- Similar rules for other commands

\[
\sigma \triangleright e \leftrightarrow^* \text{ true} \\
\sigma ; (\text{if } e \text{ then } c_1 \text{ else } c_2) \longrightarrow \sigma ; c_1
\]

\[
\sigma \triangleright e \leftrightarrow^* \text{ false} \\
\sigma ; (\text{if } e \text{ then } c_1 \text{ else } c_2) \longrightarrow \sigma ; c_2
\]

\[
\sigma \triangleright e \leftrightarrow^* \text{ true} \\
\sigma ; (\text{while } e \text{ do } c) \longrightarrow \sigma ; (c; (\text{while } e \text{ do } c))
\]

\[
\sigma \triangleright e \leftrightarrow^* \text{ false} \\
\sigma ; (\text{while } e \text{ do } c) \longrightarrow \sigma ; \text{noop}
\]

Important:
No rules to evaluate unsafe commands like (if 7 then c else c')
Summary of semantics

- Semantics are rules for evaluating programs safely
- Programs that cannot evaluate are unsafe by definition
- Two types of reductions, for expressions and commands

\[
\sigma \upRightarrow e \rightarrow e' \\
\sigma; c \rightarrow \sigma'; c'
\]
Type checking

Declarations $\Delta ::= x_1 : T_1; \ldots; x_n : T_n$

- Typing for expressions
  $\Delta \vdash e : T$
  - Reads: with respect to declarations $\Delta$, expression $e$ has type $T$

- Typing for commands
  $\Delta \vdash c$
  - Reads: with respect to declarations $\Delta$, command $c$ is well typed

Important:
A compiler can check efficiently whether a program is well typed
Typing for expressions

- To type check a variable, find its declaration
  \[
  (x : T) \in \Delta \\
  \Xi \vdash x : T
  \]

- Both true and false have type bool

  \[
  \Delta \vdash \text{true} : \text{bool} \quad \Delta \vdash \text{false} : \text{bool}
  \]

- A number has type int

  \[
  \Delta \vdash n : \text{int}
  \]
Typing for expressions (cont’d)

- \((e_1 + e_2)\) has type int if both \(e_1\) and \(e_2\) have type int

\[
\begin{align*}
\Delta \vdash e_1 : \text{int} \quad &\quad \Delta \vdash e_2 : \text{int} \\
\Delta \vdash e_1 + e_2 : \text{int}
\end{align*}
\]

- Other expressions are typed similarly

\[
\begin{align*}
\Delta \vdash e_1 : \text{int} \quad &\quad \Delta \vdash e_2 : \text{int} \\
\Delta \vdash e_1 \times e_2 : \text{int}
\end{align*}
\]

\[
\begin{align*}
\Delta \vdash e_1 : \text{int} \quad &\quad \Delta \vdash e_2 : \text{int} \\
\Delta \vdash e_1 \leq e_2 : \text{bool}
\end{align*}
\]

\[
\begin{align*}
\Delta \vdash e_1 : \text{int} \quad &\quad \Delta \vdash e_2 : \text{int} \\
\Delta \vdash e_1 = e_2 : \text{bool}
\end{align*}
\]
Typing for commands

\[ \Delta \vdash c \]

- \[ \Delta \vdash \text{noop} \]
  \[ \Delta \vdash (x : T) \in \Delta \quad \Delta \vdash e : T \]
  \[ \Delta \vdash x = e \]

- \[ \Delta \vdash c_1 \quad \Delta \vdash c_2 \]
  \[ \Delta \vdash c_1 ; c_2 \]

- \[ \Delta \vdash e : \text{bool} \quad \Delta \vdash c_1 \quad \Delta \vdash c_2 \]
  \[ \Delta \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 \]

- \[ \Delta \vdash e : \text{bool} \quad \Delta \vdash c \]
  \[ \Delta \vdash \text{while } e \text{ do } c \]
Typing: Example

Note: $\Delta = f : \text{int}; c : \text{int}; m : \text{int}$

\[
\Delta \vdash c : \text{int} \quad \Delta \vdash m : \text{int} \quad \Delta \vdash f = f \ast c \quad \Delta \vdash c = c \ast 1
\]

\[
\Delta \vdash c \leq m : \text{bool} \quad \Delta \vdash f = f \ast c; c = c + 1
\]

\[
\Delta \vdash \text{while} \ (c \leq m) \ \text{do} \ (f = f \ast c; c = c + 1)
\]
Typing for memory

- Typing for memory ensures that variables in memory hold values of expected type

\[ \Delta \vdash \sigma \]
Type-safety

- Suppose $c$ is a well-typed command
- Suppose $\sigma$ is a well-typed memory
- $\sigma; c \rightarrow^* \sigma '; c'$

- Then $c'$ is not stuck, i.e., it is noop, or it can reduce further
Type-safety (formally)

Theorem 6.2 (Safety for commands). Suppose the following hold:

1. \( \Delta \vdash c \)
2. \( \Delta \vdash \sigma \)
3. \( \sigma; c \rightarrow^* \sigma'; c' \)

Then either \( c' = \text{noop} \) or there are \( \sigma'' \) and \( c'' \) such that \( \sigma'; c' \rightarrow \sigma''; c'' \).

- Why is this theorem “type-safety”?
- Theorem says that a well-typed program will either terminate or keep reducing
- By definition, such a program is safe
- Therefore, a typed program always remains safe
Summary of types and type-safety

- Types specify expected program behavior
- Semantics define what safe behavior is
- Type-safety: Typing => always follow semantics => always safe behavior
- Reminder: Not all typed languages are type-safe
What else can types do?

- Prevent null-pointer dereferencing and dangling pointers (e.g., Cyclone)
- Enforce array bounds (e.g., DML)
- Prevent information leaks (e.g., Jif, Sif)
- Enforce access control (e.g., PCML5, Aura, PCAL)
- Ensure correctness of protocols (e.g., F7)
Language-based security

- **Objective**
  - Design security into the language
  - Compiler rules out insecure programs
  - Compiler does not run the program (no testing)

- **What’s insecure?**
  - Buffer overflows
  - Information-flow leaks
  - Violations of access rights
  - ... 

- **One approach is based on** *types*