Language-Based Security: Overview of Types

Deepak Garg

Foundations of Security and Privacy
October 27, 2009
Language-Based Security

- **Objective**
  - Compiler techniques to rule out insecure programs

- **What's insecure?**
  - Buffer overflows
  - Information flow leaks
  - Violation of access rights
  - ...

- **Our approach**
  - Use **Types**
  - Assumption: compiler does not want to run the program (no testing)
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 argument and returns an integer (type int -> int)
Types: Overview

- **Examples from C (Contd.):**
  - **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(); // readint: 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) // c <= n: boolean
    {
        f = f * c; // *: (int,int) -> int
        c++; // ++: int -> int
    }
    return f; // f: int
}

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

Types are specifications of program behavior
Type-Incorrect Program

int readint(); // readint: 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) // c <= n: boolean
    {
        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
}
Why Types?

- 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 don't exist in the compiler output (usually!)
Types Reject Bad Programs

- A compiler *may ensure that data and code are used only as specified* by types without running the program (type-checking)
- Types can be used to *reject* programs that:
  - Use strings as integers
  - Use integers as pointers
  - Have dangling pointers
  - Cause null-pointer exceptions
  - Cause array overflows
  - Allow buffer overflows, stack alteration
  - Leak secret information

Next week

Eliminate crashes and bugs; improve software reliability

Improve security
Type-Safety Definition

- A **program** is called **safe** if it is not stuck and has not crashed.
- A **language** is called **type-safe** if well-typed programs in it always remain safe.
- Presence of types does not imply type-safety.
  - E.g., C has types but is not type-safe.
- Common cause for lack of type-safety is **unsafe casts**.
  - `int x = 3; char * y = (char *) x;`
  - `y` has 3 in it, but 3 may not be the address of a char.
## Some Type-casts in C

<table>
<thead>
<tr>
<th>Type-cast</th>
<th>Safe in C?</th>
</tr>
</thead>
<tbody>
<tr>
<td>char * x; int * y = (int *) x</td>
<td>No</td>
</tr>
<tr>
<td>int * x; void * y = (void *) x</td>
<td>Yes</td>
</tr>
<tr>
<td>void * x; int * y = (int *) x</td>
<td>No</td>
</tr>
<tr>
<td>int x; char * y = (char *) x</td>
<td>No</td>
</tr>
<tr>
<td>char * x; int y = (int) x</td>
<td>Yes</td>
</tr>
<tr>
<td>int x; float y = (float) x</td>
<td>Yes</td>
</tr>
<tr>
<td>float x; int y = (int) x</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Buffer-Overflow with scanf in C

```c
void readstring(char str[8])
{
    scanf("%s", str);
}

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

User may provide input longer than 7 bytes, causing alteration of stack
void readstring(char str[8])
{
    scanf(“%s”, str);
}

void main()
{
    char buf[8];
    readstring(buf);
    ...
}
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 programs (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 formal meaning of “type-safety”
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
  
  \[ e ::= x \mid \text{true} \mid \text{false} \mid n \mid e_1 + e_2 \mid e_1 \times e_2 \mid e_1 \leq e_2 \mid e_1 = e_2 \]

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

  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
decl
  f : int; c : int; m : int
begin
  m = 10;   // Computer 10!
  f = 1;
  c = 1;
  while c <= m do
    f = f * c;
    c = c + 1
  // Output f here
end
Type-Safety Through Semantics

- Semantics define safe program computation
- Unsafe programs are those that cannot compute
- **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 => Not stuck (Type-safety)
Semantics of Expressions

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

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

• Expressions **compute** by evaluation to values

• Formalized by a **reduction relation**:

\[
\sigma \triangleright e \rightarrow e'
\]

In memory \( \sigma \), \( e \) reduces to \( e' \)

Stores \( \sigma \) ::= \( x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n \)
Semantics of Expressions (Contd.)

- To evaluate a variable, read its value from memory.

\[(x \mapsto v) \in \sigma\]
\[\sigma \triangleright x \mapsto v\]

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

\[
\begin{align*}
\sigma \triangleright e_1 & \mapsto e_1' \\
\sigma \triangleright e_1 + e_2 & \mapsto e_1' + e_2 \\
\sigma \triangleright v_1 + e_2 & \mapsto v_1 + e_2' \\
\text{add}(n_1, n_2) = n & \\
\sigma \triangleright n_1 + n_2 & \mapsto n
\end{align*}
\]
Semantics of Expressions (Contd.)

- Similar rules for other expressions

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

\[
\begin{align*}
\sigma \triangleright e_2 & \leftrightarrow e'_2 \\
\sigma \triangleright v_1 * e_2 & \leftrightarrow v_1 * e'_2 \\
\sigma \triangleright v_1 & \leftrightarrow v'_1 \\
\sigma \triangleright v_1 <= e_2 & \leftrightarrow v_1 <= e'_2 \\
\sigma \triangleright v_1 & \leftrightarrow v'_1 \\
\sigma \triangleright v_1 == e_2 & \leftrightarrow v_1 == e'_2
\end{align*}
\]

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

**Important**

No rules to evaluate unsafe expressions like \((\text{true} * 7)\)
Semantics of Commands

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

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

\[
\sigma ; c \rightarrow \sigma' ; c'
\]

Starting from memory $\sigma$, command $c$ reduces to command $c'$ and updates memory to $\sigma'$
Semantics of Commands (Contd.)

- 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 \(c_2\).

\[
\sigma ; c_1 \longrightarrow \sigma' ; c'_1 \\
\sigma ; (c_1 ; c_2) \longrightarrow \sigma' ; (c'_1 ; c_2) \\
\sigma ; (\text{noop} ; c_2) \longrightarrow \sigma ; c_2
\]
Semantics of Commands (Contd.)

- Similar rules for other commands

\[
\begin{align*}
\sigma \triangleright e &\rightsquigarrow^* \text{true} \\
\sigma ; (\text{if } e \text{ then } c_1 \text{ else } c_2) &\longrightarrow \sigma ; c_1 \\
\sigma \triangleright e &\rightsquigarrow^* \text{false} \\
\sigma ; (\text{if } e \text{ then } c_1 \text{ else } c_2) &\longrightarrow \sigma ; c_2 \\
\sigma \triangleright e &\rightsquigarrow^* \text{true} \\
\sigma ; (\text{while } e \text{ do } c) &\longrightarrow \sigma ; (c; (\text{while } e \text{ do } c)) \\
\sigma \triangleright e &\rightsquigarrow^* \text{false} \\
\sigma ; (\text{while } e \text{ do } c) &\longrightarrow \sigma ; \text{noop}
\end{align*}
\]

**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

\[
\begin{align*}
\sigma \triangleright e & \rightarrow e' \\
\sigma; c & \rightarrow \sigma'; c'
\end{align*}
\]
Type Checking

Declarations \( \Delta ::= x_1 : T_1; \ldots; x_n : T_n \)

- Typing for expressions

\[ \Delta \vdash e : T \]

- Typing for commands

\[ \Delta \vdash c \]

Important
A compiler can check efficiently whether a program is well typed or not.
Typing for Expressions

- To type check a variable, find its declaration

\[
(x : T) \in \Delta \\
\frac{}{\Delta \vdash x : T}
\]

- 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 (Contd.)

• \((e_1+e_2)\) has type \(\text{int}\) if both \(e_1\) and \(e_2\) have type \(\text{int}\)

\[
\Delta \vdash e_1 : \text{int} \quad \Delta \vdash e_2 : \text{int} \\
\Delta \vdash e_1 + e_2 : \text{int}
\]

• Other expressions are typed similarly

\[
\Delta \vdash e_1 : \text{int} \quad \Delta \vdash e_2 : \text{int} \\
\Delta \vdash e_1 * e_2 : \text{int}
\]

\[
\Delta \vdash e_1 : \text{int} \quad \Delta \vdash e_2 : \text{int} \\
\Delta \vdash e_1 <= e_2 : \text{bool}
\]

\[
\Delta \vdash e_1 : \text{int} \quad \Delta \vdash e_2 : \text{int} \\
\Delta \vdash e_1 == e_2 : \text{bool}
\]
Typing for Commands

\[ \Delta \vdash c \]

\[ \begin{array}{c}
\Delta \vdash \text{noop} \\
\hline
(x : T) \in \Delta \quad \Delta \vdash e : T \\
\Delta \vdash x = e \\
\end{array} \]

\[ \begin{array}{c}
\Delta \vdash c_1 \quad \Delta \vdash c_2 \\
\Delta \vdash c_1 ; c_2 \\
\end{array} \]

\[ \begin{array}{c}
\Delta \vdash e : \text{bool} \quad \Delta \vdash c_1 \quad \Delta \vdash c_2 \\
\Delta \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 \\
\end{array} \]

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

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

\[
\begin{align*}
D_1 &= \\
\frac{(f : \text{int}) \in \Delta}{\Delta \vdash f : \text{int}} & \frac{(c : \text{int}) \in \Delta}{\Delta \vdash c : \text{int}} \\
\frac{(f : \text{int}) \in \Delta}{\Delta \vdash f * c : \text{int}} & \\
\Delta \vdash f = f * c
\end{align*}
\]

\[
\begin{align*}
D_2 &= \\
\frac{(c : \text{int}) \in \Delta}{\Delta \vdash c : \text{int}} & \frac{\Delta \vdash 1 : \text{int}}{\Delta \vdash c + 1 : \text{int}} \\
\Delta \vdash c = c + 1
\end{align*}
\]

\[
\begin{align*}
\frac{(c : \text{int}) \in \Delta}{\Delta \vdash c : \text{int}} & \frac{(m : \text{int}) \in \Delta}{\Delta \vdash m : \text{int}} & \frac{D_1}{\Delta \vdash f = f * c} & \frac{D_2}{\Delta \vdash c = c * 1} \\
\Delta \vdash c <= m : \text{bool} & \\
\Delta \vdash f = f * c; c = c + 1
\end{align*}
\]

\[
\begin{align*}
D &= \\
\frac{D_1}{\Delta \vdash f = f * c} & \frac{D_2}{\Delta \vdash c = c + 1} \\
\Delta \vdash \text{while } (c <= m) \text{ do } (f = f * c; c = c + 1)
\end{align*}
\]
Typing for Memory

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

\[ \Delta \vdash \sigma \]

if and only if

\[(x : \text{int}) \in \Delta \text{ implies } (x \mapsto n) \in \sigma \text{ for some } n\]

\[(x : \text{bool}) \in \Delta \text{ implies } (x \mapsto \text{true}) \in \sigma \text{ or } (x \mapsto \text{false}) \in \sigma\]
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 leaks of secrets (e.g., Jif, Sif)
- Enforce access control (e.g., PCML₅, Aura, PCAL)
- Ensure correctness of protocols (e.g., F7)
Thank You!