Language-Based Security:

Security Types in Practice

Willard Rafnsson
Carnegie-Mellon University

Course: 18-732
Spring 2015
Topic Today: Security Types in Practice

0. Attacks
1. Software Security Architectures
2. Security Analysis of Software
3. Language-Based Security
4. Run-time Security Enforcement
Topic Today: Security Types in Practice

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

Language-based Security: Information Flow Control

Lujo Bauer
18-732
Spring 2015

(Slides with input from Anupam Datta)

Software Analysis in Practice

Willard Rafnsson
Carnegie-Mellon University
Course: 18-732
Spring 2015
Information-flow Security

policy: \( H \) input may not interfere with \( L \) output. (noninterference)

defines the attacker we guarantee security against
Information-flow Control

_policy:_ H input may not interfere with L output.

_enforcement:_ program analysis (static),
Information-flow Control

policy: H input may not interfere with L output.

enforcement: program analysis (static), program transformation (static / dynamic)
Information-flow Control

Policy: H input may not interfere with L output.

Enforcement: program analysis (static), program transformation (static / dynamic)

Type system!
Flow Examples

explicit flow:

\[ x := a; \]

secure?
depends on security levels
due to the variables are labeled with.
Flow Examples

explicit flow:

\[ x := a; \]
Flow Examples

explicit flow:

\[ x := a; \]

if

\[ a : H \]

\[ x : H \]

then

\[ |- p. \]
Flow Examples

explicit flow:

\[ x := a; \]

if
\[ a : L \]
\[ x : L \]
then
\[ |- p. \]
Flow Examples

explicit flow:

\[ x := a; \]

if

\[ a : L \]
\[ x : H \]

then

\[ |- p \]

taking something public and classifying it

levels of confidentiality

H

L
Flow Examples

explicit flow:

\[ x := a; \]

if

\[ a : H \]
\[ x : L \]

then

p leaks information!

taking something secret and publishing it
Flow Examples

explicit flow:
\[ x := a; \]

implicit flow:
\[
\text{if } a \{ \\
\quad x := 1; \\
\} \text{ else } \{ \\
\quad x := 0; \\
\} 
\]

levels of confidentiality
Flow Examples

explicit flow:
\[ x := a; \]

implicit flow:
\[
\text{if } a \{ \\
  x := 1; \\
\} \text{ else } \{ \\
  x := 0; \\
\} \]

if \[ a : H \]
then
\[ x : L \]

\[ p \text{ leaks information!} \]
and we have tools with which to
statically check the security of programs.
all is well...
... except darkness lurks in these lands.
Slightly Larger Example

if  |a-b| <= a/2 + 7  {
    x := 1;
} else {
    x := 0;
}

y := c >= a & c >= b;

https://xkcd.com/314/
Slightly Larger Example

if \(|a-b| \leq a/2 + 7\) {
    x := 1;
} else {
    x := 0;
}

y := c \geq a \& c \geq b;

x intended for A, y intended for C.
how to proceed?

actors
(principals, users)
if \(|a-b| \leq a/2 + 7\) {
  x := 1;
} else {
  x := 0;
}
y := \(c \geq a \& c \geq b\);

x intended for A,
y intended for C.
“push down” level of x.
Slightly Larger Example

\[
\text{if } \left| a - b \right| \leq a/2 + 7 \{ \\
x : = 1; \\
\} \text{ else } \{ \\
x : = 0; \\
\}
\]

\[
y : = c \geq a \& c \geq b;
\]

spot the leak? 

\[x \text{ intended for } A, \]
\[y \text{ intended for } C.\]
Slightly Larger Example

\[
\text{if } |a-b| \leq a/2 + 7 \ {\{ \\
\quad x := 1; \\
\} \text{ else } \ {\{ \\
\quad x := 0; \\
\} \\
\text{y := c} \geq a \ & c \geq b;}
\]

x intended for A, 
y intended for C.
Slightly Larger Example

if $|a-b| \leq a/2 + 7$ {
    $x := 1$;
} else {
    $x := 0$;
}

$y := c \geq a \& c \geq b$;

OK

x intended for A, y intended for C. “push down” level of y.
Slightly Larger Example

\[
\text{if } |a - b| \leq \frac{a}{2} + 7 \{ \\
\quad x := 1;
\} \text{ else } \{ \\
\quad x := 0;
\}
\]

\[
y := c \geq a \& c \geq b;
\]

spot the leak?

x intended for A, y intended for C.
Slightly Larger Example

\[
\text{if } |a-b| \leq a/2 + 7 \{ \\
\text{x := 1; } \\
\text{else } \{ \\
\text{x := 0; } \\
\text{y := } c \geq a \text{ } \& \text{ } c \geq b; \\
\}
\]

x intended for A, y intended for C.
Slightly Larger Example

```plaintext
if |a-b| <= a/2 + 7 {
    x := 1;
} else {
    x := 0;
}
y := c >= a & c >= b;

OK
```
Slightly Larger Example

```plaintext
if \( |a-b| \leq a/2 + 7 \) {
    x := 1;
} else {
    x := 0;
}
y := c \geq a \& c \geq b;
```

every program can be securely typed. but we may not like those typings. we only want to reveal whether certain birthdays are within certain intervals...

but a program, well-typed with this type signature, can leak all of b, anywhere!!! (the types alone thus don’t reassure us much…)
0-bit Leakage Policy Is Rarely Reasonable

type system too coarse/restrictive (all/nothing)

some programs leak by design.

- login interface (success/failure of attempt)
- average salary of workers
- online shopping (ratings, suggested items)

some programs have a dynamic security policy

- social network (view/post permissions)
- (un)shared files, ...
0-bit Leakage Policy Is Rarely Reasonable

type system too coarse/restrictive (all/nothing)

some programs leak by design.

- login interface (success/failure of attempt)
- average salary of workers
- online shopping (ratings, suggested items)
- social network (view/post permissions)
- (un)shared files, ...

How do you secure such programs?

How to specify a policy that changes, to e.g.

- declare leakage intent, or
- add/remove users?

How is such a policy enforced statically?

some programs have a dynamic security policy
Paragon

Paragon
Paragon
Paragon is

● “Jif w/ more expressive policies”. That is,
  ○ Java, with a rich
  ○ policy specification language, and
  ○ type system statically enforcing
    the specified policy.

Get it here:

● compiler
  https://code.google.com/p/paragon-java/downloads/list
● tutorial
  http://paragon.nowplea.se/
  http://teachmeparagon.nowplea.se/

I just spoke with Niklas Broberg (lead developer); he said they’re rolling out a new release. Feel free to approach him w/ Paragon Qs.
let’s jump right into Paragon.
import static HighLow.*; // Importing defs of high and low
public class MyClass {
    high boolean mySecret;
    low boolean myPublic;
    public void m() {
        mySecret = myPublic;
    }
}
Confidentiality

```java
import static HighLow.*; // Importing defs of high and low
public class MyClass {
    ?high boolean mySecret;
    ?low boolean myPublic;
    public void m() {
        mySecret = myPublic;
    }
}
```

reading this yields value of confidentiality level high.
writing to it has no other side-effects.
(should really be ?! high, or just high)

each method has a:

? : read effect; [default: “L”]
(highest) confidentiality level of value returned
!
write effect; [default: “H”]
(lowest) confidentiality level written to as a side-effect during invocation.
treat field read/write as get/set method call.
import static HighLow.*; // Importing defs of high and low
public class MyClass {
    high boolean mySecret;
    low boolean myPublic;
    public void m() {
        mySecret = myPublic;
    }
}

reading this
yields value of confidentiality
level high.
writing to it has no
other side-effects.

(should really be
!high, or just high)

each method has a:
? : read effect; [default: “L”]
(highest) confidentiality level
of value returned
!
write effect; [default: “H”]
(lowest) confidentiality level
written to as a side-effect during invocation.
treat field read/write as get/set method call.
import static HighLow.*; // Importing defs of high and low

public class MyClass {
  ?high boolean mySecret;
  ?low  boolean myPublic;

  public void m() {
    myPublic = false;
  }
}

Why is this program rejected?

NOT OK

levels of confidentiality

H

L

each method has a:

? : read effect; [default: “L”]
  (highest) confidentiality level of value returned

! : write effect; [default: “H”]
  (lowest) confidentiality level written to as a side-effect during invocation.
treat field read/write as get/set method call.
import static HighLow.*; // Importing defs of high and low
public class MyClass {
    ?high boolean mySecret;
    ?low  boolean myPublic;
    public void m() {
        myPublic = false;
    }
}

write effect mismatch.

This is the technical reason; why needed? (hint: pc)

NOT OK

each method has a:
? : read effect; [default: “L”]
    (highest) confidentiality level
    of value returned
! : write effect; [default: “H”]
    (lowest) confidentiality level
    written to as a side-effect during invocation.
treat field read/write as get/set method call.
import static HighLow.*; // Importing defs of high and low
public class MyClass {
    ?high boolean mySecret;
    ?low    boolean myPublic;
    public void m() {
        myPublic = false;
    }
}

```
Confidentiality

import static HighLow.*; // Importing defs of high and low
public class MyClass {
    ?high boolean mySecret;
    ?low    boolean myPublic;
    public void m() {
        myPublic = false;
    }
}

```

What is the fix?

NOT OK

levels of confidentiality

each method has a:
? : read effect;    [default: “L”]
    (highest) confidentiality level of value returned
! : write effect;   [default: “H”]
    (lowest) confidentiality level written to as a side-effect during invocation.
treat field read/write as get/set method call.

write effect mismatch.
calling this method in a high branch leaks otherwise
Confidentiality

import static HighLow.*; // Importing defs of high and low
public class MyClass {
    ?high boolean mySecret;
    ?low boolean myPublic;
    !low public void m() {
        myPublic = false;
    }
}
Confidentiality

import static HighLow.*; // Importing defs of high and low
public class MyClass {
    ?high boolean mySecret;
    ?low boolean myPublic;
    !low public void m() {
        myPublic = mySecret;
    }
}

levels of confidentiality

explicit flow

each method has a:

? : read effect; [default: “L”]
(highest) confidentiality level
of value returned

! : write effect; [default: “H”]
(lowest) confidentiality level
written to as a side-effect during invocation.
treat field read/write as get/set method call.

NOT OK
import static HighLow.*; // Importing defs of high and low
public class MyClass {
    ?high boolean mySecret;
    ?low boolean myPublic;
    !low public void m() {
        if (mySecret) {
            myPublic = true;
        } else {
            myPublic = false;
        }
    }
}

NOT OK

each method has a:
? : read effect; [default: “L”]
    (highest) confidentiality level
    of value returned
! : write effect; [default: “H”]
    (lowest) confidentiality level
    written to as a side-effect during invocation.
    treat field read/write as get/set method call.
import tutorial.HighLow.*;
import tutorial.UntrustedTrusted.*;

public class MyClass {
    String myDiary;
    String myPassword;

    public !(high + untrusted) void savePassword(){
        myDiary += myPassword;
    }
}

each method has a: ? : read effect; ![default: “L”];
    (highest) confidentiality level of value returned
! : write effect; ![default: “H”];
    (lowest) confidentiality level written to as a side-effect during invocation.
treat field read/write as get/set method call.

greatest lower bound; easily build new policies from old ones.

levels of integrity
U
T
import tutorial.HighLow.*;
import tutorial.UntrustedTrusted.*;

public class MyClass {
    String myDiary;
    String myPassword;

    public ! high + untrusted void savePassword()
    {
        myDiary += myPassword;
    }
}

each method has a:
? : read effect; [default: \textit{L}]  
(highest) confidentiality level of value returned
!
: write effect; [default: \textit{H}]  
(lowest) confidentiality level written to as a side-effect during invocation.
treat field read/write as get/set method call.

OK
import tutorial.HighLow.*;
import tutorial.UntrustedTrusted.*;

public class MyClass {
    String myDiary;
    String myPassword;
    String fire;

    public !(high + trusted) void savePassword() {
        myDiary += myPassword;
    }

    public !(high + trusted) void burnDiary() {
        myDiary = fire;
    }

    NOT OK

    policy prevents diary from being burned.
Static Policies

```java
public class HighLow {
    private static final Object lowObserver;
    private static final Object highObserver;

    public static final policy high = { highObserver : }
    public static final policy low  = { lowObserver :
        ; highObserver : }
}
```

We can define our own policy from scratch.

**actor**: just an object, representing an observer of information. 

*a basic building block* of policies.

high can be observed by highObserver, under **all** circumstances.
Dynamic Policies

```java
public class KeySeller {
    public static lock Paid;
    public final Object customer = new Object();
    ?{customer: } String customerData;
    {?{customer: Paid} String softwareKey;

    public void processPayment() {
        // customer pays for item
        if (paymentSuccessful) { open Paid; } else {...}
    }

    public !{customer: Paid} void buyKey() {
        processPayment();
        if (Paid) { customerData = softwareKey; } else {...}
    }
}
```

flow lock: just a boolean. a basic building block of dynamics in policies, defining policy state.

customer can read softwareKey when Paid is open.

runtime testing of a lock!
Here programmer declares *leakage intent* (here in a manner similar to an *escape hatch*), thus *declassifying* the key!
Dynamic Policies

Paragon ensures the only leaks that occur are those declared intentional by the programmer!
Dynamic Policies

Paragon ensures the only leaks that occur are those declared intentional by the programmer!
Dynamic Policies

Paragon ensures the only leaks that occur are those declared intentional by the programmer!

“With great power comes great responsibility.”

- understand the implications of a leak
  it’s tempting to declassify everything to pass analysis. but then there’s no security guarantee...
- can be hard when policy is dynamic
- even harder when information can leak through the current state of the policy.

Paragon tracks this!
Dynamic Policies, advanced

\[
policy \text{ workData} = \\
\begin{cases}
\text{Manager } m : \\
(\text{Manager } m) \text{ Employee } e : \text{GivesPermissions}(m, e) \\
(\text{Manager } m) \text{ Employee } e : \text{IsBoss}(m), \text{WorksFor}(e, m)
\end{cases}
\]
Dynamic Policies, advanced

{ elton : ; Employee e : ReadsFor(e, elton) }

want to support runtime changing permissions.
Dynamic Policies, advanced

want to support runtime changing permissions.

```java
void allowReading(Employee a, Employee b) {
    open ReadsFor(a, b);
    for (Employee e : employees) {
        if (ReadsFor(b, e)) open ReadsFor(a, e);
        if (ReadsFor(e, a)) open ReadsFor(e, b);
    }
}

void disallowReading(Employee a, Employee b) {
    close ReadsFor(a, b);
    for (Employee e : employees) {
        if (ReadsFor(b, e)) close ReadsFor(a, e);
    }
}
```
Dynamic Policies, advanced

want to support runtime changing permissions.

```java
void allowReading(Employee a, Employee b) {
    open ReadsFor(a, b);
    for (Employee e : employees) {
        if (ReadsFor(b, e))
            open ReadsFor(a, e);
        if (ReadsFor(e, a))
            open ReadsFor(e, b);
    }
}
void disallowReading(Employee a, Employee b) {
    close ReadsFor(a, b);
    for (Employee e : employees) {
        if (ReadsFor(b, e))
            close ReadsFor(a, e);
    }
}
```
Dynamic Policies, advanced

\{ \text{elton} : ; \text{Employee } e : \text{ReadsFor}(e, \text{elton}) \}\;

want to support runtime changing permissions.

\textbf{lock} \text{ReadsFor(Employee, Employee)}

\{ (\text{Employee } a) \text{ReadsFor}(a,a) : \\
\quad ; (\text{Employee } a, b, c) \text{ReadsFor}(a,c) : \text{ReadsFor}(a,b), \text{ReadsFor}(b,c) \} ;

now, instead of \text{allowReading} and \text{disallowReading},
we just open / close \text{ReadsFor}, and
transitive permissions update \textit{automatically}!
Tip of The Iceberg

- sane default labels
  - low read effect, high write effect
  - label polymorphism in method parameters!
- exception handling
- encapsulation: hide information-flow policy behind an interface; detect when invoking code tries to violate the policy through the interface.
- Java-Pargon bridge; .pi file summarizing information flows in a Java file
  - used to port Java standard library to Paragon!

E.g. access control in Android
Compared to Jif

Jif is a “niche” language

- models one scenario well: distributed decentralized data ownership
- can’t model other scenarios well: roles, relationships, temporal aspects, “trusted declassifiers”, ...

Integrity model not unified with confidentiality model; complicated.

Jif features such as
- Decentralized Label Model
- Robust Declassification
  can be expressed easily in Paragon
Paragon Shortcomings

No

- concurrency (yet)
- dynamic loading

Language is still maturing

- bugs
Secure Software, Perspective

Things to consider when picking the right tool:

● **platform**
  3rd party JS in browser vs. your own Paragon

● **security concerns**
  integrity needed? secure through transformation?

● **threat**
  what attacks are we considering?

● **trust**
  did we create the code? the policy? the enforcement? the OS and hardware?
Secure Software, Perspective

Things to consider when picking the right tool:

- **platform**
  - 3rd party JS in browser vs. your own Paragon

- **security concerns**
  - integrity needed? secure through transformation?

- **threat**
  - what attacks are we considering?

- **trust**
  - did we create the code? the policy? the enforcement? the OS and hardware?

We can be “forceful” when handling 3rd party non-safety-critical code.

- JSFlow
- FlowFox (Secure Multi-Execution)

We must trust the tools we use.

- Paragon
- Jif
- LIO

All tools make assumptions on the platform; some attacks fall out of scope, e.g. targeting lower levels of the platform.

- next time: robust low-level code
Acknowledgements

Paragon material in this slides is based on

- *Programming in Paragon*
  tutorial, by Bart van Delft, Niklas Broberg, David Sands
- *Paragon: Information-flow Control in Practice*
  slides, by Niklas Broberg

Images displayed constitute fair use.