A Capability-Based Module System for Authority Control

Darya Melicher
Yangqingwei Shi
Alex Potanin
Jonathan Aldrich

Carnegie Mellon University
Carnegie Mellon University
Victoria University of Wellington
Carnegie Mellon University
Modules aren’t created equal

Platform Libraries | Application | Extensions

Most trusted
- Provide key functionality
- Often examined by security experts
- Hard to compromise

Least trusted
- Provide auxiliary functionality
- Written by third parties
- May be exploited

© Darya Melicher 2017
Security goal: Ensure the principle of least authority

*Platform Libraries* | *Application* | *Extensions*
---|---|---
Most trusted | | Least trusted

*May have the most authority* | *Must have the least authority*

**Principle of least authority:**
Each module must have authority to access only the resources it needs for its operation and nothing else
Security goal: Ensure the principle of least authority

**Platform Libraries**
- FFI
- fileIO

**Word Processor App**
- logger

**Extensions**
- prettyChart

---

**Most trusted**
- May have the most authority

**Least trusted**
- Must have the least authority

**Principle of least authority:**
Each module must have authority to access only the resources it needs for its operation and nothing else.
Our approach

• **Goal**: facilitate the principle of least authority
• Implemented in Wyvern programming language
• Pure OO with first-class modules
First-class modules \(\approx\) dynamically loaded modules

The prettyChart extension module may be loaded, linked, and configured at runtime
Our approach

• **Goal**: facilitate the principle of least authority
• Implemented in Wyvern programming language
• Pure OO with first-class modules
  – modules = objects (*cf.* Newspeak [Bracha10])
Facilitating the principle of least authority

Basics

• Capability safety
Wyvern is capability-safe

To access an object (module), another object (module) must have an appropriate capability
Facilitating the principle of least authority

Basics

• Capability safety
  – No global state

• Statically typed capabilities
Capabilities are statically typed

All capabilities required by modules are known at compile time
Capabilities are statically typed

```
module def logger(fio : FileIO) : Logger
  var filePath = "~/log.txt"
  def writeToLog(s : String)
    fio.write(filePath, s)

module def prettyChart(log : Logger) : PrettyChart
  def drawPrettyChart(data : String) = ...

require fileIO as fio
val log = logger(fio)
val charts = prettyChart(log)
...```

© Darya Melicher 2017
Capabilities are statically typed

```scala
module def logger(fio : FileIO) : Logger
  var filePath = "~/log.txt"
  def writeToLog(s : String)
    fio.write(filePath, s)

module def prettyChart(log : Logger) : PrettyChart
  def drawPrettyChart(data : String) = ...

require fileIO as fio
val log = logger(fio)
val charts = prettyChart(log)
...
```
Facilitating the principle of least authority

**Basics**

- Capability safety
  - No global state
- Statically typed capabilities
  - Limited reflection
  - No downcasts

**Main properties**

- Non-transitive authority
Authority is non-transitive

prettyChart has authority over logger, but it must not have authority over fileIO

prettyChart’s authority is defined more precisely

logger provides an attenuated authority [Miller06] over fileIO
logger attenuates fileIO capability

```
module def logger(fio : FileIO) : Logger
  var filePath = "~/log.txt"
  def writeToLog(s : String)
    fio.write(filePath, s)

module def prettyChart(log : Logger) : PrettyChart
  def drawPrettyChart(data : String) = ...

require fileIO as fio
val log = logger(fio)
val charts = prettyChart(log)
...```
logger attenuates fileIO capability

```python
module def logger(fio : FileIO) : Logger
    var filePath = "~/log.txt"
    def writeToLog(s : String)
        fio.write(filePath, s)

module def prettyChart(log : Logger) : PrettyChart
    def drawPrettyChart(data : String) = ...

require fileIO as fio
val log = logger(fio)
val charts = prettyChart(log)
...
```

prettyChart doesn’t import fileIO
logger attenuates fileIO capability

```scala
module def logger(fio : FileIO) : Logger
  var filePath = "~/log.txt"
  def writeToLog(s : String)
    fio.write(filePath, s)

module def prettyChart(log : Logger) : PrettyChart
  def drawPrettyChart(data : String) = ...

require fileIO as fio
val log = logger(fio)
val charts = prettyChart(log)
...
```

- logger calls only write() on fileIO
- prettyChart doesn't import fileIO
logger attenuates fileIO capability

module def logger(fio : FileIO) : Logger
    var filePath = "~/log.txt"
    def writeToLog(s : String)
        fio.write(filePath, s)

module def prettyChart(log : Logger) : PrettyChart
    def drawPrettyChart(data : String) = ...

require fileIO as fio
val log = logger(fio)
val charts = prettyChart(log)
...

logger writes only to a particular file
logger calls only write() on fileIO
prettyChart doesn’t import fileIO
Authority is non-transitive

prettyChart has authority over logger, but it must not have authority over fileIO

prettyChart’s authority is defined more precisely

logger provides an attenuated authority [Miller06] over fileIO
Facilitating the principle of least authority

Basics

• Capability safety
  – No global state

• Statically typed capabilities
  – Limited reflection
  – No downcasts

Main properties

• Non-transitive authority
• Simplified authority analysis
Authority analysis is simplified

A security expert can deduce what authority each module has by looking only at modules’ interfaces, *not their code*.
Authority analysis is simplified

To verify that `prettyChart` has the least authority, a security expert needs to look at:

1. `prettyChart`’s imports (logger)
2. `prettyChart`’s methods’ arguments
3. `logger`’s methods’ returns
Authority analysis is simplified

```scala
module def logger(fio: FileIO) : Logger
  var filePath = "~/log.txt"
  def writeToLog(s: String)
    fio.write(filePath, s)

module def prettyChart(log: Logger) : PrettyChart
  def drawPrettyChart(data: String) = ...

require fileIO as fio
val log = logger(fio)
val charts = prettyChart(log)
...```

© Darya Melicher 2017
Authority analysis is simplified

```
module def logger(fio : FileIO) : Logger
  var filePath = "~/log.txt"
  def writeToLog(s : String)
    fio.write(filePath, s)

module def prettyChart(log : Logger) : PrettyChart
  def drawPrettyChart(data : String) = ...

require fileIO as fio
val log = logger(fio)
val charts = prettyChart(log)
```
Authority analysis is simplified

```python
module def logger(fio : FileIO) : Logger
    var filePath = "~/log.txt"
    def writeToLog(s : String)
        fio.write(filePath, s)

module def prettyChart(log : Logger) : PrettyChart
    def drawPrettyChart(data : String) = ...

require fileIO as fio
val log = logger(fio)
val charts = prettyChart(log)
```

1. prettyChart’s imports
2. prettyChart’s methods’ arguments
Authority analysis is simplified

1. prettyChart’s imports

2. prettyChart’s methods’ arguments

3. logger’s methods’ returns

module def logger(fio : FileIO) : Logger
  var filePath = "~/log.txt"
  def writeToLog(s : String)//: Unit
    fio.write(filePath, s)

module def prettyChart(log : Logger) : PrettyChart
  def drawPrettyChart(data : String) = ...

require fileIO as fio
val log = logger(fio)
val charts = prettyChart(log)
...
Authority analysis is simplified

To verify that prettyChart has the least authority, a security expert needs to look at:

1. prettyChart’s imports (logger)
2. prettyChart’s methods’ arguments
3. logger’s methods’ returns

None of these involve fileIO0, and so prettyChart is guaranteed to have no authority over fileIO.
Authority analysis is simplified

A security expert can deduce what authority each module has by looking only at modules’ interfaces, *not their code*
Under the hood, authority analysis is facilitated by an Authority Safety Theorem

To verify that prettyChart has the *least* authority, a security expert needs to look at:

1. prettyChart’s imports (logger)
2. prettyChart’s methods’ arguments
3. logger’s methods’ returns

**Authority Safety Theorem:**

If prettyChart doesn’t import fileIO, it may get access to fileIO only if it:

1. Creates a new fileIO
2. Receives fileIO as a method argument
3. Calls a method that returned fileIO
Facilitating the principle of least authority

Basics

• Capability safety
  – No global state

• Statically typed capabilities
  – Limited reflection
  – No downcasts

Main properties

• Non-transitive authority
• Simplified authority analysis
Security goal: Ensure the principle of least authority

Platform Libraries | Application | Extensions

Most trusted | Least trusted

May have the most authority | Must have the least authority

Principle of least authority:
Each module must have authority to access only the resources it needs for its operation and nothing else

Facilitating the principle of least authority

Basics
- Capability safety
  - No global state
- Statically typed capabilities
  - Limited reflection
  - No downcasts

Main properties
- Non-transitive authority
- Simplified authority analysis

Authority is non-transitive

Platform Libraries | Word Processor App | Extensions

java | fileIO | logger

prettyChart has authority over logger, but it must not have authority over fileIO
prettyChart’s authority is defined more precisely
logger provides an attenuated authority [Miller06] over fileIO

Authority analysis is simplified

Platform Libraries | Word Processor App | Extensions

java | fileIO | logger

PrettyChart

A security expert can deduce what authority each module has by looking only at modules’ interfaces, not their code

© Darya Melicher 2017
Extra slides
Pure modules don’t bear capabilities

fileIO uses a system resource and is authority-bearing module, aka resource module

Whereas listFactory is NOT authority-bearing module, aka pure module