# Wyvern: Improving Architecture-Based Security via a Programming Language

Alex Potanin

# Software Security is a Big Problem

18 OCT 2017 NEWS

## Report: 88% of Java Apps Vulnerable to Attacks f… Known Security Defects

Michael Hill Acting Editor , Infosecurity Magazine

Email Michael Follow @MichaelInfosec

A new report from CA Veracode has exposed the pervasive risks companies face from vulnerable open source components.

Why Not Wa…

Home > Enterprise Java > Software Development

NEWS

## New bug neutralizes latest Java security updates

### New security vulnerability bypasses the Java plug-in's protection

By Gregg Keizer

Computerworld | JANUARY 28, 2013 12:54 PM PT

Java's new security settings, designed to block drive-by browser attacks, can be bypassed by hackers, a researcher announced Sunday.

The news came in the aftermath of several embarrassing zero-day vulnerabilities, and a recent commitment by the head of Java security that his team would fix bugs in the software.

**More about Java security**

- Why it's time to deprecate the Java Plug-in
- After silence on Java flaws, Oracle now says it cares
- InfoWorld's Security Adviser

The Java security provisions that can be circumvented were introduced last December with Java 7 Update 10 and let users decide which Java applets are

MORE LIKE THIS

What the latest Java flaw really me…

Two-year-old Java flaw emerges due to broken patch

Researchers find serious flaw in lat… JRE for desktops, servers

BBC Sign in | News | Sport | Weather | Shop | Reel | Travel | M…

NEWS

Home | Video | World | Asia | UK | Business | Tech | Science | Stories | Entertainment

Technology

## Health records 'put at risk by security bugs'

Share

FEATURED NEWS

### Teen Becomes First to Earn $1M in Bug Bounties with HackerOne

He is also the all-time top-ranked hacker on HackerOne's leaderboard, out of more than 330,000 hackers competing for the top spot.

by Tara Seals

March 4, 2019

### RSAC 2019: Container Escape Hack Targets Vulnerable Linux Kernel

A proof-of-concept hack allows


### RSAC 2019: Microsoft Zero-Day Allows Exploits to Sneak Past Sandboxes

Researchers say that Microsoft won't issue a patch for the issue.

by Tara Seals

March 5, 2019


### BSides SF 2019: Remote-Root Bug in Logitech Harmony Hub Patched and Explained

Users of Logitech's Harmony Hub get long-awaited answers about the critical bugs that left their home networks wide open to attack.

by Tom Spring

# Why Systems are Vulnerable?

- We "know" how to code securely
  - Follow the rules: CERT, Oracle, ...
  - Technical advances: types, memory safety
- But we still fail too often!
- Root causes
  - Coding instead of engineering
  - Human limitations
  - Unusable tools

# Our Approach: Usable Architecture-Based Security

Secure systems development

**Engineering:**
An architecture/design perspective

**Usability:**
A human perspective

$\lambda$

**Formal Modelling:** A mathematical perspective

# The Wyvern Programming Language

- Designed for security and productivity from the ground up
- General purpose, but emphasising web, mobile, and IoT apps

http://wyvernlang.github.io/

TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

Carnegie Mellon University

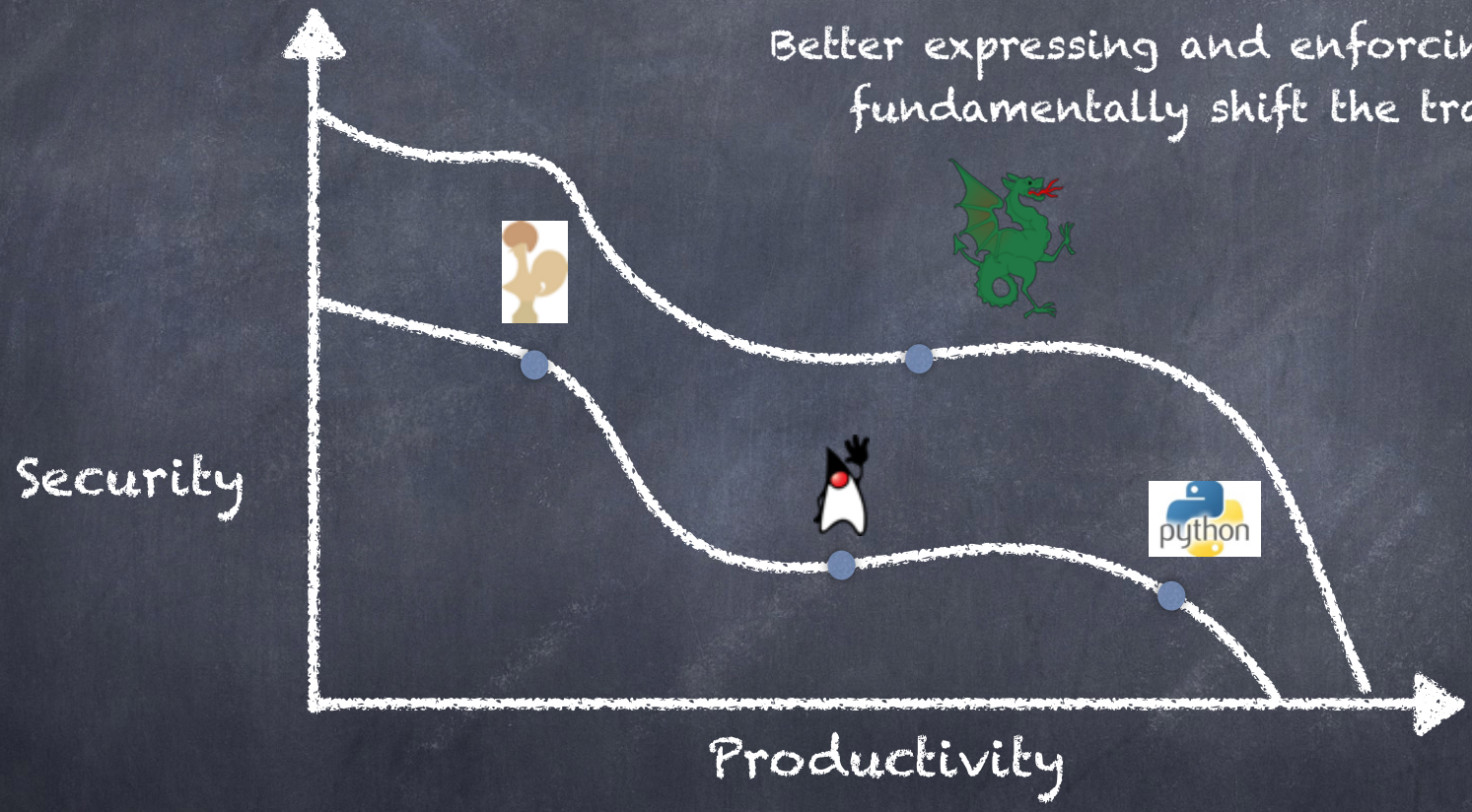
# The Wyvern Programming Language

- But you might ask: "Isn't there a trade off between security and productivity?


- What is Wyvern's secret sauce?

# Shifting the Tradeoff Curve

Better expressing and enforcing design could fundamentally shift the tradeoff curve



# Wyvern

- Design goals
  - Sound, modern language design
    - Type- and memory- safe, mostly functional, advanced module system
  - Incorporate usability principles
  - Security mechanisms built in

# Wyvern

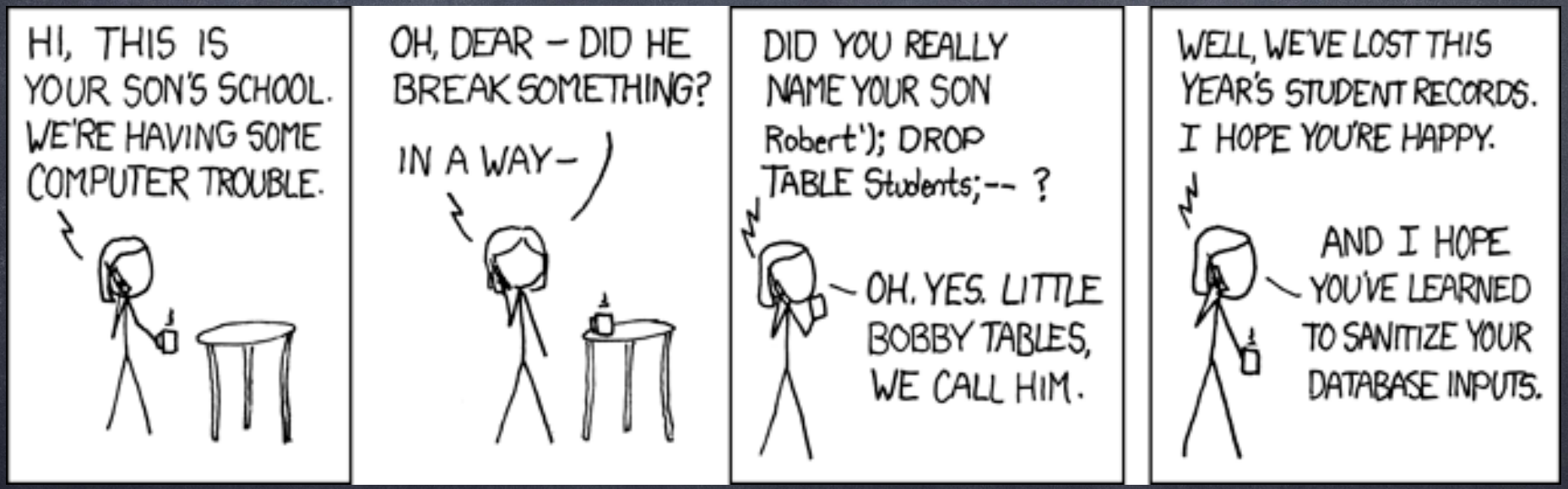
- The Wyvern Approach: Usable Design-Driven Assurance
  - Usable mechanisms to express and enforce large-scale design
  - Support for built-in assurance of critical properties, especially security
- Key mechanisms for expressing and enforcing design
  - Modules and architecture express high-level design
  - Extensible notation expresses code-level design
  - Types, capabilities, and effects are used to enforce design

# Hello, world!

```
require stdout

stdout.print("Hello, world!\n")
```

# Wyvern Demo: Immutability

# SQL Command Injection



# SQL Injection: a Solved Problem?

```
PreparedStatement s = connection.prepareStatement(
    "SELECT * FROM Students WHERE name = ?;");
s.setString(1, userName);
s.executeQuery();
```

Prepare a statement with a hole

Fill the hole securely

- Evaluation

- Usability: unnatural, verbose

- Design: string manipulation captures domain poorly

- Language semantics: largely lost - just strings
  - No type checking, IDE services, ...

# Wyvern: Usable Secure Programming

- A SQL query in Wyvern:

```
connection.executeQuery(~)
    SELECT * FROM Students WHERE name = {studentName}
```

~ introduces a domain-specific language (DSL) on the next indented lines

Semantically rich DSL. Can provide type checking, syntax highlighting, autocomplete, ...

Safely incorporates dynamic data - as data, not a command

- Claim: the secure version more natural and more usable
  - No empirical evaluation, yet

**Cyrus Omar**
Assistant Professor (Sep. 2019-)
Computer Science and Engineering
University of Michigan

# Technical Challenge: Syntax Conflicts

- Language extensions as libraries has been tried before
  - Example: SugarJ/Sugar* [Erdweg et al, 2010; 2013]

```
import XML, HTML

val snippet = ~

    How do I <b>parse</b> this example?
```

Is it XML or HTML?

# Syntax Conflicts: Wyvern's Solution

metadata keyword indicates we are importing syntax, not just a library

```
import metadata XML, HTML
```

No ambiguity: the compiler loads the unique parser associated with the expected type XML

```
val snippet : XML = ~

    How do I <b>parse</b> this example?
```

Syntax of language completely unrestricted - indentation separates from host language

# Technical Challenge: Semantics

Q: Is it safe to run custom parser at compile time?
A: Yes - immutability types used to ensure imported metadata is purely functional, has no network access, etc.

```
import metadata SQL
val connection = SQL.connect(...)
val studentName = input(...)
connection.executeQuery(~)
    SELECT * FROM Students WHERE name = {studentName}
```

Language definition includes custom type checker - can verify query against database schema

Splicing (as in genes) theory ensures capture-avoiding substitution in code generated by SQL extension - safe to use host language variables

SQL extension has access to variables and their types in Wyvern host language

# Wyvern TSL's

- Libraries cannot extend the base syntax of the language
- Instead, notation is associated with types.

"Type-Specific Languages" (TSLs)

- A type-specific language can be used within delimiters to create values of that type.

"Safely-Composable"

# Example

```
serve : (URL, HTML) -> ()
```

```
serve(`products.nameless.com`, ~)
  :html
    :head
      :title Product Listing
      :style ~
        body { font-family: %bodyFont% }
    :body
      :div[id="search"]
        {SearchBox("Products")}
      :ul[id="products"]
        {items_from_query(query(db,
           <SELECT * FROM products COUNT {n_products}>))}
```

base language
URL TSL
HTML TSL
CSS TSL
String TSL
SQL TSL

# How do you enter and exit a TSL?

- In the base language, several inline delimiters can be used to create a TSL literal:

```
`TSL code here, ``inner backticks`` must be doubled`
'TSL code here, ''inner single quotes'' must be doubled'
{TSL code here, {inner braces} must be balanced}
[TSL code here, [inner brackets] must be balanced]
<TSL code here, <inner angle brackets> must be balanced>
```

- If you use the block delimiter tilde (~), there are no restrictions on the subsequent TSL literal.
  - Indentation ("layout") determines the end of the block

# How do you associate a TSL with a type?

```
casetype HTML =
    Text of String
  | DIVElement of (Attributes, HTML)
  | ULElement  of (Attributes, HTML)
  | ...
  metadata = new : HasParser
    val parser : Parser = new
      def parse(s : TokenStream) : ExpAST =
          (* code to parse specialized HTML notation *)
```

```
objtype Parser =
  def parse(s : TokenStream) : ExpAST
```

```
casetype ExpAST =
  Var of ID
| Lam of (Var, ExpAST) | Ap of (Exp, Exp)
| CaseIntro of (TyAST, String, ExpAST) | ...
```

# Why not associate a grammar with a type?

```
casetype HTML =
    Text of String
  | DIVElement of (Attributes, HTML)
  | ULElement  of (Attributes, HTML)
  | ...
  metadata = new : HasParser
    val parser : Parser = ~
      start ::= ":body" children::start
                  `HTML.BodyElement(([], %children%))`
              | ...
```

Grammars are TSLs for Parsers!

Quotations are TSLs for ASTs!

# TSL Benefits

- Modularity and Safe Composability
  - DSLs are distributed in libraries, along with types
  - No link-time errors possible
- Identifiability
  - Can easily see when a DSL is being used
  - Can determine which DSL is being used by identifying expected type
  - DSLs always generate a value of the corresponding type
- Simplicity
  - Single mechanism that can be described in a few sentences
  - Specify a grammar in a natural manner within the type
- Flexibility
  - A large number of literal forms can be seen as type-specific languages
  - Whitespace-delimited blocks can contain arbitrary syntax

# TSL Limitations

- Decidability of Compilation: Because user-defined code is being evaluated during parsing and type checking, compilation might not terminate.
- No editor support, but subject of interesting related work at the University of Michigan by Cyrus's research group...

# Wyvern Demo: Type-Specific Languages

# Our Approach: Usable Architecture-Based Security

DSL support in Wyvern

**Engineering:**
Express design in domain-specific way

**Usability:**
Natural syntax, enabling IDE support

$\lambda$

**Formal Modelling:** Type safety, variable hygiene, conflict-free extensions

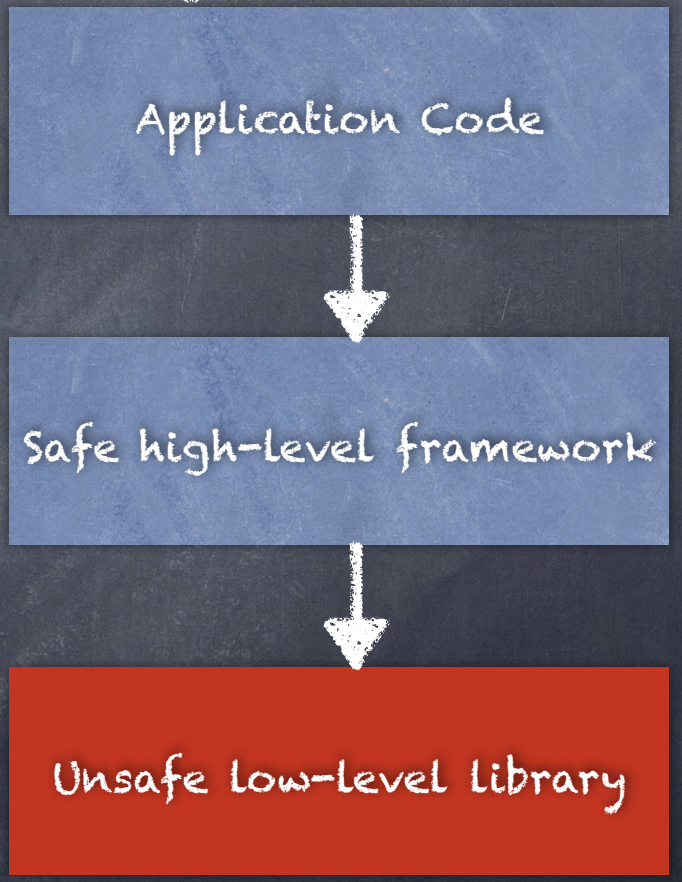
# Resource Use

- SQL extensions are nice!
  - But what if people use a low-level, string-based library anyway?
  - More broadly, what if people misuse resource-access libraries?

# An Old Idea: Layered Architectures

[Dijkstra 1968]

- Lowest layer: an unsafe, low-level library
  - provides basic access to resources
- Middle layer: a higher-level framework
  - enforces safety invariants over resources
- Top layer: the application
- Code must obey strict layering
  - Application must only use the safe framework
- Many variants:
  - Secure networking framework
  - Safe SQL-access library
  - Replicated storage library

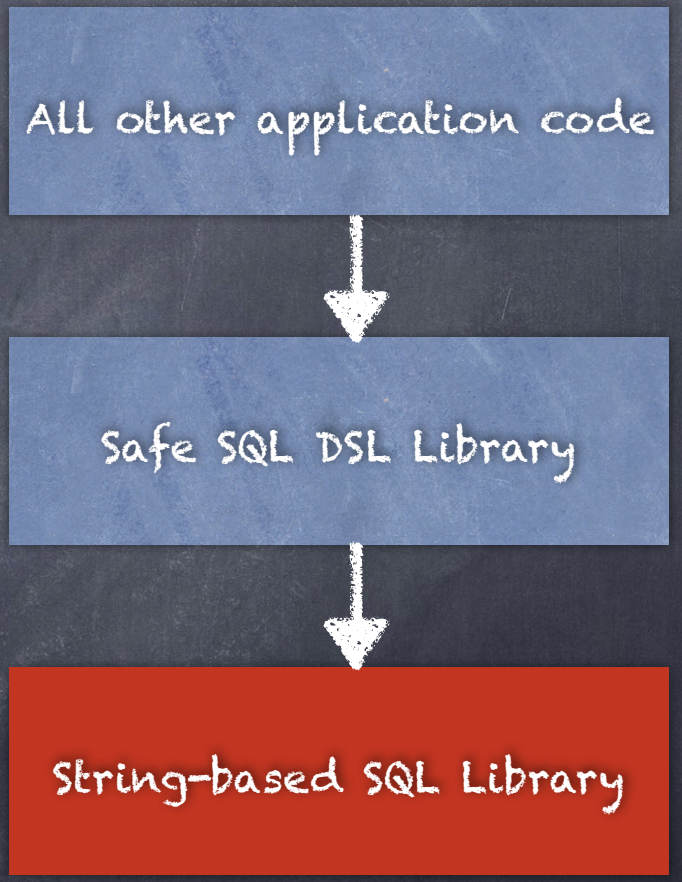



RQ: Can we use capabilities to enforce layered resource access?

* Capability: an unforgeable token controlling access to a resource [Dennis & Van Horn 1966]

# Architecture: Principle of Least Privilege (PoLP)

- Every module must be able to access only the resources necessary for its legitimate purpose [Saltzer & Schroeder 75]
- Architectural layering example: Only Safe SQL Library may access the low-level SQL interface



# Module Linking as Architecture

```
require db.stringSQL
```

To access external resources like a database, main requires a capability from the run-time system. A capability is an unforgeable token controlling access to a resource.

```
application.run()
```

# Module Linking as Architecture

We can import code modules, but they have no ambient authority to access resources (cf Newspeak). sqlApplication cannot access the database by itself.

```
require db.stringSQL


import db.safeSQL

import app.sqlApplication


val sql = safeSQL(stringSQL)

val application = sqlApplication(sql)


application.run()
```

We must instantiate a sqlApplication object, passing it the resources it needs. We pass only a capability to the safe library.

# Module Linking as Architecture

```
module def sqlApplication(safeSQL : db.SafeSQL)
def run() : Int
    // application code
```

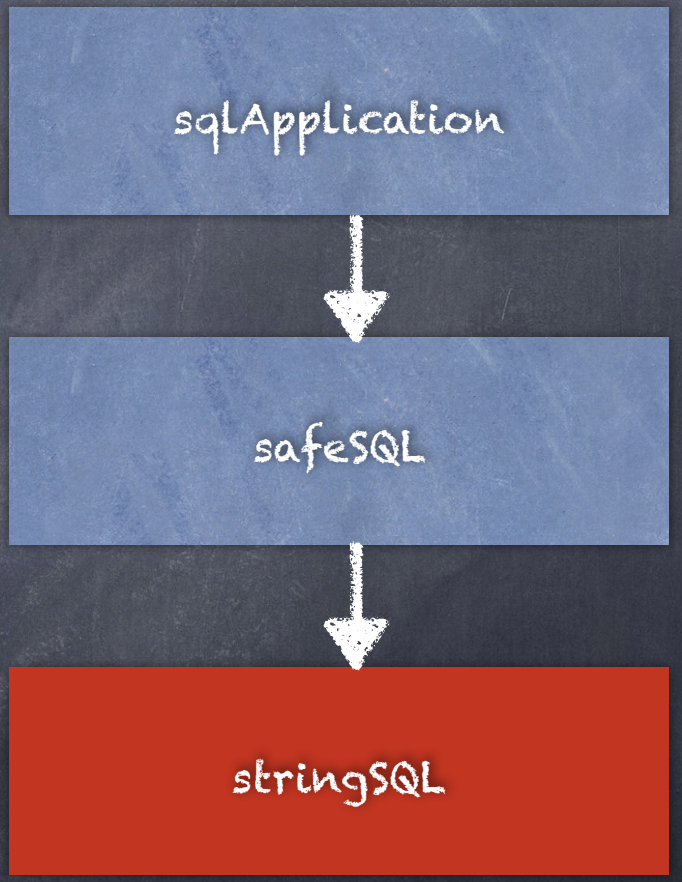
```
require db.stringSQL
```

```
import db.safeSQL

import app.sqlApplication
```

```
val sql = safeSQL(stringSQL)

val application = sqlApplication(sql)
```

```
application.run()
```

```
module def safeSQL(strSQL : db.StringSQL)
// implement ADT in terms of strings
```



# How Hard to Link it ALL Up?

- Most Wyvern modules don't have state, can be freely imported
- Statically tracked: stateful modules/objects and resource types

Provides access to OS resource

```
resource type File
    def write(s : String)
```

```
type SetM
    resource type Set
        def add(v : Int)
        def isMember(v : Int) : Bool
    def makeSet() : Set

module setM : SetM

module def client(aFile : File)
import setM ...
```

Type of modules is pure; no static state. Objects created by module may be stateful resources, though.

Resources must be passed in; pure modules can just be imported.

- resource types capture state or system access: other types do not
  - Useful design documentation; e.g. MapReduce tasks should be stateless
  - Supports powerful equational reasoning, safe concurrency, etc.

# Checking PoLP with Effects

```
// in signature of the rawSQL module
effect UnsafeQuery
type Connection
def connect(...) : Connection
def query(q:String) : {UnsafeQuery} Data

// client code
def getData(input : String) : Data
    rawSQL.query("SELECT * FROM Students WHERE name = '" + input + "';")
```

The unsafe SQL library defines an UnsafeSQL effect

Query operations have an UnsafeQuery effect

Error: getData() must declare effect rawSQL.UnsafeQuery

Has effect rawSQL.UnsafeQuery

NB! In Wyvern Effect is a "Resource.Operation" pair.

# Checking PoLP with Effects

```
// in signature of the rawSQL module
effect UnsafeQuery
type Connection
def connect(...) : Connection
def query(q:String) : {UnsafeQuery} Data

// client code
def getData(input : String) : {rawSQL.UnsafeQuery} Data
    rawSQL.query("SELECT * FROM Students WHERE name = '" + input + "';")
```

The unsafe SQL library defines an UnsafeSQL effect

Query operations have an UnsafeQuery effect

All dangerous code marked with effect

Has effect rawSQL.UnsafeQuery

NB! In Wyvern Effect is a "Resource.Operation" pair.

# Effect Abstraction

- Issue: won't users of the safeSQL library have an UnsafeQuery effect, if safe SQL is built on rawSQL?

The safeSQL functor uses a rawSQL module

```
module def safeSQL(rawSQL : RawSQL) : SafeSQL
type SQL
    metadata ...
abstract effect SafeQuery = rawSQL.UnsafeQuery
def query(SQL) : {SafeQuery} Data
    ...
```

Defines a SQL ADT with metadata for parsing

The SafeQuery effect is defined in terms of UnsafeQuery. This definition is abstract - hidden from clients.

Now clients have effect safeSQL.SafeQuery

Q: Can't any library do this, potentially hiding unsafe queries?
A: Potentially, but can mechanically check only trusted libraries do so

# Effect System Usability

Client Code

↓

Safe SQL DSL Library

- Isn't it a pain to declare all these effects?
  - Case in point: exception specifications in Java
- We can bound a module's effects by its capabilities
  - No need to effect-annotate the module
  - Does assume capability-safety (cf JS Frozen Realms)

Client can have effect safeSQL.SafeQuery (and nothing else)

```
module def client(safeSQL : SafeSQL) : Client

import ...
```

Imports may not be resources - no effects.

If safeSQL defines higher-order functions, make sure the argument is allowed to have the SafeQuery effect (cf contravariant subtyping).

# Our Approach: Usable Architecture-Based Security

Effects and capabilities in Wyvern

**Engineering:**
Architectural restrictions on resource use

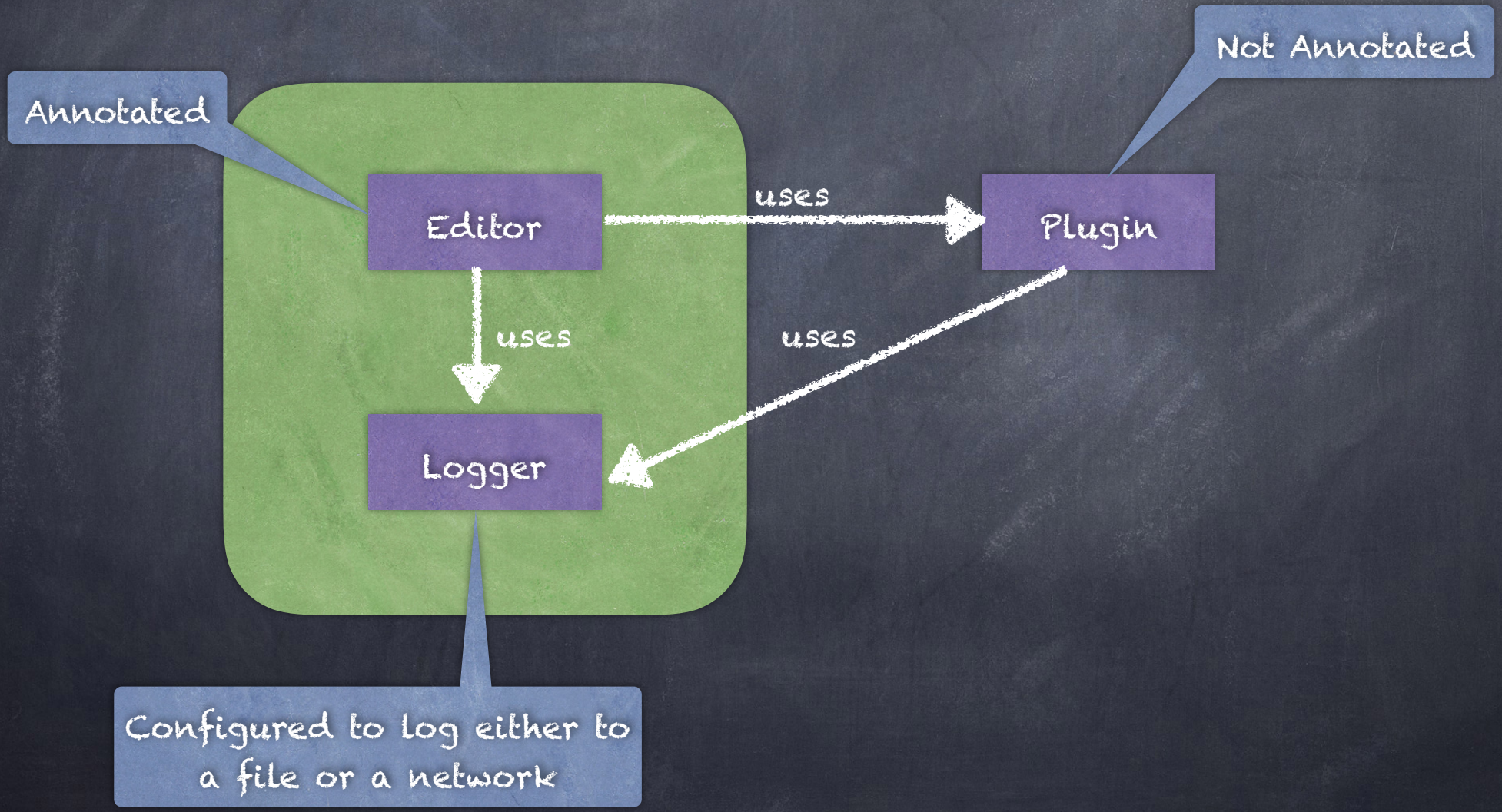
**Usability:**
Bound effects based on architecture

$\lambda$

**Formal Modelling:** effect- and capability- safety, effect bounds

# Wyvern Demo: Effects

# Wyvern Demo: Effects



# Higher Order Effects

```
module def repeaterPlugin(defaultLogger : Logger)

var logger : Logger = defaultLogger

def setLogger(logger : Logger) : Unit

    logger = newLogger
```

Our solution lifts polymorphism to the module level where the state is created

We would like to assign this function an effect polymorphic type

But the function might assign to local state, so the effect of newLogger must be bounded by the overall effect of the module (cf polymorphism and state more generally)

# Alternative: Effect Inference

- Effect Inference
  - Only applies if you have the code
  - Usability issues
    - can fail because of something deep in the code
    - can succeed, then fail if the code changes

# Sidenote: Wyvern Formalisation

- Built up from simply typed Lambda Calculus with recursive records
- Via classes translated to objects
- Via modules translated back to classes and objects system
- As "onion layers" with type members and effects added progressively

# Sidenote: Type Members

- More expressive than type parameters but harder to reason about
- Recent DOT result on soundness that we extended by exploring how to achieve decidability
- We support both type members with structural subtyping and
- Nominal declaration of explicit subtype relationships of objects that include bounded type members

# Wyvern Demo: Capabilities

# Sealers / Unsealers

```
require stdout

import wyvern.String
import wyvern.option
type Option = option.Option

resource type SealedBox
    def shareContent():Unit

resource type BrandSealer
    def seal(object:Option):SealedBox

resource type BrandUnsealer
    def unseal(box:SealedBox):Option

resource type BrandPair
    var name:String
    var sealer:BrandSealer
    var unsealer:BrandUnsealer
```

# Sealers / Unsealers

```
def makeBrandPair(name:String):BrandPair
    var shared:Option = option.None()
    def makeSealedBox(object:Option):SealedBox
        val newBox:SealedBox = new
            def shareContent():Unit
                shared = object
        newBox
    new
        var name:String = name
        var sealer:BrandSealer = new
            def seal(object:Option):SealedBox
                makeSealedBox(object)
        var unsealer:BrandUnsealer = new
            def unseal(box:SealedBox):Option
                shared = option.None()
                box.shareContent()
                var result:Option = shared
                result
```

# Sealers / Unsealers

```
// Simple example of using brand pair from E Wiki:
var alexBrandPair:BrandPair = makeBrandPair("Alex")
var jonathanBrandPair:BrandPair = makeBrandPair("Jonathan")

var alexBox:SealedBox = alexBrandPair.sealer.seal(option.Some("Alex's"))
var jonathanBox:SealedBox = jonathanBrandPair.sealer.seal(option.Some("Jonathan's"))

string = alexBrandPair.unsealer.unseal(alexBox).getOrElse(() => "NOTHING")
stdout.print(string + "\n")

string = jonathanBrandPair.unsealer.unseal(jonathanBox).getOrElse(() => "NOTHING")
stdout.print(string + "\n")

string = alexBrandPair.unsealer.unseal(jonathanBox).getOrElse(() => "NOTHING")
stdout.print(string + "\n")

string = jonathanBrandPair.unsealer.unseal(alexBox).getOrElse(() => "NOTHING")
stdout.print(string + "\n")
```

# Mint Example (E)

```
resource type Mint
    def makePurse(balance:Int):Purse
    def print():Unit

resource type Purse
    def getBalance():Int
    def sprout():Purse
    def getDecr():SealedBox
    def deposit(amount:Int, src:Purse):Unit
    def print():Unit
```

# Mint Example (E)

```
def makeMint(name:String):Mint
    var brandPair:BrandPair = makeBrandPair(name)
    new (selfMint) =>
        def makePurse(balance:Int):Purse
            var balance:Int = balance
            val decr = (amount:Int) => (balance = balance - amount)
            new (selfPurse) =>
                def getBalance():Int = balance
                def sprout():Purse = selfMint.makePurse(0)
                def getDecr():SealedBox = brandPair.sealer.seal(option.Some(decr))
                def deposit(amount:Int, src:Purse):Unit
                    brandPair.unsealer.unseal(src.getDecr()).getOrElse(() =>
                                                        ((a:Int) => (-1)))(amount)
                    balance = balance + amount
                def print():Unit
                    stdout.print("Purse that has ")
                    stdout.printInt(balance)
                    stdout.print(" bucks from mint named " + brandPair.name + "\n")
        def print():Unit
            stdout.print("Mint named " + brandPair.name + "\n")
```

# Mint Example (E)

```
var carolMint:Mint = makeMint("Carol")
carolMint.print()

var aliceMainPurse:Purse = carolMint.makePurse(1000)
aliceMainPurse.print()

var bobMainPurse:Purse = carolMint.makePurse(0)
bobMainPurse.print()

var paymentForBob:Purse = aliceMainPurse.sprout()
paymentForBob.print()

paymentForBob.deposit(10, aliceMainPurse)
paymentForBob.print()

bobMainPurse.deposit(10, paymentForBob)
bobMainPurse.print()
aliceMainPurse.print()
```

# Caretaker

```
require stdout

import wyvern.String
import wyvern.option
type Option = option.Option

resource type Carol
    def playWith1():Unit
    def playWith2():Unit

type Bob
    def playWith(carol:Carol):Unit
```

# Caretaker

```
var bob:Bob = new
    def playWith(carol:Carol):Unit
        carol.playWith1()
        carol.playWith2()

var carol1:Carol = new
    def playWith1():Unit
        stdout.print("Playing on the playground 1\n")
    def playWith2():Unit
        stdout.print("Playing on the playground 2\n")

stdout.print("Using Carol directly:\n")
bob.playWith(carol1)
```

# Caretaker

```
resource type Revoker
    def revoke():Unit

// TODO: We need Wyvern to support forwarding of methods
// somehow for this to work generically.
resource type CarolRevoker
    def carol():Carol
    def revoker():Revoker
```

# Caretaker

```
def makeCarolRevoker(carol:Carol):CarolRevoker
    var target:Option = option.Some(carol)
    new
        def carol():Carol
            new (thisCarol) =>
                var blankCarol:Carol = new
                    def playWith1():Unit = stdout.print("playWith1 REVOKED\n")
                    def playWith2():Unit = stdout.print("playWith2 REVOKED\n")
                def playWith1():Unit
                    target.getOrElse(() => thisCarol.blankCarol).playWith1()
                def playWith2():Unit
                    target.getOrElse(() => thisCarol.blankCarol).playWith2()
        def revoker():Revoker
            new
                def revoke():Unit
                    target = option.None()
```

# Caretaker

```
stdout.print("Creating Carol with caretaker.\n")
var carolRevoker:CarolRevoker = makeCarolRevoker(carol1)

stdout.print("Doing it with Carol via caretaker:\n")
var carol2:Carol = carolRevoker.carol()
bob.playWith(carol2)

stdout.print("Doing it with Carol via caretaker after revoking:\n")
var revoker:Revoker = carolRevoker.revoker()
revoker.revoke()
bob.playWith(carol2)
```