42: Enforced modular security

The clean garden security model is hopeless
We all live in the dark forest.

Programming at scale

- One programmer for a few days, no planned maintenance.
  Usually the customer is the single programmer.
  security?

- A single team/company for a few months, maintenance plan.
  Usually the customer is a wealthy individual or a small company.
  Security!

- A community effort for many years, maintaining IS developing.
  Usually there is no clear 'customer', but a large user base.
  Libraries, frameworks, OS, big open source projects..
  SECURITY !

Programming at scale

- Small scale:
  programming is mostly to express behavior.
- Larger scale:
  programming is mostly to restrict behavior.

- Freedom is slavery!

Correctness vs Security

- **Correctness**

  the code always does the right action
- **Security**

  the code never does the wrong action

  Different scopes; example:
- Correctness: the code is right up to stack/memory overflow
- Security: the code is right up to cosmic ray bitflip/hardware bug

Correctness vs Security

- Crucial point: the code may simply not do any action.
- Example: a secure but not correct implementation for anything would be

```
throw new Error("Nope");
```

- Example: a correct but not secure implementation for anything would be

```
int doStuff(int x){
  try{rec(aNumber);}
  catch(StackOverflowError o){ formatHD();}
  return correctlyDoStuff(x);
  }
void rec(int i){if(i==0){return;} rec(i-1);}
```

Modular Correctness/Security

- To do anything at scale we need to be able to reason about units of code in isolation.
- **Modular correctness**: (M can be a library and C the user code)

  The module M can be placed in any context C and when C calls a functionality of M, M always answer according to its specification.

- **Modular security**:

  The module M can be placed in any context C and code running under the control of M will never do the wrong action

Security is not just network security

- **Fact**

  We are getting better at network security

  Compromising a system by sending tailored data is getting harder and harder.

- **Fact**

  Hackers are not going to simply stop bothering.

  They are going to try to hack our code using other kinds of vulnerabilities.

Traditionally, security focused a lot on network security. Now that network security is finally getting stronger, hackers are using other ways to get their ways.

Software supply chain attacks

- Attackers can inject malicious code into one of our dependencies.

  For example, they become 'trusted' contributors to one of the open source libraries used by our system. In this way, when updating the library, we will silently import adversarial code in our security critical process.

- Eventually, some malicious code will make its way into our program.

Indeed, my interest in SW security started when as soon as my friend got to work in some bank security critical code, he spotted a back door left from a former programmer. My friend decided to report the issue and get it fixed. Someone else may have not been so honest.

Malicious code in the process.

- Malicious code is now running in our process.
- In most languages, at this point it is game over.

  The malicious code can do any action.

- 42 can modularly enforce security even in the presence of malicious code.

  (Note: 42 does not enforce correctness)

What does it mean to do an action?

- Most languages allow any piece of code to do any actions (for example I/O).
- Typically this happens by indirectly calling native static methods.
- If static methods/fields are forbidden, we can get a more pure OO setting where all behavior is obtained by method calls to objects.
- Put simply, if there's no object to do the task, the task can't be done.

The rule of Object Capabilities

NO OBJECT

NO ACTION

Java with Object capabilities

- For example, in Java we can do

```
System.out.println("...");
```

- This relies on the static field out and can be accessed everywhere
- An Object capability variant of Java could look like

```
void main(System s){ s.out.println("..."); }
```

- And 'System' would have no public constructors.

Object capabilities in 42

- Example: File System access

```
Fs = Load:{ reuse [L42.is/FileSystem] }

Code = { class method Void (mut Fs f)[_] = (
  S s=f.read(Url"data.txt")
  f.write(on=Url"data.txt",content=S"SomeContent")
  Debug(s)
  ) }

Main1 = Code(f=Fs.Real.#$of())
```

- What if we do not trust 'Code' with all the power of the File System

Object capabilities in 42

- Example: File System access

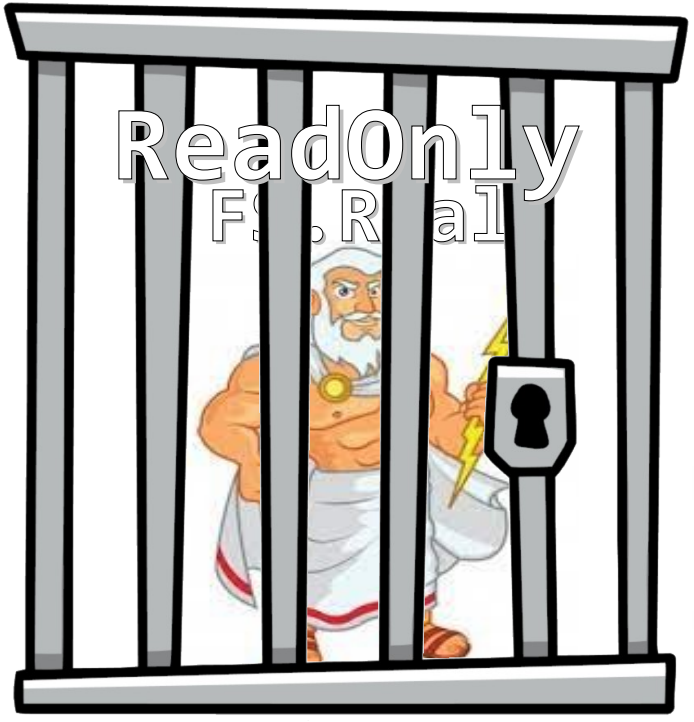
```
Fs = Load:{ reuse [L42.is/FileSystem] }

Code = { reuse [TrustySource.com/justTrustMe] }
  //class method Void (mut Fs f)[_] = (..)
  //what is this code doing today?
  //what will it do tomorrow?


Main1 = Code(f=Fs.Real.#$of())
```

- What if we do not trust 'Code' with all the power of the File System

Restricting the power of Fs

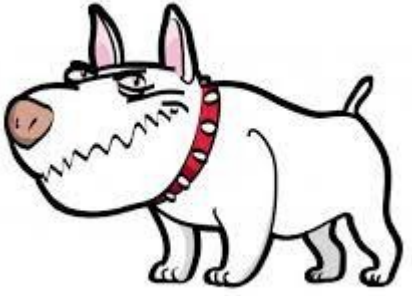
Restricting the power of Fs

Restricting the power of Fs



Delegate the read methods on the captive capability,

Throw errors on the write methods.

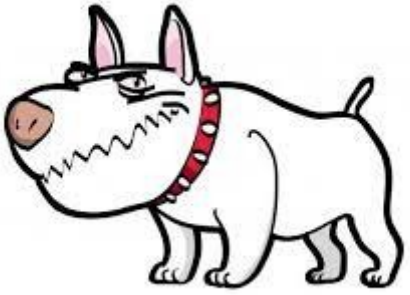


Restricting the power of Fs

```
ReadOnly = Public:{[Fs]
  mut Fs inner

  @Public class method mut This #$of() = This(inner=Fs.Real.#$of())
  @Public class method mut This(mut Fs inner)

  method read(that) = this.#inner().read(that)
  method readBase64(that) = this.#inner().readBase64(that)

  method makeDirs(that) = error X""
  method write(on,contentBase64) = error X""
  method delete(that) = error X""
  method write(on,content) = error X""
  }
SaferMain = (
  fs = ReadOnly.#$of() //The caged Real.Fs
  Code(f=fs) //We do not cage the untrusted code!
  )//We cage the weapons that the untrusted code can use
```

Restricting the power of Fs

- Now 'Code' does not have the permission to write files. Even if 'Code' is malicious, the 'Fs.Real' instance is encapsulated into a watchdog object that only allows them to call the 'read' methods.



  - 'Code' is unable to create another Fs.Real instance.
  - 'Code' is unable to freely access the Fs.Real instance from the 'ReadOnly' instance.
  - Since there are no static fields, 'Code' can not use those to access other 'Fs.Real' instances.
  - 'Code' can not access the FileSystem without going through an Fs.Real instance: all native code must be in #$ created objects

Three roles for programmers

- **Security architects:** they write capability classes, like 'ReadOnly'. Those classes can wrap and refine other capabilities.
- Any security condition that can be expressed as a check performed by a wrapper object can be modularly enforced by 42.

- **Main programmer:** they instantiate capabilities written by Security architects and they pass it to untrusted code.

- **Regular programmers:** they write most of the code, and they are responsible to write correct$^{(enough)}$ code but they do not need to worry about any security consideration.
- If they 'somehow can' do an action, that is a valid action.

Third party libraries/untrusted code

- Most libraries will not contain capability classes, but they will take in input capability objects.
- Those libraries will not be able to do anything outside of the security actions approved by the main programmer.

- Few core libraries will contain capability classes and their code will need to be manually verified.

It can't be done in other languages

- Even when coding in other languages, having a self contained part of the code defining security actions would be a good programming practice.

- This would statistically reduce the amount of security bugs in your application, but it would be no match against carefully crafted adversarial code.

- Requirement for capabilities:
  - Capability classes must not be freely instantiable
  - Reflection and other tricks must not allow to access hidden fields
  - Static fields may become a hidden communication channel between multiple untrusted code units.
  - Full access to native code must be restricted.
- Some other richer patterns are (only) possible in 42.

Object invariants => Security

- 42 enforce that object invariants can never be observed broken.
- Thus object invariants can enforce security

  If DBAccess has invariant and

  we need a DBAccess to access the DB

  Then only valid DBAccess objects will access the DB
- Invariants can encode the security requirements

Object invariants => Security

- For example, we want to ensure Person names in the DB are never empty.
- We can encode this by enforcing an invariant on Persons in the code.

- What happens if the user do

```
new Person("")
```

  - Correctness solution: the person will have a default non empty name, "Bob".
  - Security solution: the code will throw an error and no person is created.

- In Java/C# it is hard but possible; In C/C++/Python is simply hopeless:
  - Python: user code can always disassemble and tweak any other code
  - C/C++: user code can always send the whole system into undefined behavior, after that the code of M can do anything.
- In 42 it is quite trivial (but that would be another talk)

More?

Today I only focused on the security aspects of 42 and Object capabilities

Other 42 features:

- automatic unobservable parallelism
- ensured correct caching
- unbeatable representation invariants
- reference capabilities for aliasing and immutability
- trusted deterministic computations
- meta-programming and programmatic refactoring
- information flow control for confidentiality/integrity
- a blossoming ecosystem of libraries

Website / YouTube

- See the 42 tutorial in video format at ...
  (Intro to 42 playlist)
  https://www.youtube.com/watch?v=9RZlkdg3YBU&list=PLWsQqjANQic8c5wG3LfSe-mMiBKfOtBFJ

  (https://L42.is)
  (https://www.youtube.com/MarcoServetto)

But I do not want restrictions when I code

- It is turtles all the way down!
- An assembly programmer may be against C preventing them from freely decide how to allocate registers.
- A C programmer may be against Python preventing them from freely handle memory management.
- A Python programmer may be against Java preventing them to handle self modifying code (monkey patching) and requiring more strict typing rules.
- A Java programmer may be against Haskell preventing them from freely handling I/O and mutation.
- A Haskell programmer may be against Agda preventing them from handling termination and well foundness of data-structures.