Towards Secure Systems Programming Languages

Gilad Bracha
Convergence: Viruses, Worms and Terrorists

- Mainstream Operating Systems are insecure
- Cyber-Pearl harbor is only a matter of time
- Expect every Windows™ machine on the planet to have its disk erased one sunny day soon
Is it just Windows?

• No, if Linux or MacOS™ were as prominent a target, we would hear more about their problems

• However ...
Is it just Windows (3)

• Other platforms are less of a problem, because their foundations are cleaner

• Any platform in a monopoly position poses an inherent security problem
The Problem of Monoculture

The Irish Potato blight of the 19th century was a result of agricultural monoculture. Today we have an OS monoculture.
The Problem of Monoculture

• If the whole world uses one platform, a single vulnerability can be leveraged worldwide

• Unless Windows is perfect, its very ubiquity is a security risk

  – D. Geer et al., CyberInSecurity: The Cost of Monopoly

• Same is true for programming languages

• Diversity is good
What do we do?

Possible answers:

• Antivirus Software
• Security Audits
• Training and Education
• Secure Programming Languages
• Simpler Designs
What do we do?

Possible answers:

- Antivirus Software
- Security Audits
- Training and Education
- Secure Programming Languages
- Simpler Designs
Training and Education

- Human error is the prime cause of security breaches
- Leads to the argument that technology is not the issue
- Human error is not going away - that’s why AI is so popular
- Technology that is less error-prone is crucial
What do we do?

Possible answers:

• Antivirus Software
• Security Audits
• Training and Education
• Secure Programming Languages
• Simpler Designs
Secure Programming Languages

An unreliable programming language generating unreliable programs constitutes a far greater risk to our environment and to our society than unsafe cars, toxic pesticides, or accidents at nuclear power stations.

– C.A. R. Hoare, The Emperor's Old Clothes, 1980 Turing Award Lecture
Secure Programming Languages

• A major problem in all system software is that it is written in C or a variation thereof
• C’s long reign is coming to an end
• C++ has already lost the battle in application software
• Systems software is the next frontier for high level programming languages
• We need to write the OS in a HLL that runs on top of a VM
• Today, virtual machines run on top of operating systems
• To resolve this, expect the two to merge
Examples of this Trend

• OVM
• Klein
• Oberon
• OOVM
Characteristics of Secure Programming Languages

- Safe access to memory
- Encapsulation/Modularity
- Type safety
- Well defined, tractable semantics
- Flexibility
Characteristics of Secure Programming Languages

• Safe access to memory
• Encapsulation/Modularity
• Type safety
• Well defined, tractable semantics
• Flexibility
Characteristics of Secure Programming Languages

• Safe access to memory:
  • Pointer safety
  • Garbage Collection
  • Dynamic Type Safety
• Range checks on array access
Characteristics of Secure Programming Languages

- Safe access to memory
- Encapsulation/Modularity
- Type safety
- Well defined, tractable semantics
- Flexibility
Encapsulation and Modularity

• Modularity and Security go hand in hand
• Small modules, strictly decoupled, enhance reliability and security
• Module coupling for performance and “integration” is a problem
Encapsulation enables Capabilities

- Capabilities, Security and Objects are a natural fit
- Objects in today’s mainstream languages are not encapsulated, so the fit is imperfect
Public Fields

• Uninhibited access to representation makes for bad software engineering and poor security

• No discernible advantage except in most naive implementations
Pure Object Orientation

• Mainstream languages distinguish objects from “primitive” data such as numbers and characters

• Trend is to blur this distinction: autoboxing in C#™, coming in next release of Java™

• Ultimately, Smalltalk’s approach **Everything is an Object** will dominate
Pure OO and Encapsulation

- Objects are characterized only by their observable behavior
- Different implementations are interchangeable
Class based Encapsulation vs. Object-based Encapsulation

class C {
    private String key;
    public void copyKeyFrom(C c) {
        key = c.key; // legal: key is private to C
    }
}
Class based Encapsulation vs. Object-based Encapsulation

class C {

    private String key;

    public void copyKeyFrom(C c) {
        key = c.key; // illegal: key private to this
    }
}
Class based Encapsulation vs. Object-based Encapsulation

- Object-based encapsulation can be enforced by context-free syntax
- Class-based encapsulation requires a type checker
Characteristics of Secure Programming Languages

- Safe access to memory
- Encapsulation/Modularity
- Type safety
- Well defined, tractable semantics
- Flexibility
The Paradox of Type Systems

- Type systems help reliability and security by mechanically proving program properties
- Type systems hurt reliability and security by making things complex and brittle
Mandatory Typing

Well known advantages

- Machine-checkable documentation
- Types provide conceptual framework
- Early error detection
- Performance advantages
Mandatory Typing

Disadvantages:

• Brittleness/Rigidity
• Lack of expressive power
How Mandatory Typing Undermines Security

- Once a mandatory type system is in place, people rely on it for:
  - Optimization
  - Security Guarantees
- If type system fails, behavior is completely undefined
Example: Class Loaders

Class loading becomes incredibly subtle (cf. Liang and Bracha, OOPSLA 98)

• Class loaders define name spaces for types
• Inconsistent namespaces mean inconsistent types
• JVM has nominal type system
• Failure to detect inconsistencies across class loaders leads to catastrophic failure (forgeable pointers, privacy violations etc.)
Wait, type systems shouldn’t fail! A good type system will be formally proven to be sound and complete

• Real systems tend to be too complex to formalize
• Formalizations make simplifying assumptions
• These assumptions tend to be wrong
• Implementations tend to have bugs
How Mandatory Typing Undermines Security

• Type Systems are subtle and hard
• *Relying* on them is dangerous
Having our Cake and Eating it too

- Performance disadvantage is greatly overstated
- Importance of performance also overstated
- Other advantages of static types can be had without the downside
- Enter **Pluggable, Optional Type Systems**
Having our Cake and Eating it too

- Performance disadvantage is greatly overstated
- Importance of performance also overstated
- Other advantages of static types can be had without the downside
- Enter **Pluggable, Optional Type Systems**
Optional Type Systems

- Run-time semantics are independent of type system
- Type annotations are optional
Optional Type Systems

- Run-time semantics are independent of type system
- Many problematic language constructs are ruled out, e.g.,
  - Public fields aren’t possible
  - Class-based encapsulation is prohibitively expensive
- Type-based overloading is not an option
Pluggable Type Systems

- Type systems can enforce confinement
- Type systems can track information flow
- Rather than try and force a single solution, view type systems as plug-ins
Characteristics of Secure Programming Languages

- Safe access to memory
- Encapsulation/Modularity
- Type safety
- Well defined, tractable semantics
- Flexibility
Tractable Semantics

- As in type systems, formal description is **not** key
- Simplicity, regularity, uniformity are
- If these hold, formalization is plausible, accurate and useful
- If not, formalization is intractable and/or inaccurate and of limited utility
Characteristics of Secure Programming Languages

- Safe access to memory
- Encapsulation/Modularity
- Type safety
- Well defined, tractable semantics
- Flexibility
Flexibility

• Rigid systems are robust until they break
• We need systems that are less rigid and more malleable
Telecomm Atoll

- A man-made atoll in the ocean, where buggy mobile phones are dumped.
- If a device has a major security flaw in its software, it could cost hundreds of millions to replace.
- On-the-fly patching is attractive.
Reflection

• Java introduced limited form of reflection, *introspection*, to a wide audience

• Reflection now an essential component of business software
class Introspector {
    public static void main(String[] args) {
        Introspector i = new Introspector();
        Class myClass = i.getClass();
        for (Method m : myClass.getMethods())
            System.out.println(m);
    }
}
Reflection

More sophisticated uses to come

- *Self modification*

- *Intercession*
Patching System Software on the Fly

Systems should not reboot. Instead, patch running system, e.g:

- Phone Switches (see Erlang)
- Mobile phones
Patching System Software on the Fly

Patching must be well founded semantically

- Writing files into directories doesn’t cut it
- Reflective self modification is a principled way to do patching
- Language semantics structure patching
Patching System Software on the Fly

We do need to be very careful

- Authentication is a big issue
- Proper design ensures single entry point
- Patching can be destabilizing
What do we do?

Possible answers:

• Antivirus Software
• Security Audits
• Training and Education
• Secure Programming Languages
• Simpler Designs
Simplicity
Simplicity

• Need to get away from Rube Goldberg Virtual machines

• Looking for the analog of “few moving parts”

• Less rules, not more
Simplicity

• How do you decide when things are “as simple as possible, but no simpler”?

• One useful guideline: avoid example-driven design

• Use cases are valuable for analysis, not synthesis
Simplicity

• I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.
  -- C.A. R. Hoare, The Emperor's Old Clothes

• Dijsktra’s parable: EWD594 in Selected Writings on Computing: A Personal Perspective
  Edsger W. Dijkstra