RE-TRUST
Design Alternatives on JVM

Paolo Falcarin

(Politecnico di Torino - Italy)

paolo.falcarin@polito.it
http://softeng.polito.it/falcarin
Trento, December, 19th 2006

Tamper-Detection

- Tamper-detection goals
  - Detect malicious modifications to program
  - Cause the program to fail if modified
- Check the executable version itself
  - E.g., compare program with a hash of itself
- Self-checking relies on code checkers
  - whose position is hidden in the application
  - whose behavior is often obfuscated

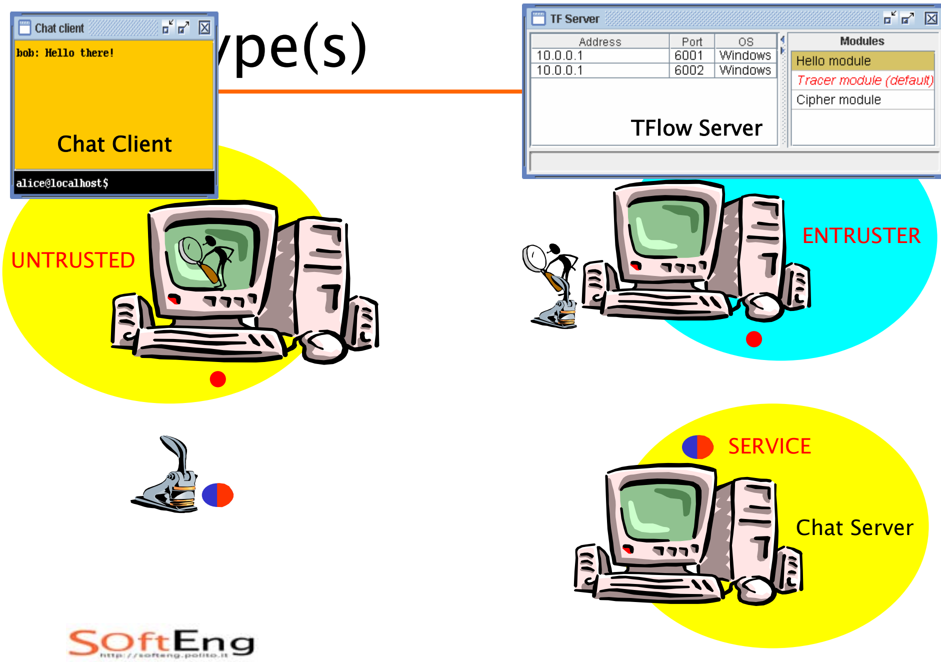
Issues on self-checking

- However, no technique can prevent all attacks
  - Goal is to increase the cost for the attacker
- Current self-checking techniques:
  - bundled within the application
  - can be identified and disabled
  - Nobody will notice!!!

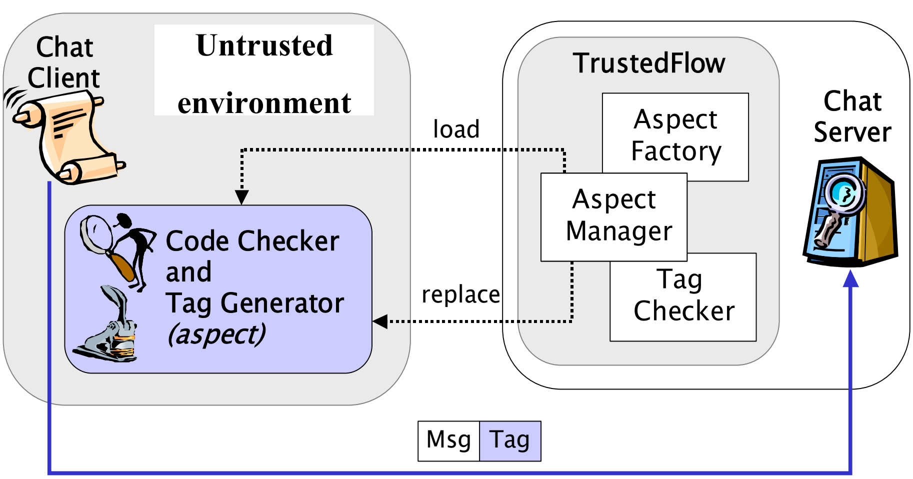
RE-TRUST approach

- Two main benefits:
  - *remote verification* that self-checking has been performed
  - *continuous replacement* of self-checking code
- 2 Prototypes implemented:
  - Using Dynamic AOP
  - Using JVMTI

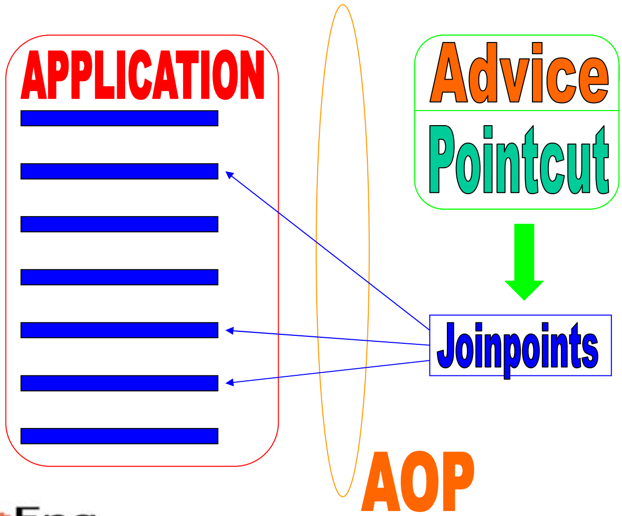
ype(s)



Prototype 1: Dynamic Aspect



Aspect-Oriented Programming



SOftEng

Aspect Weaving

- Aspect is "extra-code" that modularizes the implementation of a crosscutting concern
- The final code is obtained by weaving base code and aspect code
  - At compile time with an aspect compiler
  - At run time with a dynamic-AOP JVM

SOftEng

Why AOP ?

Aspect Oriented Programming (AOP)

- Modularizes self-checking code
- Aspect has a privileged view on the whole code
- Trusted Tag generator in dynamic aspect
  - It can be continuously updated
- Client code is NOT aspect-aware

Dynamic AOP with PROSE

- PROSE is an extension to standard JVM with dynamic AOP features
- Aspects can be remotely added/removed at run-time
- In PROSE terminology:
  - An Aspect is a Java class
  - It contains many Crosscut objects
  - A crosscut is a pointcut-advice pair

The Tag Generator Crosscut

```
public Crosscut tagGenerator = new MethodCut() {
 public void METHOD_ARGS(PrintWriter p, String msg) {
  StringBuffer tag = new StringBuffer(msg);
  tag.append( crypt( counter, key ));
  tag = hash(tag);
  p.println(tag);
  p.println(counter); p.flush();
  counter++;
 }
 protected PointCutter pointCutter() {
  PointCutter socket =
        Within.method("println").AND(type("PrintWriter"));
  return Executions.before().AND(socket);
 } };
```


Bytecode checking

- Using BCEL Java library
- Extract the actual method signature and class name.
- Keyed-hash of bytecode method is compared with the original one
- If they differ the key is nullified
  - =>The tag generator sends wrong tags
  - =>The server detects tampering

The Bytecode Checker Crosscut

```
public Crosscut bytecodeChecker = new MethodCut() {
  public void METHOD_ARGS(ANY x, REST pp) {
    String className =
        thisJoinPoint().getThis().getClass().getName();
    String method =  thisJoinPoint().getSignature();

    if (!checkBytecode(className, method))
       key=null;
  }
  protected PointCutter pointCutter() {
   return Executions.before().
        AND(Within.packageTypes("it.polito.chat.*"));
  } };
```

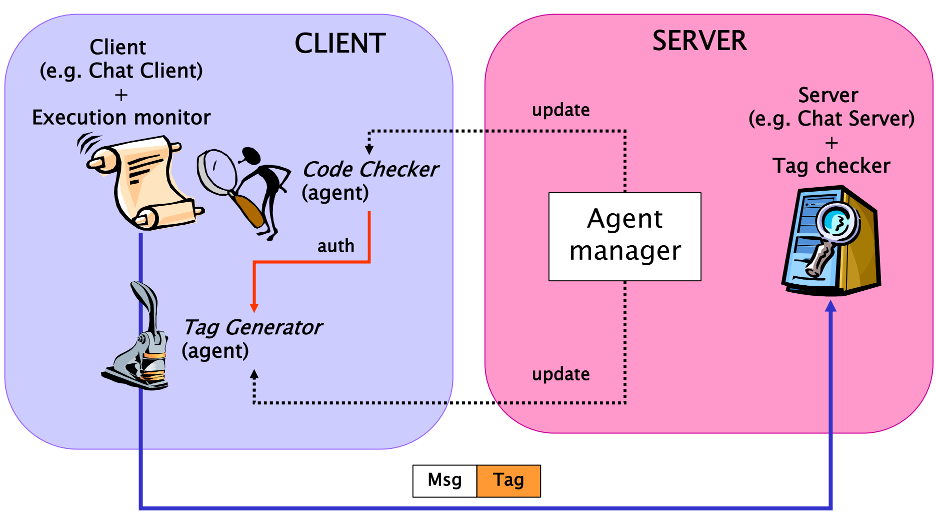

Sandbox

- Java "sandbox" restricts the operations an application can do.
  - protecting from hostile programs downloaded from un-trusted servers
- We face with the dual problem:
  - The trusted aspects sent by the trusted server cannot trust the environment they will be deployed in.
  - Deployed aspect forbids many “dangerous” activities to the running application.

The Sandbox Crosscut

```
public Crosscut shield = new MethodCut() {
 public void METHOD_ARGS(ANY x, REST pp) {
    key=null;
 }
 protected PointCutter pointCutter()        {
   PointCutter native = Within.method(NATIVE_MODIFIER);
   PointCutter fork =
        Within.method("exec").AND(type("java.lang.Runtime"));
   PointCutter loader = Within.subType("java.lang.ClassLoader")).
        AND(NOT(Within.type("java.security.SecureClassLoader"));
   return Executions.before().AND(native.OR(fork).OR(loader));
 }
};
```


Prototype 2: JVMTI



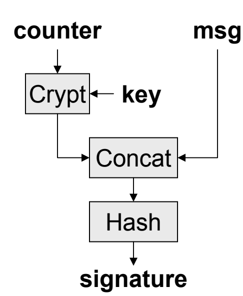
SoftEng

Prototype 2 (JVMTI)

- Execution interception with JVMTI
  - Run JVM (Java 5) in agent mode
  - An agent plugged-in using the JNI interface
  - It downloads module libraries at run-time
- Transparent tag insertion
  - Call to socket write are intercepted
  - And Data buffer is tagged
- Method entry is monitored
  - Method byte-code is continuously verified
- With JVMTI also program's memory can be accessed

Tag generation

- Tag is uid:counter:sign:plainmsg
  - Where uid is ip:port
- Counter-mode block cipher
  - Crypt: Blowfish
  - Hash: sha-1



Integrity check

- Module contains
  - List of crypto hashes (each method)
  - Symmetric key
- Keyed hash recomputed each time a method is called
- Hash compared with “good” copy

Threats: Disablement

- Dynamic Module allows:
  - integrity-checking code better modularized
  - Clearly separated from client
    - –It is inserted at run-time
- Disabling checking

  =>stops tag generation

  =>tampering detected by TFC

  =>server can block untrusted client

Threats: Replacement

- Discovery of secret key…to create correct tags
- Replace aspect to disable checking but sending correct tags:
  - Attacker must obtain module code (coming at run-time)
  - Replacement must be applied before TTG expires
- New module checks previous one

Safety features of the JVM

- Unspecified memory layout: JVM stores Application in different data areas
  - Java stacks (one for each thread)
  - A method area where bytecodes are stored
  - A heap, where objects created are stored
- When the JVM loads a class file, it decides where to store the bytecodes.
- An attacker cannot predict where the class' data will be stored
- The way in which a JVM lays out its inner data depends on JVM implementation

Threats: debuggers

- Dynamic AOP relies on debugger
- Prototype2 relies on JVMTI
- In both cases attackers cannot run client in debug mode
  - Is this enough to thwart them?
- Attacker should be smart to discover the checker behavior
  - Difficult access to mobile code
  - Automating this attack before a new module arrives is not trivial

Open issues: platform

- All software-based techniques rely on an un-trusted external platform:
  - the JVM, the O.S., the hardware.
- This is a problem for ALL integrity-checking techniques
- Our Prototype could (?) check authenticity of:
  - underlying VM
  - O.S. & HW configuration

Conclusions

- The aspect/agent checker is:
  - Not bundled within application
  - Its strategy is not visible through static code analysis
  - Easily configurable
  - Can check previously installed ones
  - Could check the platform itself

Possible Enhancements

- Module obfuscation
- Module factory
- Server authentication
- Network protocol design
- Measures
- O.S. configuration checking
- Integration with HW
- Hiding key with White-box Cryptography

SoftEng http://softeng.polito.it