Improvements using mobility for remote entrusting

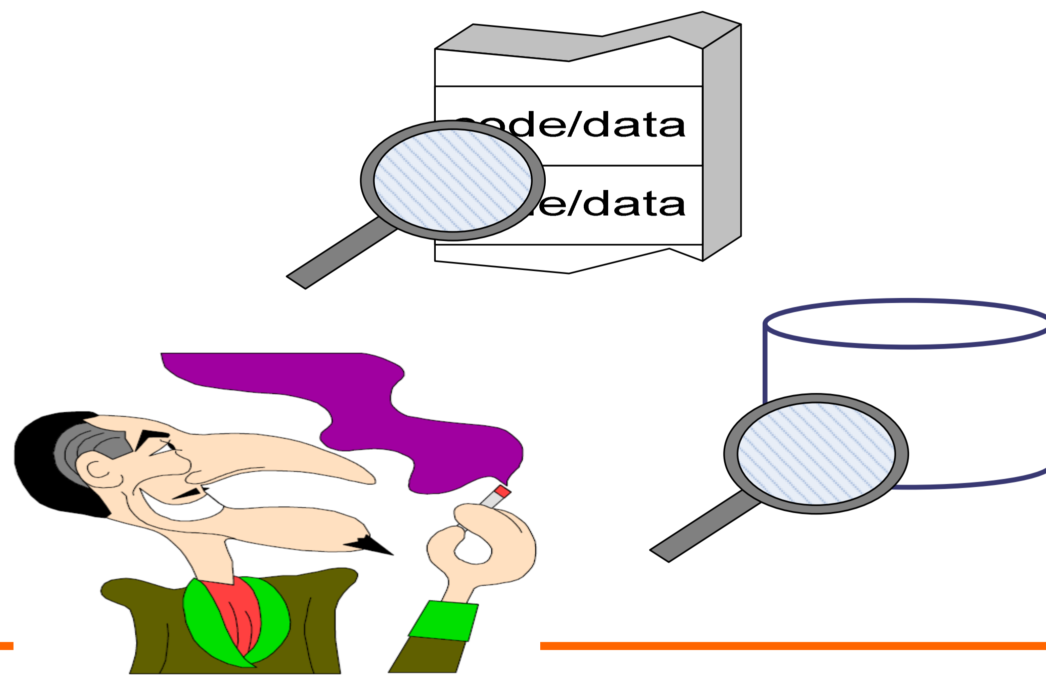
Paolo Falcarin

http://softeng.polito.it

RE-TRUST Meeting, 19 Dec 2007

Why mobile code?

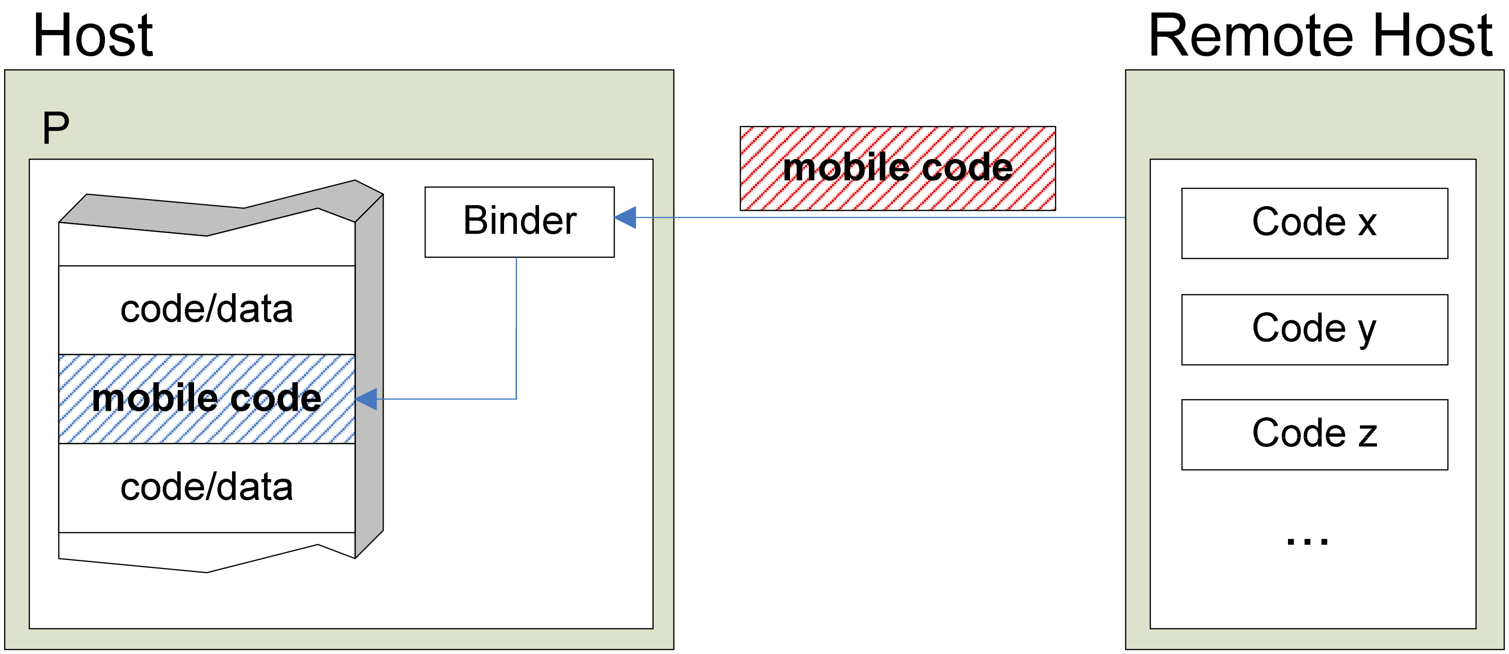
- Protections are typically embedded in application code
- The Attacker can look at executable and modify it (disassembler, debugger)



Remote entrusting with mobility

- Mobile code can be:
  - Integrity-checker
  - Functional code
- Mobile code is replaced during execution by trusted server
- Server needs a library of different integrity-checkers ready to be sent

Mobile Code



Host is untrusted

Remote Host: send mobile code

Binder: it is responsible of proper installing of mobile code (interlocking)

Mobile code Binder

- Two main categories of binder
  - Embedded in application native code
  - Extension to VM for managed code
- Former prototypes on JVMs
  - Dynamic AOP
  - Java 5 JVMTI interface
- Recent prototype in native code

Mobile code and JVM

- Dynamic AOP platform:
  - allows add and replace aspect/classes as integrity checkers
  - Easy design and mobility handling
  - Performances were not good
- JVM 5 extension on JVMTI interface
  - Allows read-access to code memory image
  - Mobility to be implemented from scratch and not easy to write modules
  - Better performance

Safety features of the JVM

- Unspecified memory layout: JVM stores Application in different data areas
- 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

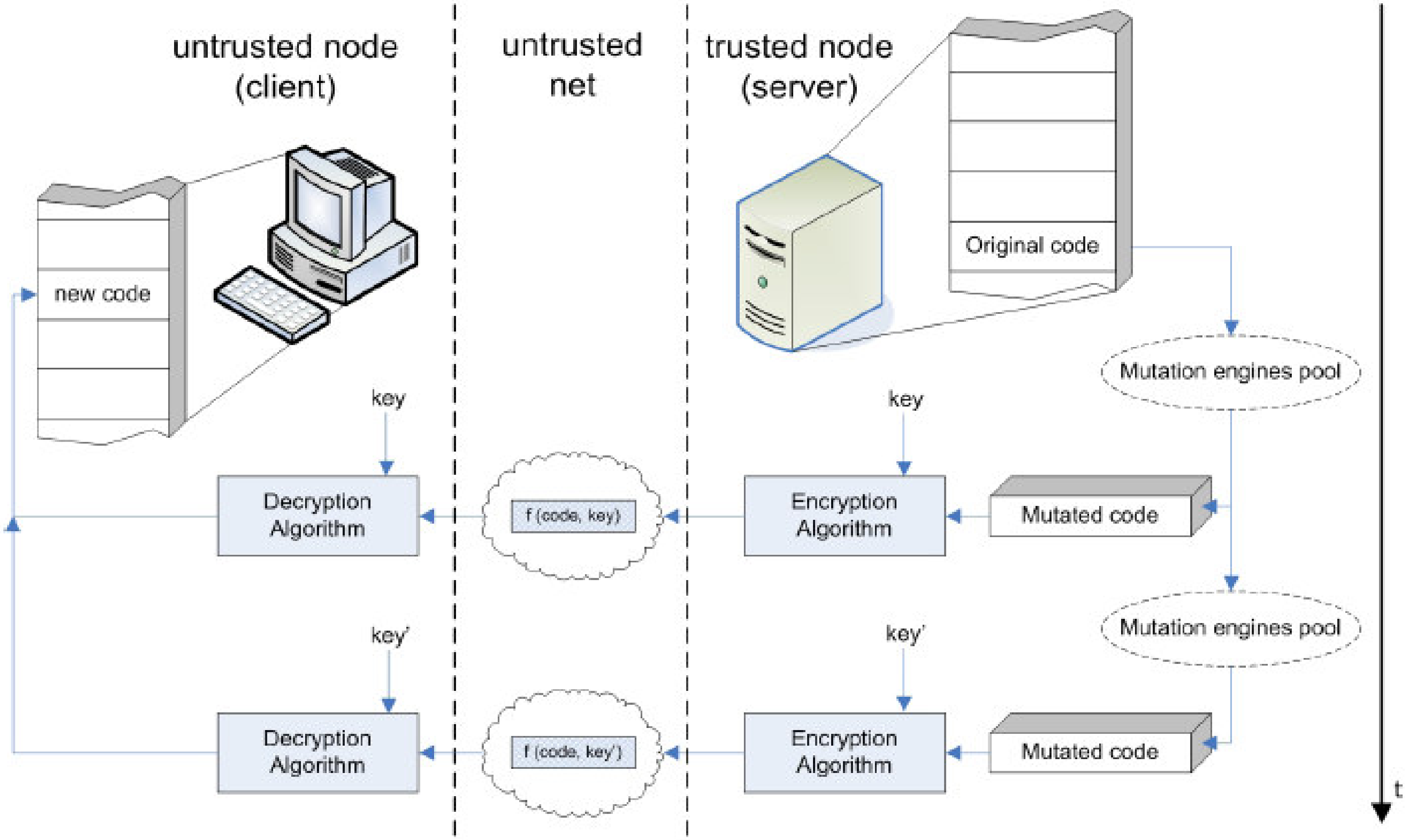
JVM and debugger

- Dynamic AOP and JVMTI rely on debugger
- 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

Problems with JVMs prototypes

- Key was embedded in mobile module
- Discovery of secret key…to calculate checksums
- Replace aspect to disable checking but sending correct tags:
  - Attacker intercepts mobile code
  - Hijack it to check original application
  - …While tampered one is running

Architecture with native code



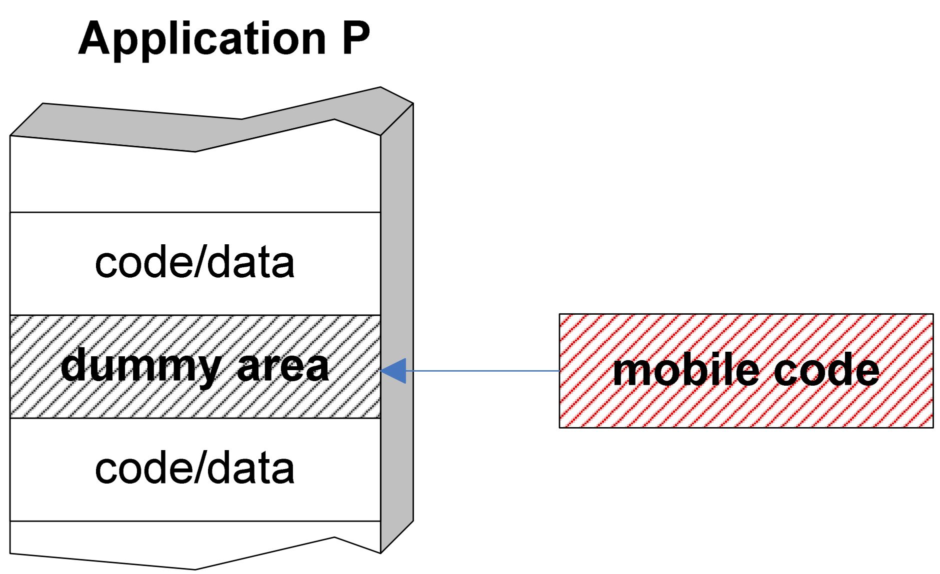
Trusted node

- It is the mobile code provider
  - It has a pool of integrity-checkers
  - Send such checkers to the untrusted node
  - The more checkers we have

    =>the more robust is the protection

Code replacement

- Binder: receives code from trusted node and insert mobile code in application memory
- A dummy area is instrumented in the application as a placeholder



Mutation Engine Pool

- Collection of Integrity checkers
  - Each one has a different algorithm
  - They can be parameterized by hash key
  - Each checker can be mutated depending on mutation rules
- Mutation Goal: attacker will find hard to automatically recognize such checkers by patter-matching
  - Similar to virus behavior

Fooling the checkers

- Van Oorschot et al. find out how to fool checkers:
  - Modified Operating System to intercept when an instruction of the application read from code segment
  - System call is modified: checkers will always check original code while tampered one is running

Self-Modifying Code

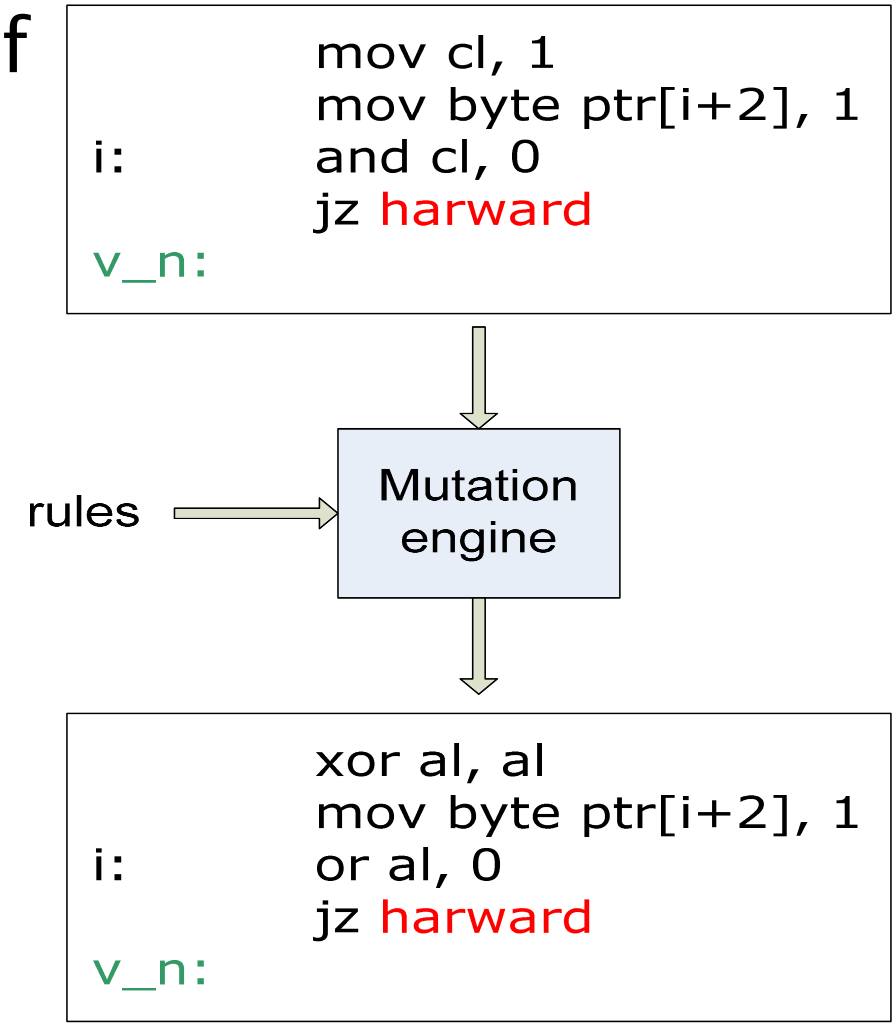
- Can be used to avoid former attack on checkers
- Self-modifying code alters its own instructions at run-time
- Data segment contain original code, used for checksum, while code segment contains code which is actually executed.
- The executable file structure is different from the one created in memory at run-time
- If attacker finds out checker function and calculate checksum on the executable files they are useless.

One step further

- Binder is embedded in application
- At load-time Binder downloads checkers and some functional code
- Then it (self-)modifies the surrounding application in order to have a new memory structure
- Executable is then different from memory image
- Then Binder can handle mobile code replacement

Example of self-modifying code

- This code modifies itself
  - and cl,1 is executed



- This is another version of the same code

Mutations on Checkers

- Modify assembler code structure without changing its behavior
- Used to produce many version of checkers
- Similar to obfuscation on assembly
- Example: recombination of operators or registry renaming

Prototype

- Protections applied:
  - Code Checksum
  - Invariant checking
- How they are combined
  - 2 Different Checkers calculate hash
  - They differ for one invariant
- Prototype tested on
  - Developed in C++
  - OS: Windows XP but working on Linux

Experimental Results

- Advantages
  - Cross-platform
  - Code relocation
  - Application structure in memory different form executable file
  - Customization for each instance
  - New Protections can be plugged in
- Weak points
  - Complex code development/instrumentation

Communication Protocol

- Authentication
  - ISO Symmetric Key Three-Pass Mutual Authentication
  - Open issues:
    - Need to save client private key on untrusted host
    - Algorithm is computationally expensive

Private Key on Client

+ Keep on server temporary key of client
+ Client uses at boot time temporary key and not the private key

- Client must save its private key and last temporary key

Key generation

- Use client code as data source
- Function embedded in mobile code arrives from server and selects a subset of bytes of code to make key
- Mobile code and its function periodically updated at run-time
- The function can be customized for each client instance

The prototype

- Key Generation
  - First communication made with key hidden by server in client executable
    - steganography
  - Client and server generate temp key using function sent by server
  - Use this temporary key for next communications
  - New module=> new key

Steganography with images

- Advantages:
  - Modify Less Significant Bits of each pixel with information to hide
  - Such modification does not damage image quality
  - Easy to implement
  - Image modifiable at run-time
- Disadvantages:
  - Image not always available in all programs

Steganography in code

- Same operations in i386-like architectures can be expressed in 2 ways:
  - Add %eax, $50
  - Sub %eax, $50
- This sequence can encode a bit
  - Add-sub -> 1
  - Sub-Add -> 0
- Disadvantages:
  - Codice read from executable file
  - File not modifiable at run-time
  - Attacker may find which memory areas are used by function in mobile code