Trust Model (D2.1/D3.1)

Thomas Herlea

Katholieke Universiteit Leuven
Department ESAT/SCD-COSIC

RE-TRUST 5th Quarterly Meeting
December 2007, Leuven

Outline

Assets

- impossible to enumerate
- depend on the application in question
- are among its **business goals**
- limit the attacks we care about

Business Goals

- still too varied
- not trivial to express
- but original program satisfies them
- assets == **properties of execution**?
- and obfuscation? CED/CEF?
- consider properties of transformed programs, too

Properties of Execution

- correctness
- sequentiality
- completeness
- limitedness
- timeliness
- execution cardinality
- ...

Secondary Assets

- adding defences adds assets
- primary assets $\Leftarrow$ original program
- secondary assets $\Leftarrow$ extra protection code
  - program confidentiality
  - crypto key confidentiality
  - protection code interlocking
  - dependence on trusted server

Trusted Elements

- the trusted server
- cryptographic primitives
- trusted client hardware
- equipment manufacturers
- certification authorities
- network access providers
- ...

Attacker Model

- goals
- capabilities
- limitations

Attacker Goals

- roughly: "break business goals"
- other goals not of interest
- refusal to execute == attack?
- typically: "break some business goals, preserve some business goals"

Attacker Capabilities

- complete control over the untrusted execution environment
- mounting environmental attacks
  - using system libraries, I/O, networking, virtualisation, ...
- mounting static attacks
  - using disassemblers, decompilers, flipping bits, changing instructions, constants, ...
- mounting dynamic attacks
  - using stepping, conditional breakpoints, changing variables and control flow, ...

Attacker Limitations

- no tampering with the trusted entities
- probabilistic polynomial time algorithm (security parameter)
- more radical tampering is more expensive
- subject to limits derived from physical laws (timing)

Outline

Strategy

We seek a formal framework that supports a unified view of:

- business goals
- attacks
- distributed execution
- trust properties

and supports reasoning about them.

Code Integrity?

Good:

- straightforward capturing of business goals
- straightforward checking (in principle)

...but not good enough:

- does not “see” environmental attacks
- may complain about harmless changes

What a Trace Is

- low level description of one program execution
- sequence of elementary computations
- both operators and operand values
- timestamped elements, for reassembling distributed traces
- inspired by theory of Abstract Interpretation
- $t \in T$, $T = \{\text{elementary computation}, \text{timestamp}\}^*$

Attitude to Traces

- seem a promising candidate
  - effective tool
  - very fine-grained
  - good for theory
- still have to prove themselves
  - possibly inefficient tool
  - not directly accessible to the defender
  - unproven in practice
- we have not yet decided on the limits

Goals and Traces

- all traces of a correct execution satisfy the defender's goals
- possibly other traces, too
- other traces do not
- model defender's goal as a **predicate**

$$
\begin{aligned}
D : T &\rightarrow \{0,1\} \\
t & \quad 1 \text{ if goals are met} \\
& \quad 0 \text{ else.}
\end{aligned}
$$

- by definition, attacker's goal is $A = \overline{D}$

Reasoning with Goals

Let's express a conjunction of business goals.

- in English: "satisfy all subgoals D_i"
- with predicates: $D = (D_1 \wedge D_2 \wedge \cdots \wedge D_n) = \bigwedge_i D_i$
- and the right of the client not to execute the program at all?
- retry in English: "satisfy all **or nothing**"
- retry with predicates:

$$D = (D_1 \wedge D_2 \wedge \cdots \wedge D_n) \vee \overline{D_1 \vee D_2 \vee \cdots \vee D_n} = \bigwedge_i D_i \vee \overline{\bigvee_i D_i},$$

- attacker: $A = \overline{D} = \overline{\bigwedge_i D_i \vee \overline{\bigvee_i D_i}} = \bigvee_i \overline{D_i} \wedge \bigvee_i D_i$
- paradox: $A \Rightarrow \bigvee_i D_i$

Execution and Traces

In Re-Trust traces depend on:

- the program's binary P
- the execution context C
- the inputs from the trusted server I_s
- the inputs from the untrusted client I_c

An "execution engine" generates the trace:

$$E(P, C, I_s, I_c) = t$$

Attacks and Traces

The actions of an attacker will change the trace:

- skipping instructions cause missing trace elements
- inserting instructions cause extraneous elements
- out of order execution causes out of order sequences of elements
- changing values in memory causes changed trace elements
- running a debugger is like inserting instructions

Tampering Formalism (tentative)

Given that $D(E(P, C, I_s, I_c)) = 1$, successful tampering means that the attacker obtains $A(E(P', C', I'_s, I'_c)) = 1$ by producing:

- $P' = P + \delta$ - a tampered binary
- C' - an instrumented context
- I'_s - a way to alter communication with the trusted server
- I'_c - a subset of the client inputs

The attack algorithm ("simulation") is denoted

$$\mathcal{S}(P, C, k) = (\delta, C', I'_s, I'_c)$$

Protection Formalism (tentative)

Re-Trust is looking for

- θ ($\theta(P)$ is program P, protected)
- and the corresponding server input I_s

such that $\mathcal{S}$ succeeds with negligible probability:

$$Pr[\mathcal{S}(\theta(P), C, k) = (\delta, C', I'_s, I'_c) | A(E(\theta(P) + \delta, C', I'_s, I'_c)) = 1] \leq neg(k).$$

If θ were able to ensure this alone, execution could be performed offline. If it existed, perfect obfuscation could be θ. In practice, it seems more feasible to use a reactive approach using I_s.

Reactive Protection (tentative)

At every step, the defender would like the trace up to now:

- to still allow D to be satisfied in the future (trace is still "D-satisfactory") and
- not to allow A to be satisfied any more in the future (the trace is already "A-unsatisfactory").

As this might not be possible, the defender would like at all times to have a "fragile trace", i.e. a trace that can turn into an "A-unsatisfactory" trace in one step:

- automatically, triggered by the slightest tampering or
- under the defender's control, by withholding the next benign piece of server input (which should be hard to guess)

Evidence and Verdicts

Evidence would have to:

- be produced by the untrusted client
- in an unforgeable way
- return one "tag" based on the current trace
- and allow the existence of a validation function.

A verdict function would have to:

- work on the sequence of tags returned during an execution
- return 1 only if the tag sequence corresponds to a "D-satisfactory" trace
- return 0 only if the tag sequence corresponds to a "D-unsatisfactory" trace

Goals Revisited

Security requirements examples

- the client should not be able to forge a trace based on other trace it has seen
- the client should not be able to inject specific additional instructions into a trace
- the server should be able to detect that tags from a client come from traces it was authorized to produce
- ensure atomicity of transactions

Performance requirements examples

- computing the verdict is cheaper than executing the original program on the server
- computing the verdict plus executing the protected program on the client should not be much more expensive than executing the unprotected program on the client

Outline

Conclusion

- Traces are powerful tool
- Traces are not well-defined on multi-threaded platforms
- Traces can require complex predicates for simple business goals
- Traces can not express the integrity of some static properties
- We believe they are a useful evaluation tool