Expressing Type-Flaw Attacks in a Strongly Typed Language

Iliano Cervesato  iliano@itd.nrl.navy.mil

ITT Industries, Inc  @  NRL - Washington DC

http://www.cs.stanford.edu/~iliano/

2nd International Workshop on Foundations for Secure/Survivable Systems and Networks

Tokyo, October 27th, 2001
Expressing Type-Flaw Attacks in a Strongly Typed Language

Outline

- Type-confusion attacks
- Type-Flaw Attacks in MSR
- Simulation with Dolev-Yao Intruder

Work in progress

Type flaws
Example
Positions
Contribution
MSR 2.0
Example
Typing
DAS
Execution
Intruder
Type flaws
Simulation
DY Intruder
Big steps
Type flaws
Type-Flaw Attacks

• Functionalities seen as “types”
  ➢ Names
  ➢ Nonces
  ➢ Keys, ...

• Violation
  ➢ Principal misinterprets data

• Type flaw/confusion attack
  ➢ Intruder manipulates message
  ➢ Principal led to misuse data
Example: NSL [Millen]

A → B: \{A, n_A\}_{kB}
B → A: \{n_A, n_B, B\}_{kB}
A → B: \{n_B\}_{kB}

\[ I \rightarrow A: \{n_A, n_B, B\}_{kB} \]

Confusion 1:
name/nonce

Confusion 2:
pair/nonce

B is fooled!

"Unlikely type violation"

Expressing Type-Flaw Attacks in a Strongly Typed Language
Type-flaw attacks are serious threats

- Push type-free specifications
  - Catch all “normal” attacks
  - ... and type-confusion attacks too
  - Types are not real!
Most type-flaw attacks are unrealistic

- Push typed specification languages
  - Catch “real” attacks
  - Types guide search ⇒ fast
  - Type-flaw attacks too low-level anyway
Prog. Languages vs. Security

- Types in programming languages
- Types in security

Expressing Type-Flaw Attacks in a Strongly Typed Language
... in Reality

Type discriminants
- Data length
- Redundancy
- Explicit checks

- Resolve many situations ...
- ... but not all

"I so far found only one realistic type-flaw attack"  [Meadows]
Contribution

- Reconcile
  - Typed languages
  - Type violations
- User specifies confusable types
  - Flexible
  - Abstract
- Support efficient simulation

Expressing Type-Flaw Attacks in a Strongly Typed Language
Expressing Type-Flaw Attacks in a Strongly Typed Language

MSR

- Follows the Dolev-Yao abstraction
- Based on
  - Multiset rewriting, linear logic, type theory
- Used to prove
  - Undecidability of protocol verification
  - Completeness of Dolev-Yao intruder
- Related to
  - strands
  - CIL
  - spi-calculus, ...

Type flaws
MSR 2.0
Example
Typing
DAS
Execution
Intruder
Type flaws
Simulation

Expressing Type-Flaw Attacks in a Strongly Typed Language
What’s in MSR 2.0?

- Multiset rewriting with existentials
- Dependent types w/ subsorting
- Memory predicates
- Constraints

Expressing Type-Flaw Attacks in a Strongly Typed Language
Expressing Type-Flaw Attacks in a Strongly Typed Language

The Dolev-Yao Model of Security

- **Symbolic data**
  - No bits
- **Black-box cryptography**
  - No guessing of keys
- **Partially abstract data access**
  - Knowledge soup
  - Found in most protocol analysis tools
    - Tractability

Found in most protocol analysis tools

- **Type flaws**
  - **MSR 2.0**
  - **Example**
    - Typing
    - DAS
  - **Execution**
  - **Intruder**
    - Type flaws
  - **Simulation**
Roles

- **Generic roles**

\[
\exists L': \tau_1'(x_1) \times \ldots \times \tau_n'(x_n)
\]

\[
\forall A \quad \exists y: \tau'.
\]

\[
\forall x: \tau.
\]

\[
lhs \rightarrow \exists y: \tau'.
\]

\[
rhs
\]

- **Anchored roles**

\[
\exists L: \tau_1'(x_1) \times \ldots \times \tau_n'(x_n)
\]

\[
\forall A \quad \exists y: \tau'.
\]

\[
\forall x: \tau.
\]

\[
lhs \rightarrow \exists y: \tau'.
\]

\[
rhs
\]

Expressing Type-Flaw Attacks in a Strongly Typed Language
Expressing Type-Flaw Attacks in a Strongly Typed Language

\[ \forall x_1 : \tau_1. \]
\[ \quad \quad \quad \quad \quad \vdots \]
\[ \forall x_n : \tau_n. \]

\[ \exists y_1 : \tau'_1. \]
\[ \quad \quad \quad \quad \quad \vdots \]
\[ \exists y_n : \tau'_n. \]

Rules

ls \rightarrow rs

- N(t)  Network
- L(t, ..., t)  Local state
- M_A(t, ..., t)  Memory
- \(\chi\)  Constraints

- N(t)  Network
- L(t, ..., t)  Local state
- M_A(t, ..., t)  Memory
Expressing Type-Flaw Attacks in a Strongly Typed Language

NSL Initiator

∀ A

∀ B: princ

∀ k_B: pubK B

• → ∃ n_A: nonce.

L(A, B, k_B, n_A) \rightarrow N({n_A, n_B, B}_k_A)

∀ k'_A: privK k_A

∀ n_A, n_B: nonce

∃ L: princ \times princ(B) \times pubK B \times nonce.

∀ B: princ

∀ k_B: pubK B

\rightarrow ∃ n_A: nonce.

L(A, B, k_B, n_A) \rightarrow N({A, n_A}_k_B)

∀ A

∀ B: {A, n_A}_k_B

B \rightarrow A: {n_A, n_B, B}_k_A

A \rightarrow B: {n_B}_k_B
Expressing Type-Flaw Attacks in a Strongly Typed Language

NSL Responder

\[
\exists L: \text{princ}(B) \times \text{pubK } B^{(KB)} \times \text{privK } k_B \times \text{nonce.}
\]

\[
\forall k_B: \text{pubK } B
\]
\[
\forall k'_B: \text{privK } k_B
\]
\[
\forall A: \text{princ}
\]
\[
\forall n_A: \text{nonce}
\]
\[
\forall k_A: \text{pubK } A
\]

\[
N(\{A, n_A\}_{KB}) \rightarrow \exists n_B: \text{nonce.}
\]

\[
\exists L(\text{princ}(B), k_B, k'_B, n_B)
\]

\[
\forall n_B: \text{nonce}
\]

\[
N(\{n_B\}_{KB}) \rightarrow \bullet
\]
Types of Terms

- A: princ
- n: nonce
- k: shK A B
- k: pubK A
- k\': privK k
- ... (definable)

Types can depend on term
- Captures relations between objects
Subtyping

- Allows atomic terms in messages

- Definable
  - Non-transmittable terms
  - Sub-hierarchies

\[
\text{princ} :: \text{msg} \\
\text{nonce} :: \text{msg} \\
\text{pubK A} :: \text{msg}
\]
Type Checking

\[ \Sigma \vdash P \]

\[ \Gamma \vdash \cdot: \tau \]

- **Catches:**
  - Encryption with a nonce
  - Transmission of a long term key
  - Circular key hierarchies, ...

- **Static and dynamic uses**

- **Decidable**

**Expressing Type-Flaw Attacks in a Strongly Typed Language**
Data Access Specification

- **Catches**
  - $A$ signing/encrypting with $B$'s key
  - $A$ accessing $B$'s private data, ... 

- **Static & Decidable**

- **Gives meaning to Dolev-Yao intruder**
  - Completeness
  - Reconstructibility

$\Sigma \vdash A \Gamma \vdash r$

$r$ is DAS-valid for $A$ in $\Gamma$

$P$ is DAS-valid in $\Sigma$
Configurations

\[ C = [S]^R_\Sigma \]

State
- \( N(t) \)
- \( L_1(t, \ldots, t) \)
- \( M_A(t, \ldots, t) \)

Active role set

Signature
- \( a : \tau \)
- \( L_1 : \tau \)
- \( M : \tau \)

Expressing Type-Flaw Attacks in a Strongly Typed Language
Execution Model

\[ P \rightarrow C \rightarrow C' \]

- Activate roles
- Generates new role state pred. names
- Instantiate variables
- Apply rules
- Skips rules
Variable Instantiation

\[ \Sigma \vdash \dagger : \tau \]
\[ \sum [S]^R (\forall x: \tau. r, \rho)^A \sum \rightarrow [S]^R ([\dagger/x]r, \rho)^A \sum \]

Type checking guarantees proper usage
Rule Application

\[ r = F, \chi \rightarrow \exists n : \tau. G(n) \]

- **Constraint check**
  \[ \Sigma \models \chi \]  
  (constraint handler)

- **Firing**
  \[
  [S_1]^{R(r,\rho)^A} \Sigma \rightarrow [S_2]^{R^\rho A} \Sigma, c : \tau
  \]
  \[ S, F \]
  \[ S, G(c) \]
  \[ c \text{ not in } S_1 \]

Expressing Type-Flaw Attacks in a Strongly Typed Language
Execution with an Attacker

\[ P, P_I \triangleright C \rightarrow C' \]

- **Selected principal(s):** \( I \)
- **Generic capabilities:**
  - Well-typed
  - DAS-valid
- **Modeled completely within MSR**

Expressing Type-Flaw Attacks in a Strongly Typed Language
Expressing Type Violations?

- Impossible!

\[ \Sigma \vdash \dagger : \tau \]

\[ [S]^R (\forall x: \tau.r, \rho)^A \Sigma \rightarrow [S]^R ([\dagger/x]r, \rho)^A \Sigma \]

Typing forces principal to play by the rules
Expressing Type Violations!

Distinguish

- Static type-checking
- Dynamic type-checking

Expressing Type-Flaw Attacks in a Strongly Typed Language
Subtyping Revisited

• Most rules have a rigid format

• Subtyping provides hook

\[
\begin{align*}
\tau' :: \tau & \quad \Gamma \vdash \tau : \tau' \\
& \quad \Gamma \vdash \tau : \tau \\
& \quad \Gamma \vdash \tau : \tau \\
\end{align*}
\]

Extend subtyping with confusible types

Expressing Type-Flaw Attacks in a Strongly Typed Language
A First Solution

princ :: msg
nonce :: msg
pubK A :: msg
classic intrusion

princ :: nonce
dynamic extension

msg :: nonce

• Works ...
• ... but very raw
  - not every msg mistaken as a nonce
  - unwanted recursion

Expressing Type-Flaw Attacks in a Strongly Typed Language
Towards a Polymorphic MSR

princ :: msg
nonce :: msg
pubK A :: msg
pair α β :: msg
nonce+ :: msg
princ :: nonce+
nonce :: nonce+
pair princ nonce :: nonce+

Confusable nonces

pair : type -> type -> type.
_,_ : α -> β -> pair α β.

Fine grained
Captures what we want
Recursion is up to us

Expressing Type-Flaw Attacks in a Strongly Typed Language
Summary

• Type violation (attacks) expressible in MSR

• Simple

• Flexible
  - You decide confusable types
  - Shades of gray in black/white positions

Types are good
Simulation

- No attacker
  - Prototype

- With attacker
  - Verification
    - Model checking
    - Theorem proving
    - Process equivalence
The Dolev-Yao Intruder

- Intercept / emit messages
- Decrypt / encrypt with known key
- Split / form pairs
- Look up public information
- Generate fresh data
- Found in most protocol analysis tools
- Modeled completely within MSR
- Generated automatically (mostly)
Intruder Simulation Approaches

- Take protocol text into account?
  - Blind / Focused

- Size of intruder steps
  - Small / Big

- Intruder representation
  - Explicit / Implicit

Expressing Type-Flaw Attacks in a Strongly Typed Language
Expressing Type-Flaw Attacks in a Strongly Typed Language

Graphically...

- MSR
- Paulson
- SPI
- NPA
- Strands
- MSR, ...
- Casper
- CAPSL

Good for proving theorems

Type flaws
MSR 2.0
Simulation

DY Intruder
Big steps
Type flaws
Intruder Activity

No need to remember

No need to construct
Intruder Activity Comparison

Disassembly
- Blind
  - Take pieces apart until
    - Atomic
    - Key unavailable
- Focused
  - Anticipate message contents
  - Memorize only what is needed

Assembly
- Blind
  - Put pieces together until meaningful message is built
- Focused
  - Build only usable messages
Big-Step Message Disassembly

- Take typing derivation of (incoming) messages
- Encryption defines regions
- 1 role for each message
  - 1 rule for each region
  - Interface rule

\[ \Gamma \vdash t : \tau \]
**NSL – 1st Message**

\[ \exists \forall: \text{princ} \times \text{msg.} \]

\[ \forall m: \text{msg} \quad N(m) \quad \rightarrow \quad L(I, m) \quad M_I(m) \]

\[ \forall A: \text{princ} \quad L(I, \{A, n_A\}_{kB}) \quad \rightarrow \quad M_I(A) \quad M_I(n_A) \quad M_I(k'_B) \]

\[ \forall k_B: \text{pubK B} \quad M_I(\{A, n_A\}_{kB}) \quad \rightarrow \quad M_I(n_A) \quad M_I(k'_B) \]

\[ \forall k'_B: \text{privK k}_B \quad M_I(k'_B) \]

\[ \forall n_A: \text{nonce} \]

- \(M_I(m)\) “forgotten” as soon as \(k'_B\) is known
- Special case if \(k'_B\) known right away

Expressing Type-Flaw Attacks in a Strongly Typed Language
Big-Step Message Assembly

- Take typing derivation of (outgoing) messages
- Encryption defines regions
- 1 role for each region
- Extras for generated data

\[ \Gamma |- t : \tau \]
NSL – 1st Message

\[
\begin{align*}
\forall m: \text{msg} & \quad M_I(m) \to N(m) \\
\forall A, B: \text{princ} & \quad M_I(A) \to N(\{A, n_A\}_{k_B}) \\
\forall k_B: \text{pubK B} & \quad M_I(n_A) \to M_I(A), M_I(n_A), M_I(k_B) \\
\forall n_A: \text{nonce} & \quad M_I(k_B) \to M_I(A), M_I(n_A), M_I(k_B) \\
\forall A, B: \text{princ} & \quad M_I(A) \to \exists n_A: \text{nonce.} \\
\forall k_B: \text{pubK B} & \quad M_I(k_B) \to N(\{A, n_A\}_{k_B}) \\
\end{align*}
\]

What about confusable types?

Expressing Type-Flaw Attacks in a Strongly Typed Language
Creating Confusion

- Mark confusable objects
- Add rules for each option

\[
\exists L: \text{princ } \times \text{nonce}^+.
\]

\[
\forall C: \text{princ} \quad M_I(C) \rightarrow \angle(I, (C,n))
\]

\[
\forall n: \text{nonce} \quad M_I(n) \rightarrow \angle(I, n), M_I(n)
\]

\[
\forall C: \text{princ} \quad M_I(C) \rightarrow \angle(I, C), M_I(C)
\]

\[
\forall A,B: \text{princ} \quad \angle(I,n)
\quad M_I(A) \rightarrow N([A, n]_{k_B})
\quad M_I(k_B) \rightarrow M_I(A), M_I(k_B)
\]

Expressing Type-Flaw Attacks in a Strongly Typed Language
Making Sense of Confusion

\[ \exists L: \text{princ} \times \text{msg.} \quad \exists L': \text{princ} \times \text{nonce}^+ \]

\[ \forall m: \text{msg} \quad N(m) \quad \rightarrow \quad L(I, m) \quad M_I(m) \]

\[ \forall A: \text{princ} \quad L(I, \{A, n_A\}_{k_B}) \quad \rightarrow \quad M_I(A) \quad L'(I, n_A) \quad M_I(k'_B) \]

\[ \forall n: \text{nonce} \quad L'(I, n) \quad \rightarrow \quad M_I(n) \]

\[ \forall A: \text{princ} \quad L'(I, A) \quad \rightarrow \quad M_I(A) \]

\[ \forall A: \text{princ} \quad L'(I, (A, n)) \quad \rightarrow \quad M_I(A) \quad M_I(n) \]

Expressing Type-Flaw Attacks in a Strongly Typed Language
Further Optimizations

- Fold added rules in (unless confusion type is recursive)
  - Type-check in static type system
  - Bigger steps

- Simplify result using DAS rules
  - More compact
  - Formalizes “regions”
  - Automation
Future Work

- Polymorphic MSR
- Strategies