Applications of language-based security

Thanks to Andrei Sabelfeld for making his lecture slides available, on which much of this material is based.

Web services

- Download software (e.g., *Quicken* for tax)
- Billed as you compute (hypothetical model) – depending on complexity of tax calculation (multi-state, Enron, etc.)
- Hooks for automatic updates

Another example: File sharing services for file exchange by users; for example, music files.

In reality…“scumware” (AS)

Downloaded software often arrives with:

- Pop-up ads (unsolicited)
- Execution of arbitrary, unchecked code
- Data mining operations (attack on privacy): How do you know Quicken is not copying your private information, e.g., SSN, salary, when you send data for billing ?

Tension between contradictions

On the one hand:

- Object-oriented programming: widely adopted because of promise of reusable software components.
- Languages like Java and C# allow extensible components to be used in many contexts (platform independence). Large software assembled using “components”.

On the other:

- Excellent opportunities for attackers
  - Easy to distribute worms, viruses, etc.
  - Attack once, run everywhere.

Critical to ensure the security of information flowing between multiple sites, but do not use a sledgehammer.

Defenses against malicious code (Torben)

Several approaches, based on:

- Analysis
- Rewriting
- Monitor
- Audit trails

to enforce *security policies* like:

- Confi dentiality (sensitive data should not flow to untrusted site)
- Integrity (untrusted data should not flow to trusted site)
- Availability (of resources/services)
- Accountability (who authorized/performed a sensitive operation).

The state of the practice

- Access Control: Prevents unauthorized release of information. But not *propagation* of information once access is granted.
- Firewalls: Permit selective communication.
- Encryption: Secures communication channel. But leaks possible at end points.
- Antivirus scanning: Rejects known attacks. Defenseless against new attacks.
- Digital signatures: Authenticates code producer. But what is the security guarantee? What security policy can it support?
- Sandboxing: Do not allow foreign code to perform sensitive operations. Inflexible – this is why JDK changed to stack inspection.

Confidentiality

- Want *End-to-end* confidentiality: there is no insecure information flow in the application.
- Standard security mechanisms provide no end-to-end guarantees.
- An application is a *program*: hence look inside the program. This yields:
  - *Semantics-based security specification*: robust semantic specification of end-to-end security policies; strong reasoning principles about program semantics.
  - *Static security analysis*: enforcement of end-to-end security policies; specification of analysis as a *security type system*... compile time type checking.

Security type system

Idea: Attacker should not be able to view changes in sensitive data. So classifying data as ℓ (for Low) and h (for High), want to disallow “bad flows”:

- $\ell :=$ h
- if h then ℓ := 0 else $\ell :=$ 1

Attacker is considered Low.

Semantically, what policy is guaranteed to hold?

The JDK getSigners bug

```
public class Class {
    private Identity [] signers;
    public  Identity[] getSigners() { return signers;}
}
```

The call to Class.getSigners() can be used to create an alias between the *private* array signers and a malicious client. Then the client can install itself as a valid signer by updating the alias.