A Web Security Scenario

Client (web browser) -> Web Server: URL Request
A Web Security Scenario

Client (web browser)

Web Server

web page
A Web Security Scenario
A Web Security Scenario

Client (web browser)

URL request

Web Server

Ad Network

Ad Server
A Web Security Scenario

Client (web browser)

Web Server

Ad Network

Ad Server
A Web Security Scenario

- Four principals: client, page publisher, ad network, ad publisher
- What are some security requirements each principal is likely to have?
- Which existing technologies can be used to meet those requirements?
- How can we assess/measure the “security” of the resulting system?
Trust

• Trust model: Who trusts whom to do what?
• Trusted Computing Base (TCB): The set of all system components that must be trusted in order to maintain system security
  – Security meta-goal: minimize the TCB
• What is the trust model in our web scenario?
• What is the TCB? How can we make it smaller?
Attacks / Threats

• Threat model: set of assumed attacker capabilities
  – attacks outside the model may succeed!
  – threat model assumptions = security system limitations

• What is a reasonable threat model for our web scenario?
The Security Crisis

• NVD Vulnerability Disclosures by year

– These are just the “big ones” that make the news!

• MITRE CVE Top “Unforgivable” Vulnerabilities:
  – buffer overflow
  – XSS
  – SQL injection
  – directory traversal
  – world-writable files
  – direct admin script requests
  – homegrown crypto
  – authentication bypass
  – large check-use windows
  – privilege escalation
  – undocumented account
  – integer overflow

• Why do these still occur? Why do standard approaches fail?
Security Crisis and TCB Minimization

• Let’s play a game: I’m thinking of a piece of software.
  – Most of you have it and have used it.
  – If it fails, it could delete or divulge all your personal files.
  – Microsoft makes it.
  – Can you guess which software I’m thinking of?
Misguided Fixes

• People who haven’t studied the field think the solution is “obvious”:
  – Naïve idea #1: “If everyone just used linux…”
  – Naïve idea #2: “Just configure your permissions properly!”
  – Naïve idea #3: “Just use RBAC/XACML/latest fancy access control model.”

• In reality, two hard issues involved:
  – Minimal Trusted Computing Base
  – Principle of Least Privilege
Least Privilege

• **Principle of Least Privilege**: “Every program and every user of the system should operate using the least set of privileges necessary to complete the job.” [Saltzer & Schroeder, 1975]

• Hard problem: What is the least set of privileges necessary to complete the job? How do we compute it?

• No finite set of roles or permission options suffices to meet PoLP in all cases!

• Richer classes of enforceable policies get us closer, though.
Major Classes of Security Policies

• **Integrity** – preventing improper or unauthorized change to data or resources
  – Example: ad may not delete your files

• **Availability** – continued access to data or resources
  – Example: ad may not expand to occlude the rest of the page

• **Confidentiality** – concealment of data or resources
  – Example: ad may not send your browsing history to your employer
Defining Policies Formally

- **Security Policy** – specification of allowed (or, equivalently, disallowed) behaviors
  - **Safety Policies** – some “bad” thing shouldn’t happen (integrity)
  - **Liveness Policies** – some “good” thing should eventually happen (availability)

- **Safety + Liveness = all policies** [Alpern & Schneider, 1985]
Important LBS Technologies

• Automated theorem-provers
  – machine-assisted, machine-checked proofs of security

• In-lined Reference Monitors
  – insert dynamic security checks into untrusted code

• Type-checkers
  – advanced type systems can encode security properties

• Model-checkers
  – statically verify that code model obeys a security property

• Certifying Compilers
  – transform source code into object code and an independently verifiable proof that the object code is safe to execute
Software Lifecycle

- **Design & Development**
  - Security vulnerabilities in non-malicious code
    - type-safe programming languages
    - formal verification
    - code synthesis
  - Malicious code (viruses, worms, etc.)

- **Deployment (Download, Install, Load)**
  - Antivirus scanning
  - Code-signing
  - Type-safe target codes (e.g., Java bytecode)
  - Independently verifiable certificates

- **Execution**
  - Runtime monitoring
  - Automatically generated self-monitoring code

- **Recovery**
  - Auditing (logging)
  - Rollback (reversible computation, restore points)
  - Legal action
An Example: Memory Safety

- Memory safety: ?

- Traditional security model:
  - program is a black box
  - OS/hardware intercepts every memory access

- Language-based security model:
  - program is a sequence of instructions
  - analyze the sequence to identify potential violations
  - insert dynamic memory checks into program
An Example: Memory Safety

- Memory safety: Programs may not access unallocated memory addresses

- Traditional security model:
  - program is a black box
  - OS/hardware intercepts every memory access

- Language-based security model:
  - program is a sequence of instructions
  - analyze the sequence to identify potential violations
  - insert dynamic memory checks into program
// Load a user-specified webpage. Prepend “http://”
// to the url if necessary.
void loadurl(char *url)
{
    char buffer[8192];

    if (strncmp(user_input, "http://", 7))
    {
        sprintf(buffer, "http://%s", user_input);
        getwebpage(buffer);
    }
    else getwebpage(url);
}
The stack grows downward.

- `char *url` (4 bytes)
- `return address` (4 bytes)
- `old stack pointer` (4 bytes)
- `char[] buffer` (8192 bytes)
Reasons for the Language-based Approach

• **Rigor**
  – We have a science of programming languages.
  – Lets us prove things about software.

• **Efficiency**
  – enforce security “from inside” the software
  – fewer context switches, smarter security checks

• **Flexibility**
  – no need for custom OS/hardware

• **Power/expressiveness**
  – can enforce history-based policies
  – enforce confidentiality and not just integrity
History-based Policies

• policy: no network-sends after file-reads
• Traditional approach: none?
• Language-based approach: In-lined Reference Monitors
  – perform an automated program transformation:
  – inject a new state bit: `send_ok := 1`
  – after each file-read instruction, add an assignment: `send_ok := 0`
  – before each send instruction, add a guard: 
    `if (!send_ok) then throw SecurityException`
  – new program satisfies policy!
Information-flow Policies

• policy: don’t divulge my credit card number
• Traditional approach:
  – monitor outgoing network traffic
  – block any transmission containing the relevant bit sequence
• Language-based approach:
  – analyze dataflow of program
  – reject flows from high-security sources to low-security sinks (e.g., network sends)
Decidability

• Is this really possible with a static analysis?
• The halting problem
  – Exercise: Reduce memory safety to halting problem
• Escape hatches:
  – conservative rejection
  – limit the domain (e.g., Java bytecode only)
  – dynamic checks