Integrity and Confidentiality in Web Applications

Jeff Johnson
Where We're At

• We have been mostly focusing on
  – Fixing vulnerabilities inherent to languages
  – Finding/protecting against programmer follies

• Today we will look at security at a slightly different angle
  – Assume we have good programmers
  – Concern ourself with the environment in which the code is run
  – Focus on trust
Security in Web Apps

Client
- Fast Response
- Untrusted
- Exposed to User

INTERNET

Server
- Slow Response
- Trusted
- Secret
function submitRPC(s) {
    if (validate(s)) {
        ajaxWrapper.submit(s);
        box.display("submitted!");
    } else { // bad input
        box.display("bad input");
    }
}

FireBug and/or Proxy

function submitRPC(s) {
    //if (validate(s)) {
    //    ajaxWrapper.submit(s);
    //    box.display("submitted!");
    //} else { // bad input
    //    box.display("bad input");
    //}
}
Attack From the Client-Side

Bottom Line:

NEVER trust ANYTHING Client gives you
NEVER store secrets at the Client
Web Apps: Current State of the Art

• Write two closely-coupled programs glued together with AJAX
  – Client-side: Javascript
  – Server: Java, C#, etc.

• Programmer reasons about security
  – Who should do what computations?
    • Remember: the client-side is so very responsive but so very untrusted
  – What data goes where?

• This is difficult for many reasons
Alleviating Stress: Two Approaches

- **Swift**
  - Make the app's security requirements explicit
  - Use static checking to enforce
  - Secure by design

- **Ripley**
  - Replicate untrusted computation in a trusted environment
  - Compare results to detect misbehaving clients
Secure Web Applications via Automatic Partitioning

Swift

• Key insight: reasoning about security is difficult because it is:
  • Across two entities
  • Not explicit
  • Not enforced or checked

• Swift says: Write **one** program using security labels
  • Enforce security with a static checker
  • Compiler: split code between client and server based on labels
1) Write source with security labels (in Jif programming language)

2) Check labels, convert to WebIL (what can go on client, server?)

3) Determine placement, convert to JS and Java

4) Runtime
Write Source Using Jif

- Java-like syntax
- Plus labels for specifying read/write capabilities between 'principals'

Confidentiality policy

int {bob -> alice} x;

Integrity policy

String {bob <- alice} s;
Can strengthen/weaken policies as needed (dangerous but necessary):

- **declassify**
  change confidentiality of an expression or a statement

- **endorse**
  changes integrity of an expression or a statement
Jif in Swift

• Two principals
  – “*” denotes server
  – “client” denotes client

• * is trusted
  – In Jif terms: * actsfor client

• Bottom line
  – client sees data no greater than {* -> client}
  – client produces data no greater than {* <- client}
Example: Labels

```c
int secret;
int count;
...
void guess(int i)
{
    if (i == secret) {
        messagebox.display("winner");
    } else {
        count++;
        messagebox.display("looser");
    }
}
```
void guess(int i)
{
    if (i == secret) {
        messageBox.display("winner");
    } else {
        count++;
        messageBox.display("looser");
    }
}
Labeling Methods

```plaintext
text {-->*; <--*} secret;
int {-->client; <--*} count;
...
void guess{-->client}(int i)
where authority(*)
{
    if (i == secret) {
        messagebox.display("winner");
    } else {
        count++;
        messagebox.display("looser");
    }
}
```
int {*->*; *<-*} secret;
int {*->client; *<-*} count;
...

void guess{*->client} (int i)
where authority(*)
{
    if (i == secret) {
        declassify({*->*} to {*->client})
        messageBox.display("winner");
    } else {
        count++;
        declassify({*->*} to {*->client})
        messageBox.display("looser");
    }
}
Endorse

```c
int {*->*; *<-*} secret;
int {*->client; *<-*} count;
...
void guess{*->client}(int i)
where authority(*)
{
    if (i == secret) {
        declassify({*->*} to {*->client})
        messagebox.display("winner");
    } else {
        endorse(i, {*<->client} to {*<-*})
        count++;
        declassify({*->*} to {*->client})
        messagebox.display("looser");
    }
}
```
Next Step: WebIL

• Analyze source and verify the security specified by labels
• Determine what data/computations can go where by transforming to WebIL
• Not committing to placement yet
• This step enforces and guarantees our security model expressed in source
Example: Source to WebIL

```c
int {*-}; {*<} secret;
int {*-}client; {*<} count;
...
void guess{*-}client(int i)
where authority(*)
{
    if (i == secret) {
        declassify({*-} to {*-}client)
        messagebox.display("winner");
    } else {
        endorse(i, {*<}client to {*<})
        count++;
        declassify({*-} to {*-}client)
        messagebox.display("looser");
    }
}
```

C – Client
S – Server
? – Optional
h – high integrity
Next Step: Partitioning

• Goal: Place code to optimize performance without harming security

• Main cost is network traffic

• Approach:
  • Approximate weighted control flow graph over the whole program
  • Translate into an integer programming problem
  • Reduce to maximum flow problem
  • Solve
Partitioning

Message cost of edge $e$:

$$W_e \times (X_e + Y_e)$$

where

$X_e = 1$ if STMT1 is on Client and STMT2 is on Server

$Y_e = 1$ if STMT1 is on Server and STMT2 is on Client

Minimize sum of message costs over the whole CFG while keeping the security policies in place

See the paper for more details...
Result of Partitioning

block1 (S):
  if (i == secret) goto block2;
  else goto block3;

block3 (SC):
  count++;
  goto block4;

block5 (SC):
  // end

SERVER

block2 (C):
  messagebox.display("winner");
  goto block5;

block3 (SC):
  count++;
  goto block4;

block4 (C):
  messagebox.display("looser");
  goto block5;

block5 (SC):
  // end

CLIENT
Last Step: Translation

Located WebIL code

Java client code

Javascript client code

Swift client runtime

GWT runtime library

Java servlet framework

Swift server runtime

Java server code

Web Browser

Web Server

HTTP
Swift Runtime

Program point 1

Program point 2
## Evaluation

<table>
<thead>
<tr>
<th>Example</th>
<th>Jif</th>
<th>Java target code</th>
<th>JavaScript</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Server</td>
<td>Client</td>
</tr>
<tr>
<td>Null program</td>
<td>6 lines</td>
<td>0.7k tokens</td>
<td>0.6k tokens</td>
</tr>
<tr>
<td>Guess-a-Number</td>
<td>142 lines</td>
<td>12k tokens</td>
<td>25k tokens</td>
</tr>
<tr>
<td>Shop</td>
<td>1094 lines</td>
<td>139k tokens</td>
<td>187k tokens</td>
</tr>
<tr>
<td>Poll</td>
<td>113 lines</td>
<td>8k tokens</td>
<td>17k tokens</td>
</tr>
<tr>
<td>Secret Keeper</td>
<td>324 lines</td>
<td>38k tokens</td>
<td>38k tokens</td>
</tr>
<tr>
<td>Treasure Hunt</td>
<td>92 lines</td>
<td>11k tokens</td>
<td>11k tokens</td>
</tr>
<tr>
<td>Auction</td>
<td>502 lines</td>
<td>46k tokens</td>
<td>77k tokens</td>
</tr>
</tbody>
</table>
Discussion

• The good
  – Security policies are explicit and checked
  – One program makes it easier to reason about security
  – Minimizes network traffic

• The bad
  – Labeling is verbose and slightly confusing (~20-30% of lines are annotated)
  – Difficult to retrofit legacy code (that may already be split)
  – Still only as good as the programmer's ability to reason about security
Do Better?

• Can we ensure confidentiality and integrity without creating extra work for the programmer?
• Confidentiality – no (why not?)
  – programmer must mark confidential information as being such
• Integrity – yes
Integrity

• Integrity is directly tied to the location of computation
• Solution: move untrusted computation to trusted location
  – Move client-side computation to server
• But then we loose performance...
  – Better: replicate client-side computations on server, compare results
Ripley: Automatically Securing Web 2.0 Applications Through Replicated Execution

K. Vikram, A. Prateek, B. Livshits
Ripley Execution Model

CLIENT

- Client Events (Clicks, etc.)
- COMPUTATION (Generate RPC)
- Send RPC to Server

SERVER

- Replay Events
- COMPUTATION (Generate RPC)
- Compare Results
  - MATCHES: Pass to Server
  - DOESN'T MATCH: Terminate Session
What Ripley Does and Doesn't Do

• Ripley ensures integrity of Client-run code
• Protects against any attacks involved in manipulating client-side code/state that results in malformed RPCs
  – Remember example from before
• Not for protecting against malformed input that is accepted by the client and server code
  – Ripley protects the developer-intended protection for the application
Implementation Workflow

Volta Program

Client (C) .NET IL

Server (S) .NET IL

Mirror (C') .NET IL

Client (C) JS

S + Ripley

TIER-SPLITTING

INSTRUMENT to capture events

IL to JS

ADD Ripley layer
Writing one program (similar to Jif)

```java
class C1 {
    void foo() { ... }
    void bar() { ... }
}

[RunAt("server")]
class C2 {
    void baz() { ... }
    void faz() { ... }
}
```
Volta Advantages

• Write one program
• Glue together with RPCs
• No funny JS (e.g. innerHTML editing)
• Produces fast Client mirror (C) in .NET IL
C' Instrumentation

- Intercept *primitive* and *custom* events via bytecode rewriting
- Transfer events to server
  - Batch: Queue events, send packet-sized set
  - Flush queue when client generates an RPC call

Client (C') JS

Event \{ type, DOM object id, other info \} → RPC
Adding Ripley Checker to S
Replicating C' execution at C

• Run C as Jit-ed .NET IR
• Use lightweight browser emulator
  – headless
  – no rendering or layout computation
  – simple DOM manipulation interface
  – Each DOM node has unique id for event replay
Evaluation
Hotmail
 Extras and Optimizations

• 0-latency RPCs
• MAC-ing RPCs
• Dependency analysis
Discussion and Conclusion