Self-propelled Instrumentation

Wenbin Fang
Paradyn Project

Paradyn / Dyninst Week
College Park, Maryland
March 26-28, 2012
Overview

- **Self-propelled instrumentation**
  - Inject a fragment of code into the target process
  - Instrumentation automatically follows control flow
    - Within process
    - Cross process / host boundary

- **Features**
  - 1st party execution in the same address space
  - On demand, function call level instrumentation

- **Applications**
  - Fault diagnosis, security analysis, …
Big Picture

- **Injector**: Process to inject shared library
- **Agent**: Shared library
  - User-specified payload functions
  - Instrumentation engine

---

Application Process

- a.out
- libc.so
- libpthread.so
- Agent.so
- Payload Functions
- Instrumentation Engine
void payload(SpPoint* pt){
    // Do something
}

void main () {
    pthread_create...
    printf(...)
}

void foo () {
    connect(...)
}
Reincarnation

Previous life (2006)
- Parallel implementation with Dyninst
- x86 Linux
- For research on fault diagnosis [1]

Reincarnation (2012)
- Leverage Dyninst toolkits
- x86/x86_64 Linux
- Initially for security analysis, also for more applications

Injector

- **Inject a shared library before a process executes**
  - Use dynamic linking to load agent (LD_PRELOAD)

- **Inject a shared library for a process in execution**
  - Use Dyninst toolkit (ProcControlAPI)
Agent – Contents

- **User-provided payload functions**
  - Entry payload: executed before each function call
  - Exit payload: executed after each function call

- **Instrumentation Engine**
  - Mainly based on PatchAPI
  - Also based on other Dyninst toolkits, e.g., Stackwalker, InstructionAPI, …
Agent – In payload function

- **Query PatchAPI CFG structures**
  - Functions / Basic blocks / Edges
  - Enables sophisticated code analysis

- **Query runtime information related to the current function call**
  - Arguments / Return value

- **Detect system events**
  - Communication events, e.g., send/recv
  - Security events, e.g., setuid
Example: Print func names and # of calls to printf

```c
Agent.so

void payload(SpPoint* pt) {
    // Print Callee Name
    // Print count of printf
}

void main () {
    printf(...);
    foo();
    ...
}

void foo() {
    printf();
}
```

Output:

CALL: `printf`  
# `printf`: 1

CALL: `foo`

CALL: `printf`  
# `printf`: 2
Payload functions

Single-threaded, single process

```c
int count = 0;

void payload(SpPoint* pt) {
    PatchFunction* f = Callee(pt);
    if (!f) return;
    string name = f->name();
    printf("CALL:%s\n", name.c_str());
    if (name.compare("printf") == 0) {
        ++count;
        printf("# printf: %d\n", count);
    }
    Propel(pt);
}
```

Multi-threaded, single process

```c
int count = 0;
SpLock count_lock = 0;

void payload(SpPoint* pt) {
    PatchFunction* f = Callee(pt);
    if (!f) return;
    string name = f->name();
    printf("CALL:%s\n", name.c_str());
    if (name.compare("printf") == 0) {
        sp::Lock(&count_lock);
        ++count;
        sp::Unlock(&count_lock);
        printf("# printf: %d\n", count);
    }
    Propel(pt);
}```
Intra-process propagation

### Agent.so

<table>
<thead>
<tr>
<th>Function</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>83f0:</td>
<td>push %ebp</td>
</tr>
<tr>
<td></td>
<td>83f1:</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td></td>
<td>83f3:</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>8400:</td>
<td>jmp Patch1</td>
</tr>
<tr>
<td>foo</td>
<td>8430:</td>
<td>push %ebp</td>
</tr>
<tr>
<td></td>
<td>8431:</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td></td>
<td>8433:</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>8444:</td>
<td>jmp Patch2</td>
</tr>
<tr>
<td></td>
<td>8449:</td>
<td>mov %ebp,%esp</td>
</tr>
<tr>
<td></td>
<td>844b:</td>
<td>xor %eax,%eax</td>
</tr>
<tr>
<td></td>
<td>844e:</td>
<td>pop %ebp</td>
</tr>
<tr>
<td></td>
<td>844f:</td>
<td>ret</td>
</tr>
</tbody>
</table>

### Patch1
- call payload(foo)
- call foo
- jmp 0x8405

### Patch2
- call payload(printf)
- call printf
- jmp 0x8449
Intra-process propagation challenges

- Instrumentation Engine
  - Call insn size < jump insn size
  - Call block size < jump insn size
  - Cannot find a suitable springboard

- Relocate call block
- Find springboard block
- Use trap or ignore
Example: Print remote process name

Host 1 - 192.168.0.2 : 5370

Agent.so

void payload(SpPoint* pt) {
   // Print remote process Name
}

void main () {
   connect(…)
   recv(…)
   send(…)
}

Output for 192.168.0.2:5370:
CONNECT: (192.168.0.2:5370, 192.168.0.31:8080)
READ FROM: 192.168.0.31:8080
WRITE TO: 192.168.0.31:8080

Call payload

Host 2 - 192.168.0.31 : 8080

Agent.so

void payload(SpPoint* pt) {
   // Print remote process name
}

void main () {
   accept(…)
   send(…)
   recv(…)
}

Output for 192.168.0.31:8080:
ACCEPT: (192.168.0.31:8080, 192.168.0.2:5370)
WRITE TO: 192.168.0.0.2:5370
READ FROM: 192.168.0.0.2:5370

Call payload
Payload functions for IPC

```cpp
void payload(SpPoint* pt) {
  PatchFunction* f = sp::Callee(pt);
  if (!f) return;

  if (IsConnect(f)) {
    fprintf(fp, "CONNECT: (%s, %s)\n", 
        LocalProcessName(),
        RemoteProcessName());
  } else if (IsIpcWrite(f)) {
    fprintf(fp, "WRITE TO: %s\n", 
        RemoteProcessName());
  } else if (IsIpcRead(f)) {
    fprintf(fp, "READ FROM: %s\n", 
        RemoteProcessName());
  }
  sp::Propel(pt);
}
```

Utilities
- Detect communication events.
- Get process name
  - e.g., 192.168.0.2:8080

IPC mechanisms
- Pipe
- TCP
Inter-process propagation

Main procedure for inter-process propagation

1. Detect the initiation of communication at the local site.
   - connect, write, send ...
2. Identify the remote process
3. Inject the agent into the remote process
4. Start following the flow of control in the remote site

```c
void main () {
    connect(…)
    recv(…)
}
```

```c
void main () {
    accept(…)
    send(…)
}
```

Agent.so

Process A

Agent.so

Process B

inject via ssh / scp
Applications

- **Existing:**
  - Fault diagnosis in distributed systems
  - Software security analysis

- **Potential:**
  - Error reporting in distributed systems
  - Performance profiling
  - Consistency analysis in distributed systems
  - Taint checking
  - More ...
Status and Future work

○ **Status**
  ○ x86 / x86_64 Linux
  ○ IPC: pipe, tcp
  ○ Testing and debugging to make it robust enough to work on complex programs, e.g., Google Chrome

○ **Future work**
  ○ Develop tools for software security analysis
  ○ Support more protocols / mechanisms for inter-process propagation
  ○ Port to Microsoft Windows
Questions?