A Case for Live, Adversary-Aware Program Testing

Trent Jaeger
Systems and Internet Infrastructure Security Lab
Penn State University
Testing Goal

• Find potential vulnerabilities...
  ‣ Key question: What is a vulnerability?
Vulnerabilities

• A program vulnerability consists of three elements:
  ‣ A flaw
  ‣ Accessible to an adversary
  ‣ Adversary has the capability to exploit the flaw

• Claim: Testing techniques focus on subset of these elements
  ‣ But all conditions must be present for a true vulnerability
Consider a university department webserver …

GET /~student1/index.html HTTP/1.1

Diagram:

- /etc/passwd
- student1/public_html
- student2/public_html
- faculty1/public_html
- Apache Webserver

Link
Attack Video
What Just Happened?

- Program received an *unexpected* resource
  - 😈 when expecting 🖕️ (unexpected attack surface)
  - 🙁 when expecting 😈 (confused deputy)
Vulnerabilities

• Flaw finding
  ‣ Opening a file is not necessarily a flaw
  ‣ Checks for correct name filtering and/or binding is expensive, so not done unless a reason

• Restricting exploits
  ‣ Mandatory access control is insufficient
  ‣ Victim needs to communicate with potential adversaries and access sensitive resources

• Adversary access
  ‣ Not tracked systematically
• We actively change the namespace whenever an adversary can write to a binding used in resolution
  ‣ Fundamental problem: adversaries may be able to write directories used in name resolution

• Generate an adversarial test case and see how program reacts – live (unoptimized) for <8% overhead

Vulnerable!
Launch Phase

1. Find bindings
2. Find adversary access
3. Launch attack (modify namespace)
4. Continue system call

```
fd = open("/var/mail/root", O_APPEND)
```

```
delete("/var/mail/root");
symlink("/etc/passwd", "/var/mail/root")
```
Detect Phase

1. Victim accepts resource
2. Record vulnerability
3. Rollback namespace
4. Restart system call

Victim (user root)

write(fd)

User-space

Kernel

/etc
/var
/mail

passwd

root (symbolic link)
# Results - Vulnerabilities

<table>
<thead>
<tr>
<th>Program</th>
<th>Vuln. Entry</th>
<th>Priv. Escalation DAC: uid-&gt;uid</th>
<th>Distribution</th>
<th>Previously known</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbus-daemon</td>
<td>2</td>
<td>messagebus-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>landscape</td>
<td>4</td>
<td>landscape-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>Startup scripts (3)</td>
<td>4</td>
<td>various-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>mysql</td>
<td>2</td>
<td>mysql-&gt;root</td>
<td>Ubuntu</td>
<td>1 Known</td>
</tr>
<tr>
<td>mysql_upgrade</td>
<td>1</td>
<td>mysql-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>tomcat script</td>
<td>2</td>
<td>tomcat6-&gt;root</td>
<td>Ubuntu</td>
<td>Known</td>
</tr>
<tr>
<td>lightdm</td>
<td>1</td>
<td>*-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>bluetooth-applet</td>
<td>1</td>
<td>*-&gt;user</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>java (openjdk)</td>
<td>1</td>
<td>*-&gt;user</td>
<td>Both</td>
<td>Known</td>
</tr>
<tr>
<td>zeitgeist-daemon</td>
<td>1</td>
<td>*-&gt;user</td>
<td>Both</td>
<td>Unknown</td>
</tr>
<tr>
<td>mountall</td>
<td>1</td>
<td>*-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>mailutils</td>
<td>1</td>
<td>mail-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>bsd-mailx</td>
<td>1</td>
<td>mail-&gt;root</td>
<td>Fedora</td>
<td>Unknown</td>
</tr>
<tr>
<td>cupsd</td>
<td>1</td>
<td>cups-&gt;root</td>
<td>Fedora</td>
<td>Known</td>
</tr>
<tr>
<td>abrt-server</td>
<td>1</td>
<td>abrt-&gt;root</td>
<td>Fedora</td>
<td>Unknown</td>
</tr>
<tr>
<td>yum</td>
<td>1</td>
<td>sync-&gt;root</td>
<td>Fedora</td>
<td>Unknown</td>
</tr>
<tr>
<td>x2gostartagent</td>
<td>1</td>
<td>*-&gt;user</td>
<td>Extra</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

**19 Programs** | **26** | **21 Unknown**
Adversary Models

• How should we identify adversaries of a program?
Adversary Models

• How should we identify adversaries of a program?

• Researchers have evaluated research prototypes where adversaries are
  ‣ Not a root process
  ‣ Not a process with the same user id

• But, lots of root processes and processes with your user id – reasons not to trust them
  ‣ Compromised root network daemons and user processes
  ‣ ACL Policy may be modified by compromised processes
Adversary Models

• Instead we have explored using available mandatory access control (MAC) policies
  ‣ Fine-grained: confine root processes
  ‣ Immutable: mandatory system policy
  ‣ Restrictive: least privilege MAC policies

• To define a conservative adversary models
  ‣ What are the minimal set of subjects that a process must trust when it executes?
    • Those that already have the permissions sufficient to compromise the process
Subjects That Can Attack Already

• Subjects a process must trust… [ASIACCS 2012]
  ‣ Subjects that can modify the process’s executable file
  ‣ Subjects that can modify the kernel objects
  ‣ Subjects that can modify executable files of these subjects
    • Applied transitively

• Practical?
  ‣ Only 81 of over 2000 system call sites in Linux system service programs access resources modifiable by these adversaries
Testing Conclusions

• What are the lessons from STING that we can use?

• Adversary accessibility
  ‣ Vulnerabilities require adversary accessibility
  ‣ Can leverage conservative adversary models based on available MAC policies

• In-vivo
  ‣ Test the program when it is in a state that accessible to adversaries
  ‣ Launch attacks (in some way) to test program reaction, while keeping the program running (low overhead)

• Live, Adversary-Aware testing can be practical