Two papers

• Automated Exploit Generation, Oakland, NDSS 2011
  – Automatically generate attacks

• Vigilante: End-to-end Containment of Internet worms, SOSP 2005
  – Automatically generate and distributed defenses

• Both are about speed
AEG

• What’s the goal here?

• Why?
A bit of background

- Brumley’s, “Automated Patch-based Exploit Generation”, Oakland 2008
Structure of AEG

• Given program: find bugs and generate exploit

• Three main parts
  – Source code analysis
  – Binary analysis
  – Exploit generation
Assumptions
Bug finding via: symbolic execution

• From verification community
• Simulate program source using symbolic expressions for variables
  – Test if safety predicate is violated
  – If so, report input predicate that allowed this

• What’s the problem here?
Preconditioned symbolic execution

• Fancy name for “a heuristic”
• Basic idea: instead of checking all possible paths, focus on paths that are more likely to be exploitable

• What’s an example of this?

• Further helped by path prioritization... what’s this?
Having found a bug...

• Is it exploitable?

• Why can’t we evaluate this at the source level?

• Parallel binary analysis identifies low-level state associated with symbolic state
Exploit generation as constraint solving

• Combines:
  – Symbolic predicate on buggy program state
  – Low-level information on current stack/register state
  – Predicate on exploitability (i.e., can overwrite control data and get return without crash)

• Run through STP constraint solver
Assumptions/limitations

- Only look at control-flow hijacks
- Only stack-overflow and format string bugs
- Only two kinds of exploit predicates
  - (ret to stack, ret to libc)
- Doesn’t generate exploits to evade defenses
- Preconditioning assumption

- Still very impressive...
Speed

• For example programs generated exploits quickly
  – Typically a minute or less (20mins in one case)

• Is this a metric we care about?
• What if it were faster? Slower?
Vigilante

- What is the goal here?
- Why?
Worms: context

• Autonomous, active code that can replicate to remote hosts via vulnerability
• Because they propagate autonomously, they can spread very quickly
• This was a really big deal between 2001-2004
How to think about worms

• Reasonably well described as infectious epidemics
  – Simplest model: Homogeneous random contacts

• Classic SI model
  • $N$: population size
  • $S(t)$: susceptible hosts at time $t$
  • $I(t)$: infected hosts at time $t$
  • $\beta$: contact rate
  • $i(t)$: $I(t)/N$, $s(t)$: $S(t)/N$

\[
\frac{dI}{dt} = \beta \frac{IS}{N} \quad \frac{di}{dt} = \beta i(1 - i)
\]

\[
\frac{dS}{dt} = -\beta \frac{IS}{N}
\]

\[
i(t) = \frac{e^{\beta(t-T)}}{1 + e^{\beta(t-T)}}
\]

courtesy Paxson, Staniford, Weaver
A pretty fast outbreak:
Slammer (2003)

- First ~1min behaves like classic random scanning worm
  - Doubling time of ~8.5 seconds
  - CodeRed doubled every 40mins

- >1min worm starts to saturate access bandwidth
  - Some hosts issue >20,000 scans per second
  - Self-interfering (no congestion control)

- Peaks at ~3min
  - >55million IP scans/sec

- 90% of Internet scanned in <10mins
  - Infected ~100k hosts (conservative)

Vigilante

• Focused on network worms exploiting previously unknown vulnerability

• Goals:
  – Identify new attack/exploit as it happens
  – Distribute information quickly and securely
  – Build defenses against vulnerability (try to generalize)
Vigilante’s components

- Detection
- SCA generation
- SCA distribution
- SCA verification
- Protection
Detectors

• Some hosts run exploit detectors
  – Taint tracking, no execute bits, etc...

• Assumptions?
Dynamic dataflow analysis

- high coverage and low false positive rate
- allows direct extraction of verification information
Self-certifying alerts

• Detector generates an SCA that describes the vulnerability
  – Then distributes SCA to other vulnerable hosts

• Receiving hosts validate SCA
  – In sandboxed environment

• What problem does this solve?
Distribute SCA via P2P network

• Why via P2P network?
• What is goal here?
Protection

• Hosts generate “filter” from SCA
• **Dynamic data and control flow analysis**
  – run vulnerable application in a sandbox
  – track control and data flow from input messages
  – compute conditions that determine execution path
  – filter blocks messages that satisfy conditions
• Specific filters have **zero** false positives
• General filters (relaxation by heuristics) can have false positives
  – Why bother with these?
**Vulnerability filter generation**

```
mov al, [netbuf]
mov cl, 0x31
cmp al, cl
jne out
xor eax, eax
loop:  
mov [esp+eax+4], cl
mov cl, [eax+netbuf+1]
inc eax
test cl, cl
jne loop
out:
```

**netbuf**

<table>
<thead>
<tr>
<th></th>
<th>0x31</th>
<th>0x24</th>
<th>0x67</th>
<th>0x42</th>
<th>0x0</th>
</tr>
</thead>
</table>

**Conditions:**

- `netbuf[0] == 0x31`
- `netbuf[1] != 0`
- `netbuf[2] != 0`

Network

Filter

Vulnerable Application
Time to generate SCAs

<table>
<thead>
<tr>
<th>Dynamic dataflow</th>
<th>SCA generation time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slammer</td>
<td>18</td>
</tr>
<tr>
<td>Blaster</td>
<td>206</td>
</tr>
<tr>
<td>CodeRed</td>
<td>2667</td>
</tr>
<tr>
<td>Slammer (NX)</td>
<td>2</td>
</tr>
</tbody>
</table>
Time to verify SCAs

- Slammer: 10 ms
- Blaster: 18 ms
- CodeRed: 75 ms
Time to generate filters

<table>
<thead>
<tr>
<th>Filter</th>
<th>Generation Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slammer</td>
<td>24</td>
</tr>
<tr>
<td>Blaster</td>
<td>273</td>
</tr>
<tr>
<td>CodeRed</td>
<td>3402</td>
</tr>
</tbody>
</table>
For next time...

- Class starts 15 mins late (CNS)

- Chromium browser paper