Fence: Protecting Device Availability With Uniform Resource Control

Tao Li†, Albert Rafetseder†, Rodrigo Fonseca*, Justin Cappos†
†New York University  *Brown University
Motivation
Motivation
Typical Causes

- DropBox sync
- Browser tabs
- Virus scan
- Software updates
- ...

Scan causes CPU to overheat

Some weeks back I wrote “Death to Javascript”, in which I related the problem my wife has reported, of web pages that tie up her computer. She’s been seeing this more and more often lately. But today we got lucky; she was able to identify a specific page, on CNN.com, that causes this to happen.

Warning: save your work and close down other applications before you click this link in the Fortune blog, because it’s likely to paralyze your computer. I’ve tested this with Firefox on my PC, and after this page has finished loading the CPU usage is maxed at 100%—even though I’m doing nothing, and there’s no data being transferred over the network. (Yea, I’m running a 1.3 GHz CPU, but my wife’s computer is much faster and has the same problem.)

The interesting thing is that when I view the same web page in the Opera browser, with Javascript enabled, my CPU usage is only about 21%. I know that Opera has a more efficient Javascript interpreter, but I wonder if it also takes precautions against infinite loops?

While searching for “javascript kills browser” I found this set of horrible examples. If I click the second “DON’T CLICK ME” link in Firefox, the forever loop will send my CPU utilization to 100%. Ditto in Opera. So whatever CNN is doing wrong, it’s not a simple infinite loop.

Perhaps some enterprising Javascript wizard will take a look at the...
Goal: Performance Isolation

● Control performance, battery, heat, etc.
  ○ Do not kill -- useful-but-gluttonous apps

● Do not require OS / hardware changes
  ○ Applicable to sandboxes, browsers, etc.
  ○ Run everywhere (Linux, Windows, Mac, Android, OpenWrt, etc.)

● Focus on mechanism
  ○ Necessary for policies to function
Why is performance isolation hard?

- Multiple contended resources
  - Separate mechanisms / policies
  - Creating an overarching policy is difficult
- Some controls are missing
  - Gaps in enforcement -> lack of isolation
- Legacy systems tend to be work-preserving
Key Idea: Uniform Resource Control

- Unifying resource abstraction
  - Two axes per resource: fungible/renewable
    - Fully defines mechanism
  - Easy to cover new resources
  - Easy to implement policies
Resource Abstraction Questions

● Fungible:
  ○ Are items of this type interchangeable?
    ■ Yes (disk space) vs No (TCP port)

● Renewable:
  ○ Are items replenished over time?
    ■ Yes (Network bandwidth) vs No (RAM)
## Resource Controls

<table>
<thead>
<tr>
<th>Not Fungible</th>
<th>Fungible</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>is_item_allowed()</strong></td>
<td><strong>tattle_add_item()</strong></td>
</tr>
<tr>
<td>“Check if permitted”</td>
<td><strong>tattle_remove_item()</strong></td>
</tr>
<tr>
<td>e.g. UDP port</td>
<td>“Restrict total used”</td>
</tr>
<tr>
<td></td>
<td>e.g. File Descriptors</td>
</tr>
<tr>
<td><strong>tattle_quantity()</strong></td>
<td><strong>tattle_quantity()</strong></td>
</tr>
<tr>
<td>“Rate limit”</td>
<td>“Rate limit”</td>
</tr>
<tr>
<td>e.g. Network b/w</td>
<td>e.g. CPU</td>
</tr>
</tbody>
</table>

Not Fungible: Resources that are not fungible cannot be divided or combined.

Fungible: Resources that are fungible can be divided or combined.
Enforcement mechanism

- **Polling**
  - Find value, stop / rate limit if over
    - e.g. CPU uses job control interface (SIGSTOP / SIGCONT)
  
- **Interposition**
  - API code changes to add interposition

- **Which depends on implementation**
Example Implementation Changes

def sendmessage(destip, destport, msg, localip, localport):  # 117 lines
...

# get the OS's UDP socket
sock = _get_udp_socket(localip, localport)

# Send this UDP datagram
bytessent = sock.sendto(msg, (destip, destport))

...
def sendmessage(destip, destport, msg, localip, localport):

# 117 lines + 10 lines
...

# check that we are permitted to use this port...
if not fence.is_item_allowed('UDPport', localport):
    raise ResourceAccessDenied('...')

# get the OS's UDP socket
sock = _get_udp_socket(localip, localport)

# Send this UDP datagram
bytessent = sock.sendto(msg, (destip, destport))

...
def sendmessage(destip, destport, msg, localip, localport):
    # 117 lines + 10 lines
    ...
    # check that we are permitted to use this port...
    if not fence.is_item_allowed('UDPport', localport):
        raise ResourceAccessDenied('...')
    # get the OS's UDP socket
    sock = _get_udp_socket(localip, localport)
    # Register this socket descriptor with fence
    fence.tattle_add_item('outsocketsopened', id(sock))
    # Send this UDP datagram
    bytessent = sock.sendto(msg, (destip, destport))
    ...

Example Implementation Changes

socket: Fungible, non-renewable
network b/w: fungible, renewable
Uses of Fence

- Seattle Testbed’s Repy sandbox
  - Seattle ~= Peer-to-peer PlanetLab
  - Tens of thousands of diverse devices
- Lind
  - NaCl / POSIX sandbox
- Sensibility Testbed
  - Privacy preserving sensing on Android
Limitations

- Resource consumption must be visible
  - HW / OS hide info
- Minimizes performance impact
  - “Worst case” limits
- Scope of policies
  - Unclear how complete Fence is
  - Worked for us in practice
Evaluation

- How well does Fence work vs legacy controls?
- How well does Fence work across platforms?
- How much overhead does Fence incur?
- Can realistic policies be expressed in Fence?
- How diverse of resources can be metered?
- How hard is it to add resources to Fence?
Evaluation

- How well does Fence work vs legacy controls?
- How well does Fence work across platforms?
- How much overhead does Fence incur?
- Can realistic policies be expressed in Fence?
- How diverse of resources can be metered?
- How hard is it to add resources to Fence?
Fence vs Legacy Controls

- Video on disk (Dell Inspiron 630m w/ Ubuntu 10.04)
- “hog” everything
- worst setting for hog
- best setting for video

https://www.youtube.com/watch?v=OGe8QpPbtz4
Fence vs Legacy Heat / Battery

- Heat / battery
- “hog” everything
Fence vs Legacy Heat / Battery

- Heat / battery
- “hog” everything
Fence vs Legacy Heat / Battery

- Heat / battery
- “hog” everything
Fence vs Legacy Heat / Battery

- Heat / battery
- “hog” everything
Fence vs Legacy Heat / Battery

- Heat / battery
- “hog” everything
Evaluation

- How well does Fence work vs legacy controls?
- How well does Fence work across platforms?
- How much overhead does Fence incur?
- Can realistic policies be expressed in Fence?
- How diverse of resources can be metered?
- How hard is it to add resources to Fence?
Expressing Policies: Cinder

Power draw policy from Cinder [Roy Eurosyst 2011]
Stores energy w/ a tap (token bucket)
Polling using ACPI (updates every 15 seconds)

Program: Richards benchmark in a run / sleep cycle
Expressing Policies: Cinder (cont)
Expressing Policies: Cinder (cont)
Expressing Policies: Cinder (cont)

150 LOC!
Conclusion

● Performance isolation is still a challenge

● Uniform Resource Control
  ○ Same simple reasoning for all resources
  ○ Fungible / Renewable
  ○ Easy to implement / use
  ○ Effective in practice
NYU is Hiring!

- Secure software distribution
  - Adoption by Python, Ruby, Docker, LEAP, CoreOS, Go, Rust, Haskell, OCaml, etc.
  - Plausible standard for many new domains

- Hiring Post Doc / Research Professor / Dev
Questions?
Fence vs Legacy Controls (cont)

<20% frames for each legacy tool
(Combining tools, only gives 22% of frames)

99% of frames for Fence
<table>
<thead>
<tr>
<th>Not Renewable</th>
<th>Fungible</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDP ports</td>
<td>Threads</td>
</tr>
<tr>
<td>TCP ports</td>
<td>Memory (RAM)</td>
</tr>
<tr>
<td>Network read / write</td>
<td>Storage Space</td>
</tr>
<tr>
<td></td>
<td>Open Sockets</td>
</tr>
<tr>
<td></td>
<td>Open Files</td>
</tr>
<tr>
<td></td>
<td>CPU</td>
</tr>
<tr>
<td></td>
<td>File read / write</td>
</tr>
<tr>
<td></td>
<td>HW random</td>
</tr>
</tbody>
</table>