A Secure Access Control Mechanism against Internet Crackers

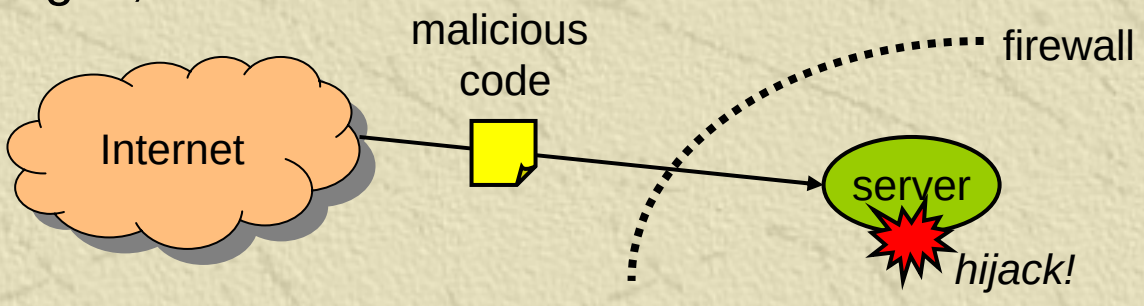
Kenichi Kourai* Shigeru Chiba**

**University of Tokyo*

***University of Tsukuba*

Server Hijacks by Crackers

- Internet servers are often “hijacked.”
  - e.g. buffer overflow attacks
    - A cracker sends malicious code into a server as the input and then can take the full control of the server.
  - Preventing all types of attacks is difficult.
    - Most attacks are caused by programming or design errors.
      - These errors are left in many programs.
    - Only a part of attacks are detected by StackGuard, safe languages, etc.

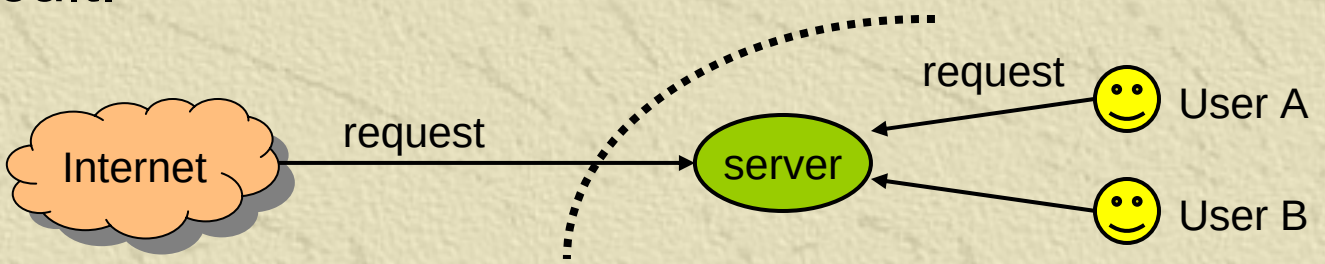


The Compacto Operating System

- Access restrictions can protect servers from attacks.
  - We have developed the Compacto operating system (OS), which provides:
    - Fine-grained access control at the system call level
      - Issues of system calls are limited by the type or arguments.
    - Access control with user/group ID (**`setuid`**/**`setgid`**)
    - Limited access to directories (**`chroot`**)
  - Compacto can impose access restrictions on a various range.
    - Application (some processes)
    - Process
    - Code fragment in a process

Danger of Removing Access Restrictions

- Servers need different access restrictions for each request.
  - It is necessary to impose/remove access restrictions.
- Removing access restrictions is dangerous.
  - Crackers can gain higher privileges after removing access restrictions from hijacked servers.
  - But, detecting whether a server is hijacked or not is difficult.

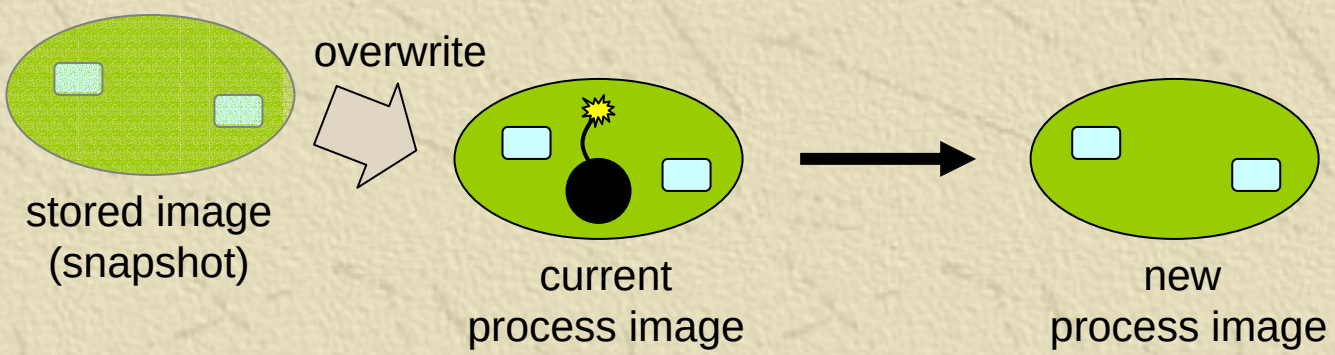


Traditional Approach: "*Spawn*"

- Spawns a new child process for every request and gives it different access restrictions.
  - The child process just terminates after handling a request.
    - Removing access restrictions is not necessary.
- Drawbacks
  - Slow
    - Too many processes are spawned.
  - Not compatible with the process pool technique
    - Today's servers often use a process pool for efficiency.

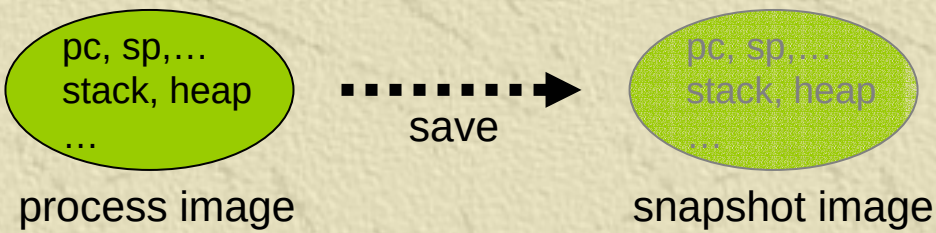
Our Approach: *Process Cleaning*

- Allows a process to remove access restrictions.
  - But, the process image is cleaned up back to the image stored by checkpointing.
    - Process cleaning achieves the same security as the “Spawn” approach.
- Benefits
  - Faster than the “Spawn” approach
  - Compatible with the process pool technique



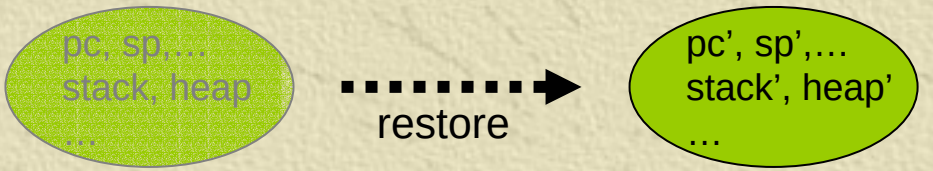
`Save_state` System Call

- **`Save_state`** takes a snapshot of a process at a safe point.
  - Where is considered as a safe point?
    - Before a server communicates with any clients.
  - The snapshot includes the whole state of a process.
    - registers, a memory image, signal handlers, open/close status of files/sockets, etc.
  - The snapshot cannot be compromised.
    - Because it is saved in the kernel address space.



Restore_state System Call

- **Restore_state** restores a safe state of a process saved by **save_state**.
  - Restoring an instruction pointer
    - Recovers the thread of control
  - Restoring the whole memory image
    - Eliminates malicious code from the memory
- After restoration, Compacto removes access restrictions from the process.



How to Use Process Cleaning

- Request-handling code in typical servers on Compacto looks like the following:

```
save_state();
accept();
if (source == Internet)
    impose strong access restrictions
else if (source == Intranet)
    impose weak access restrictions
handle a request
restore_state();
```

process cleaning & removing access restrictions

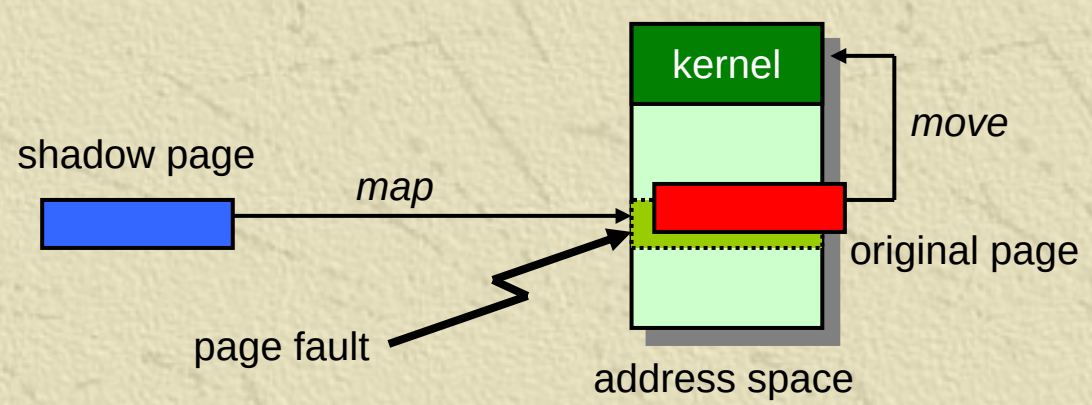
Implementation

- Saving/restoring a memory image become a performance bottleneck.
  - Naïve implementation is slow.

- Our implementation techniques
  - Efficient save of a memory image
    - Compacto saves only modified pages by *copy-on-write.*
  - Selectable strategy for restoring memory
    - Compacto allows users to select the restoration strategy according to the behavior of a server.

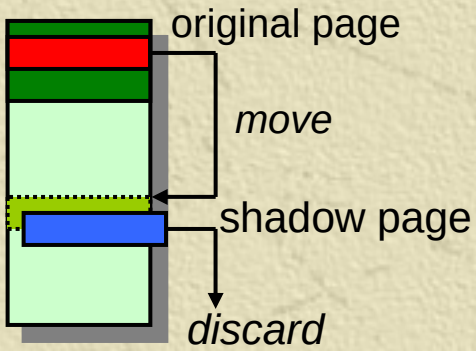
Page Save by Copy-on-Write

- Compacto allocates a shadow page at a page fault.
  - All writable pages are write-protected in the **`save_state`** system call.
  - The contents of an original page are copied to the shadow page.
    - The shadow page is modified and the original page is left unmodified.

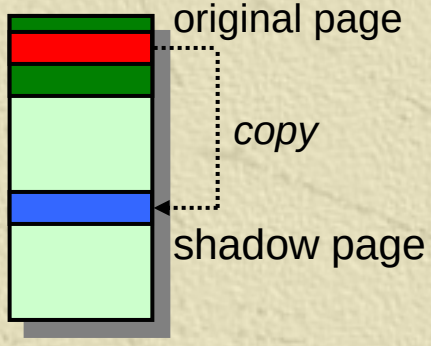


Remap/copy Strategies

- Remap strategy
  - Discards a shadow page and moves the original page back
  - Restores memory fast but may cause extra page faults



- Copy strategy
  - Copies the contents of an original page back to the shadow page
  - May reduce page faults but may cause extra page copies



Experiment

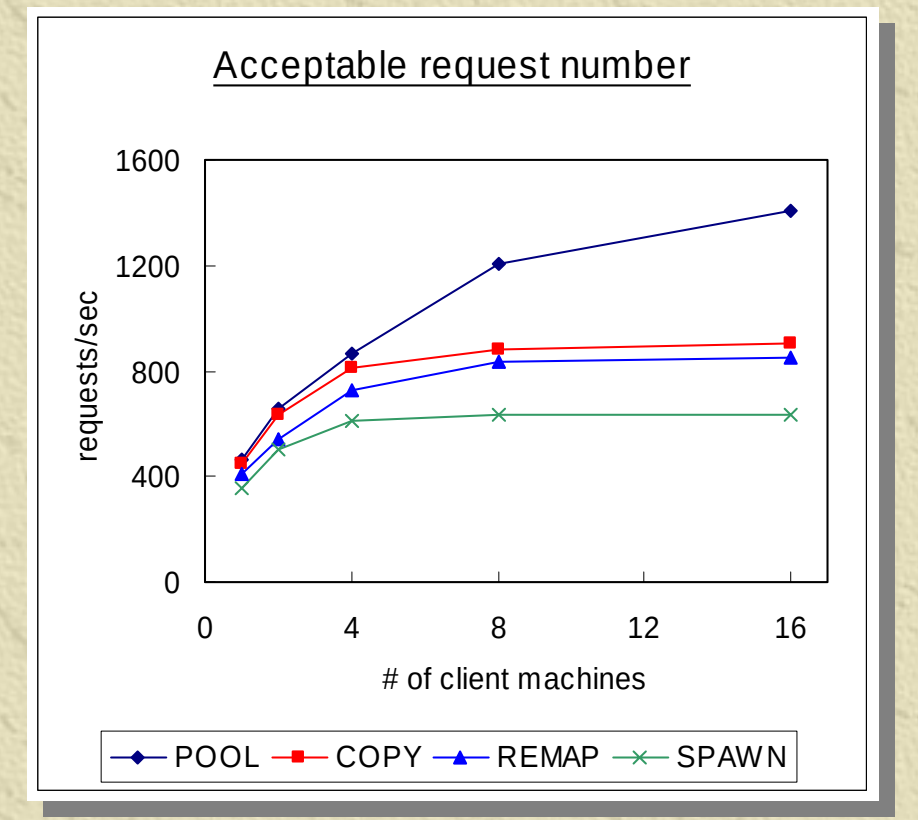
- We measured the performance of the Apache web server for 4 types of server constructs.

<table>
<tr><th colspan="3">Using a process pool</th><th>Spawning a child process</th></tr>
<tr><td>Insecure</td><td colspan="2">Process cleaning</td><td rowspan="3">SPAWN</td></tr>
<tr><td rowspan="2">POOL</td><td>Remap strategy</td><td>Copy strategy</td></tr>
<tr><td>REMAP</td><td>COPY</td></tr>
</table>

- Environments
  - Server: PentiumIII 933MHz, Compacto OS
  - Clients: Celeron 300MHz, FreeBSD 3.4
  - WebStone benchmark

Experimental Results

- Process cleaning is 40% faster than the “Spawn” approach.
- The overhead of process cleaning is low - 40%.
- The copy strategy is 8% faster than the remap strategy.



Concluding Remarks

- We proposed process cleaning.
  - It prevents hijacked servers from illegally removing access restrictions.
  - It achieves the same security as the traditional “Spawn” approach.
- Process cleaning achieves great performance improvement against the “Spawn” approach.
  - 40% faster