Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation

Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum
Intro

- applications and modern OSes do not promptly remove data from memory
- consider the host of “sensitive” information in memory
  - passwords, social security numbers, confidential documents, etc.
- don’t assume data in memory will be rewritten relatively soon
- why not preempt rewrites by zeroing out pages?
Motivation

- software bugs have memory leaks
- Windows’ “Dr. Watson” can send sensitive data in core files to a remote vendor
- 2001 study discovered 35 bugs in Linux and OpenBSD that can be used to read unauthorized sensitive data from kernel memory
- programs that handle sensitive data don’t take initiative to protect it (ex: Windows login program)
- even proactive programs lack control over kernel buffers or handling crashes
Data Lifetime

- **Ideal Lifetime:** time from first write after allocation to last read before deallocation
- **Natural Lifetime:** time from first write after allocation to first write of a different allocation
- **Secure Deallocation Lifetime:** time from first write after allocation until deallocation and zeroing
- **Holes:** occur when a subset of allocated memory is re-allocated
Long-Term Data Lifetime
Long-Term Data Lifetime

- If it’s memory, why not just reboot or shutdown?
- A soft reboot does not turn off the power
  - Some stamps remained in memory
- A hard reboot completely powers off the machine
  - Results varied
- One laptop kept some stamps 30 seconds after hard reboot
Design

- There are choices of when to clear data
- Lower layer (kernel): data is cleared when a process dies
- Higher layer (user stack/heap): data is cleared immediately after deallocation or after a timeout
- Higher layer is preferred, but authors implement both
Design Caveats

- no deallocation
  - if a process doesn’t deallocate pages, they can’t be zeroed out
- memory leaks
  - failing to free memory causes data lifetime problem
- custom allocators
  - memory allocator bypasses the C library alloc reducing effectiveness of secure deallocation
# Data Lifetime Stats

<table>
<thead>
<tr>
<th>Application</th>
<th>Run Time</th>
<th>Allocated</th>
<th>Written</th>
<th>Ideal Lifetime</th>
<th>Secure Deallocation Lifetime</th>
<th>Written, Freed, &amp; Overwritten</th>
<th>Natural Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Written</td>
<td>mean</td>
<td>stddev</td>
<td>Written &amp; Freed</td>
<td>mean</td>
</tr>
<tr>
<td>Mozilla</td>
<td>23:04</td>
<td>135 MB</td>
<td>96 MB</td>
<td>11 s</td>
<td>68 s</td>
<td>94 MB</td>
<td>21 s</td>
</tr>
<tr>
<td>Thunderbird</td>
<td>44:20</td>
<td>232 MB</td>
<td>155 MB</td>
<td>5 s</td>
<td>86 s</td>
<td>153 MB</td>
<td>10 s</td>
</tr>
<tr>
<td>ssh</td>
<td>30:55</td>
<td>6 MB</td>
<td>6 MB</td>
<td>0 s</td>
<td>0 s</td>
<td>6 MB</td>
<td>0 s</td>
</tr>
<tr>
<td>sshd</td>
<td>46:19</td>
<td>6 MB</td>
<td>6 MB</td>
<td>0 s</td>
<td>0 s</td>
<td>6 MB</td>
<td>0 s</td>
</tr>
<tr>
<td>Python</td>
<td>46:14</td>
<td>352 MB</td>
<td>232 MB</td>
<td>24 s</td>
<td>53 s</td>
<td>232 MB</td>
<td>23 s</td>
</tr>
<tr>
<td>Apache</td>
<td>1:01:21</td>
<td>57 MB</td>
<td>5 MB</td>
<td>0 s</td>
<td>0 s</td>
<td>5 MB</td>
<td>0 s</td>
</tr>
<tr>
<td>xterm</td>
<td>46:13</td>
<td>8 MB</td>
<td>8 MB</td>
<td>1 s</td>
<td>2 s</td>
<td>0 MB</td>
<td>1 s</td>
</tr>
<tr>
<td>ls</td>
<td>46:02</td>
<td>86 MB</td>
<td>23 MB</td>
<td>1 s</td>
<td>13 s</td>
<td>22 MB</td>
<td>2 s</td>
</tr>
</tbody>
</table>

- Secure Deallocation is close to ideal lifetime
- Natural lifetime shows just how vulnerable data can be
Kernel Clearing

- use TaintBochs simulator
  - allows pseudo-sensitive data to be marked “tainted”
- added a bit to Linux page structure for “polluted” pages
- free page pool is divided into 3 subsets:
  - not-zeroed, zeroed, and polluted
Kernel Clearing Optimizations

- caller may request a zeroed page
  - allocator returns a zeroed page if available else it takes a polluted page and zeros it out
- caller can indicate it will clear the page itself
  - allocator returns polluted page and leaves clearing to the caller
- caller has no preference on page type
  - allocator follows precedence: not-zeroed, zeroed, polluted
Kernel Clearing

- tracked lifetime of a password through an Apache web server to a Perl subprocess
- Apache had three tainted buffers:
  - network input buffer, input and output buffer to Perl subprocess
- Perl had tainted input buffer and numerous tainted string buffers
- All buffers had the full password in cleartext (except some Perl string buffers)
Clearing the Heap

The diagram shows the speed comparison of different benchmarks between Unmodified and Heap Clearing. The x-axis represents various benchmarks: 164.zip, 175.vpr, 176.gcc, 181.mcf, 186.crafty, 197.parser, 252.eon, 253.perfbmk, 254.gap, 255.vortex, 256.bzip2, 300.twolf, and firefox. The y-axis represents speed, with values ranging from 0 to 1.4. The column charts indicate a clear improvement in speed with Heap Clearing compared to Unmodified for most benchmarks.
Clearing the Stack
Conclusions and Issues

- It is possible to zero data at the kernel and user level with negligible overhead
- Immediate clearing can be done for applications known to handle sensitive data
- Can help to find referencing errors too
- Some issues I had with the paper:
  - What about swap space and virtual memory?
  - What if the file system is encrypted? Is data on disk