Operating System and Memory Vulnerabilities

Presented by Sam Silvestro
Buffer Overflows

Buffer overflow example

<table>
<thead>
<tr>
<th>Buffer (8 bytes)</th>
<th>Overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>USERNAME</td>
<td>1 2</td>
</tr>
</tbody>
</table>

Stack-based buffer overflow attack

BEFORE ATTACK

AFTER ATTACK

MALICIOUS CODE
# Latest CVE Details (1999–now)

## (Computer Vulnerability and Exposures)

### Vulnerabilities By Type

<table>
<thead>
<tr>
<th>Year</th>
<th># of Vulnerabilities</th>
<th>DoS</th>
<th>Code Execution</th>
<th>Overflow</th>
<th>Memory Corruption</th>
<th>Sql Injection</th>
<th>XSS</th>
<th>Directory Traversal</th>
<th>Http Response Splitting</th>
<th>Bypass something</th>
<th>Gain Information</th>
<th>Gain Privileges</th>
<th>CSRF</th>
<th>File Inclusion</th>
<th># of exploits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999</td>
<td>894</td>
<td>177</td>
<td>112</td>
<td>172</td>
<td></td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>25</td>
<td>16</td>
<td>103</td>
<td>2</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>2000</td>
<td>1020</td>
<td>257</td>
<td>208</td>
<td>206</td>
<td></td>
<td>2</td>
<td>4</td>
<td>20</td>
<td>20</td>
<td>48</td>
<td>10</td>
<td>139</td>
<td>23</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>2001</td>
<td>1677</td>
<td>402</td>
<td>402</td>
<td>297</td>
<td></td>
<td>7</td>
<td>24</td>
<td>124</td>
<td>124</td>
<td>83</td>
<td>36</td>
<td>220</td>
<td>2</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>2002</td>
<td>2156</td>
<td>498</td>
<td>553</td>
<td>435</td>
<td>2</td>
<td>41</td>
<td>200</td>
<td>103</td>
<td>127</td>
<td>76</td>
<td>199</td>
<td>2</td>
<td>2</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>2003</td>
<td>1527</td>
<td>381</td>
<td>477</td>
<td>372</td>
<td>2</td>
<td>50</td>
<td>129</td>
<td>60</td>
<td>1</td>
<td>62</td>
<td>69</td>
<td>144</td>
<td>16</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>2004</td>
<td>2451</td>
<td>580</td>
<td>614</td>
<td>408</td>
<td>3</td>
<td>140</td>
<td>291</td>
<td>111</td>
<td>12</td>
<td>145</td>
<td>96</td>
<td>134</td>
<td>5</td>
<td>59</td>
<td>5</td>
</tr>
<tr>
<td>2005</td>
<td>4935</td>
<td>838</td>
<td>1627</td>
<td>657</td>
<td>21</td>
<td>604</td>
<td>786</td>
<td>202</td>
<td>15</td>
<td>289</td>
<td>261</td>
<td>221</td>
<td>11</td>
<td>100</td>
<td>24</td>
</tr>
<tr>
<td>2006</td>
<td>6610</td>
<td>893</td>
<td>2719</td>
<td>664</td>
<td>91</td>
<td>967</td>
<td>1302</td>
<td>222</td>
<td>32</td>
<td>267</td>
<td>272</td>
<td>184</td>
<td>18</td>
<td>849</td>
<td>30</td>
</tr>
<tr>
<td>2007</td>
<td>6520</td>
<td>1101</td>
<td>2601</td>
<td>955</td>
<td>95</td>
<td>706</td>
<td>883</td>
<td>338</td>
<td>14</td>
<td>267</td>
<td>326</td>
<td>242</td>
<td>69</td>
<td>700</td>
<td>45</td>
</tr>
<tr>
<td>2008</td>
<td>5632</td>
<td>894</td>
<td>2310</td>
<td>599</td>
<td>128</td>
<td>1101</td>
<td>807</td>
<td>362</td>
<td>7</td>
<td>288</td>
<td>268</td>
<td>188</td>
<td>83</td>
<td>170</td>
<td>76</td>
</tr>
<tr>
<td>2009</td>
<td>5736</td>
<td>1038</td>
<td>2185</td>
<td>658</td>
<td>188</td>
<td>963</td>
<td>851</td>
<td>323</td>
<td>9</td>
<td>337</td>
<td>302</td>
<td>223</td>
<td>115</td>
<td>138</td>
<td>738</td>
</tr>
<tr>
<td>2010</td>
<td>4653</td>
<td>1102</td>
<td>1714</td>
<td>676</td>
<td>242</td>
<td>520</td>
<td>605</td>
<td>276</td>
<td>8</td>
<td>234</td>
<td>284</td>
<td>238</td>
<td>85</td>
<td>73</td>
<td>1501</td>
</tr>
<tr>
<td>2011</td>
<td>4155</td>
<td>1221</td>
<td>1334</td>
<td>734</td>
<td>351</td>
<td>294</td>
<td>470</td>
<td>108</td>
<td>7</td>
<td>197</td>
<td>411</td>
<td>206</td>
<td>51</td>
<td>17</td>
<td>557</td>
</tr>
<tr>
<td>2012</td>
<td>5297</td>
<td>1425</td>
<td>1459</td>
<td>833</td>
<td>423</td>
<td>243</td>
<td>759</td>
<td>122</td>
<td>13</td>
<td>344</td>
<td>392</td>
<td>250</td>
<td>166</td>
<td>14</td>
<td>623</td>
</tr>
<tr>
<td>2013</td>
<td>5191</td>
<td>1455</td>
<td>1186</td>
<td>856</td>
<td>366</td>
<td>156</td>
<td>650</td>
<td>110</td>
<td>7</td>
<td>352</td>
<td>512</td>
<td>274</td>
<td>123</td>
<td>1</td>
<td>206</td>
</tr>
<tr>
<td>2014</td>
<td>7939</td>
<td>1589</td>
<td>1522</td>
<td>841</td>
<td>420</td>
<td>304</td>
<td>1103</td>
<td>204</td>
<td>12</td>
<td>457</td>
<td>2106</td>
<td>239</td>
<td>264</td>
<td>2</td>
<td>403</td>
</tr>
<tr>
<td>2015</td>
<td>6504</td>
<td>1793</td>
<td>1830</td>
<td>1084</td>
<td>749</td>
<td>221</td>
<td>784</td>
<td>151</td>
<td>12</td>
<td>577</td>
<td>753</td>
<td>366</td>
<td>248</td>
<td>5</td>
<td>129</td>
</tr>
<tr>
<td>2016</td>
<td>6454</td>
<td>2029</td>
<td>1496</td>
<td>1313</td>
<td>217</td>
<td>94</td>
<td>498</td>
<td>99</td>
<td>15</td>
<td>444</td>
<td>870</td>
<td>662</td>
<td>86</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>2017</td>
<td>14714</td>
<td>3155</td>
<td>3004</td>
<td>2495</td>
<td>745</td>
<td>508</td>
<td>1518</td>
<td>279</td>
<td>11</td>
<td>629</td>
<td>1659</td>
<td>459</td>
<td>327</td>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>2018</td>
<td>16557</td>
<td>1853</td>
<td>2041</td>
<td>2121</td>
<td>400</td>
<td>517</td>
<td>2048</td>
<td>545</td>
<td>11</td>
<td>478</td>
<td>1238</td>
<td>247</td>
<td>461</td>
<td>31</td>
<td>4</td>
</tr>
<tr>
<td>2019</td>
<td>17344</td>
<td>1342</td>
<td>2201</td>
<td>1270</td>
<td>488</td>
<td>549</td>
<td>2390</td>
<td>465</td>
<td>10</td>
<td>710</td>
<td>983</td>
<td>202</td>
<td>535</td>
<td>57</td>
<td>13</td>
</tr>
<tr>
<td>2020</td>
<td>18325</td>
<td>1351</td>
<td>3248</td>
<td>1617</td>
<td>409</td>
<td>460</td>
<td>2178</td>
<td>401</td>
<td>14</td>
<td>966</td>
<td>1345</td>
<td>310</td>
<td>402</td>
<td>37</td>
<td>62</td>
</tr>
<tr>
<td>2021</td>
<td>10906</td>
<td>1084</td>
<td>2181</td>
<td>959</td>
<td>208</td>
<td>342</td>
<td>1335</td>
<td>247</td>
<td>2</td>
<td>459</td>
<td>551</td>
<td>149</td>
<td>206</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>157197</td>
<td>26466</td>
<td>39075</td>
<td>20302</td>
<td>6148</td>
<td>8707</td>
<td>19627</td>
<td>4979</td>
<td>188</td>
<td>8015</td>
<td>12845</td>
<td>5539</td>
<td>3265</td>
<td>2313</td>
<td>4423</td>
</tr>
<tr>
<td>% Of All</td>
<td></td>
<td>16.8</td>
<td>24.9</td>
<td>12.9</td>
<td>3.9</td>
<td>5.6</td>
<td>12.5</td>
<td>3.2</td>
<td>0.1</td>
<td>5.1</td>
<td>8.2</td>
<td>3.5</td>
<td>2.1</td>
<td>1.5</td>
<td></td>
</tr>
</tbody>
</table>
The Heap

• Used to allocate memory dynamically
• Typical interface is the `malloc()` library function in C
• “The heap” is simply the name of the address space segment dedicated to fulfilling these requests
• It can grow and shrink dynamically, as well
• Is also vulnerable to heap-based overflow attacks, similar to the classic stack-based overflow
Stack vs. Heap

**Key Differences:**

- Stack memory will never become fragmented whereas heap memory can become fragmented as blocks are first allocated and then freed.
- Stack accesses local variables only while the heap allows you to access variables globally.
- Stack variables can’t be resized whereas heap variables can be resized (see `realloc()` library function).
- Stack memory is allocated in a contiguous block whereas heap memory can be allocated in an indeterminate order.
- Stack doesn’t require deallocation of variables whereas the heap does (preferably!).
- Stack allocation and deallocation are done by compiler instructions, whereas heap allocation and deallocation is done by the programmer.
Common Heap Vulnerabilities

- Buffer over-read
  - Information leakage
    - e.g., Heartbleed
- Use-after-free
- Buffer overflow
- Double / invalid free
  - Unexpected results, program crash, hijacked control flow
Heap Vulnerabilities Reported in NIST Database

<table>
<thead>
<tr>
<th>Heap Vulnerabilities</th>
<th>Occurrences (#)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap Over-reads</td>
<td>125</td>
</tr>
<tr>
<td>Heap Over-writes</td>
<td>673</td>
</tr>
<tr>
<td>Use-after-frees</td>
<td>264</td>
</tr>
<tr>
<td>Invalid-frees</td>
<td>35</td>
</tr>
<tr>
<td>Double-frees</td>
<td>33</td>
</tr>
</tbody>
</table>

- Many vulnerabilities recently reported in a one-year period
- Buffer overflows top the list
- UAF reportedly most severe vuln’s of Chromium browser
- Allocator usually first line of defense against these attacks
Outline

Detection

1  Sampler: PMU-based Sampling to Detect Memory Errors Latent in Production Software

Prevention

2  FreeGuard: A Faster Secure Heap Allocator

3  Guarder: A Tunable Secure Allocator
Motivation

• Impossible to guarantee all errors expunged during testing
  – Specific inputs, timing, environment
  – Inevitable leakage into production

• Problems of existing detection frameworks
  – Binary Instrumentation
    • Prohibitive performance overhead
  – Compiler Instrumentation
    • Requires recompilation
    • False negatives (uninstrumented libraries)
  – Static Analysis
    • False negative, false positives
    • Scalability issues
  – Dynamic Analysis
    • Bounds checking, pointer-tracking
    • Only addresses one particular type of problem

Valgrind (PLDI’07)
Dr. Memory (CGO’11)
ASan (ATC’12)
Cppcheck
FindBugs
Coverity
Baggy Bounds Checking (SSYM‘09),
FreeSentry (NDSS’15),
DangNULL (NDSS’15)
Motivation

• Seek a solution satisfying the properties:
  – **Efficiency**: <5% overhead on average
  – **Precision**: report provides precise info to guide fixes
  – **Completeness**: Detect read/write errors on all components
  – **Accuracy**: Every report → real problem
  – **Transparency**: zero manual effort/expertise req’d of users
Overview

• Use sampling-based approach
  – Validates only sampled accesses
  – First work to utilize PMU for detecting memory errors

• Utilize commodity hardware
  – Performance Monitoring Unit (PMU)
  – ProRace PEBS Driver

• Custom memory allocator
  – Information-computable property yields fast metadata lookup
  – Segregated metadata is critical

• Non-intrusive
  – No recompilation, instrumentation
Employing Sampling

• Why use sampling?
  – Lower performance overhead; validates sampled access only
  – Non-intrusive: no recompilation or modification to apps
  – Suitability for production: large user base, many executions

• Challenges
  – Correctness
    • Validation occurs after the access
  – Completeness
    • Encourage unique sample sequences
    • Detection of double and invalid frees
  – Performance
    • Efficient mechanism to perform checking
    • Minimize amount of checking
Sample Checking Sequence

1. Invalid access occurs on object’s redzone

2. PMU samples the access

3. Sampler reads sample data, consults SM, and detects the error
PEBS Driver

Process flow using the standard Linux perf subsystem driver.

The ProRace PEBS driver eliminates the copy and translation step for delivery of sample data to the application.
Effectiveness Evaluation

- Effectiveness proportional to sampling period $P$
  - $1/P$ probability of detecting each invalid access
- Bugs often involve multiple invalid accesses
- Evaluated 12 bugs
  - For $P=5000$ over 1000 executions:
    - Averaged 378 detections (37.8%)
    - All bugs detected given sufficient numbers of executions
- Using latest hardware support $\Rightarrow$ no false positives
- Custom allocator $\Rightarrow$ 100% detection of double & invalid frees
# Effectiveness Evaluation

<table>
<thead>
<tr>
<th>Application</th>
<th>Type</th>
<th>Bug</th>
<th>First Detection</th>
<th># Detections</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP 5.6.3</td>
<td>Double free</td>
<td>CVE-2016-5772</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>PHP 5.6.3</td>
<td>Use-after-free</td>
<td>CVE-2016-6290</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>PHP 5.6.3</td>
<td>Use-after-free</td>
<td>CVE-2016-5771</td>
<td>96</td>
<td>26</td>
</tr>
<tr>
<td>PHP 5.6.3</td>
<td>Use-after-free</td>
<td>CVE-2016-3141</td>
<td>59</td>
<td>27</td>
</tr>
<tr>
<td>PHP 5.6.3</td>
<td>Use-after-free</td>
<td>CVE-2015-6835</td>
<td>29</td>
<td>33</td>
</tr>
<tr>
<td>PHP 5.6.3</td>
<td>Use-after-free</td>
<td>CVE-2015-0273</td>
<td>37</td>
<td>26</td>
</tr>
<tr>
<td>bc 1.06</td>
<td>Buffer overflow</td>
<td>Bugbench</td>
<td>292</td>
<td>8</td>
</tr>
<tr>
<td>bzip2 1.0.3</td>
<td>Buffer overflow</td>
<td>Bugbench</td>
<td>1</td>
<td>853</td>
</tr>
<tr>
<td>gzip 1.2.4</td>
<td>Buffer overflow</td>
<td>Bugbench</td>
<td>1</td>
<td>447</td>
</tr>
<tr>
<td>libtiff 4.0.1</td>
<td>Buffer overflow</td>
<td>CVE-2013-4243</td>
<td>1</td>
<td>995</td>
</tr>
<tr>
<td>polymorph 0.4.0</td>
<td>Buffer overflow</td>
<td>Bugbench</td>
<td>193</td>
<td>105</td>
</tr>
<tr>
<td>ed 1.14.2</td>
<td>Invalid free</td>
<td>CVE-2017-5357</td>
<td>1</td>
<td>1000</td>
</tr>
</tbody>
</table>

Avg = 26.2

Avg = 481.6
• Evaluation of 19 applications (13 from PARSEC)
• Sampler has < 3% overhead on average (using P=5000)
• ASan has ~45% overhead on average (typically > 70%)
• Sampler’s allocator is ~3% faster than default Linux allocator
Performance Sensitivity

- Average runtime overhead inversely proportional to sampling period
  - $P=1000 \implies 6.6\%$
  - $P=5000 \implies 2.4\%$
  - $P=10000 \implies 1.0\%$
Outline

Detection
1  Sampler: PMU-based Sampling to Detect Memory Errors Latent in Production Software

Prevention
2  FreeGuard: A Faster Secure Heap Allocator
3  Guarder: A Tunable Secure Allocator
Default Linux Allocator

- Designed to perform well
  - Bump pointers + freelists
  - Not designed for security purposes
- Prepends metadata before actual objects
- Provides no randomization
  - Result: easy to determine when an object will be allocated
- Poor handling of double/invalid frees

An example of Linux allocator’s object metadata placement

- prev size: 16
- cur size: 32

allocated space
Existing Secure Allocators:
OpenBSD and DieHarder

- Both are BIBOP-style secure allocators
  - “Big Bag of Pages”
  - Each “bag” of pages holds objects of a specific size class
- Both feature randomization
  - DieHarder = \( \log n \) bits of entropy
  - OpenBSD = 2 ~ 10 bits
- Both impose high performance overhead
  - Evaluated runtime overhead:
    - OpenBSD \( \approx 34\% \)
    - DieHarder \( \approx 88\% \), up to 10X
  - Use bitmaps
  - Utilize many more system calls (\texttt{mmap})
  - Use the same heap for different threads
    - May result in significant performance loss for dominant multithreaded apps
Motivation of FreeGuard

- Performance-oriented allocators are not secure
- Secure allocators are too slow
FREEGUARD's BIBOP-style Layout

- Acquire large block and divide into multiple heaps
- Per-thread sub-heap design
- Sub-heaps made up of 16 bags
- All bags have same size
- Each bag serves a specific size class
- Size classes increase by powers-of-two
FREEGUARD’s BIBOP-style Layout

- When one heap is exhausted, the thread utilizes the bag (with the same index) in the next heap.
- Uniform layout:
  - Bags have same size
  - Same number of bags per sub-heap
  - Same number of sub-heaps per heap
  - All quantities are powers of two
FREEGUARD’s BIBOP-style Layout

- **Information-computable**
  - Given an address:
    - Size class
    - Owning thread
    - Starting address
    - Metadata location
  - Enables constant-time deallocation
  - Detects all double and invalid frees
Freelist-based Design

• Linked-list maintains deallocated objects
• Enables constant-time allocation

Together, with information-computable layout, FreeGuard takes only a single trial for allocation and deallocation
• Others may require many trials due to use of bitmaps or hashes to locate objects by address
Secure Design – Segregated Metadata

- BIBOP style can separate metadata with the actual heap
- Metadata is placed in a separate area, preventing metadata-based attacks
Secure Design – Randomization

- Four-way randomization: increase attack complexity
  - Each per-thread/per-bag pair has 4 freelists & 4 bump pointers
  - On allocation, choose one-of-four freelists and possibly from bump pointers
  - If empty, use the bump pointer with the same index
  - Some chance we fall back to bump pointer, even if freelist is non-empty

\[ \text{rand()} = 14 \]
Secure Design – Guard Pages

- Guard pages have all permission bits turned off
- Static guard pages
  - Placed at end of every bag
- Random guard pages
  - Distributed throughout bag
  - Default proportion: 10% of each bag
  - Helps prevent heap spraying, buffer overflow attacks
Secure Design – Canaries

• Canary values with neighbor checking
  – Checks canary values of freed object’s neighbors
  – Helps the timely detection of overflows
  – Skips neighbor if not in-use or inside guard pages

Heap objects:
Handling Large Objects

- Objects >512KB classified as large objects
  - Allocated separately using mmap, which is similar to DieHarder, but different from OpenBSD
  - After deallocation, memory is returned to OS using munmap, helping to prevent use-after-free attacks
  - Hashtable is used to track the size of each object
  - Object is placed at the end of the block to help detect overflows
## Security Feature Comparison

<table>
<thead>
<tr>
<th>Security Feature</th>
<th>Linux</th>
<th>DieHarder</th>
<th>OpenBSD</th>
<th>FreeGuard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully-segregated metadata</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Randomized allocation</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Guard pages</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Check overflows on free</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Over-provisioned allocation</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Detects double/invalid frees</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

✓ indicates the allocator has this feature

☒ indicates the implementation has some weakness

- **FreeGuard** provides feature-set comparable to DieHarder and OpenBSD, but:
  - FreeGuard and OpenBSD do not support heap over-provisioning
  - FreeGuard has a lower randomization entropy
- FreeGuard can detect all double and invalid frees; OpenBSD detects double frees with very low probability, while DieHarder does not report either
- FreeGuard and DieHarder better prevent use-after-free of large objects over OpenBSD
Outline

Detection

1 Sampler: PMU-based Sampling to Detect Memory Errors Latent in Production Software

Prevention

2 FreeGuard: A Faster Secure Heap Allocator

3 Guarder: A Tunable Secure Allocator
1. Exhibit either low or unstable entropy
2. Unstable entropy dependent on size class, execution phase, inputs, and applications
Design Purpose

• Reliable Security
  – Stable allocation entropy across:
    • Size classes
    • Inputs
    • Execution phases
    • Applications

• Tunable security
  – User may specify the bits of entropy
  – Balances performance budget with security needs
Supplying the Specified Entropy

• We could use a simple array as the object buffer
  – 1 out of 256 objects = 8 bits of entropy

• Challenges with this approach:
  – How to handle deallocations?
    • How to efficiently find space to reinsert freed objects?
  – How to avoid repopulating array after every allocation?
    • < 256 objects $\Rightarrow$ < 8 bits of entropy
  – How to avoid excessive checking cycles?
    • Upon allocations, probability of choosing empty slot
Combining Allocation and Deallocation Buffers

- Provides minimum of $E$-bits of user-specified entropy
  - Every thread has pair of allocation and deallocation buffers per size class
  - Allocation holds $2^{E+1}$ objects
    - Never allow buffer to fall below half full $\Rightarrow$ ensures minimum $E$ bits entropy
  - Allocation buffer is filled from top of heap if no freed objects are available
Tunable Security – Overprovisioning

- Dedicates a portion of heap objects as “never use”
  - Guarder’s default factor = 1/8
  - Thus, each object has 1/8 probability of being excluded from future use
- Helps thwart buffer overflow
- During allocation buffer filling step:
  - 1/8 of objects will be selected randomly for exclusion
  - Corresponding slot marked empty
  - Remaining 7/8 of non-empty slots will be pulled into allocation buffer
Performance Evaluation

- 21 applications evaluated
  - PARSEC
  - 8 real-world
- < 3% overhead, on average (arithmetic mean)

All values normalized to performance of default Linux allocator
Performance Evaluation

— Two reasons why Guarder performs faster
  • Avoids use of central lock
  • Due to the following design

<table>
<thead>
<tr>
<th>Trials</th>
<th>DieHarder</th>
<th>OpenBSD</th>
<th>FreeGuard</th>
<th>Guarder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>1.99</td>
<td>3.79</td>
<td>1</td>
<td>1.77</td>
</tr>
<tr>
<td>Maximum</td>
<td>93</td>
<td>45</td>
<td>1</td>
<td>131</td>
</tr>
<tr>
<td>Deallocation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>12.40</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Maximum</td>
<td>141</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Data collected with Guarder’s default tunable parameter of 9 bits of entropy.
# Security Feature Comparison

<table>
<thead>
<tr>
<th>Security Feature</th>
<th>Linux</th>
<th>DieHarder</th>
<th>OpenBSD</th>
<th>FreeGuard</th>
<th>Guarder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully-segregated metadata</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Randomized allocation</td>
<td>✓</td>
<td>✓</td>
<td>☹</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Guard pages</td>
<td>☹</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Check overflows on free</td>
<td>☹</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Over-provisioned allocation</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Detects double/invalid frees</td>
<td>☹</td>
<td>✓</td>
<td>☹</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

✓ indicates the allocator has this feature
☹ indicates the implementation has some weakness

- Guarder provides the most complete feature-set as compared to existing works
- Provides the strongest guarantee with respect to randomization entropy
Entropy of Secure Allocators

- Guarder exhibits reliable entropy
- Allows users to specify the entropy (e.g., 9 bits here)
Why Does Tunable Matter?

- Values normalized to default settings
- Higher security indicates higher overhead
Comparison of Existing Security Allocators

DieHarder (CCS'10)

OpenBSD

FreeGuard (CCS'17)

Guarder

Linux
Conclusion

• Works have improved on many of these problems, albeit incrementally, in the past couple of years
• However, we are always stuck in the position of trading performance for security
• Unless a hardware-based breakthrough occurs that features transparency and minimal to no performance loss, protecting against these vulnerability will remain an open problem
• Alternatively, we could abandon programming languages without built-in bounds-checking, but this is unlikely and, in many cases, impossible
• Thank you! Questions?