HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities

Jason Gionta, William Enck, Peng Ning
JIT-ROP
Two Attack Categories

• Injection Attacks
  – Code Integrity
  – Data Execution Prevention

• Code-Reuse Attacks
Code-Reuse Attacks – Simple ROP

What an attacker knows?
- Buffer overflow
- Static binary load location
- Static location of gadgets

Overwritten Return Address

```
xor rax, rax
ret
add $0x1, rcx
ret
lea rbx, [rsp+8]
ret

"buffer overflow"
ret
```
Code-Reuse Attacks – Break ROP

What an attacker knows?
• Buffer overflow
• Static binary load location
• Static location of gadgets

Protection: ASLR

Overwritten Return Address
What an attacker knows?

- Buffer overflow
- Static binary load location
- Static location of gadgets
- Return address leak

Protection: ASLR

Overwritten Return Address
Code-Reuse Attacks – Break ROP (Again)

What an attacker knows?
- Buffer overflow
- Static binary load location
- Static location of gadgets
- Return address leak

Protection: Fine-Grained ASLR

Overwritten Return Address
Code-Reuse Attacks – ROP + Disclosure

What an attacker knows?
- Buffer overflow
- Static binary load location
- Static location of gadgets
- Return address leak
- Code Disclosure

Protection: Fine-Grained ASLR
Memory Disclosure Vulnerabilities

• Leak raw memory contents
• Used for bypassing modern protections
  – ASLR - Pwn2Own2013, Pwn2Own2014
  – Fine-grained ASLR - Just-In-Time Code Reuse [Snow et al. 2013]

• Disclosure Protections
  – XnR — [Backes et al. 2014]
    • Limit code reads so small set of memory
    • Does not handle legitimate reads
    • Heuristic based detection

• Observation: commodity systems lack of fine-grained read permissions
HideM: Protect Userspace Code from Disclosure

• Assumption: Fine-Grained ASLR Deployed
• Enable fine-grained read permissions on executable memory
  – Prevent the majority of code from being read
• Enforce permissions seamlessly on Commercial-Off-The-Shelf (COTS) binaries
• Target commodity systems to ease adoption
• GOAL: Unreliable exploitation
  – Adversary must guess contents of code
Challenges

• Execute permissions imply read permissions
  – A *present* userspace page can always be read
  – **Solution**: apply *code hiding* to differentiate memory access based on CPU operation

• Executable pages often contain read-only data
  – Allow legitimate reads of executable pages
  – **Solution**: generate and apply *code reading* policy per executable page

• Protecting COTS binaries without symbols
  – **Solution**: light-weight binary analysis to identify data embedded in code pages
Code Hiding: Primitive

- Enables execute-only permissions on memory
  - Access based on CPU operation
- Based on PaX and advanced rootkit hiding
- Leverage split TLB architecture

```
pushl %ebp
movl %esp, %ebp
subl $8, %esp
```
Fine-grained Read Permissions

- Generate and enforce code reading policy
  - Identify read data in code prior to execution
  - Embed as part of COTS binary
- Associate binary data locations with load time memory
- Apply code reading policy per page

Pre-execution

Execution

Legacy/COTS Binary

Symbol + Data Analysis

Legacy/COTS Binary

Data Locations

Load Binary

Page Fault

Is Hidden Page?

yes

no

Continue execution

Data Locations

Load Information

Policy Generation

Read Policy

Apply Policy

Legacy/COTS Binary

Data + Analysis

Legacy/COTS Binary

Data Locations
Identifying Data in Code - Types

- Two types of data read in code pages
  - DT-1: read-only data never executed
  - DT-2: executed data that needs to be read
- Provide DT-1 and DT-2 ranges with binary
Identifying Data in Code – How to Identify

• DT-1: read-only data never executed
  – Binary structure
  – Recursive disassembly to identify ICF targets
    • Based on Zhang and Sekar [Usenix Sec’13]
    • Identify jump-tables
    • Disassembly errors identify gaps (unknown regions)

• DT-2: executed data that needs to be read
  – Binary analysis – identify DT-2
    • Immediate values that result in a valid code address
    • Instruction pointer relative values
Applying Read Policy

- All HideM protected pages have userspace access denied by default
- Generate shadow read page based on policy
  - First page fault copy DT-1 and DT-2 to read page
  - Remove DT-1 from code page
- Apply shadow page with code hiding
  - Prime TLB
Hardening Against ROP Exploits

• Readable pages contain DT-2 data
• Adversary can build exploits from only DT-2
  – Limited to 4 bytes in length
    • Identifiable by ROP Runtime Protections
• Add noise to readable pages in place of non-read code
  – Mimics DT-2 data
Empirical Evaluation

• Implementation of HideM
  – Linux kernel 3.10.12
    • Intel x86 64-bit
  – Dynamic library loading support
    • glib 2.18

• Platform
  – IBM LS22 blade server
    • Two Quad-Core AMD Operton 2384 processors
    • 32GB of RAM
    • 1024 4KB page TLB entries per core
  – Ubuntu 12.04.4 LTS 64-bit
Empirical Evaluation: Application Set

- 28 applications and required shared libraries converted to HideM
  - 442 total binaries converted / 441 MB
  - 13 binaries required manual analysis of data in code
- 9 non-trivial applications
  - Wireshark, dumpcap, gimp, gedit, lynx, python, emacs, lynx, smplayer
- 19 SpecCPU 2006 applications
  - Perlbench, bzip2, gcc, mcf, gobmk, hmmer, sjeng, libquantum, h264ref, omnetop, astar, xalancbmk, milc, namd, dealII, soplex, povray, lbm, sphinx3
Empirical Evaluation: Performance

- Runtime overhead: Percent increase in runtime
  - Maximum: 6.5% increase; Minimum: 2% decrease
  - Average 1.49%, median 0.51%

• Model the probability of exploitation based on knowledge of HideM
  – Adversaries dump memory and search for gadgets
  – Identify unique gadgets required for exploit
  – Choose a location for each unique required gadget
    • Duplicate gadgets at different locations

- Probability of exploitation against HideM
  - Based on “Unordered sampling without replacement”
    - N gadgets for an exploit
    - $U_g$ total unique gadgets
    - $U_{vg}$ number of unique valid gadgets
    - $S_{vg}$ number of valid unique gadgets for a specific gadget
    - $S_g$ total number of gadgets for a specific valid unique gadget

\[
\prod_{n=1}^{N} \left( \frac{U_{vg} - n - 1}{U_g - n - 1} \right) \left( \frac{S_{vg}}{S_g} \right)
\]

- Probability of exploitation against HideM
  - \( \frac{S_{vg}}{S_g} \) is gadget specific
  - Replace \( \frac{S_{vg}}{S_g} \) with an observed average for the distribution

\[
\prod_{n=1}^{N} \left( \left( \frac{U_{vg} - n - 1}{U_g - n - 1} \right) \left( \frac{S_{vg}}{S_g} \right) \right)
\]

• Use two tools to find ROP gadgets in memory
  – ROPGadget
  – RP++

• Gadgets limited to 4 bytes in length

• Calculate probability of exploitation for tested binaries given N=1
  – Only one valid gadget required to exploit

- 5 highest and lowest exploit probability (N=1)

<table>
<thead>
<tr>
<th>Binary Name</th>
<th>Before HideM</th>
<th>After HideM</th>
<th>Distribution</th>
<th>Guessing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exec HideM (A/U)</td>
<td>Valid HideM (A/U)</td>
<td>(S_{vg} / S_{g})</td>
<td>Exec HideM (A/U)</td>
<td>Valid HideM (A/U)</td>
</tr>
<tr>
<td>dumpcap</td>
<td>3/3</td>
<td>0/0</td>
<td>0.0</td>
<td>2270/1817</td>
</tr>
<tr>
<td>mcf</td>
<td>118/22</td>
<td>2/2</td>
<td>.0734</td>
<td>73/61k</td>
</tr>
<tr>
<td>h264ref</td>
<td>226/20</td>
<td>1/1</td>
<td>.1429</td>
<td>141/118k</td>
</tr>
<tr>
<td>lbm</td>
<td>127/20</td>
<td>1/1</td>
<td>.2000</td>
<td>76/64k</td>
</tr>
<tr>
<td>bzip2</td>
<td>143/21</td>
<td>2/2</td>
<td>.1397</td>
<td>78/65k</td>
</tr>
<tr>
<td>dgimp</td>
<td>519/21</td>
<td>45/17</td>
<td>.0869</td>
<td>353/286k</td>
</tr>
<tr>
<td>wireshark</td>
<td>197/22</td>
<td>20/14</td>
<td>.1579</td>
<td>128/106k</td>
</tr>
<tr>
<td>lynx</td>
<td>164/20</td>
<td>15/11</td>
<td>.1932</td>
<td>107/89k</td>
</tr>
<tr>
<td>gedit</td>
<td>54/19</td>
<td>6/5</td>
<td>.5067</td>
<td>37/31k</td>
</tr>
<tr>
<td>lyx</td>
<td>1387/28</td>
<td>236/21</td>
<td>.2124</td>
<td>789/626k</td>
</tr>
</tbody>
</table>

- 10 highest exploit probability
- Gadgets limited to 4 bytes
Conclusion

• HideM provides protection against code disclosure
  – Hides codes from being read
  – Applies code reading policy to enable selective fine-grained reads of code
    • Supports C++ exception handling

• Supports COTS binaries
  – Identifies data locations through offline static analysis, minimal manual verification

• Existing systems can be retrofitted for protection
• Limited impact on performance
Thanks

- Questions?

✉ jjgionta@ncsu.edu
💻 gionta.org