Nibbler: Debloating Binary Shared Libraries

Ioannis Agadakos, SRI International
Di Jin, Brown University
David W. King, Columbia University
Vasileios P. Kemerlis, Brown University
Georgios Portokalidis, Stevens
Software Exploitation with Code Injection

- Software exploitation through code injection by Elias Levi ~1996
Software Exploitation with Code Re-use attacks

- With **DEP** injecting and executing code is **no longer possible**
- Attackers must re-use parts (gadgets) from the existing app **code base**
Program Code Base

• Can be split in two broad parts:
  1. Main application code
  2. Shared library code
Application Code Bloating with Interest

VLC Media player ~ x50, Δτ=19 years

Chrome Browser ~ x17, Δτ=9 years

*Source openhub, https://www.openhub.net
Shared Library Bloating

- Code added due to library generalization
  - New features
  - Backwards compatibility
- Applications may need only a **subset** of functions
  - As time progresses libraries become generalized
    - libc Debian 2.0= **1952** symbols, Debian 8.8= **2890**
What is our Goal?

“Disable unused code (external functions) from an application’s shared libraries without recompilation”
2. Nibbler: Overview and Design

- Applications do not need all accessible library code (functions)
- Disable unused code without changing memory layout
- Without recompiling
- Without breaking apps

Assumptions

- Metadata
  - Relocation Information, Symbols
- Binary code with:
  - No obfuscation
  - No mixing data with code
- No dynamic loading (dlopen, dlsym)
  - We handle it, **but with weaker** guarantees
Overview

• Structurally identical libs
• But.. Code unused is disabled (int3)
• In the end the app utilizes libraries with disabled code

In the end..
• LD...LDPATH=./app/nibbled/
• RPATH=/app/nibbled/

Disassembly

FCG Reconstruction

Compose FCGs and detect unused functions

Database

Thinned Libraries

Symbols
Relocations
Erased code

© 2018 SRI International. All Rights Reserved. Proprietary
Application Hot Path – Code that is Needed

- To collect all dependencies:
  - Analyze main programs code
  - Find external direct symbols referenced
  - Detect direct symbol dependencies
  - Detect indirect symbol dependencies

0. System Startup
1. Main
2. Direct References
3. Indirect References
4. dlopen
0. System Startup Sequence

Linux Program Startup. Figure by Patrick Hogan
**Function Call Graph Recovery – In module**

1. Local calls are resolved implicitly
2. External references are stubs
3. Address Taken references are recorded
4. We do not try to resolve targets
FCG Fallacies - Part 1
(libc where else..)

- Symbols with the same label

<table>
<thead>
<tr>
<th>Value</th>
<th>Size</th>
<th>Type</th>
<th>Bind</th>
<th>Vis</th>
<th>Ndx</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1f577</td>
<td>74</td>
<td>FUNC LOCAL DEFAULT</td>
<td>0</td>
<td>11</td>
<td>read_int</td>
<td></td>
</tr>
<tr>
<td>685a0</td>
<td>90</td>
<td>FUNC LOCAL DEFAULT</td>
<td>0</td>
<td>11</td>
<td>read_int</td>
<td></td>
</tr>
<tr>
<td>68cb0</td>
<td>84</td>
<td>FUNC LOCAL DEFAULT</td>
<td>0</td>
<td>11</td>
<td>read_int</td>
<td></td>
</tr>
</tbody>
</table>

- Fall through functions, engulfed functions
FCG Fallacies- Part 2

• Symbol visibility & binding

<table>
<thead>
<tr>
<th>Value</th>
<th>Size</th>
<th>Type</th>
<th>Bind</th>
<th>Vis</th>
<th>Ndx</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>6d310</td>
<td>292</td>
<td>FUNC</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>12</td>
<td>putchar@GLIBC_2.2.5</td>
</tr>
<tr>
<td>eb2e0</td>
<td>153</td>
<td>FUNC</td>
<td>WEAK</td>
<td>DEFAULT</td>
<td>12</td>
<td>iswalpha@GLIBC_2.2.5</td>
</tr>
<tr>
<td>22130</td>
<td>170</td>
<td>FUNC</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>12</td>
<td>strip</td>
</tr>
<tr>
<td>8bcc0</td>
<td>45</td>
<td>IFUNC</td>
<td>WEAK</td>
<td>DEFAULT</td>
<td>12</td>
<td>strcasestr@GLIBC_2.2.5</td>
</tr>
</tbody>
</table>

• Function Resolvers (IFUNC) special handling
  • malloc, memcpy, strcpy etc

• Add all AT’s from all libraries
  • Context sensitive improvement removes some AT’s

• System specific sequences
Context Sensitive Approach

- Removes some AT functions
- Go over the recovered FCG and tag all taken function addresses
  - Detect address transfer through eg. mov instructions
- If a function is AT and it’s not in rodata or taken in the recovered FCG; **remove it**
- Sanitization of function pointer arrays/structures
Composing Application Scope FCG

1. Create a dependency graph
2. Do reverse topological ordering
   1. In linux ld.so is always in the bottom, app on top
3. Start building full FCG by bottom up composition
4. Observe the loader’s rules for each system e.g. Strong, Weak symbols

Library Thinning

- The nodes in the reconstructed graph are needed functions
- The complementary of that set is the un-needed code
  - Overwrite each byte by ‘int3’ inst (0xCC)
  - Topology of the binary does not changed
Notes on Dynamic Loading

- Profiling is **required** to identify symbols

Consider the following case taken from nsswitch.c in

```c
/*Construct shared object name*/
__stpcpy(__stpcpy(__stpcpy(__stpcpy(shlib_name,
    "libnss_"),
   ni->name),".so"),
__nss_shlib_revision);
ni->library->lib_handle=__libc_dlopen(shlib_name);
```

- **Methodology**
  - Use test cases provided with application
  - Run and invoke common functionality
    - Eg. See a video in YouTube in chrome

<table>
<thead>
<tr>
<th>Total Packages</th>
<th>C/C++</th>
<th>Dlopen and/or asm</th>
</tr>
</thead>
<tbody>
<tr>
<td>25,526</td>
<td>9,792</td>
<td>1,351 (13.8%)</td>
</tr>
</tbody>
</table>

Ubuntu 18.08 bionic universe
Three (3) different O/S
  - Debian 8, Debian 9, and Ubuntu 16.04
Diverse application test bed more than 200 binaries
  - Nginx, MySql, SPEC CINT2016, Coreutils
  - GUI and CLI apps
  - C/C++
    - Chromium, ~146 library dependencies!!!

How do we know we don’t break stuff?
  - We use bundled test cases!
## Overview: Disabled Code

<table>
<thead>
<tr>
<th>App(s)</th>
<th>Set Size</th>
<th>Code Reduction (Lib. Set)</th>
<th>Functions</th>
<th>Code (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of Bin.</td>
<td># of Lib.</td>
<td>Vanilla</td>
<td>Removed</td>
</tr>
<tr>
<td>a) Coreutils</td>
<td>104</td>
<td>11</td>
<td>4754</td>
<td>2796</td>
</tr>
<tr>
<td>b) SPEC</td>
<td>11</td>
<td>5</td>
<td>7808</td>
<td>5729</td>
</tr>
<tr>
<td>c) Nginx</td>
<td>1</td>
<td>7</td>
<td>8599</td>
<td>7015</td>
</tr>
<tr>
<td>d) MySQL</td>
<td>1</td>
<td>8</td>
<td>7979</td>
<td>5524</td>
</tr>
<tr>
<td>a) + b) + c) + d)</td>
<td>117</td>
<td>16</td>
<td>14438</td>
<td>10622</td>
</tr>
</tbody>
</table>

### Coreutils (libc-2.19)
- Symbols present in every program in the set: 36.4%
- Symbols not present in any program in the set: 5.0%
- Mixed: 58.6%

### Debian 8 (libc-2.19)
- Symbols present in every program in the set: 37.2%
- Symbols not present in any program in the set: 21.0%
- Mixed: 41.8%
Versus Static Compilation

- However:
  - Static code space is not exactly 1-1 with dynamic
  - Functions that are added in static compilation do not always exist in dynamic scope (e.g. glibc static)
  - Shared libraries might have different editions for static functions (e.g. libdl)
Notes for Gadget Reduction

- We use 8 categories (Arithmetic, Logic etc) as defined in “Just-In-Time Code Reuse” (Snow)
- Overall gadget reduction is proportional to code reduction
- In some cases achieved 100% reduction of some categories
- Just by using Nibbler CFI schemes can be augmented
  - Less valid targets
  - Less valid backward edges
- Although Nibbler was able to defend against 3 real exploits
  - Attacks are still possible e.g. Nginx (CVE-2013-20283),

![Gadget Reduction Diagram]
Is Gadget Reduction a good metric?

- A natural question is that removed code was not used in the first place.
- This is true (mostly), there can be cases where unneeded code is actually executed and could trigger a vulnerability.
- Even though not used these functions act as a dead weight for highly effective defenses such as continuous rerandomization.

```
<table>
<thead>
<tr>
<th>main</th>
<th>libx.so</th>
<th>libz.so</th>
</tr>
</thead>
<tbody>
<tr>
<td>symA@liba</td>
<td>symA@libx</td>
<td>syma@libz</td>
</tr>
<tr>
<td></td>
<td>symb@libx</td>
<td>symb@libz</td>
</tr>
<tr>
<td></td>
<td>symc@libc</td>
<td>symc@libz</td>
</tr>
</tbody>
</table>
```
Empowering Continuous Re-randomization: Shuffler

- We used the ``nibbled'' version
- ~20% Improvement on Nginx
- Linear improvement when cpu is the bottleneck
- While the reduction we achieved overall in Nginx is ~80%:
  - Shuffler actually has extra dependencies libshuffle.so
  - At the time, libcrypto which is heavily ``nibbled'' was not supported
  - by shuffler so all of the symbols were includes
4. Related Work

- Automated software winnowing (Malenca et al, 2015)
  - Removes unused functions
  - Requires source code/recompilation

- CodeFreeze (black hat 2015)
  - Windows
  - Fuzzing/whitelisting
  - Runs at load time

- Debloating software through Piece-Wise Compilation and Loading (Usenix 2018 Ahn Quach et al)
  Libraries **must** be compiled to include a FCG in a special section
  Runs on loading time
  Similar results
  Handles dynamic case as Nibbler

- Bintrimmer Redini et al (DIMVA 2019)
  - Uses VSA to reduce potential targets

It is very interesting that all related systems have similar results!!
Conclusions & Future Plans

• Tackled the phenomenon of library bloating
• Studied thoroughly In more than 200 binaries we showed that:
  • We can remove 73.57% of the Functions in scope corresponding to 48% of code and 52% of gadgets
  • Without recompilation or source code
  • Without breaking applications in a sound way
  • Zero performance overhead, minimal memory cost (common thinned libraries)
• We augment existing defenses:
  • CFI particularly in regard to backward edges
  • Continuous re-randomization techniques in a fashion linear to the code removed
• Identified that there is an additional vector –code that cannot be removed- that needs to be addressed and offered directions for future research