Semi-automated Feature-Debloating of Binary Software

Dr. Kevin W. Hamlen
Eugene McDermott Professor
Computer Science Department
The University of Texas at Dallas

ONR TPCP Software Security Summer School (SSSS)
August 3, 2020


*supported in part by ONR Award N00014-17-1-2995, NSF Award #1513704, and an endowment from the Eugene McDermott family.
Binary Control-flow Trimming

- **Objective:** Erase (“debloat”) unwanted/unneeded features in binary software without the aid of source code

- **Motivating Example:** Linux Bash + Shellshock

  - Discovered September 2014
  - Bash shells execute certain environment variable texts as code(!!)
  - Allows attackers to remote-compromise most Linux systems
  - Window of vulnerability: 25 years(!!)
  - Probably NOT originally a bug!
    - introduced in 1989 to facilitate function-import into child shells
    - never clearly documented, eventually forgotten
Research Challenges

- Can we automatically erase unneeded (risky) functionalities from binary software?
  - Admins might not even know that the undesired functionality exists, and therefore cannot necessarily demonstrate bugs/vulnerabilities.
  - Demonstration of desired functionalities will usually be incomplete.
    - Large input spaces (e.g., unbounded streams of network packets)
  - No assumptions about code design/provenance
    - Arbitrary source languages
    - Arbitrary compilation toolchains
    - Simplifying assumption: not obfuscated (we can at least disassemble it)

- Can we do so without introducing significant inefficiencies?
  - No virtualization layers introduced
  - “Debloated” code should be runnable on bare hardware
Basic Workflow

(1) Demonstrate representative desired functionalities by running the target software on various inputs in an emulator/VM.

(2) Submit resulting logs along with original binary code to de-bloater.

(3) If resulting de-bloated binary is unsatisfactory (e.g., needed functionalities missing), then repeat with more/better tests.
Binary Control-flow Trimming Architecture

original binary

test suite

traces

conservative disassembler

IRM rewriter

trimming policy (CCFG)

policy learner

trimmed binary
Stepwise Usage

1. CCFI-protect binary with a permit-all policy
   - `rewriter-makeout.py --learn --target $BCFT_TARGET_BINARY`

2. run new binary in emulator (PIN) on training inputs
   - `pin -i ... -o ... -- $PROGRAM $ARGS`

3. learn a CCFI policy from the traces logged by the emulator
   - `learner.py $PROGRAM_TRACES_DIR`

4. replace the permit-all policy with the learned policy
   - `rewriter-makeout.py --policy $POLICY_FILE --target $BCFT_BINARY`
Experiments and Evaluations

- **Performance:**
  - SPEC CPU Benchmark.
  - Lighttpd, Nginx web-servers.
  - Proftpd, pureftpd, vsftpd ftp-servers.

- **Test-suite for accuracy and security:**

<table>
<thead>
<tr>
<th>Program</th>
<th>Test Suite</th>
<th>Debloated Functionalities</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>Its own source code.</td>
<td>-m32 (accuracy)</td>
</tr>
<tr>
<td>Ftp-servers</td>
<td>Random files mixed with commands (e.g. rm).</td>
<td>SITE, DELETE (security, accuracy)</td>
</tr>
<tr>
<td>Browsers</td>
<td>Quantcast top 475K URLs.</td>
<td>Incognito, cookies</td>
</tr>
<tr>
<td></td>
<td></td>
<td>add/delete(accuracy)</td>
</tr>
<tr>
<td>ImageMagic convert</td>
<td>Converting random jpps to png.</td>
<td>resizing(accuracy)</td>
</tr>
<tr>
<td>Exim</td>
<td>Random emails to a specific address.</td>
<td>-ps (security), -oMs(accuracy)</td>
</tr>
<tr>
<td>Node.js</td>
<td>Java scrip code not using serialize().</td>
<td>serialize()(security)</td>
</tr>
</tbody>
</table>
Successfully removed Shellshock vulnerability using only the pre-Shellshock test-suite shipped with bash.

<table>
<thead>
<tr>
<th>Program</th>
<th>CVE numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bash</td>
<td>CVE-2014-6271, -6277, -6278, -7169</td>
</tr>
<tr>
<td>ImageMagic</td>
<td>CVE-2016-3714, -3715, -3716, -3717, -3718</td>
</tr>
<tr>
<td>Proftpd</td>
<td>CVE-2015-3306</td>
</tr>
<tr>
<td>Node.js</td>
<td>CVE-2017-5941</td>
</tr>
<tr>
<td>Exim</td>
<td>CVE-2016-1531</td>
</tr>
</tbody>
</table>
Limitations and Scope

➢ **DON’T** use this if...
  - … you have full source code and can recompile all system components.
  - … you want to shrink the software’s memory image.
  - … it is difficult/impossible to demonstrate all critical functionalities.
    - (In future research we want to relax this restriction.)

➢ **DO** use this if...
  - … you don’t have or don’t trust some/all of the source code for the software.
  - … the software has *no formal specification* of correctness/security.
  - … you have no developer cooperation for finding/fixing bugs/features.
  - … you want to run the code natively (no VM).
Obvious Approach: Code Byte Erasure
Obvious Approach: Code Byte Erasure
Obvious Approach: Code Byte Erasure

Two Problems:

(1) Too much gets erased (needed functionalities broken)
(2) Too many “bad” functionalities retained!
void access_database() {
    bool (*check)(void);
    char vul_buf[N];
    check = &security_check;
    ...
    scanf("%s", vul_buf);
    if (check()) {
        grant_privileges();
    }
}
void access_database() {
    bool (*check)(void);
    char vul_buf[N];
    if (authenticated)
        check = weak_check;
    else
        check = strong_check;
    scanf("%s", vul_buf);
    if (check()) {
        grant_privileges();
    }
}
Contextual Control-flow Integrity (CCFI)

- Basic implementation strategy
  - Replace each jump/branch/call instruction in the original code with a *check-then-jump* sequence
  - The “check” code updates and consults a saved *context history* of previous jumps.

- Requirements
  - ALL jump/branch/calls must be replaced
  - saved context history must be protected from attacker modification

- Prior work
  - non-contextual CFI enforcement is well-established
  - contextual CFI is very hard to implement efficiently
    - PathArmor [Van Der Veen et al.; USENIX Sec ‘15]: only checks system API calls, has high overhead

- Main challenge #1: How to learn a CCFI policy without a spec?
- Main challenge #2: How to enforce such fine-grained CCFI efficiently?
Learning CFG Policy

- Decision Trees at every branch site.

What is the impending target?

Target t1
- Target t2
- Target t3

YES

YES

NO
Learning Contextual CFG Policy

origin o

Target t1
Target t2
Target t3
Target t4
Target t5

What is the impending target?
What was the target before that?
Or even before that?
void access_database() {
  bool (*check)(void);
  char vul_buf[N];

  if (authenticated)
    check = weak_check;
  else
    check = strong_check;

  scanf("%s", vul_buf);

  if (check()) {
    grant_privileges();
  }
}
Policy Representation

- Lookup table.

\[
hash(\chi) = \bigoplus_{i=1}^{|\chi|} (\pi_2 \chi_i) \ll (|\chi| - i)s
\]

\[
hash(\chi e) = (hash(\chi) \ll s) \oplus (\pi_2 e)
\]
Hash Table Sizes

A table of size $n$ B can whitelist $8n$ contexts.
# Guard Checks

<table>
<thead>
<tr>
<th>Description</th>
<th>Original code</th>
<th>Rewritten Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional Jumps</td>
<td>\textit{\texttt{jcc }l}</td>
<td>call \textit{\texttt{jcc_fall} .quad }l</td>
</tr>
<tr>
<td>Indirect calls</td>
<td>call \texttt{r/[m]}</td>
<td>\texttt{mov r/[m], %rax} call \texttt{indirect_call}</td>
</tr>
<tr>
<td>Indirect Jumps</td>
<td>\texttt{jmp r/[m]}</td>
<td>\texttt{mov %rax, -16(%rsp)} \texttt{mov r/[m], %rax} call \texttt{indirect_jump}</td>
</tr>
<tr>
<td>Variable Returns</td>
<td>\texttt{ret }n</td>
<td>\texttt{pop %rdx} \texttt{lea n(%rsp), %rsp} \texttt{push %rdx} \texttt{jmp return}</td>
</tr>
<tr>
<td>Returns</td>
<td>\texttt{ret}</td>
<td>\texttt{mov (%rsp), %rdx} \texttt{jmp return}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Label</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{indirect_jump:}</td>
<td>push %rax common-guard mov -8(%rsp), %rax ret</td>
</tr>
<tr>
<td>\texttt{indirect_call:}</td>
<td>push %rax common-guard mov \texttt{jmp return}</td>
</tr>
<tr>
<td>\texttt{return:}</td>
<td>common-guard ret</td>
</tr>
<tr>
<td>\texttt{jcc_fall:}</td>
<td>jcc \texttt{jmp fall_l}</td>
</tr>
<tr>
<td>\texttt{jcc_back:}</td>
<td>jcc \texttt{jmp back_l}</td>
</tr>
<tr>
<td>\texttt{jump_l:}</td>
<td>xchg (%rsp), %rax mov (%rax), %rax \texttt{jmp condition_jump}</td>
</tr>
<tr>
<td>\texttt{fall_l:}</td>
<td>xchg (%rsp), %rax \texttt{lea 8(%rax), %rax} \texttt{jmp condition_jump}</td>
</tr>
<tr>
<td>\texttt{back_l:}</td>
<td>xchg (%rsp), %rax \texttt{lea 8(%rax), %rax} xchg (%rsp), %rax \texttt{jmp return}</td>
</tr>
<tr>
<td>\texttt{condition_jump:}</td>
<td>push %rax common-guard pop %rax xchg (%rsp), %rax ret</td>
</tr>
</tbody>
</table>
## Context Protection with Wide Registers

<table>
<thead>
<tr>
<th>Guard Name</th>
<th>Legacy-mode</th>
<th>SHA-extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>before-check</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1:movd $r, %xmm11</td>
<td>1:movd $r, %xmm11</td>
</tr>
<tr>
<td></td>
<td>2:psubd %xmm12, %xmm11</td>
<td>2:psubd %xmm12, %xmm11</td>
</tr>
<tr>
<td></td>
<td>3:pxor %xmm11, %xmm13</td>
<td>3:sha1msg1 %xmm14, %xmm13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4:sha1msg2 %xmm13, %xmm13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5:psllrdq $4, %xmm13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6:pxor %xmm11, %xmm13</td>
</tr>
<tr>
<td>check</td>
<td>4:movd %xmm13, $r</td>
<td>7:movd %xmm13, $r</td>
</tr>
<tr>
<td></td>
<td>5:and (max_hash - 1), $r</td>
<td>8:and (max_hash - 1), $r</td>
</tr>
<tr>
<td></td>
<td>6:bt $r, (HASH_TABLE)</td>
<td>9:bt $r, (HASH_TABLE)</td>
</tr>
<tr>
<td></td>
<td>7:jnb TRAP</td>
<td>10:jnb TRAP</td>
</tr>
<tr>
<td>after-check</td>
<td>8:pextrd $3, %xmm14, $r</td>
<td>11:psllldq $4, %xmm14</td>
</tr>
<tr>
<td></td>
<td>9:psllrdq $4, %xmm14</td>
<td>12:psllw $1, %xmm14</td>
</tr>
<tr>
<td></td>
<td>10:pxor %xmm11, %xmm14</td>
<td>13:pxor %xmm11, %xmm14</td>
</tr>
<tr>
<td></td>
<td>11:movd $r, %xmm11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12:pxor %xmm11, %xmm13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>13:psllld $1, %xmm13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>14:psllld $1, %xmm14</td>
<td></td>
</tr>
</tbody>
</table>
Tuning Policy Strictness
Decision Trees and Entropy

- High entropy node = high uncertainty = incomplete testing

```c
void dispatch(void (*func)()) {
    func();
    LOG();
}
```
Relaxing the policy

- Relaxation philosophy:
  - Relaxed policy is always as strict as non-contextual CFI.
  - Relaxations merely identify some context as irrelevant to the enforcement decision.

- Parameters
  - $\lambda = \# \text{ times the node observed in all traces}$
  - $\gamma = \# \text{ traces in which node is observed}$
  - $N = \text{ total traces}$
  - $M = \# \text{ children}$

$$score(n) = \frac{\gamma}{N} - \frac{1}{M^2} \sum_{m=1}^{M} \frac{\lambda_m}{\lambda} \log_M \frac{\lambda_m}{\lambda}$$
# Accuracy

<table>
<thead>
<tr>
<th>Sample Size</th>
<th>proftpd</th>
<th>vsftpd</th>
<th>pure-ftpd</th>
<th>exim</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.48</td>
<td>0.38</td>
<td>0.41</td>
<td>0.25</td>
</tr>
<tr>
<td>100</td>
<td>0.37</td>
<td>0.23</td>
<td>0.28</td>
<td>0.53</td>
</tr>
<tr>
<td>FN</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sample Size</th>
<th>epiphany</th>
<th>uzbl</th>
<th>convert</th>
<th>gcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.93</td>
<td>0.92</td>
<td>0.45</td>
<td>0.64</td>
</tr>
<tr>
<td>100</td>
<td>0.81</td>
<td>0.83</td>
<td>0.64</td>
<td>0.54</td>
</tr>
<tr>
<td>FN</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>
Reachable Code Reduction

<table>
<thead>
<tr>
<th>Software</th>
<th>Code Reduction Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>proftpd</td>
<td>50</td>
</tr>
<tr>
<td>vsftpd</td>
<td>40</td>
</tr>
<tr>
<td>pure-ftpd</td>
<td>45</td>
</tr>
<tr>
<td>exim</td>
<td>90</td>
</tr>
<tr>
<td>convert</td>
<td>90</td>
</tr>
<tr>
<td>gcc</td>
<td>90</td>
</tr>
<tr>
<td>epiphany</td>
<td>70</td>
</tr>
<tr>
<td>uzbl</td>
<td>60</td>
</tr>
</tbody>
</table>
Run-time Overhead

![Bar chart showing run-time overhead for various benchmarks and applications. The chart displays the runtime overhead (%) for each item, with the median shown as well.]
CFI ≠ Debloating

- Policies enforced by prior CFI works:
  - Source-aware CFI solutions: CFG derived from source code semantics
  - Binary-only CFI solutions: Approximate the source CFG from binary semantics
  - Both approaches preserve developer-intended, consumer-unwanted edges.

- Prior contextual CFI solution:
  - PathArmor [Van Der Veen et al.; USENIX Security 2015]
    - Contextual checks only performed at system call sites
    - Insufficient granularity to debloat fine-grained code blocks from software
    - Performance overhead too high if applied to every branch instruction
Comparison with RAZOR [Qian et al. (USENIX’19)]

<table>
<thead>
<tr>
<th></th>
<th>RAZOR</th>
<th>Control-flow Trimming</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Strategy</strong></td>
<td>Heuristics applied to code structure and traces</td>
<td>Machine learning (decision trees)</td>
</tr>
<tr>
<td><strong>Policy Expressiveness</strong></td>
<td>Static CFI</td>
<td>Contextual CFI</td>
</tr>
<tr>
<td><strong>Debloating rate</strong></td>
<td>~71%</td>
<td>~71%</td>
</tr>
<tr>
<td><strong>Performance Overhead</strong></td>
<td>1.7%</td>
<td>1.9%</td>
</tr>
</tbody>
</table>
Conclusion

Main achievements
- Binary software debloating using incomplete test-suite and no source code
- First fine-grained contextual CFI enforcement at every branch site with high performance (1.8% overhead)

Challenges for Future Research / Transition
- Highly interactive software (diverse traces) can create high training burden. Could couple with directed fuzzers to improve training effectiveness.
- Training process automatically detects uncertainties and ambiguities. Feed this information back to (non-expert) users to help them refine the training?
THANK YOU

QUESTIONS?