Large-Scale Evaluation of a Vulnerability Analysis Framework

Nathan S. Evans, Azzedine Benameur, Matthew C. Elder
Symantec Research Labs
Outline

• Overview

• MINESTRONE: Architecture and Detection Technologies

• Test and Evaluation: Architecture, Test Suite, and Results

• Technology Improvements and Results

• Lessons Learned
Overview

• Software vulnerability detection is an ongoing research problem
  – Existing commercial tools address non-overlapping subsets of vulnerabilities, have high false positive rates, and require security expertise to use
  – Vulnerability detection experimentation/evaluation is an open problem

• IARPA STONESOUP Program: “Securely Taking On New Executable Software of Uncertain Provenance”
  – Develop and demonstrate technology that provides comprehensive, automated techniques that allow end users to safely execute new software of uncertain provenance
  – Addressing 8 “weakness” classes across 3 target language classes

• MINESTRONE Team: Columbia University (PI: Sal Stolfo), George Mason University (GMU), and Symantec
Overview: STONESOUP/MINESTRONE Program Targets

• Target classes of vulnerabilities (including example CWE numbers):
  – Number handling (e.g., integer overflow/underflow, sign conversion: #190, #191)
  – Resource drains (e.g., failure to release memory, structures, devices: #400, #404)
  – Tainted data/input validation errors (#78, #134)
  – Error handling (e.g., unhandled exceptions/error status codes: #248, #252)
  – SQL injection / command injection (#78, #89)
  – Concurrency handling (e.g., race conditions, thread safety: #362, #366)
  – Buffer overflows/underflows/out of bounds accesses/memory safety (#121, #122)
  – Null pointer errors (#476)

• Target language classes:
  – Type-safe languages (Java, C#)
  – Type-unsafe languages (C, C++)
  – Binaries (x86, Windows or Linux)
MINESTRONE: System Architecture

Software of Unknown Provenance (SOUP) → MINESTRONE System Composer → REASSURE Container → DYBOC Container(s) → ResMon Container → NumHandling Container
MINESTRONE: Detection Technologies

• DYBOC
  – Source-to-source transformation
  – Memory-based errors (overflows and underflows)
  – Runtime checks (custom memory allocator)

• REASSURE (TheRing)
  – Self-contained mechanism for healing software using rescue points
  – Detects program crashes and gracefully recovers

• ResMon
  – Resource monitoring tool
  – Monitors CPU, memory, disk space; enforces a threshold or absolute value

• Number Handling
  – Unsafe signed/unsigned conversion, overflow, etc.
  – Clang-based (enforced at runtime)
Test and Evaluation: Architecture

- MITRE developed testing framework and API
- We/Symantec developed the interface to interact with the test harness
Test and Evaluation: Test Suite Base Programs

- Relatively large real-world open source projects
  - Improvement over phase 1 test suite
  - Integration more difficult

- Base programs used:
  - Cherokee, 200KLOC
  - Nginx, 200KLOC
  - GNU wget, over 50KLOC
  - GNU grep, ~10KLOC
  - zshell, over 150KLOC
  - libwww, ~80KLOC
  - tcpdump, over 200KLOC
Test and Evaluation: Test Suite

• Vulnerability classes:
  – Null pointer: 1530 test cases
  – Memory corruption: 1810 test cases
  – Resource drain: 900 test cases
  – Number handling: 1237 test cases

• Input sources:
  – Environment variable
  – Command line arguments
  – File
  – Network
  – Shared Memory
  – Clipboard
Test and Evaluation: Results

• Terminology
  – Processed
  – Unaltered
  – Rendered unexploitable
  – Base Score
  – Final Score

• Caveats, issues:
  – Memory alignment required for many test cases
  – Our memory-based tools intentionally do not return aligned memory!
  – Clang issues building certain test cases
  – CIL issues building certain test cases
  – Some unintentional test case bugs
### Test and Evaluation: Results, Overall

<table>
<thead>
<tr>
<th>Weakness Class</th>
<th>Processed</th>
<th>Unaltered</th>
<th>Base Score</th>
<th>Final Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Corruption</td>
<td>90.00%</td>
<td>96.64%</td>
<td>84.35%</td>
<td>88.19%</td>
</tr>
<tr>
<td></td>
<td>(1629/1810)</td>
<td>(1558/1629)</td>
<td>(1374/1629)</td>
<td>(1374/1558)</td>
</tr>
<tr>
<td>Null Pointer</td>
<td>100%</td>
<td>82.55%</td>
<td>82.55%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>(1530/1530)</td>
<td>(1263/1530)</td>
<td>(1263/1530)</td>
<td>(1263/1263)</td>
</tr>
<tr>
<td>Resource Drains</td>
<td>90.56%</td>
<td>72.76%</td>
<td>54.85%</td>
<td>75.38%</td>
</tr>
<tr>
<td></td>
<td>(815/900)</td>
<td>(593/815)</td>
<td>(447/815)</td>
<td>(447/593)</td>
</tr>
<tr>
<td>Number Handling</td>
<td>90.78%</td>
<td>72.40%</td>
<td>49.78%</td>
<td>68.76%</td>
</tr>
<tr>
<td></td>
<td>(1123/1237)</td>
<td>(813/1123)</td>
<td>(559/1123)</td>
<td>(559/813)</td>
</tr>
</tbody>
</table>
Test and Evaluation: Results, Memory Corruption

- Overall: 88.19% final score
- CWEs successfully detected (>90%): 120, 124, 129, 170, 415, 416, 590, 761, 785, 805, 806, 822, 824, 843

<table>
<thead>
<tr>
<th>CWE</th>
<th>Processed</th>
<th>Unaltered</th>
<th>Base Score</th>
<th>Final Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>126</td>
<td>79.22%</td>
<td>100%</td>
<td>26.22%</td>
<td>26.22%</td>
</tr>
<tr>
<td></td>
<td>(61/77)</td>
<td>(61/61)</td>
<td>(16/61)</td>
<td>(16/61)</td>
</tr>
<tr>
<td>127</td>
<td>100%</td>
<td>87.23%</td>
<td>32.97%</td>
<td>37.8%</td>
</tr>
<tr>
<td></td>
<td>(94/94)</td>
<td>(82/94)</td>
<td>(31/94)</td>
<td>(31/82)</td>
</tr>
<tr>
<td>134</td>
<td>81.03%</td>
<td>94.68%</td>
<td>39.36%</td>
<td>41.57%</td>
</tr>
<tr>
<td></td>
<td>(94/116)</td>
<td>(89/94)</td>
<td>(37/94)</td>
<td>(37/89)</td>
</tr>
</tbody>
</table>
Test and Evaluation: Results, Number Handling

- Overall: 68.76% final score
- CWEs successfully detected: 194, 195, 197, 369

<table>
<thead>
<tr>
<th>CWE</th>
<th>Processed</th>
<th>Unaltered</th>
<th>Base Score</th>
<th>Final Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>196</td>
<td>97.36% (148/152)</td>
<td>70.94% (105/148)</td>
<td>0% (0/148)</td>
<td>0% (0/105)</td>
</tr>
<tr>
<td>682</td>
<td>98.98% (98/99)</td>
<td>71.42% (70/98)</td>
<td>4.08% (4/98)</td>
<td>5.71% (4/70)</td>
</tr>
<tr>
<td>839</td>
<td>96.63% (115/119)</td>
<td>77.39% (89/115)</td>
<td>11.3% (13/115)</td>
<td>14.6% (13/89)</td>
</tr>
</tbody>
</table>
Test and Evaluation: Results, Resource Drain

- Overall: 75.38% final score
- CWEs successfully detected: 401, 459, 674, 771, 773, 774, 775, 834

<table>
<thead>
<tr>
<th>CWE</th>
<th>Processed</th>
<th>Unaltered</th>
<th>Base Score</th>
<th>Final Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>789</td>
<td>93.2% (96/103)</td>
<td>69.79% (67/96)</td>
<td>34.37% (33/96)</td>
<td>49.25% (33/67)</td>
</tr>
<tr>
<td>835</td>
<td>94.62% (88/93)</td>
<td>72.72% (64/88)</td>
<td>2.27% (2/88)</td>
<td>3.12% (2/64)</td>
</tr>
</tbody>
</table>
Technology Improvements

• Format string vulnerability detection
  – During initial T&E, we had no protection against format string vulnerabilities
  – Implemented FormatGuard-inspired protection

• Improved stack-to-heap source transformation
  – Original stack-to-heap transform not fine grained
  – Modified CIL source to protect individual stack-allocated variables
Technology Improvements:
Format String Vulnerability Detection

```c
int main() {
    printf ("%08x.%08x.%08x.%08x.%08x\n", 42);
}
```

```c
int main() {
    safe_printf (1, "%08x.%08x.%08x.%08x.%08x\n", 42);
}

/* Implementation of safe version of printf */
int safe_printf (int num_args, const char *fmt, ...) {
    ...
    size_t count_args = parse_format (fmt, 0, NULL);
    if (count_args > num_args)
        fprintf (stderr, "Format string error!\n");
    ...
}
```
Technology Improvements: Improved Stack-to-Heap Source Transformation

```c
/* Original code */
void init () {
    char buf1[10]; char buf2[10];
    ...
    memset (buf1, 'a', 10);
    memset (buf2, 'b', 10);
    ...
}

/* Heapify transformed code, heap variable */
struct init_heap {
    char buf1[10]; char buf2[10];
};
void init () {
    struct init_heap *init_vars;
    init_vars = malloc (sizeof(struct init_heap));
    ...
    memset (init_vars->buf1, 'a', 10);
    memset (init_vars->buf2, 'b', 10);
    ...
    free (init_vars);
}
```
Technology Improvements: Improved Stack-to-Heap Source Transformation

/* Altered heapify transformed code, heap variable */
struct init_heap_buf1 {
  char buf1[10];
};
struct init_heap_buf2 {
  char buf2[10];
};
void init() {
  struct init_heap_buf1 *init_buf1;
  struct init_heap_buf2 *init_buf2;
  init_buf1 = malloc (sizeof(struct init_heap_buf1));
  init_buf2 = malloc (sizeof(struct init_heap_buf2));
...  memset (init_buf1->buf1, 'a', 10);
  memset (init_buf2->buf2, 'b', 10);
...  free (init_buf1); free (init_buf2);
Technology Improvements: Results

<table>
<thead>
<tr>
<th>CWE</th>
<th>Original Score</th>
<th>Original no memalign</th>
<th>Heapify score</th>
<th>Heapify no memalign</th>
<th>Cherokee excluded</th>
</tr>
</thead>
<tbody>
<tr>
<td>126</td>
<td>26.22% (16/61)</td>
<td>47.05% (16/34)</td>
<td>62.12% (41/66)</td>
<td>83.67% (41/49)</td>
<td>95.92% (47/49)</td>
</tr>
<tr>
<td>127</td>
<td>37.8% (31/82)</td>
<td>73.80% (31/42)</td>
<td>37.07% (33/89)</td>
<td>71.73% (33/46)</td>
<td>89.13% (41/46)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CWE</th>
<th>Original Score</th>
<th>Format score</th>
<th>Cherokee excluded</th>
</tr>
</thead>
<tbody>
<tr>
<td>134</td>
<td>41.57% (37/89)</td>
<td>87.17% (68/78)</td>
<td>98.72% (77/78)</td>
</tr>
</tbody>
</table>
Lessons Learned

• Building a large corpus of vulnerability injected programs is hard
  – Number handling test cases are hard: developer’s intent vs. vulnerability (whitelisting helped)
  – Resource consumption test cases should leverage a “learning phase” not ulimits

• Focus should be on easy to use repeatable test cases, not architecture

• Obfuscation of details makes everyone’s life harder
  – Scoring
  – Vulnerability source
  – Co-processes
Lessons Learned

• Assumptions/misunderstandings between performers/evaluators cause big problems
  – Memory alignment
  – Network configuration

• Scoring should be based on the program under test not the co-process output

• Discard all unnecessary files or you’ll need a 6Tb file share
Summary

• T&E was successful, more so than phase 1

• Difficult task to create large, reliable test suite, infrastructure

• Using open source software a double edged sword

• Scores improved with minor fixes (once IO pairs known)

• MINESTRONE continues on to phase 3
Thank you!

Nathan Evans  nathan_evans@symantec.com
Azzedine Benameur  azzedine_benameur@symantec.com
Matthew Elder  matthew_elder@symantec.com