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

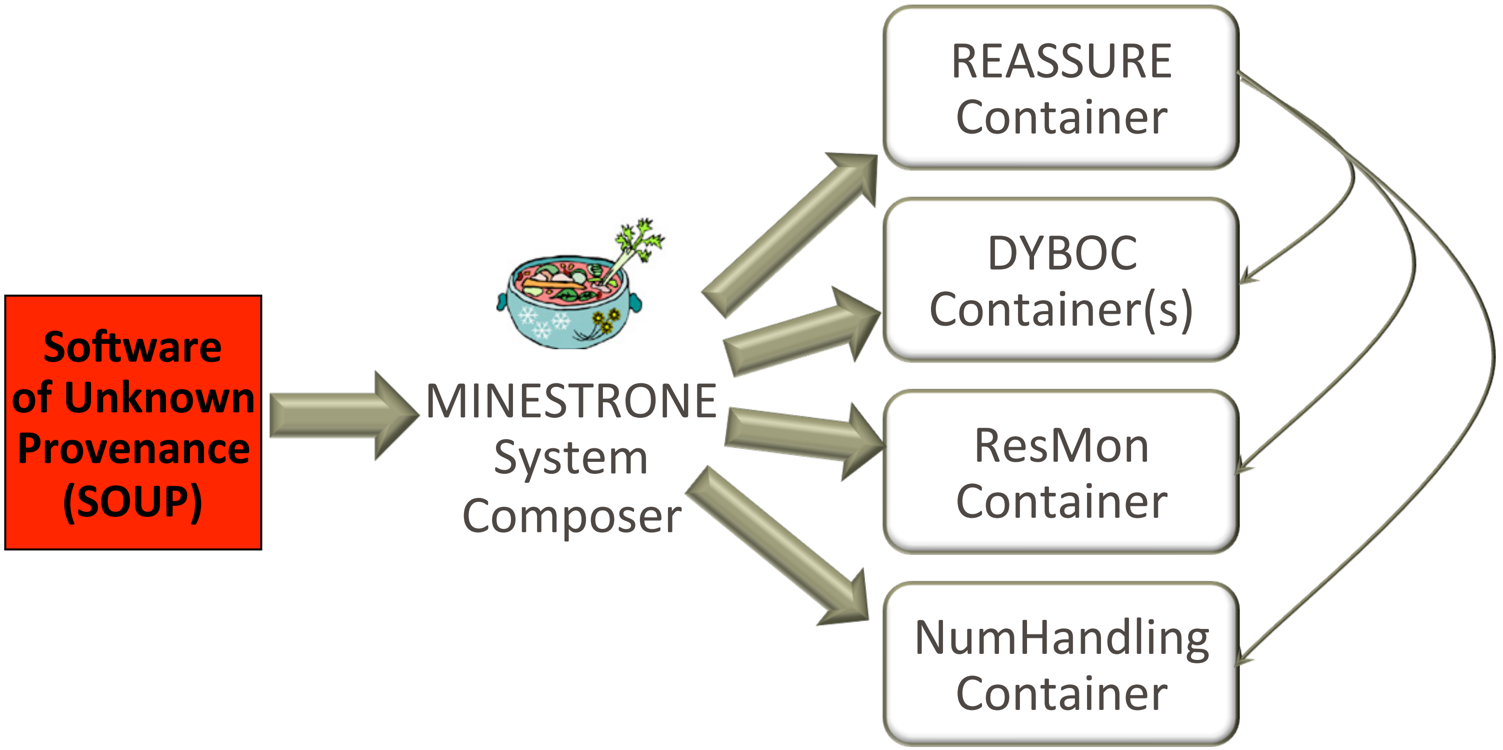
Overview

- 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



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

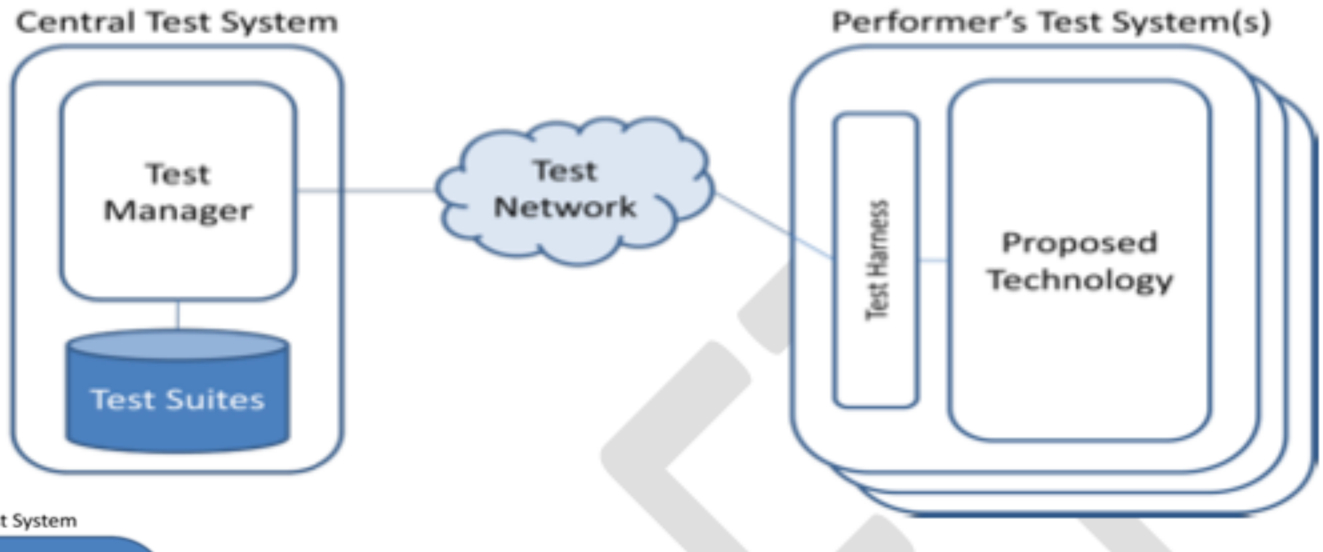


Figure 6-1 Test Environment

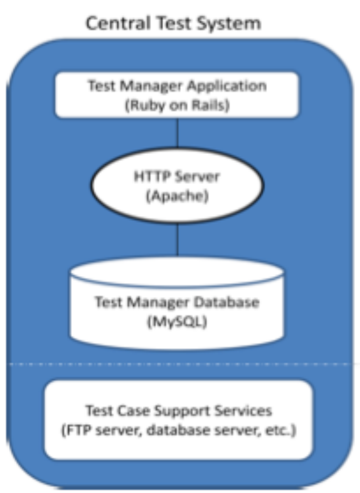


Figure 6-2 Central Test System

- 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

Weakness Class	Processed	Unaltered	Base Score	Final Score
Memory Corruption	90.00% (1629/1810)	96.64% (1558/1629)	84.35% (1374/1629)	88.19% (1374/1558)
Null Pointer	100% (1530/1530)	82.55% (1263/1530)	82.55% (1263/1530)	100% (1263/1263)
Resource Drains	90.56% (815/900)	72.76% (593/815)	54.85% (447/815)	75.38% (447/593)
Number Handling	90.78% (1123/1237)	72.40% (813/1123)	49.78% (559/1123)	68.76% (559/813)

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

CWE	Processed	Unaltered	Base Score	Final Score
126	79.22% (61/77)	100% (61/61)	26.22% (16/61)	26.22% (16/61)
127	100% (94/94)	87.23% (82/94)	32.97% (31/94)	37.8% (31/82)
134	81.03% (94/116)	94.68% (89/94)	39.36% (37/94)	41.57% (37/89)

Test and Evaluation: Results, Number Handling

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

CWE	Processed	Unaltered	Base Score	Final Score
196	97.36% (148/152)	70.94% (105/148)	0% (0/148)	0% (0/105)
682	98.98% (98/99)	71.42% (70/98)	4.08% (4/98)	5.71% (4/70)
839	96.63% (115/119)	77.39% (89/115)	11.3% (13/115)	14.6% (13/89)

Test and Evaluation: Results, Resource Drain

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

CWE	Processed	Unaltered	Base Score	Final Score
789	93.2% (96/103)	69.79% (67/96)	34.37% (33/96)	49.25% (33/67)
835	94.62% (88/93)	72.72% (64/88)	2.27% (2/88)	3.12% (2/64)

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

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

```
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

```
/* 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

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

CWE	Original Score	Format score	Cherokee excluded
134	41.57% (37/89)	87.17% (68/78)	98.72% (77/78)

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