Large-Scale Evaluation of a Vulnerability Analysis Framework

Nathan S. Evans
Symantec Research Labs
Herndon, VA
nathan_evans@symantec.com

Azzedine Benameur
Symantec Research Labs
Herndon, VA
azzedine_benameur@symantec.com

Matthew C. Elder
Symantec Research Labs
Herndon, VA
matthew_elder@symantec.com

***Abstract*—Ensuring that exploitable vulnerabilities do not exist in a piece of software written using type-unsafe languages (e.g., C/C++) is still a challenging, largely unsolved problem. Current commercial security tools are improving but still have shortcomings, including limited detection rates for certain vulnerability classes and high false-positive rates (which require a security expert's knowledge to analyze). To address this there is a great deal of ongoing research in software vulnerability detection and mitigation as well as in experimentation and evaluation of the associated software security tools. We present the second-generation prototype of the MINESTRONE architecture along with a large-scale evaluation conducted under the IARPA STONESOUP program. This second evaluation includes improvements in the scale and realism of the test suite with real-world test programs up to 200+KLOC. This paper presents three main contributions. First, we show that the MINESTRONE framework remains a useful tool for evaluating real-world software for security vulnerabilities. Second, we enhance the existing tools to provide detection of previously omitted vulnerabilities. Finally, we provide an analysis of the test corpus and give lessons learned from the test and evaluation.**

I. Introduction

Software security is, or should be, one of the most important concerns for development teams. As such, many secure development methodologies have arisen [1], [2]. While these methods prevent a number of naïve mistakes they can never prevent all vulnerabilities. Software companies rely extensively on vulnerability detection tools [3], [4] to help test software for security vulnerabilities. These tools significantly reduce the number of bugs and are fine grained. However, they are plagued by a large number of false positives, which requires expert domain knowledge to identify real vulnerabilities. The effectiveness of existing commercial tools is limited: a study by the NSA [5] demonstrated that the combination of five vulnerability detection tools against a C and C++ test suite left 40% of the vulnerabilities unidentified.

MINESTRONE [6] is a software vulnerability testing framework for C and C++ languages. It is a hybrid framework combining multiple tailored detection tools that works on both source code and binaries and provides an architecture for replicated execution and confinement. Previous experimentation using an independent test and evaluation suite demonstrated MINESTRONE's effectiveness at detecting two types of vulnerabilities: memory corruption and null pointer errors [7]. Research on MINESTRONE and experimentation with an improved test suite has continued and is described in this paper. In particular, the MINESTRONE prototype addresses and is tested against four vulnerability types now (the first two are new): number handling, resource drain, memory corruption, and null pointer errors. The test suite improvements include the usage of real-world programs for testing, which enables testing against much larger, more realistic test programs: from 1KLOC in [7] to 200+KLOC in the current paper.

The remainder of the paper is organized as follows. First, we present related work. Next, we detail our methodology and the test suite used for our experiment and present the experimental setup used for evaluation of our framework. We then present and discuss the results and improvements we have made to the current technologies to increase detection of vulnerabilities. Finally, we share our lessons learned and conclude.

II. Related Work

There are a number of existing commercial tools for finding vulnerabilities in source code. Previous studies found that these tools have significant gaps in detection capabilities across the tools and there is little overlap in the detection capabilities of the tools [5]. The end result is that in practice an organization needs to run as many different vulnerability detection tools as possible in order to get the best detection coverage. Another common drawback of vulnerability detection tools, especially those based on static analysis, is the high rate of false positive detections.

Symbolic execution tools [8] can be used to identify software bugs or vulnerabilities. While their results are fine grained they have proven to be hard to use with real-world programs [7] and suffer from a huge state exploration problem.

Fuzzing frameworks [9], [10] are used to test the robustness of a program but can also be used to identify bugs or vulnerabilities. They generally reveal major bugs that lead to a program crash but fail to identify fine-grained vulnerabilities.

The Common Weakness Enumeration (CWE) Initiative [11] is a community-developed effort aimed at maintaining a list of well-known software vulnerabilities. CWEs from this list are used by the STONESOUP test and evaluation team as a means of grouping vulnerabilities into weakness classes, which are tested against MINESTRONE during test and evaluation.

III. Hypothesis, Methodology, and Test Suite

A. Hypothesis and Methodology

The primary hypothesis that we test in this work is that the MINESTRONE framework, composed of multiple detection technologies combined with I/O redirection, can mitigate four different classes of common software weaknesses with a detection rate of 80%. The four software weakness classes that we test are memory corruption, null pointer, resource drain, and number handling errors. Mitigation, in the context of this research and as defined by the funding program, is defined in a particular manner as described in the following paragraph and described in more detail in [7]. The target detection rate of 80% in our hypothesis was set as the performance goal of the funding program.

Our methodology is straightforward and summarized here; for full details of the test infrastructure, test artifacts, and test result descriptions, please refer to [7]. In short, we were provided (by the project's independent test and evaluation team) a testing architecture comprised of a test manager and a number of test harnesses. The MINESTRONE framework is integrated into each individual test harness, which connects to the test manager to retrieve test cases to run. Each test case is comprised of both good and bad I/O pairs. The test harness runs each of these pairs, without knowing whether they are good or bad, and returns the result of each I/O pair to the test manager. This result includes whether or not MINESTRONE detected a vulnerability, and the results and side effects of program execution. Only if all good I/O pairs are not reported as "bad" and bad I/O pairs are reported as "bad" does the test case count as correctly "rendered unexploitable".

A secondary hypothesis that we test is that two important improvements that we made to the underlying MINESTRONE technologies can improve our detection rates for particular memory corruption CWEs.

For the results presented in this paper, we report on the test results that we achieved during the full test and evaluation, and then selectively report on the results of our technological improvements.

B. Test Suite

The test suite chosen for the second test and evaluation differs from the first test suite in a number of important ways. The first test suite was made up of a number of small, manufactured test cases. The vulnerabilities were hand written and hand injected into the code. For this reason, the test cases were limited in number (340 total) and littered with unintended vulnerabilities and program errors that limited the veracity of the evaluation.

The test suite for the second test and evaluation was made up of real-world open source projects ranging in size from tens of thousands to hundreds of thousands of lines of code. Choosing to use relatively large open source projects has a number of advantages over small hand written test cases. First, these projects usually have many contributors, and are open for scrutiny by the public. This generally leads to more testing, reducing the number of unintended program errors and vulnerabilities. Second, these code bases are large enough that vulnerabilities can be injected in a large number of places, allowing the input required to trigger the seeded code to be variegated and thus more thoroughly evaluate the detection technologies. Finally, using extant software allows the test and evaluation team to focus on creating realistic vulnerabilities and spend less time developing contrived test cases.

Another major change made for the second test and evaluation was the automatic seeding of vulnerabilities using a source rewriting framework called ROSE [12]. One main goal from IARPA was to have a test corpus where the vulnerabilities were automatically injected in different regions of the code base, so that performers would not be able to predict that a certain modified piece of code contained a vulnerability simply based on where it was located. ROSE provided the ability to combine vulnerabilities and base programs in virtually infinite combinations. Lastly, ROSE was used to obfuscate the base programs so that simply comparing the original did not reveal whether or not a vulnerability had been seeded. All in all, ROSE made it so the test corpus was not limited in size by the engineering effort required to create a test case (which was the limit in the first test and evaluation).

Here we list the software used as base programs for the test and evaluation described in this paper:

- **Cherokee** a fast and versatile, well-maintained multi-process web server, over 200KLOC.
- **Nginx** a web server purpose built for speed, well-maintained and popular, nearly 200KLOC.
- **GNU wget** a ubiquitous and full-featured command line web client, over 50KLOC.
- **GNU grep** well-known command line tool for pattern matching on files, nearly 10KLOC.

- **zshell** popular command interpreter with many features, over 150KLOC
- **libwww** relatively old, unmaintained command line web client and API, nearly 80KLOC.
- **tcpdump** widely used packet capture and packet analysis command line tool, over 200KLOC

IV. Experimental Setup

MINESTRONE is an architecture that enables the combination of multiple detection technologies into a single, integrated system via container-based virtualization and I/O redirection technology. The MINESTRONE framework uses lightweight virtualization technology (OpenVZ [13]) to isolate various vulnerability detection and mitigation technologies. A single composer controls the execution of programs in each of these containers and I/O redirection is used to replay user input in each of the containers. (For full details of the MINESTRONE architecture, we refer the reader to our previous work [7].)

A. Detection Components

In this section we present the core detection technology components that comprise MINESTRONE, which are key to the vulnerability detection capabilities.

DYBOC [14] is a source-to-source transformation tool that augments the source code to detect stack- and heap-based buffer overflow and underflow attacks.

REASSURE is an error recovery mechanism that allows the software to recover from unforeseen errors or vulnerabilities [15], [16]. It reuses existing code locations that handle certain anticipated errors for unanticipated ones as well.

ResMon is a resource monitoring tool that stops excessive usage of resources, including file descriptors, network traffic, CPU, memory, and number of processes.

IOC number handling is a Clang-based tool which inserts dynamic checks on numerical operations to detect common errors, such as unsafe unsigned-to-signed conversion and addition/subtraction.

1) Format String Vulnerability Detection: One weakness included in the class of memory corruption test cases is the uncontrolled format string vulnerability CWE-134. Attackers can use this weakness to leak stack addresses; information which can be used to exploit the program. This vulnerability has been around as long as the `printf` family of functions. The issue is that at compile time (or source code analysis time) the number of arguments to these functions is known, but if the format string is variable it is difficult or impossible to count the number of and type of arguments. When more arguments are specified than were supplied, the memory addresses on the stack immediately following the legitimate arguments are accessed. An example of code vulnerable to this weakness is given in Figure 1[1].

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

Fig. 1. Trivial example of format string vulnerability.

```
int main() {
  safe_printf ("%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");
  ...
}
```

Fig. 2. Example of format string solution.

The solution to this problem is to combine static analysis with runtime analysis. Specifically, at or before compilation, any `printf`-style functions are augmented to include the actual number of arguments as the first argument to `safe_printf`-style functions, which at runtime verify that the format string supplied contains the correct (or fewer) arguments than are available. This solution was first described in FormatGuard [17], and we have implemented a similar solution in MINESTRONE to prevent these types of vulnerabilities. We demonstrate the result of the source-to-source transformation of the program given in Figure 1 and the safe version of the `printf` function in Figure 2.

Our implementation uses tools from CIL [18] (described in IV-A2) to automatically transform source files before compiling to replace any of the `printf`-style functions with `safe_printf` versions. The main difference between the FormatGuard solution and ours is in the implementation. FormatGuard required the inclusion of header files that used `#DEFINE`'s to overload the built in functions and automatically replace them with their safe versions. Our implementation uses source code rewriting to replace these functions, which we believe makes the protection a bit more flexible (no header includes or modification is necessary). It also makes the resulting code more easily readable, as the source is modified and the safe functions are easily recognized. FormatGuard also relied on merging the runtime protection into a custom-built libc library, with the expectation that all applications running on the system would be built with the safe versions of the vulnerable functions. In our case, we simply link each application with a library that

[1]This example is contrived; in this case static analysis could easily detect the problem. Normally the format string would be read from user input, but we place it here for sake of visibility.

```
/* 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);
}
```

Fig. 3. Example of base CIL stack to heap 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);
}
```

Fig. 4. Example of new CIL stack to heap transformation.

includes the protected functions. This can be done as the program is built, or at runtime with `LD_PRELOAD`. Again, this is a small distinction but it makes our version a bit easier to use as it works with stock versions of libc.

While the FormatGuard concept has been around for a long time, it has not been widely adopted. Our implementation and addition to the MINESTRONE framework significantly increased our detection of these format string vulnerabilities in the test corpus we were tested against. These results are detailed in Section V-C.

2) Improved Stack-to-Heap Transform: We protect against stack buffer over/under flows by transforming every stack buffer allocation to a heap-allocated buffer. These heap-allocated buffers that were moved from the stack are then protected by the DYBOC technology, which wraps memory allocations with guard pages to detect over and under-flows. In order to achieve this transformation, we leverage a source transformation framework called CIL [18].

Our initial implementation of the stack-to-heap buffer transformation utilized an existing module in CIL, called "heapify". This built-in transformation moves all stack-allocated arrays per function into a newly created struct for which a pointer is created. This pointer is then assigned memory sufficient for the function variables at the function entry point, and it is `free`'d upon function return. A simple example showing the initial code and the heapify-transformed code is given in Figure 3.

This transformation works fine to protect against stack corruption, but it unfortunately only moves any buffer over or under runs to the heap. Thanks to tricks of modern compilers, the program stack generally has protections against traditional uses of overflows. For instance, in the example shown in Figure 3, imagine a test case which attempts to alter the contents of `buf2` by overflowing `buf1`. On most modern systems, this overflow will not hit `buf2` due to stack alignment protections. However, with the code transformed to the heap using the CIL built-in heapify alteration, this type of overflow will be allowed (and cause the "expected" functionality). It could be argued that the overflow is actually mitigated by transforming the source in this way, as the program is only accessing valid memory inside of a struct. It is not uncommon to find C code with known memory offsets directly into structs as opposed to using member names. We believe that those direct accesses should be allowed, but that we still need to protect against the overflow of stack variables.

Our solution is to modify the CIL source to add a new heapify transformation that makes fine-grained stack overflows detectable. The implementation is straightforward: instead of allocating a single struct for all array variables declared per function, we allocate a struct for *each* array variable, regardless of the parent function that it exists in. An example of the resultant transformed code (for the same source as in Figure 3) is given in Figure 4.

We combine our modified stack-to-heap transformation with the heap protection functionality provided by DYBOC to catch these fine-grained stack overflows. While this additional step may seem like overkill, we have found it to enhance our detection of injected stack overflows in the test corpus used by the independent test and evaluation team to test the MINESTRONE framework. These improvements are discussed in Section V-C.

B. Experimental Environment

Our target experimental environment for this test and evaluation is a 32-bit Linux system with OpenVZ container virtualization enabled. Each of the detection components is placed in its own lightweight replica. Source code (C/C++) is provided per test case.

V. Experimental Results

A. Terminology

The following terminology is used to present results:

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)

TABLE I
SUMMARY OF SCORING RESULTS ON THE MEMORY CORRUPTION TEST CASES.

Processed: The base stage with no instrumentation was successful at running the test case.

Unaltered: The test program with instrumentation and good inputs had the expected output.

Base Score: Percentage of test cases correctly *rendered unexploitable* for all bad I/O pairs and the execution was *unaltered* for all good I/O pairs, out of all processed test cases.

Final Score: This score represents the percentage of test cases for which a technology correctly *rendered unexploitable* all bad I/O pairs and the execution was *unaltered* for all good I/O pairs, out of all *unaltered* test cases. This is the final score that is used for evaluation, as the test cases with altered functionality were caused by errors in test case design, not due to MINESTRONE altering execution.

B. Detailed Results

Table I shows the scoring results for the four weakness classes evaluated by the MINESTRONE framework. It gives the combined results incorporating all the detection technologies. The memory corruption scores are nearly 90%, however some of the weaknesses we expected to handle were not. We explore why this occurred, and our solutions, in Section V-C. The null pointer test case results are relatively uninteresting. These errors are almost trivially detected by the REASSURE component, and after controlling for program alteration due to test case and evaluation problems we see a clear 100% detection rate. We can also see that the resource drain and number handling weakness classes had both the lowest percentage of unaltered execution, and the lowest base and final scores. The results for the number handling test cases are well below our target of 80% detection. The discussion following Tables III and VII serves to explain why this was the case.

The final scores for the resource drain test cases are good, though it must be noted that the base score is significantly lower than the final score. Part of the reason for the 75% detection rate is that the bounds given for the resource drain test cases were based on a heuristic. The test and evaluation team used ulimit values in Linux to set the absolute bounds for resources, and the resource drain detection technology simply used a percentage of that absolute bound as the condition to catch on. In many of the undetected test cases, this bound was not strict enough to catch the injected resource drain. For the programs with altered functionality, the opposite problem occurred. For instance, a limit on the number of file descriptors was set for the program under test. However, due to our container-based technology, many more file descriptors were required to execute the test. While the per-container limit was not reached, the host limit was, causing execution to fail.

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)

TABLE II
SUMMARY OF MEMORY CORRUPTION TEST CASE RESULTS BY CWE NUMBER.

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)

TABLE III
SUMMARY OF NUMBER HANDLING TEST CASE RESULTS BY CWE NUMBER.

Table II gives the breakdown of memory corruption test case results based on the corresponding CWE number of the injected vulnerability. We show only those CWEs where the detection was below 85%. (The other memory corruption CWEs tested - 120, 124, 129, 170, 415, 416, 590, 761, 785, 805, 806, 822, 824, and 843 - all had high rates of detection, 87%-100%.) We capture less than 40% of CWE-126, 127 and 134. CWE-126 is a buffer over read, and 127 is a buffer under-read. We expected the DYBOC component to catch these vulnerabilities. Upon closer inspection, we discovered that these are stack based buffer overflows, and as such weren't caught due to our lack of a fine-grained stack to heap transformation. Additionally, CWE-134 covers format string vulnerabilities, which at the time of initial testing were not covered by the MINESTRONE framework. We discuss the improvements after adding our new heapify and format string functionality in Section V-C.

For most number handling CWEs (194, 195, 197, and 369) tested, the IOC number handling component catches nearly 100% of vulnerabilities after excluding those with altered functionality. Table III shows the three

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)

TABLE IV
SUMMARY OF RESOURCE DRAIN TEST CASE RESULTS BY CWE NUMBER

Base program	Processed	Unaltered	Base Score	Final Score
ZSHELL	44.31% (113/255)	71.68% (81/113)	62.83% (71/113)	87.65% (71/81)
CHEROKEE	97.56% (280/287)	94.64% (265/280)	76.78% (215/280)	81.13% (215/265)

TABLE V
SUMMARY OF MEMORY CORRUPTION TEST CASE RESULTS BY BASE TEST CASE

Base program	Processed	Unaltered	Base Score	Final Score
ZSHELL	66.66% (92/138)	0% (0/92)	0% (0/92)	0% (0/0)
TCPDUMP	100% (127/127)	0% (0/127)	0% (0/127)	0% (0/0)

TABLE VI
SUMMARY OF RESOURCE DRAIN TEST CASE RESULTS BY BASE TEST CASE

Base program	Processed	Unaltered	Base Score	Final Score
GREP	100% (171/171)	0% (0/171)	0% (0/171)	0% (0/0)
LIBWWW	100% (140/140)	66.42% (93/140)	36.42% (51/140)	54.83% (51/93)

TABLE VII
SUMMARY OF NUMBER HANDLING TEST CASE RESULTS BY BASE TEST CASE

CWEs for which the IOC had a poor detection rate: 196, 682, and 839. The high amount with altered functionality should also be addressed. A large number of innocuous numeric conversions exist in the base programs used for the test and evaluation (for example, setting an unsigned int to negative 1). In order to deal with these, a whitelist was created based on the known "allowed" conversions. However, these whitelists were developed based on a small number of "good" test cases that we were provided prior to the test and evaluation. As such, there were many places where a true positive numerical conversion was flagged, but counted as altered functionality because it was considered innocuous.

For resource drain test cases, the detection rate for most CWEs (401, 459, 674, 771, 773, 774, 775, 834) tested was good (81%-100%), but in general the number of test cases with altered functionality was high. This was largely due to unexpected limits being enforced in the MINESTRONE environment. In Table IV, we can see that the resource drain technology performed abysmally on CWE-835. This CWE is based on an unterminated infinite loop. In these cases, the program simply would never terminate. This particular weakness was not covered by our resource exhaustion detectors.

Figure V shows the memory corruption test cases by base program for those with unexpected performance. zshell is notable due to the low number of processed and unaltered test cases. The cause of these problems is twofold. First, the escaping of backticks and quotation marks (which we had to pass through multiple other shell scripts) often was lost before the program was executed. This caused many of the zshell test cases to fail to be processed. The second issue was a bug in the stage 2 (where MINESTRONE was executed) phase of processing, where an incorrect hostname was given that caused the test case to fail. Also shown is Cherokee, as our heap protection failed to build with Cherokee, which brings down the final score.

Table VI shows that the resource drain component failed to properly run any zshell or tcpdump test cases. This was caused by the scoring, which compared output (stdout) for these base programs. The resource drain component was writing extraneous output, causing the output to appear altered.

Figure VII shows two base programs that gave the number handling component difficulty. grep contained an unexpected number handling error that we were unable to whitelist. This caused all of the test cases, both good and bad, to be detected as an error. As such, all grep test cases were considered to have altered functionality.

C. Technology Improvements

Sections IV-A2 and IV-A1 detailed two improvements made to the MINESTRONE component technologies for detecting fine-grained stack buffer overflow and underflows as well as a FormatGuard-inspired format string protection mechanism. Tables VIII and IX show the improved results for the relevant CWEs that covered stack-based memory corruption (126, 127) and format string vulnerabilities (134). Note that these test cases also covered many with the "memory alignment" requirement discussed in Section VI. Because our memory protection tools intrinsically will not work with programs that require sequential memory alignment, these are discarded for the final results (identified by "no memalign" in the table). Another interesting point is that due to the build system for Cherokee, our stack-to-heap transformation and format string prevention are unable to be applied. In the case of CWE-134, for instance, 9/10 of the missed test cases were Cherokee-based. Similarly, 8/13 of the missed test cases were Cherokee-based for CWE-127 and 6/8 for CWE-126. Thus, we also provide the scores excluding Cherokee.

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)

TABLE VIII
SUMMARY OF IMPROVEMENTS TO SCORES BY FINE GRAINED BUFFER PROTECTION.

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

TABLE IX
SUMMARY OF IMPROVEMENTS TO SCORES BY FINE GRAINED BUFFER PROTECTION.

VI. LESSONS LEARNED

In this section we share our experience working with a large-scale test suite and testing infrastructure.

The test suite we used for our experimentation was provided by MITRE in the IARPA STONESOUP program [19]. However, this time, leveraging the experiences and lessons from [7], this test suite was developed using a mature compiler architecture [12] to inject vulnerabilities. Efforts were also made to obfuscate the vulnerable code by using randomly generated variable names and extensive usage of pointer-to-pointer declarations (up to 15 levels of indirection). While this test suite was significantly improved in size and quality, it still suffered from some mistakes that forced the evaluation to throw out programs with altered functionality.

The following issues were encountered with the test suite, specific to particular weakness classes, providing lessons learned for future vulnerability testing efforts.

Memory Layout: 25% (417/1629) of the memory corruption test cases were discarded due to an incorrect memory layout assumption. The designers made the assumption that memory was allocated linearly: meaning two consecutive malloc calls guaranteed a contiguous range of memory. There is is no guarantee that consecutive malloc calls will return contiguous allocations. These test cases explicitly checked for contiguous memory in order to run, which always failed with our primary memory corruption detection components (which by design never return contiguous memory segments).

Numerical Errors: Numerical errors are one of the hardest problems to detect. Sometimes an integer overflow is used on purpose by developers. Distinguishing the programmer's intent with such a "bug" is hard if not impossible. The test suite was riddled with operations undefined by the C standard, which we reported as vulnerable. This led to a lot of discussion as to what a false positive was.

Ambiguity in Resource Drains: The resource drain test cases were particularly troublesome, for a few reasons. The limits used as guidelines for the test and evaluation were the ulimit values on the system. As such, the resource drain technology was forced to set an arbitrary variable percentage of the ulimit value as the new limit. This was a rather clunky mechanism, and was thrown off by our multi-container approach, which had multiple subprocesses using the same base limit in the host. A better approach would be to specify (via configuration file) the acceptable limits for the program under test, so our technology would not be interfered with by system-wide (or process-wide) limits. An even better approach would be to allow a training phase, where some good inputs are provided to figure out a baseline, which would then be used as a comparison point by the resource drain technology.

A number of issues were related to the testing infrastructure. The testing architecture is, at a high level, made of two distinct components: the test manager and the performer's architecture. The test manager handles the delivery of test cases, over the network to the performer's architecture, as well as scoring. Many of the issues encountered were related to assumptions made about the performer's architecture, and these point to lessons learned for future vulnerability testing infrastructures and large-scale experimentation.

Testing Co-Process Results: The test manager expects the performer's architecture to report the execution status after the program under test terminates. However, in some cases the program under test was backgrounded and killed after a co-process finished. In the MINESTRONE framework, the co-process is only executed once, while I/O redirection is used to replay its behavior. As such, the reported status (after the co-process finishes) is only valid for the first component of the MINESTRONE framework. This is a violation of the assumption that the program under test would be treated as a blocking process whose termination indicated a completed test case run.

Use of Co-Process Binaries: Some test cases required co-processes to be executed. For example, a TCP server would require a TCP client to send some message to it to trigger a vulnerability. The problem was that the co-processes shipped in compiled binary form instead of source code. This led to many execution errors as some of the co-processes required a different libc version than the one present in the test infrastructure. The testing infrastructure should provide all dependencies, such as co-process programs, in source code form.

Hardcoded IP Addresses: A class of network test cases used hardcoded IP addresses (or localhost) to reach co-processes (e.g., a web server). This led to several errors in our architecture where the program under test runs in a separate container (where the co-

process was not running locally). After discussions with the test and evaluation team we were able to convince them to replace hardcoded IP addresses (which limit portability of the test infrastructure) with hostnames that allowed us some flexibility and set up simple redirection with /etc/hosts. However, despite being aware of this constraint, many of the final test cases included still exhibited this error.

Scalability: After each test case execution the performer's architecture reports its status to the test manager and also sends back all the test case execution results (binaries, buildfiles, logs, etc.). However, our architecture builds the test cases as many times as we have different detection technologies, resulting in at least five builds. This design caused us to need over 10TB just for the results of the test and evaluation in our off-site testing. In the end scalability was such an issue that we had to modify the test harness to delete unimportant files before sending them to the test manager. During the on-site test and evaluation this design also caused big problems. The storage for the test managers, shared over NFS, was quickly overwhelmed when hundreds of test harnesses (and multiple test managers) were all running concurrently. Their storage was exhausted during a small-scale dry run before the actual test and evaluation, we do not know what steps were taken internally to solve the problem. A better approach would be to identify only modified files and keep a de-duplicated data store of files.

VII. Conclusion

In this paper, we have presented the second-generation prototype of the MINESTRONE architecture for software vulnerability detection and mitigation. We tested our prototype against real-world programs up to 200+KLOC with injected vulnerabilities. We identified weaknesses in our first-generation prototype and significantly improved the detection technology for fine-grained stack over/under flows as well as format string vulnerabilities. Our results showed a 100% improvement in detection of these vulnerabilities in this real-world test corpus. We also identified and outlined pitfalls and possible solutions in a large-scale test and evaluation architecture. In the next phase of the STONESOUP program, MINESTRONE will be tested on even larger programs (1000+KLOC), with a larger variety of test case inputs. In our future work we plan to extend the prototype to handle concurrency bugs, which are one of the hardest to detect.

Acknowledgment

This work was supported by the US Air Force through Contract AFRL-FA8650-10-C-7024. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government or the Air Force. The authors would like to thank Angelos Keromytis and the MINESTRONE team. Acknowledgment is also extended to the members of MITRE who worked on the test suite and test infrastructure.

References

[1] (2010) Microsoft security development lifecycle. [Online]. Available: "http://www.microsoft.com/en-us/download/details.aspx?displaylang=en&id=12285"

[2] N. R. Mead and T. Stehney, "Security quality requirements engineering (square) methodology," in *Proceedings of the 2005 Workshop on Software Engineering for Secure Systems&Mdash;Building Trustworthy Applications*, ser. SESS '05. ACM, 2005, pp. 1–7.

[3] (2013) Coverity development testing platform. [Online]. Available: http://www.coverity.com/

[4] (2013) Hp fortity. [Online]. Available: https://www.fortify.com

[5] J. Merced. (2012) Source code analysis tool evaluation. [Online]. Available: http://www.iarpa.gov/stonesoup_Merced_DHSAWGbrief.pdf;http://www.iarpa.gov/images/files/programs/stonesoup/Stonesoup_Proposer_Day_Brief.pdf

[6] A. D. Keromytis, S. J. Stolfo, J. Yang, A. Stavrou, A. Ghosh, D. Engler, M. Dacier, M. Elder, and D. Kienzle, "The minestrone architecture combining static and dynamic analysis techniques for software security," in *Proceedings of the 2011 First SysSec Workshop*, ser. SYSSEC '11, 2011, pp. 53–56.

[7] A. Benameur, N. S. Evans, and M. C. Elder, "Minestrone: Testing the soup," in *Presented as part of the 6th Workshop on Cyber Security Experimentation and Test*. USENIX, 2013.

[8] C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08, 2008, pp. 209–224.

[9] (2014) Spike. [Online]. Available: https://www.immunitysec.com/resources-freesoftware.shtml

[10] (2014) The cert basic fuzzing framework (bff). [Online]. Available: http://www.cert.org/vulnerability-analysis/tools/bff.cfm

[11] (2008) The common weakness enumeration (cwe) initiative. [Online]. Available: http://cwe.mitre.org/

[12] e. a. Quinlan, D.J. Rose compiler infrastructure. [Online]. Available: http://rosecompiler.org/

[13] (2012) Openvz linux containers. [Online]. Available: http://www.openvz.org/Main_Page

[14] S. Sidiroglou, G. Giovanidis, and A. D. Keromytis, "A dynamic mechanism for recovering from buffer overflow attacks," in *Proceedings of the 8th Information Security Conference (ISC)*, 2005, pp. 1–15.

[15] G. Portokalidis and A. D. Keromytis, "Reassure: A self-contained mechanism for healing software using rescue points," in *In: Proceedings of the 6th International Workshop on Security (IWSEC)*, 2011, pp. 16–32.

[16] S. Sidiroglou, O. Laadan, A. D. Keromytis, and J. Nieh, "Using rescue points to navigate software recovery (short paper)," in *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, 2007, pp. 273–278.

[17] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier, "Formatguard: Automatic protection from printf format string vulnerabilities," in *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, ser. SSYM'01. USENIX Association, 2001, pp. 15–15.

[18] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "Cil: Intermediate language and tools for analysis and transformation of c programs," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. CC '02, 2002, pp. 213–228.

[19] (2011) Securely taking on new executable software of uncertain provenance (stonesoup) program. [Online]. Available: http://www.iarpa.gov/Programs/sso/STONESOUP/stonesoup.html