AFRL-IF-RS-TR-2004-94
Final Technical Report
April 2004

STATIC SECURITY ANALYSIS FOR OPEN SOURCE SOFTWARE

Secure Software, Inc.

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. M137

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

AFRL-IF-RS-TR-2004-94 has been reviewed and is approved for publication

APPROVED: /s/
ELIZABETH S. KEAN
Project Engineer

FOR THE DIRECTOR: /s/
JAMES W.CUSACK, Chief
Information Systems Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** APRIL 2004	**3. REPORT TYPE AND DATES COVERED** **FINAL Sep 01 – Mar 04**

4. TITLE AND SUBTITLE STATIC SECURITY ANALYSIS FOR OPEN SOURCE SOFTWARE	**5. FUNDING NUMBERS** C - F30602-01-C-0161 PE - 62301E PR - CHAT TA - 00 WU - 02
6. AUTHOR(S) John Viega	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Secure Software, Inc. 4100 Lafayette Center Drive, Suite 100 Chantilly VA 20151	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington VA 22203-1714 AFRL/IFSA 525 Brooks Road Rome NY 13441-4505	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-2004-94

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Elizabeth S. Kean/IFSA/(315) 330-2601 Elizabeth.Kean@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT *APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 Words)*

The goal of the project, “Static Security Analysis for Open Source Software”, was to explore technologies to improve the security of software by helping to automate security analysis. The project successfully improved upon the best published analysis techniques and made several releases publicly available as open source software. The analysis techniques developed under this effort reduce both false positives and false negatives compared to previous techniques. Additionally, the tools developed are highly scalable and extensible. Some of these tools were adopted by other projects within the DARPA Composable High Assurance Trusted Software Program.

14. SUBJECT TERMS static analysis techniques, secure software, software auditing	**15. NUMBER OF PAGES** 15
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UL

Table of Contents

1. Technical Objectives 1
2. Technical Approach 1
   Catscan 2
      Figure 1 : Catscan Architecture 2
3. Results.......... 9
4. Key Accomplishments to Date Summary.......... 11
5. CDRLs Delivered 11
6. References.......... 11

1. Technical Objectives

One of the tenants of the open source movement is that “many eyeballs” will look at source code and eradicate flaws. Many people believe this principle applies to security. However, there is plenty of evidence to suggest that the “many eyeballs” phenomenon may not actually help security much. There are many examples of significant, simple security bugs in open source software that persisted for over a decade, including problems in the Washington University FTP Daemon and the MIT Kerberos implementation.

There are many reasons for this phenomenon. Few people look at source code for security, and those that do often are not expert enough to do a good job. In practice, anything that makes it more difficult to audit software for security tends to be a deterrent to analysis. The effectiveness of the “many eyeballs” phenomenon would be vastly improved if high quality security auditing were significantly easier to perform, and could be performed at low or no cost.

The goal of this project is to explore technologies to improve the security of software (particularly open source software) by helping to automate security analysis. By trying to codify as much expertise as possible into tools, we hope to make security analysis something that can be done by developers without security expertise as part of their development workflow.

We focus on static analysis techniques that may limit false positives, but also look at simpler techniques, as they are still cost effective.

2. Technical Approach

The naïve way of doing security analysis is simple pattern matching. The PI has written a tool to do this (ITS4), and a replacement tool (RATS) was largely completed under this effort. These tools do not perform real analysis, but report only the most superficial symptoms of problems. For example, consider the following C program:

```
int main(int argc, char **argc) {
  char progname[1024];

  if (strlen(argv[0]) >= 1024) {
    abort();  /* Don't allow buffer overflow */
  {
  strcpy(progname, argv[0]);
  printf("Program name is %s\n", progname);
  return 0;
}
```

This program has no security problems, because it checks to make sure that argv[0] will not overflow the buffer progname. However, RATS and ITS4 only see that “strcpy” is called with two variables as arguments. Since these tools know that strcpy is often misused, they flag the operation as a potential problem of high severity.

Nonetheless, such pattern-matching tools are useful for auditors, when no tool support is the alternative. Since building "real" static analysis tools is a difficult task (and still a research area) and extending such tools to work with new languages is a costly and time-consuming chore, there is a real practical value in such tools. Therefore, our strategy on this effort was two-fold. Primarily, we focused on building a practical static analysis tool, based on research from David Wagner, and using our own extensions. Additionally, as a low-priority thread, we extended our pattern-matching tool RATS to deal with three popular languages, Perl, Python and PHP. In the rest of this section, we will discuss the approach for our static analysis tool. Our pattern matching tool is straightforward, and the techniques used in this tool are discussed in [1].

Catscan

The static analysis tool we built under this effort is called Catscan. We started with research from David Wagner [2]. Wagner's tool was a research tool written in ML, and not expected to scale to real systems. Our primary goal was to build a practical tool based on his technique, and to research practical improvements to his technique.

Catscan operates by translating source code into a graph-based format and analyzing the graph against a set of constraints, which are rules describing security-relevant behaviors of important API calls. The high-level architecture of Catscan is shown in Figure 1.

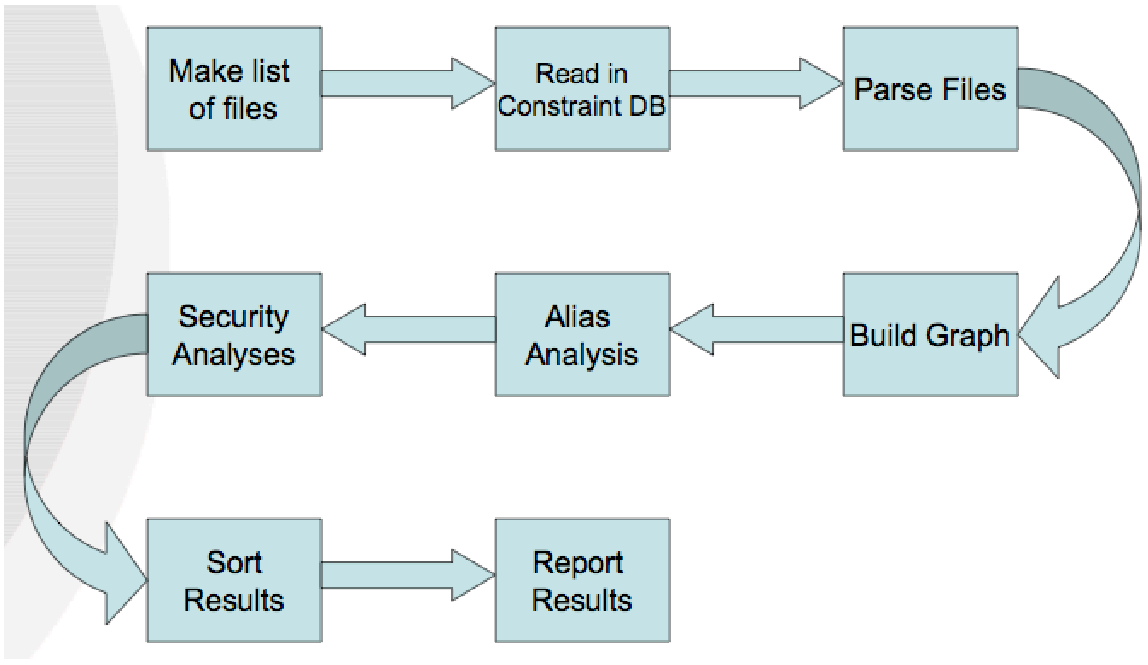


Figure 1: Catscan Architecture

The first stage is to make a list of files that constitute a program. This can either be done via a command line or through a JAVA GUI that we have built. The JAVA GUI is not currently distributed as part of the Catscan package available from our web site, but has been shipped concurrently with this report, and can be redistributed at the governments will. We found that auditors were inconvenienced by the use of a GUI, and would have rather had integrating with development environments such as Visual Studio. Also, the unpolished prototype nature of the GUI makes it of little use to those who are comfortable with the command line.

The second stage in Catscan is to read in the constraint database. Constraints are behavioral descriptions of security behaviors that are used in our static analysis. Like Wagner, we perform a buffer overflow analysis. We also introduce two new analyses, a race condition analysis for so-called "time-of-check, time of use" race conditions [3] and a format string analysis for cases where malicious input can modify the format string of a C input/output routine.

The constraints allow complex expressions and allow multiple constraints to be associated with any function call. We additionally have advanced features, such as support for variable argument functions. We implemented this module using the standard compiler tools, Flex and Bison.

Here are some example constraints:

```
strlen  := len($1) - 1 <= $ret;
calloc  := $1*$2 <= alloc($ret);
fprintf := format($2);
fopen   := use($1);
```

The first two constraints describe behavior of function calls that is relevant to the detection of buffer overflows. The first says that the value returned by strlen() is in a range from 0 to one less than the actual length of the first argument. The "one less than" qualifier is due to the NULL that delimits the end of the first argument. We calculate NULL values as part of a buffer, but strlen() does not.

The second constraint simply says that calloc() returns allocated memory, the size of which is determined from multiplying the first two arguments together. We will see how these constraints are used below.

The third constraint specifies that the second argument to fprintf() is a format string. Our analysis tries to detect when hostile input may be used to manipulate a format string.

The final constraint says that the first argument of fopen() may be involved in a specific type of race condition. If we find "checks" and "uses" on matching data, then we have found a possible race condition.

When constraints are read in, they are translated internally into a tree format that is an equivalent representation with the benefit of being easier to manipulate programmatically.

In addition to constraints, Catscan also reads in the RATS database. For problems outside the scope of the three problem areas for which we perform static analysis, we perform a RATS-like analysis.

The next phase of Catscan is parsing files. We do this using a version of GCC we modified. This modification has now been used by others, including Drexel. It basically lets GCC parse into its internal representation, then dumps that representation to disk.

Catscan then reads the GCC output from disk and converts it into a graph. The nodes in the graph are variables in the program. The nodes can be annotated with analysis data, as discussed below. The edges in the graph represent explicit data flows in the program.

Next, we perform a simplistic alias analysis. In C, an alias is some value that points to another storage location. For example, consider the following common piece of code:

```
char *p;
for (p=c; *p; p++) …
```

In this code, when c gets assigned to p, p is pointing to the same string c is. This is an alias; p is said to alias c. Now, when we operate on p, we're actually operating on c. This is an issue because, if we do not track aliases, operations on p might look secure, but aren't, because we failed to take into account the behavior of things that operated directly on the variable c.

Our alias analysis technique is a "fully insensitive" technique. We basically assume that, if p aliases c, it always aliases c. This leads to imprecision in the analysis, but does not actually do that bad of a job.

Once we have gathered alias information, we perform our analysis. The results of that analysis are sorted based on severity (a field in our database) and then presented for output in a format that is easy to manipulate (e.g., it is imported by our Java GUI).

Our analysis algorithm for format string problems is simple. We traverse the graph in a canonical order, looking to see if there is a data flow from any external input of the program to any argument denoted as a format string in our constraint library. If we see such a data flow, we report an error.

Our analysis for file-based race conditions is similar. If we see data used a operation flagged as a "check", we look to see if there are any data flows from the check operations to "use" operations on the same data. Generally, this means that, if a piece of data is flagged with both a check operation and a use operation, we report it. Also, if there is an alias such that the alias is flagged one way and the non-alias is flagged the other, we will still report.

Buffer overflow analysis uses the same graph traversal approach, but uses a more complex algorithm that models strings in a useful way, as suggested by Wagner [2]. For each integer in the program, we keep track of the range of possible values that integer might take on. We do not worry about keeping a precise set of values, we only track the worst-case bounds. Strings (arrays of characters), we model as a pair of integers, and we track ranges on both. The first integer range represents the allocated size of a string. The second integer range represents the actual size of the string as used.

The goal of the buffer overflow analysis is to determine whether the actual size of a buffer might exceed the allocated size, in which case, we report a potential buffer overflow problem. We do this by building our data flow graph, propagating information through the graph, and then comparing allocated ranges to actual ranges to see if the actual size might exceed the allocated size.

To create the graph, we add a node for each integer variable we encounter, and two nodes for each string variable we encounter (one annotated as the allocated length, one as the actual length). We initialize every range to "null", indicating we have no data. Structures cause us to add multiple nodes for each data element. We also add "dummy" nodes for returns from functions. We then add directed data-flow edges between these nodes every time we see a data flow in the program (e.g., on assignment). We also add edges to the graph based on our alias analysis.

Then, we match the actual graphs to constraint graphs. On a match, we may generate one or more edges between participating variables, as appropriate. That is, some functions may specify that they cause data flows that aren't explicit in the language. For example, the constraint for strcpy(dst, src) denotes that there is a data flow from src to dst. Then, we annotate the edge, so that we can later determine the exact effects of the data flow.

The only effect a data flow can have is to expand the possible range associated with a node. If there's a flow into a node with a bigger range than that node, then that node inherits the bigger range. We do this analysis for all nodes, and take into account the effect of cycles in the graph (anything that changes eventually grows to infinity, and we are able to catch this trend and draw the obvious conclusions quickly).

For example, consider the following program:

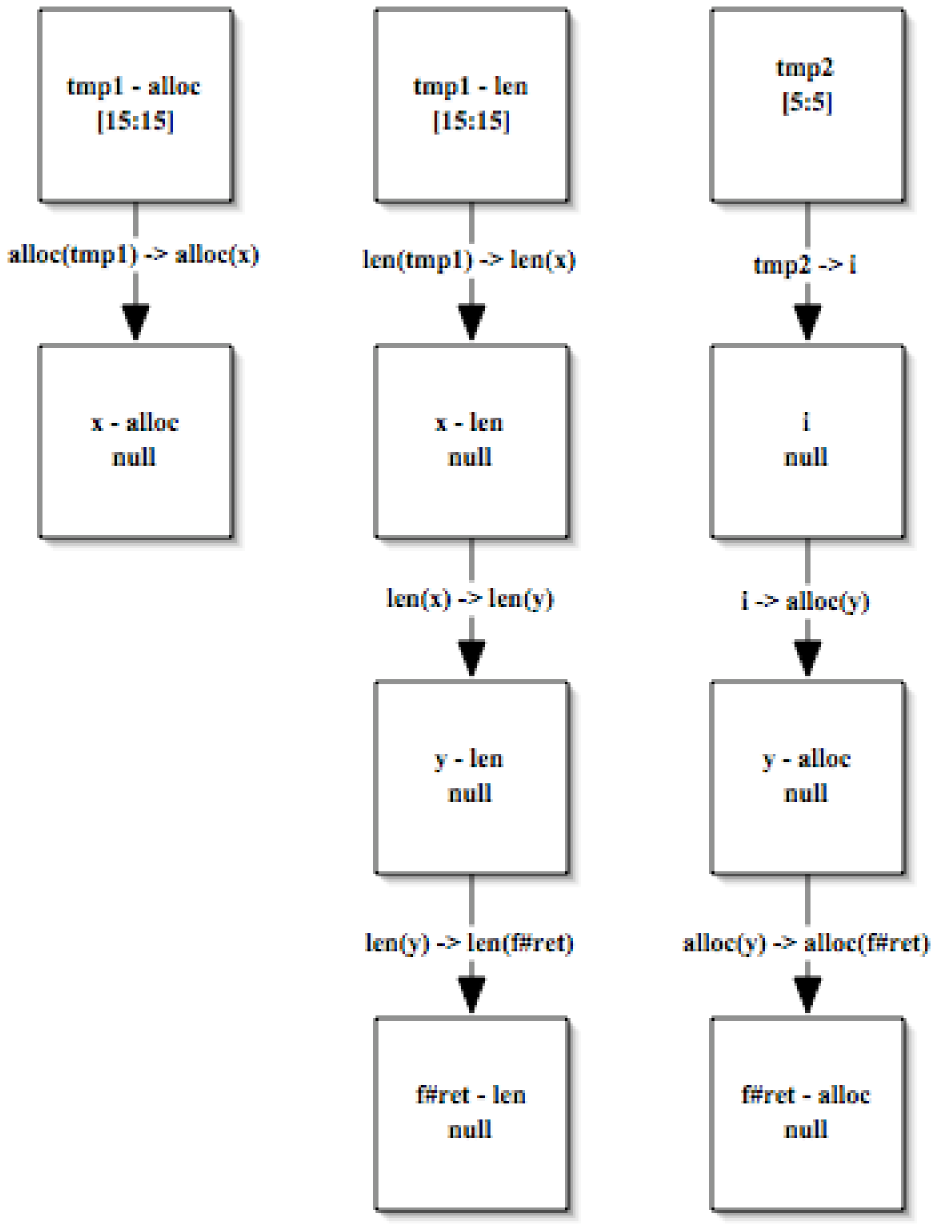
```
int main() {
  f("this is a test.", 5);
}

char *f (char *x, int i) {
  char y[i];
  strcpy(y, x);
  return y;
}
```

From this program, we extract a list of variables, x, i, y and three artificial variables, one we'll call f#ret, which indicates the return of f, and then what we'll call tmp1 and tmp2, which indicate the constants used when calling f in line 2. We count function calls as data flows, in addition to assignments.

We generate the following conceptual relationships while comparing this code to the constraint database:

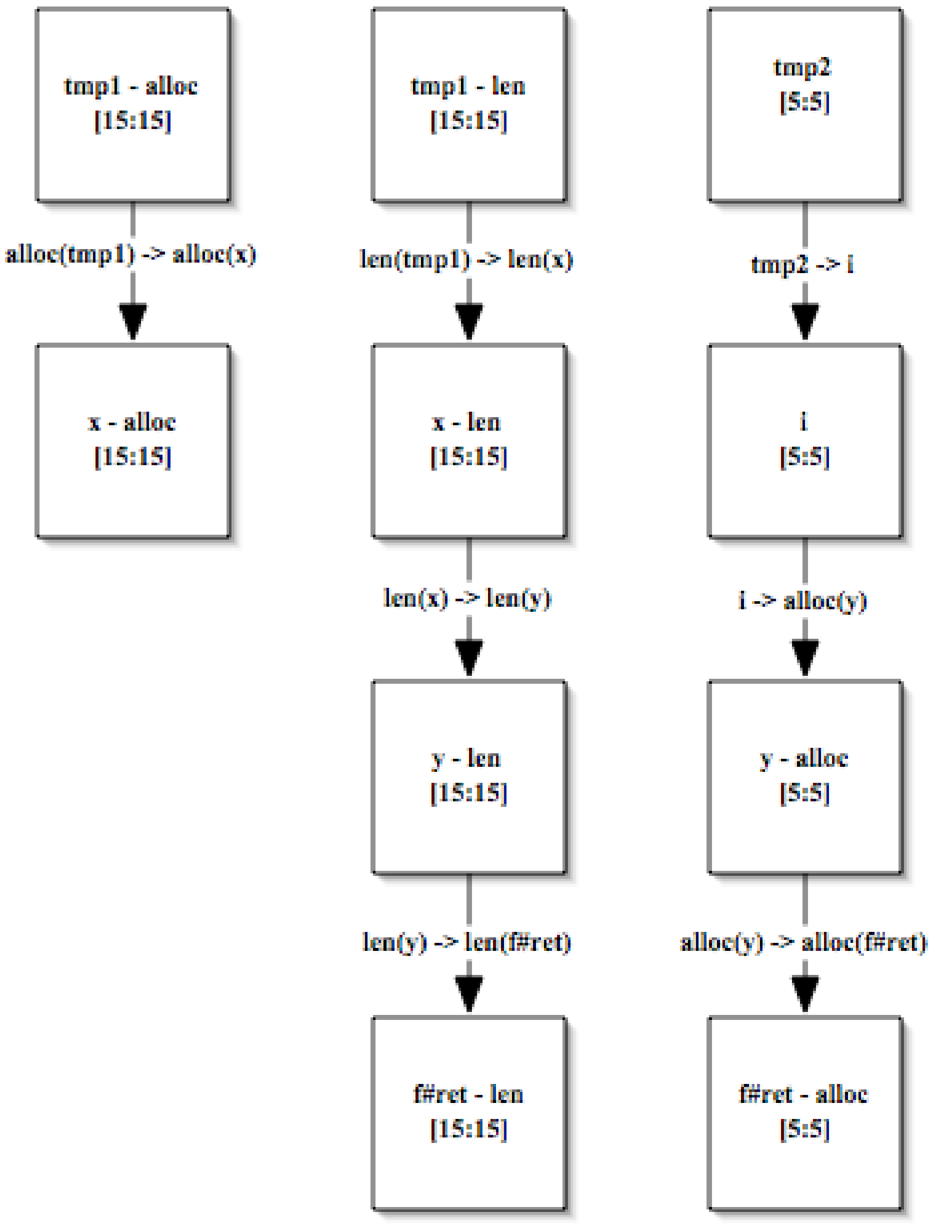
```
# These from line 2 (read -> as "gets pushed into"):
# 15 and 5 are constant, so they're used as initial values.
15 -> len(tmp1);
15 -> alloc(tmp1);
5 -> tmp2;

# These represent the data flow in the function call
len(tmp1) -> len(x);
alloc(tmp1) -> alloc(x);
tmp2 -> i;

#  The first line of the body of f.
i  -> alloc(y);

# The strcpy.
len(x) -> len(y);

# The return statement.
len(y) -> len(f#ret);
alloc(y) -> alloc(f#ret);
```

From this information, we build the following graph:



Now, we propagate range data through the graph, keeping in mind our constraints, which might cause us to do things like multiply range information (for example, see the calloc constraint above).

In this graph, the propagations are all copy operations. Therefore, the [15:15]s flow down as does the [5:5]. We then end up with the following graph:



The [15:15] for the allocated length of x indicates that, in all circumstances, the only value that x can ever take on is 15.

Once we've propagated data through our graph, we compare the allocated range to the actual range for all strings. We see that tmp1 and x never overflow (x can't if tmp1 can't, as x is an alias of tmp1), as they are always allocated to 15 bytes and are always 15 bytes in length.

We also see that y and f#ret always overflow, because they represent memory that is allocated to 5 bytes, but is actually made to store 15 bytes. In fact, these two variables are pointing to the exact same memory (f#ret is an alias of y).

Our analysis will report these two problems.

Our implementation is coded in C, and has scaled to programs of arbitrary size. We have run it on a few hundred thousand lines of code and it ran in about an hour, which we believe is reasonable.

Our tool improves upon the Wagner work in many ways. Here are the most significant improvements:

1. It performs analyses that the Wagner tool does not perform. While the Wagner tools only performs buffer overflow analysis, our tool performs two other static analyses, and also performs a RATS-like analysis.
2. It handles aliasing, which eliminates a major source of false negatives, problems that should be reported, but are not.
3. It models C structures and arrays of strings, eliminating another major source of false negatives.
4. It better models memory allocation, so as to reduce false positives significantly.
5. It has a notion of function calls and returns, making more accurate analyses possible. One technique would be to run the analysis on a function-by-function basis, then run a more sophisticated algorithm on the intra-function information, one that takes control-flow order into account.
6. It is more scalable and efficient, being written in C and built with the intention of releasing it for practical use.
7. Catscan reports information about where buffers get used, so that auditors can better track down problems.

3. Results

We completed the Catscan tool and released it to the Internet. It is currently available from http://www.securesoftware.com/download_form_catscan.htm, and is available under an open source license. We also built a Java GUI, which we include in our deliverable submission.

Additionally, we released several updates of RATS under the contract, with the blessing of our program manager. The tool was extended to handle three new programming languages, PHP, Perl and Python. This is the auditing tool most frequently used by those performing security application audits. Several consulting firms such as Guardent have extended RATS for their own personal work, extending it to handle even more languages, such as Java and Visual Basic. This tool is available from http://www.securesoftware.com/download_form_rats.htm.

We ran our Catscan tool on many open source software packages, including prominent packages such as Apache, Bash, CUPS, EMACS, lpr, nettools, OpenSSL, qmail, sendmail, wget and WU-FTPD. As a result of these runs, we did notice that, while our tool gives fewer false positives than tools like ITS4 and RATS, it still gives a large number of false positives.

We did an in-depth comparison of our performance on WU-FTPD, vs. the performance of RATS. We tested WU-FTPD 2.4 academic, looking only for potential buffer overflows. The RATS tool flagged 349 spots in the code as potential problems. Catscan flagged only 158. Of the flagged problems, RATS denoted 276 as being "high risk". Catscan only denoted 96 as high risk.

Assuming that all outputs are false positives, this is a massive reduction in false positives. In reality, not all problems were false positives. In fact, we noted 23 that are valid overflows, though few if any of them were security-critical. For example, there were many places where a malicious configuration file can overwrite an internal buffer. However, since it requires administrative privileges to write the configuration file, the attacker gains nothing by injecting a bogus configuration file, as the result would be privileges the attacker already needed to have!

The remainder of the false positives in the Catscan output are due to the insensitive nature of our analysis. We could do far better if we had efficient algorithms for processing control flow, particularly loops.

In general, the Catscan tool is far better at separating out low-risk problems from high-risk problems than is RATS. Particularly, we often know that an overflow is possible, but that the overflow is "just a few bytes", which means an exploit is far less likely.

Nonetheless, Catscan still produces a large number of false positives. And, some reports generated by the tool are confusing to the end auditor, particularly when they are wrong due to bad assumptions, or when our modeling techniques are inadequate.

On the bright side, the false positive rate seemed to be far lower for race conditions than it was for buffer overflows. In most cases where there was a bad data flow leading to us giving a warning, there seemed to be a matching control flow (which our tool didn't model). That is, for that particular problem, our approximation techniques could theoretically fall over, but model what tends to happen in practice very well.

Format string problems have an almost zero false positive rate in this analysis. As with race conditions, false positives are possible. However, when our analysis links external inputs to a format string parameter, there is a real data flow. And, when there is a real data flow from an untrusted input to one of these format string parameters, it is extremely rare for any checking to occur, and the problems are almost always exploitable when no checking is done. For that reason, in our testing, we did not find a single format string false positive.

The research performed on this contract spawned several "start-from-scratch" iterations within our company, as we continually learned more about how to build effective analyses. After a year, we started to make significant breakthroughs. Now, nearly two years later, we have made incredible progress. Our new tools have false positive rates that are below 10%, and we see how to improve on those results even further. As a result of the technology that indirectly came out of this contract, our company has received venture financing from Charles River Ventures in Boston, a top-rate venture capital firm.

4. Key Accomplishments to Date Summary

- Developed working static analysis engine.
- Made substantial improvements to Wagner's best-of-breed research techniques.
- Implemented two new static analysis techniques.
- Released the tool as open source software.
- Enhanced popular RATS tool with support for three new languages and released the results to the open source community.
- Audited numerous free software packages for a limited set of problems.
- Assisted other projects in achieving their goals. Particularly, parts of our software were adopted by Drexel, and we provided porting assistance to NYU.

5. CDRLs Delivered

- Our Catscan tool is available on the CD submitted concurrently with this report, and is additionally available for download from: http://www.securesoftware.com/download_form_catscan.htm
- Our tool RATS is available on the CD submitted concurrently with this report, and is additionally available for download from: http://www.securesoftware.com/download_form_rats.htm
- Our Java front-end is available on the CD submitted concurrently with this report.

6. References

[1] John Viega, J.T. Bloch, Tadayoshi Kohno, Gary McGraw. Token-based scanning of source code for security problems. ACM Transactions on Information and System Security 5(3), August 2002.
[2] David Wagner, Jeffrey Foster, Eric Brewer, Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In Proceedings of NDSS 2000.
[3] Matt Bishop, Michael Dilger. Checking for Race Conditions in File Accesses. Computing Systems 9(2), 1996.