ITS4: A Static Vulnerability Scanner for C and C++ Code

John Viega, J.T. Bloch, Tadayoshi Kohno, Gary McGraw
Reliable Software Technologies
21351 Ridgetop Circle, Suite 400, Dulles, VA 20166
phone: (703) 404-9293, fax: (703) 404-9295
email: {viega, jtbloch, yoshi, gem}@rstcorp.com
http://www.rstcorp.com

Abstract

We describe ITS4*, a tool for statically scanning security-critical C and C++ source code for vulnerabilities. Compared to other techniques, our results indicate that this approach stakes out a new middle ground on accuracy, while being efficient enough to give real-time feedback to a developer during coding. Our technique is also simple enough that it can easily be applied to C++, despite the complexities inherent in the language. We have used our tool to find new remotely-exploitable vulnerabilities in a widely distributed software package, as well as a major piece of e-commerce software.*

Our tool, along with its source code, is available from http://www.rstcorp.com/its4/.[1]

1 Introduction

The C and C++ programming languages are a detriment to writing secure code, because the languages and their supporting libraries make it easy for programmers to add vulnerabilities to their code inadvertently.

For example, the C standard library defines the `gets` routine, which takes as a parameter a pointer to a character, `s`. The `gets` function reads text from the standard input of a program, placing the first character at the location to which `s` points. Subsequent data are placed consecutively after `s` in memory. Bytes are read from the standard input until a newline or end of file character is reached, at which point the buffer is terminated with a null character. The programmer has no way to specify how big a buffer is being passed to `gets`. As a result, if the buffer is n bytes and an attacker tries to write $n+m$ bytes into the buffer when running the program, the attack will always be successful, as long as the data does not include newlines.

There are two significant risks in this case. First, variables adjacent in memory to the buffer can easily be overwritten. If such variables store security-critical data such as an access control list, then a wily attacker can modify the data to great advantage. The second risk is that an attacker will be able to trick the program into running arbitrary code. Such stack overflow attacks are perhaps the most common security flaw in applications today. The technical details of such attacks are discussed widely in the security community. [6, 17]

In practice, the presence of `gets` in a program almost always signals a security problem. Nonetheless, this function has remained in the standard C library since the early days of the language. Many similar problems pervade the library. Some well-known "gotchas" include `sprintf`, `strcpy` and `strcat`. Wagner[17] discusses more subtle buffer overflow problems with common C functions, including the so-called "safe" alternatives to these functions, including `strncpy` and `strncat`.

The problem of widespread security vulnerabilities easily finding their way into C and C++ programs is by no means restricted to buffer overflow conditions, even though they are the most common type of error. For example, `system` and `popen`, two library calls for running programs through the command shell, are both notoriously difficult to use correctly.

Nonetheless, these functions are commonly used in security-critical applications. Indeed, so are the well-known unsafe string operations, including `strcpy` and `sprintf`. For example, `sendmail` version 8.9.3 boasts 285 individual calls to `strcpy` alone. If these problems are so well known, why are they still encountered so often?

We believe based on personal experience that there are several factors contributing to this problem:

1. Despite the fact these problems are well known,

[1] Note to reviewers: this page is expected to go live on February 22. Source code is available earlier by request.

they are not universally known. Programmers who have heard about a problem aren't always thinking about it when they use a questionable call. Many programmers give no consideration to security at all until after all the code has been written.

2. Programmers often know that a particular call may introduce problems, but do not know what the potential problems are.

3. Programmers are often unaware of what should be corrected to avoid a known problem.

4. Programmers are likely to take the easiest approach, hoping that their use of a hazardous construct is either not exploitable, or that no one will figure out that they have a problem (the "security through obscurity" argument).

Unfortunately, there are few good sources of information about writing secure software. Such sources would help alleviate problems through education, but will not represent a complete solution because the programmer must remain security conscious.

Adding to these obvious problems, there are other categories of bugs that are far less well known, and far more subtle. For example, synchronization issues such as race conditions can often lead to security vulnerabilities. The "time-of-check-time-of-use" (TOCTOU) category of file-based race conditions identified by Bishop and Dilger[5] is a good example. Many programs that use temporary, publicly writable storage space are susceptible to being raced by a malicious process. Problems arise when a process checks information on a file (such as whether or not it already exists), then later uses the file, assuming that the recently checked information is still true. For example, a setuid text editor might open a temporary file *"/tmp/foo"* after checking to see that it does not already exist. After the check, but before the file is actually opened, a malicious attacker symbolically links a temporary file with the same name to */etc/passwd*. The attacker then types his new password file into the text editor and saves it, at leisure.

Many programmers would never think that such an attack was even a possibility. Even worse, fewer programmers would know how to avoid race conditions and other hazards. For example, one common solution among people supposedly in the know is to create a temporary file name that is meant to be hard to guess by appending a unique string that is some transformation on the output of a system random function such as `rand`. Unfortunately, such solutions are poor, since most random number generation routines generate reproducible output based on a seed value. Choosing a secure seed is itself a difficult software security problem.

We believe that in an ideal world, the programmer should need to know nothing about security; the abstractions and tools used in programming should be so good that there is miniscule chance of the programmer ever writing code that contains a security bug. Of course, this goal is unrealistic. Determining whether "untrusted" data is able to affect "trusted" data in a general purpose manner is quite a complex problem, and currently requires the programmer to annotate variables with what is essentially a security policy [14]. There is currently no realistic hope that this task could ever be completely automated.

The C and C++ languages are unlikely to become inherently more secure anytime soon. To make up for this shortcoming, we believe that programming environments should attempt to ease the burden of writing secure software for the end programmer. For example, both editors and program compilers can be made to examine code for potential security violations.

Such a paradigm works well for more mundane errors. Editors catch some errors, especially those of a syntactic nature. Compilers are more powerful, detecting syntactic errors, and more complex problems.

Why have the editor catch errors when compilers are available? There are benefits. The main benefit is that the user receives more immediate feedback from an editor than a compiler. Plus, editors must be interactive, real-time applications, whereas compilers are generally slow. Every bug the editing environment catches can potentially spare the programmer an additional compile when building and testing a program.

We see similar parallels in the area of static software vulnerability detection. On one end of the spectrum, "quick-and-dirty" approaches should be available to the programmer as early in the development cycle as possible (preferably as the programmer types), even if they forego a significant amount of precision. Our work falls in this space. On the other end of the spectrum, compilers (or similar tools) should be capable of performing a much higher-assurance static security analysis at build time, even if such an analysis is time consuming.

2 The problem with `grep`

`ITS4` was developed to address the need for a practical, widely applicable tool to help people identify potentially unsafe constructs in C and C++ code. While we certainly would find such a tool useful in the course of developing our own security-critical software, the primary motivation was to save ourselves time when performing security audits of C and C++ source.

Before ITS4, we would use `grep` at the command line as one part of a source code audit (as we believe many people do). The primary goal was to identify locations at which a program might fall prey to the same old bag of tricks. We almost exclusively looked for call sites to standard library functions with known issues. While this technique was indeed useful for finding actual vulnerabilities, we found it to be lacking in several respects:

1. **Too much expert knowledge is required.** There are dozens, or even hundreds, of vulnerable system calls; many rarely appear in the wild. We found it very hard to remember everything for which we should search, and found ourselves too lazy to spend a lot of time looking up such information in our rather poor notes and our scattered references.

   By contrast, we believe that a good tool lowers the requirement for possessing expert knowledge by keeping a database of vulnerabilities. This database would include a description possible problems, hints on how to tell if there really is a problem, and suggested fixes.

2. **Using `grep` is too inflexible.** It would be useful for the code auditor to be able to sort data intelligently. For example, an auditor may wish to look at vulnerabilities in order on a per-file basis, instead of looking at all `strcpy`s followed by all `sprintf`s, etc. Also, an auditor might want to look at all buffer overflow problems at once, followed by all TOCTOU problems. Unfortunately, `grep` alone cannot readily provide this sort of functionality; a special-purpose tool is necessary.

   More importantly, it would be useful to perform other forms of analysis in addition to the `grep`, to help refine the results. For example, a heuristic for detecting race conditions[5] may help keep the auditor from having to check dozens of calls. Grep does not provide a good framework for such analysis, since it affords no data structures representing the program (e.g., there are no parse trees or token streams).

3. **There tend to be too many false positives.** Since `grep` is only performing simple string matching, its false positive rate can be quite frustrating. We've found that when a user has to sift through high proportions of false positives, it is common for a user not to examine individual instances closely or at all.[2] We call this the "get done, go home" phenomenon. We postulate that this phenomenon contributes to the fact that there are several reported cases of significant vulnerabilities escaping notice during security audits[17].

2.1 False positives

It is not unheard of to see risky function names mentioned in comments or string literals.[3] Similarly, it is not uncommon practice for someone to use a macro definition such as:

```
#define safe_strcpy(dst, src) \
   (sizeof(dst) < strlen(src) ?\
      strcpy(dst, src) : abort())
```

Every instance of `safe_strcpy` encountered by `grep` will be flagged as a potential problem. To get rid of all those instances, `grep` must be rerun, with a more complex command line.

From our experience, these problems combined happen surprisingly often in real applications. See Section 6.1 for some results in this area.

Another type of false positive is a call that can almost trivially be ruled out from context. For example, running `grep` on `ssh` version 1.2.26 yields 134 calls to `sprintf`. 80 of these calls can be quickly classified as "very unlikely to be a problem" once the context is examined. Consider the following examples from that distribution:

```
sprintf(hex + 2 * 1, "%02x", byte);
sprintf(buf, "select: %.100s\r\n",
                    strerror(errno));
```

In the first case, it might be possible to scribble into memory, but the odds are incredibly low that anything interesting would be possible even if we could, since we only get to write a single byte. Generally, we would ignore any such problems in our audit, because experience tells us that it is not a good use of our time to examine such cases.

In the second case, the programmer has explicitly specified that no more than 100 bytes of `strerror(errno)` are to be copied into `buf`. Generally, that indicates someone is pretty sure `buf` is at

[2] In fact, if a user doesn't find a vulnerability fairly quickly, we often find people claiming that the code is secure without finishing their audit!

[3] Reviewers, we plan on determining how common this practice actually is in several big packages, including `sendmail` and `wu-ftpd`.

least that long. Of course, mistakes can happen. However, we generally classify these call sites as "low risk", and only examine them if time permits.

Similarly, the following can potentially be exploitable in theory, but seldom is in practice:

```
strcpy(dst, "\n");
```

If an experienced user can trivially prioritize all the code shown above, a good tool should be able to do the same in an effort to avoid the "get done, go home" phenomenon as much as possible.

2.2 False negatives

Note that, theoretically, false negatives are possible with `grep`; i.e., it is possible to fail to report a call site, even though it is in the source and will be parsed by the compiler. However, such (potential) false negatives do not tend to be a significant problem in practice. Consider the following cases:

1. **A function we wish to flag is used in a macro.** This case is not much of a problem, especially if we remember to grep through header files as well as the source. If we do so, we are bound to flag the macro definition.

   Additionally, finding one of our functions of interest in a macro can help reduce the amount of spam we must wade through to perform useful analysis. For example, `sendmail` 8.9.3 defines the following macro:

   ```
   #define newstr(s)\
      strcpy(xalloc(strlen(s) + 1), s)
   ```

   This macro is clearly never going to lead to any buffer overflow problems, given no bugs in `xalloc` [4]. However, `newstr` is called from 179 sites, saving us much manual inspection. If the macro were not so clearly safe, it would be pretty easy for us to also `grep` for `newstr`.

   `ITS4` behaves the same as `grep` does in the face of macros.

2. **A function we wish to flag is applied indirectly through a function pointer.** Consider the following code:

   ```
   char *(*fp)(char *, const char*);
   fp = strcpy;
   ...
   (*fp)(dst, src);
   ```

   While `grep` will flag the assignment to a function pointer, it obviously will not catch applications through that pointer. Such call sites must be discovered by manual inspection. This situation is not even caught by the technique presented in [17], though it is certainly possible for static analysis to flag these call sites if a sufficiently powerful alias analysis is performed.

   If the pointer to `strcpy` is obtained without specifying the name of the function (e.g., by calculating the location from a known offset to another function) no warning results—a true false negative (assuming the call was actually a real vulnerability). However, such code is generally not found outside of obfuscated C contests.

   `ITS4` behaves like `grep` with respect to function pointers.

3. **A call to a function we wish to flag is separated across multiple lines.** For example:

   ```
   int main(int argc, char **argv)
   {
     char buf[100];
     if(argc > 1)
       str\
   cpy(dst,argv[1]);
     return 0;
   }
   ```

   In this case, `grep` will produce a true false negative. However, in practice, breaking a single token across multiple lines is rare (except perhaps in compiler test suites and more obfuscated C contents).

   `ITS4` does not share this problem because of our parsing strategy.

3 Why not more precise analysis?

3.1 Parsing strategy

`ITS4` performs only simplistic analysis on source code (as described in more detail in the next section). A large part of the reason why `ITS4` does not perform analysis of any real sophistication is because of its parsing strategy.

`ITS4` breaks a non-preprocessed file up into a series of lexical tokens, and then matches patterns in the stream of tokens. Matching code is added by hand,

[4] But it may be disastrous when the system runs out of memory.

Package	*Total counted lines*	*Percent passive*
wu-ftpd-2.4	6613	8.65%
net-tools-1.33	8493	9.73%
sshd-1.2.26	21336	15.45%
sendmail-8.9.3	37124	17.95%
apache-1.3.9	60543	27.54%

Table 1: Code not compiled into an average configuration

so non-regular patterns can be recognized. When performing more sophisticated static analysis, it is generally easier to use a fairly complete, easy to navigate representation of a program, such as a parse tree generated from a context-free parser.

3.1.1 False negatives

One reason we chose not to use a "real parser" was that we wanted to have a false negative rate of as close to 0 as possible. Analysis tools using traditional parsing (such as the `lint` family of tools) can only analyze a single build of a program at once, since there is currently no known technique for parsing C and C++ programs with preprocessor directives into a single abstract syntax tree.

As developers ourselves, we want to check every possible build of our program, not just the build we use to develop. As people who audit the code of others, we also want to examine the entire program easily without having to specify multiple build configurations and keep track of uncovered code.

Under the assumption that people aren't often going to analyze more than a single build, we examined several large pieces of open-source software to see how much source such an analysis will miss. We wrote a simplistic preprocessor that counts how many lines of original source (not counting system headers) will be included into an executable (we call these active lines), and how many will not be (we call these passive lines). This tool is not sophisticated enough to handle complex conditional expressions, so in those cases, we evaluate them by hand, and substitute a constant expression. We ran this tool on several large open-source projects, using default configurations for a Pentium-90 running Redhat 5.0. The tool counts lines of source and blank lines, but omits comments. We did not count lines in packaged third-party software. All preprocessor directives are ignored in our statistics. The results are shown in Table 1.

Even 8.65% of a program is quite a large portion not to consider during analysis. In the testing world, 91.35% statement coverage is not considered adequate.

Although we elide per-module data for the sake of brevity, we should note that the percentage of passive lines in individual modules can vary greatly. This means that static analysis tools can fail to analyze mission-critical modules accurately.

For example, the `net-tools` package includes code to support IPv6. However, if HAVE_AFINET6 is not defined, then none of the functionality in the IPv6 portions of `net-tools` will be examined by a static analysis tool.

Of course, multiple builds can be made. But the analyst has to figure out which builds to make, compile each, and run the entire analysis algorithm repeatedly. We currently do not know how many builds of each of the applications above would require analysis. Future work will address this question.

3.1.2 Practicality

Another reason why we chose not to use "real" parsing was the desire to immediately produce a practical, widely applicable tool that developers can use. We wanted something "quick and dirty" that avoided all the difficulties that we would encounter in "real" parsing. The most significant hurdle was the large, complex nature of C++'s syntax. Another factor was the amount of time required to design the data structures used by analysis techniques.

3.1.3 Interactivity

A third reason for not using "real" context-free parsing is that we wanted to be able to support interactive programming environments such as Emacs and Microsoft Visual C++ in real time. We would like to see potential security errors highlighted in red, like bad spelling in Microsoft Office applications. In other words, as the programmer enters code, the programming environment should recognize the likelihood of any particular piece of code being a security problem, and act appropriately.

Unfortunately, traditional parsing techniques are not suitable for meeting this goal, since they only work reliably on a semantically valid program. Highly accurate error handling in traditional parsers is notoriously difficult[1]. Also, traditional parsing considers an entire file as a unit, and thus may end up being inefficient in practice if an individual file was parsed after every few keystrokes.

However, heuristics based on regular languages are known to work fairly well in similar situations, even if they are not fully precise. For example, Emacs uses regular-expression based matching on code in order to perform syntax highlighting. Though its inferences about the syntax of an individual token are occasionally wrong, Emacs is right far more often than not. Similarly, the Microsoft Office incremental spelling and grammar checker can fail to parse an English sentence properly. Despite shortcomings, these tools are widely used and highly useful.

3.2 Current limitations of advanced static analysis for C and C++

We believe that static analysis of a quality beyond that available in ITS4 can have a tremendous impact on software security in C and C++. However, we identify several problems, some of which make a practical tool involving such technology difficult for the time being.

1. **C's liberal nature makes the language poorly suited to static analysis.** The general laxness of the C language (e.g., arbitrary pointer arithmetic and gotos) makes many types of static analysis intractable in the worst case[11]. In the average case, C's heavy reliance upon pointers makes any sophisticated analysis very difficult.

2. **The added complexities of C++ make it very difficult to analyze.** Though resent research on static analysis has made some headway into performing useful analyses on object-oriented languages in general, C++ suffers because it is both object-oriented and derived from C. Currently, object-oriented analysis techniques are still cutting-edge research; performing an accurate analysis in an environment with classes, dynamic dispatch and templates is a large challenge.

3. **Static analysis in a multi-threaded environment is difficult.** In a production environment, multi-threaded applications are quite popular on Windows platforms, and are becoming ever-more popular for Unix-based systems. Unfortunately, the potential for interaction of data between threads must be considered by any analysis tool that wishes to be correct.

4. **Better static analysis is less efficient.** ITS4, which performs a very simple analysis (described in Section 4), analyzes about 9000 lines of code per second on a Pentium-90. For sendmail 8.9.3, it took 5.916 seconds on average to scan the code in CPU time, and never more than 7.5 seconds of wall time (more detailed performance information is given in Section 4.6).

   [17] presents a static analysis technique that uses constraint solving to try to determine which buffers can potentially overflow, and by how much. That technique ignores control flow information as well as context. Their prototype tool can process sendmail in about 15 minutes on a Pentium III. It is believed that a version of the software could be made to run on the order of a few minutes if the code were better tuned for performance [16]. We anticipate that a similar analysis that handled flow and context properly would be at least an order of magnitude slower still.

These problems played a significant role in our decision to avoid complicated forms of analysis in ITS4. The conclusions we drew from our experience with static analysis is that it would take several years of solid effort to produce a robust, precise, portable and (most importantly) practical tool that does an excellent job of statically analyzing source for security vulnerabilities.

4 It's The Software, Stupid! (Security Scanner)

This section discusses version 1.0b1 of ITS4. The current version of the tool supports a command-line interface to the scanning engine, and integration with Gnu Emacs.

4.1 Initial scanning and assessment

ITS4 takes one or more C or C++ source files as input, breaking each into a stream of tokens. After scanning a file ITS4 examines the resultant token stream, comparing identifiers against a database of "suspects." The database is discussed in more detail in the next subsection.

Checking each identifier is a heuristic that is not completely accurate: security neutral identifiers may be flagged. The most obvious example is variable names. Consider the following C code:

```
#include "test.h"
int main()
{
  int strcpy;
  return 0;
}
```

Running ITS4 on this code produces the following results:

```
[viega@lima c]$ its4 test1.c
test1.c:3:(Very Risky) strcpy
This function is high risk for buffer overflows.
Use strncpy instead.
----------------
```

Obviously, we would like to avoid these false positives. However, we cannot accurately determine all identifiers that are lexically used as variables without "real" parsing. The largest problem is that the preprocessor can arbitrarily modify our identifiers. In the program above, both the `int` specifier and the variable `strcpy` could be replaced with arbitrary code.

We could make a "closed-world" assumption that our scanner gets to examine all code that will be used to build the application. However, to handle the general case correctly, we would have to implement a full preprocessor, as the programmer might do arbitrarily complex things. The problem is made worse in that the preprocessor can have arbitrarily complex expressions in conditionals, and the resulting value of each conditional can change from build to build by passing in flags at the compile line.

Fortunately, programmers don't generally pervert the preprocessor in this way— a simpler analysis usually suffices for practical applications. Of course, programmers don't generally use `strcpy` as a variable name. On one hand, we could add further complexity to our code, and in the theoretical worst case have false negatives. On the other, we could err on the side of conservatism, potentially adding false positives. We chose the conservative approach.

While our approach does seem to do what the programmer expects with regard to flagging function calls almost all the time, we have run into one case where it did not do so. In particular, when scanning `sendmail-8.9.3`, we found several uses of a variable named `stat`, which happens to clash with the `stat` call, often involved in race condition problems.

Scanning for all identifier tokens had unexpected benefits. It had been suggested that we could restrict our checks to those identifiers that are followed by a left parenthesis. We did not do so due to the potential for preprocessor abuse, and because our tool flags assignments of dangerous functions to variables (see Section 2.2).

4.2 The vulnerability database

`ITS4` reads a vulnerability database from a text file at startup, keeping the entire contents resident in memory for the lifetime of the tool. Vulnerabilities can be added to the database, removed and changed with ease.

The `ITS4` vulnerability database currently contains 131 calls culled from many sources[4, 5, 8] including the Bugtraq archives[12] and our own personal experience. The largest single class of problems in our database are race conditions involving file accesses. Functions susceptible to buffer overflows also account for many entries. Several different pseudorandom number routines are flagged because they are often used (incorrectly) to provide entropy in security-critical applications. For example, developers may use these functions to shuffle cards or generate cryptographic keys in situations where security is important[3, 9].

For each call, we store the following information:

- A brief description of the problem.
- A high-level description of how to code around the problem.
- A relative assessment of the severity of the problem, on the following scale: `NO_RISK`, `LOW_RISK`, `MODERATE_RISK`, `RISKY`, `VERY_RISKY`, `MOST_RISKY`.
- An indication of what type of analysis to perform whenever the function is found in the token stream.
- Whether or not the function can retrieve input from an external source such as a file or socket. `ITS4` has a mode that finds all points at which input can come in to the program, because we often found ourselves wanting that sort of functionality in our manual audits.

Unfortunately, the database currently has several limitations, mainly stemming from the fact that it was put together based on the limited knowledge of the authors.

1. Measures of severity should be refined based on feedback from the security community. We do not feel we were the best people to judge these values in most cases.
2. The descriptions and recommendations we provide are thin in substance.
3. Several fields would be desirable but are currently not present, such as a detailed description of the problem and a detailed code example for how to mitigate the problem.
4. The database is currently Unix specific, reflecting our lack of knowledge of Windows vulnerabilities.

We hope each of these issues can be addressed in the near future with the help of the community.

The location of the vulnerability database can be specified at the command line. As a result, it is very easy to use databases that have been modified, such as a pared down database that contains only buffer overflow information. The programmer can also specify functions for which ITS4 should check at the command line, even if they are not in the database. Also, the programmer can selectively check for particular functions, or ignore functions through command line options.

4.3 ITS4 commands

ITS4 can ignore individual occurrences of a particular function. While such a feature can be detrimental (as misuse can cause the tool to ignore actual vulnerabilities), it is useful for pruning the output as individual vulnerabilities are manually audited and eliminated.

For example, a developer may add a `strcpy` to a work-in-progress. After running ITS4, she learns about the potential problem, and fixes it by adding an explicit bounds check before the call. ITS4 cannot currently perform a sophisticated enough analysis to determine that such a check is present. As a result, it will always flag this instance of `strcpy`. It would be unfortunate for there to be no way to suppress this error.

ITS4 commands are meant to ameliorate this problem, and offers two ways to do so. First, the developer can insert in-place comments with embedded commands to the scanner. For example,

```
strcpy(buf, dst); // ITS4: ignore
```

Will be ignored. The comment usually occurs on the same line as the code it effects. However, if there is no code on the same line, it affects the subsequent line.

The case-insensitive text "ITS4:" must appear in the comment, followed by an optional list of function calls. The list may optionally be comma separated. Nothing else may appear in the comment.[5] If no calls are specified, ITS4 will ignore any call on the affected line.

When modifying the source code is not an option, the user can keep a list of ITS4 commands in a file, along with the file name and line number to which the command applies. The user specifies the location of this file on the command line.

[5]Well, okay, whitespace may also appear.

To allow auditing of code that already has embedded ITS4 commands, the tool provides a command line option to ignore all commands.

ITS4 provides other ways to reduce the amount of output, or, at the very least, to present it in a more useful way. For example, there are several different sorting methods available, and vulnerabilities can be filtered based on severity.

4.4 Analysis techniques

When ITS4 first flags a function name, it looks up a "handler" for the function in the vulnerability database. The handler is responsible for reporting the problem flagged by the scanner. If no handler is found in the database, the default handler is used, which merely adds the problem to the results database. However, handlers can be used to perform more sophisticated analysis on a program.

ITS4 performs several tricks in an attempt to reduce the number of false positives produced by the tool. However, the notion of "false positive" is slightly fuzzy in this discussion, because our tool will never throw away information about a vulnerability. In practice, we expect that users will often consider only a percentage of the output, and then only the output ranked as most severe. Consider the following C code:

```
strcpy(buf, "\n");
```

ITS4 will reduce the severity of the above use of strcpy from VERY_RISKY to the lowest available. Since the scanner only outputs vulnerabilities of MODERATE_RISK or above by default, the end user will never see the warning generated by the tool unless she specifically asks to see all warnings.

In our experience with the tool, we've found that even the most patient programmers will give up fairly quickly when the severity of all problems is RISKY or below. We believe the RISKY designation is approximately where the false positive rate starts to approach 100% rapidly.[6] Therefore, even in our own security audits, we may only look at such items if time permits, depending on the situation. This problem is discussed further in Section 6.

Currently, there are two types of analysis that ITS4 can perform to refine the initial assessment it produces. The first is checking parameters of string constants in argument parameters in unsafe string operations. The second is performing a heuristic check for

[6]Unfortunately, measuring accuracy rates is very difficult to do, because we would have to examine a large number of programs to get significant numbers, and because the manual work involved to obtain such numbers would be enormous.

race conditions, using a modification of an algorithm presented in [5].

Both analyses can be turned off at the command line.

4.4.1 Sanity checking arguments

As mentioned in Section 2, `grep` unfortunately reports many hazards that are "obviously" unlikely to be problems in practice. When performing code inspections with `grep` we would often note in frustration the things that could easily be ignored with some code that wrapped the command. The most common examples we saw were `strcpy`s that only copied a fixed string into a buffer and `sprintf`s with no string specifier (i.e., %s) in the format string. ITS4 is able to identify these obvious cases through its handler mechanism.

One handler that comes with ITS4 is the "strcpy" handler. This handler is currently used not only by `strcpy`, but also by `strcat` and `strncpy`. In each of these functions, the first argument is the target buffer, and the second is the source string. If the source string is a constant, then we should reduce the severity of this vulnerability. For example, the following call should not be flagged as severe, because the second argument is a fixed string:

```
strcpy(dst, "\n");
```

Our handler has a pointer to the current token, which is the left parenthesis immediately after the `strcpy`. If the handler finds anything other than a parenthesis, it gives up. Next, it tries to find the second argument, by scanning forward in the token stream, looking for commas at the right nesting level. If the first argument consists of nested function calls, the algorithm will work properly. For example, ITS4 has no problem with the following:

```
strcat(a(b("h(i", e(x,y,z)), "the end.");
```

If a second argument is not found or is not a string, ITS4 gives up. Otherwise, it matches the pattern, and awards the problem the lowest possible severity level.

Similar checking is performed for the `sprintf` family of functions. First, the format string is found. Then, the format string is scanned for a percent sign, followed immediately by an 's'. If no such pattern is found, ITS4 assumes that either the format string only contains formatting for numbers, or that all strings have a precision specification. Either way, the chances of exploit are greatly reduced. More checking could easily distinguish between the two possibilities.

In both of these cases, we are recognizing patterns that are not regular, due to the parenthesis matching that must be performed; `grep`-style tools cannot recognize a pattern that allows arbitrary nesting.[7] Since the programmer writing a handler can make use of the full power of the C++ language, ITS4 is certainly capable in the general case of performing an analysis that is not undecidable.[8]

These two checks were added as a proof of concept. Several other checks that would be possible to add (and at least somewhat effective) are discussed in Section 4.7.

A comparison of our technique vs. `grep` and a more sophisticated static analysis tool is presented in Section 6.

4.4.2 Race condition analysis

Our analysis also addresses race conditions in file accesses, so-called "Time-Of-Check, Time-Of-Use" (TOCTOU) problems. Bishop and Dilger discuss this type of problem extensively;[5] we introduced this problem in Section 1.

We scan for these problems in a simple way. First, TOCTOU functions are classified based on their handler into functions that can be checks and functions that can be uses (several can be both). Every time we see a function, we look at the identifier that holds the file name. We store a mapping of variables to the list of TOCTOU functions that use that variable.

```
FILE *f;

int main(){
  char *fname = argv[1];
  if(!access(fname, W_OK)){
    f = fopen(fname, "w+");
  }
  else{
    // Do error handling.
  }
  // Write stdin to f then exit.
}
```

In the example above, our mapping would contain a single key "fname" which would have an array of two elements as a value. The array's values would be the instance of `access` on line 5 and the instance of `fopen` on line 6. The mapping has a lifetime beyond that of the handler.

[7] Unless they have context-free extensions.

[8] Of course, how easy such analyses will be to write is another matter completely!

At this point, scanning continues. After scanning all tokens, ITS4 calls the handler module to perform any final analysis of the data before reporting the results. We iterate over our mapping. For any keys where there is at least one check on a variable and one use, we combine the notations into a single result, which is reported with an increased severity.

This strategy works well, but there are currently significant limitations that result in ITS4 failing to promote the severity of conditions that should probably be reported. The first problem is that ITS4 currently only recognizes identifiers or string constants as valid arguments for files. As a result, if we change the above code to the following:

```
FILE *f;

int main(){
  if(!access(argv[1], W_OK)){
    f = fopen(argv[1], "w+");
  }
  else{
    // Do error handling.
  }
  // Write stdin to f then exit.
}
```

ITS4 will not increase the severity of the code above. It easily could do so; we only need implement a function that can compare a set of tokens for equivalence.[9]

Another problem is that we do not handle aliasing. For example, if we changed the above code to the following:

```
FILE *f;

int main(){
  char *f1 = argv[1];
  char *f2 = f1;
  if(!access(f1, W_OK)){
    f = fopen(f2, "w+");
  }
  else{
    // Do error handling.
  }
  // Write stdin to f then exit.
}
```

ITS4 would not increase the severity. Approaches for improving the "false negative"s of this analysis are discussed in Section 4.7.

[9]Reviewers: This will likely be done before we release the tool.

Note also that there is still plenty of room for false positives. Having two variables with the same name is indistinguishable from a single variable, as far as our analysis is concerned. Also, our approach fails to take control flow into account, and so if the check happens after the use, they are both promoted in severity, when they should not be.

Currently, there is no similar, available tool that performs a better static analysis for us to compare ourselves against. However, in Section 7 we do discuss our tool in relation to the prototype discussed by Bishop and Dilger[5].

4.5 Environment integration

ITS4 is designed so that the front-end to the tool and the back-end for the tool are both easily removed. We did this because we hope to see ITS4 integrated into popular programming environments, such as Microsoft's Visual Studio.

In such an environment, code should be analyzed in the background while the user types. The current line can be scanned continually, and the entire file can be scanned frequently to see if there are any new constructs to flag. When such a construct is identified, it should be highlighted. Mousing over the problem could give a detailed description of the issues, and so on.

ITS4 commands would be a poor user interface for such an environment. For example, Microsoft Office allows the user to right-click on a misspelled word to ignore it; a much better user interface, in our opinion.

Currently, the only environment with which we have integrated ITS4 is Gnu Emacs. The user can either run the scan all at once, much like one would compile a program from within Emacs. Alternatively, we have bindings available that will scan the current file every time the user hits enter or moves off the current line. Problems are highlighted, and output from the scanner is placed in another buffer. This integration is only a prototype, however; it is still fairly inefficient. The biggest problem is that we invoke the ITS4 command every time. It would be easy to add a new front-end to the scanner that enables it to be a persistent server communicating with Emacs, which would make it far more usable.

4.6 Performance

We performed preliminary tests on the performance of ITS4. We measured performance on a Pentium-90 with 32M of RAM running Redhat 5.0. Generally, the machine is 93.7% idle, with under 2M of real memory free. We measured the sum of user and system time using the `time` command.

Package	*Lines*	*Avg. time* user+sys.	*lines/sec.*
wu-ftpd-2.4	9899	1.343	7370.81
net-tools-1.33	11724	1.304	8990.80
sshd-1.2.26	37343	4.129	9044.08
sendmail-8.9.3	56829	5.916	9605.98
apache-1.3.9	95665	10.755	8894.93

Table 2: Performance of `ITS4` on a P90

In this environment, we ran our scanner on all the tools mentioned in Section 3.1.1. The scanner was run ten times per tool. The `wc` command was used to count lines for this study, so comments and blank lines are included. Our results for each tool appear in Table 2.

Computed over all 50 runs, the mean number of lines per second that `ITS4` scans is 8789.54, with a standard deviation of 797.12 lines.

In the course of developing this software, we noticed some interesting anecdotal trends that help us interpret our results. First, adding analyses such as TOCTOU scanning did not have any noticeable impact on the run time of our tool whatsoever, suggesting that our tool is currently I/O bound, and not bound by the analysis.

4.7 Future Directions for `ITS4`

There are several practical improvements that can easily be made to `ITS4`. Among them:

- **Integrate with new programming environments.** We discuss this option in Section 4.5.
- **Downgrade buffer overflow severity if the destination is not stack allocated.** Overflows of dynamically allocated and static memory are generally more difficult to exploit than are overflows of stack allocated memory. `ITS4` can look for patterns that look like array declarations. For each such pattern, `ITS4` can actually parse the declaration to determine whether it is stack allocated. If not, the variable may be an alias for a stack allocated buffer. Therefore, the scanner would also need to check for allocation statements (and static declarations) before it could rule out stack allocation. With our general philosophy of conservatism, items would not be downgraded unless such an allocation could be found.[10]
- **Perform alias analysis.** More accurate TOCTOU scanning can be performed if we obtain pointer aliasing information with any degree of accuracy, even if it is not fully precise. One way to go about this is to scan through all tokens, looking for assignments and function calls, noting any aliases we see. Then, aliases can be considered in a flow-insensitive, context-insensitive light. Since we will ignore the lack of flow information and other contextual clues, we certainly will not be capable of a precise analysis. The results should be much better than no such analysis, however, assuming that it is uncommon for such an approach to decide something not helpful, such as "all variables can alias all variables."
- **Perform range analysis.** The biggest hurdle to `ITS4` performing the sort of static analysis presented by Wagner, [17] and briefly described in Section 3.2, is that the constraint generation step is difficult, given our approach to parsing the input. While we would have a very difficult time generating the same constraint sets as they do, a heuristic parse could potentially do a good job. Such work should be integrated with any sort of alias analysis performed.
- **Approximate flow information.** Even our proposed heuristic static analysis techniques could be improved in accuracy if we can extract a reasonable model of the program's control flow from the data stream alone.

5 Practical experience with `ITS4`

To date, we have applied `ITS4` as a tool to assist in our auditing of two large pieces of software. The first was I-Pay, a reference version of an electronic payment system used by many Dutch banks. Our tool helped us find a definitive break in one of the network applications that comes with this package. The second was Jitterbug, a web-based bug tracking system, which has been extensively audited for security in the past [15]. `ITS4` helped find a small number of exploitable flaws, though they are unlikely to affect many users of the software.

We have some initial conclusions based on our experiences usign `ITS4`:

1. **`ITS4` still requires a significant level of expert knowledge.** While our tool does encode a vast amount of knowledge on vulnerabilities that the developer no longer needs to keep in his head, we've found that an expert still does a much better job than a novice at taking a potential vul-

[10] We rarely see references to heap allocated memory later be used to alias the stack, so we feel comfortable downgrading this type of situation.

nerability location and manually performing the static analysis necessary to determine whether an exploit is possible. We find experts tend to be far more efficient and far more accurate at this process.

2. **Even for experts, analysis is still time-consuming.** While we have not used the tool enough to give more than anecdotal evidence, we would say that the tool only eliminates from one quarter to one third of the time it takes to perform such an analysis, because the manual analysis is so time consuming.

3. **Every little bit helps.** We feel based on our limited experience with the tool that ITS4 helps significantly with fighting the "get done, go home" effect. We noticed that in the case where ITS4 prioritizes one instance of a function call over another, we tend to be more careful about analysis of the more severe problem.

4. **It can help find real bugs.** Using ITS4, we have found security problems in two real applications. In both cases, we found the problems in the first 10 minutes of analysis that we would not have found as quickly otherwise.

   Note that although we ran our tool on several large applications such as `sendmail` and `apache`, we did not hand-audit those tools. We only spent enough time with them to gather data for purposes such as timing tests and comparative analyses with other tools.

5.1 I-Pay

We used ITS4 to audit the source code for I-Pay, "the Internet payment infrastructure for the combined Dutch banks"[10]. We were most interested in remote exploits, since the I-Pay software utilities typically run on organizational web servers and other protected machines.

ITS4 immediately flagged 160 potential problems in I-Pay, including 3 possible race conditions. We first examined all input points of the distribution, as provided by ITS4. We then examined approximately 20 of the problems reported by the regular scan. We did this over the course of two hours. We primarily searched buffer overflow candidates that ITS4 flagged as "very risky". We uncovered one problem that is exploitable over the network, and three that can be exploited locally.

Our first step was to use ITS4 in locating all sites where network or file data was read. ITS4 flagged a single call to `recv`. We saw that this call was made from a function called `netread`. We asked ITS4 to find `netread`, and nothing else. There were several instances found, but we followed the first, which was made from a function called `multiread`. We asked ITS4 to find uses of this function. It found us one, in a function called `saferead`, which was itself used only three times. Examining these three call sites showed that most interesting network communication took place from these points. The first of these three calls turned out to be a major vulnerability.

I-Pay includes a utility called `checkkey` which is used after installation to check the firewall settings of the host machine and confirm that the Triple-DES library included with I-Pay is correctly configured for encryption and decryption. When `checkkey` executes, it constructs a simple text message, which is encrypted and sent to a server specified in a configuration file. The `checkkey` program waits for a response from this server, decrypts the response upon reciept, and displays it along with status information. Unfortunately, the buffer which receives the response message is a stack-allocated 256 byte buffer, while the function charged with reading data from the socket will read up to 32766 bytes. This programming mistake will allow a malicious server or a machine masquerading as the server to introduce and execute arbitrary code on the client machine. About an hour of subsequent analysis was required to confirm that this spot was likely to be a vulnerability. A brief test confirmed that it was remotely exploitable.

The other potential problems identified by ITS4 were less serious. Several calls to `strcpy` and `sprintf` were flagged as risky, but were deemed harmless upon inspection. We did locate three other buffer overflow vulnerabilities using the tool, but they each require local access. As long as the I-Pay utilities run with low privileges on non-interactive machines, these potential flaws are likely have little or no impact.

The temporary file name selection algorithm employed by I-Pay appears quite poor, and susceptible to a race condition. We were alerted to this potential problem by ITS4, but we have not had enough time to look into the matter.[11]

5.2 Jitterbug

Jitterbug is free software, written in C, that tracks bug reports over the web. We were interested in auditing Jitterbug because we use it for other purposes, and we are skeptical of any C code we run, especially if it has network access. We learned after our analysis that Jitterbug has previously been extensively analyzed.

[11] Reviewers: We do expect to do so shortly.

The first five things reported by ITS4 were calls to `popen`. All of the calls passed input from the configuration file into `popen`. Assuming the configuration file is secure and the environment in which the program runs is also secure, these should not be problems. However, three of those `popen`s also take input from the web. One performs sufficient sanity checking of the arguments; we were unable to exploit it. The other two uses are exploitable; we were able to confirm this with an actual exploit. However, the vulnerabilities are only exploitable if one of two undocumented features are enabled (by default, they are not). Therefore, very few people, if any, are susceptible to this vulnerability. Apparently, the features were added for a single high-profile user who no longer uses the software, and, in light of the vulnerabilities found, they will be removed in the next version of the software[15].

ITS4 also found some buffer overflow conditions that were also exploitable in only very exceptional circumstances. For example, Jitterbug has the following macro definition:

```
#define MAX_USERNAME_LEN 30
```

If a user decides this number is low for her needs and changes it to a very high number because she doesn't want there to be an arbitrary limit, the user has unintentionally added a security vulnerability, because elsewhere in the program there will now be an exploitable `sprintf`.

There was also another fully exploitable buffer overflow found. However, its scope was also one of the aforementioned undocumented features [15].

We did not have time to examine all output from the tool.[12] In particular, there were five TOCTOU reports generated by the race condition analysis that we wish to explore. The `grep` command reported nearly 80 different function call sites where the called function can be involved in a TOCTOU condition. Without this tool, we would have manually examined each of the 80 calls in the context of the entire program. In this case, we will only have to consider flagged spots in the context of the list generated.

The reduction in time spent examining code is therefore expected to be large. However, do remember that our analysis does not handle aliasing.

One thing that we noticed when examining Jitterbug is that out of 78 functions ITS4 identified as potential spots for buffer overflows, 38 of them were protected by a call to a function called `check_overflow` on the line immediately preceding it. This function aborts if it detects an overflow. We did not notice any instances where the programmer used this call incorrectly. It would be nice if these false positives could be removed automatically by the tool, but currently we do not perform a sophisticated enough static analysis.

[12] Reviewers: We expect to finish this analysis shortly.

Package	`grep`	ITS4 *-anl.*	ITS4	*Lex red.(%)*	*Anl. red.(%)*
wu-ftpd	146	138	112	5.48	17.81
net-tools	160	142	103	11.25	24.38
sshd	265	238	206	10.19	12.08
sendmail	480	418	342	12.92	15.83
apache	623	168	113	73.03	8.83

Table 3: Effectiveness of `grep` compared to ITS4, without and with analysis.

6 Comparing ITS4 to other solutions

6.1 `grep`

In this subsection, we compare `grep` to ITS4, each using a database that only scans the 13 functions for which there are buffer overflow handlers. We limit the scope of our comparison in this way so that we can compare the performance of the handlers. Relative severities are ignored; either the tool reported a problem, or it did not. In the case of ITS4 with analysis, if the analysis downgraded a problem to the lowest possible setting, we considered that a failure to report.

Table 3 shows the number of vulnerabilities found by `grep`, ITS4 with analysis turned off, and ITS4 with analysis on. The next to the last column of this table shows the percent reduction of results reported compared to `grep` when smarter parsing is applied (i.e., lexing instead of `grep`). The last column shows the percent reduction of results reported that are due to our analysis. Note that, except in the case of `apache`, which is a vast outlier, our feeble analysis seems slightly more effective than better parsing.

Table 4 shows the overall reduction in vulnerabilities reported from `grep`. We believe that indicates that ITS4 users can expect results that are around 25% better than `grep`, perhaps more. However, this number will probably vary widely by application, and may also vary based on the programming style of the developer.

6.2 Buffer overflow detection via range analysis

The only other tool of which we are aware that we might possibly compare our work to is presented in [17]. Unfortunately, this comparison proves to be quite difficult:

Package	grep	ITS4	*Reduction (%)*
wu-ftpd	146	112	23.29
net-tools	160	103	35.63
sshd	265	206	22.26
sendmail	480	342	28.75
apache	623	113	81.86

Table 4: Total reduction compared to `grep`.

1. Their work is not limited to picking out function calls, as ours currently is. Therefore, they may flag some problems that we do not.
2. Their work fails to analyze approximately 17.95% of the program that `ITS4` does not fail to analyze.
3. Their output is based on different metrics than ours is. While theirs is based solely on the results of their analysis, ours is largely based on human experience, with only a small analysis component.

Nonetheless, we make some simplifying assumptions, in an attempt to compare how the tools would compare "in practice":

1. Since we do not know the configuration used to test `sendmail`, we make the assumption that it was the same as ours.
2. We assume that our tool will report everything their tool reports, and probably more.
3. They present results for how many "probable" results their tool gives. We assume that reporting our "very risky" and "most risky" classifications has the same semantic meaning. This means that, for the sake of our comparison, there are some functions our tool considers that it will never report, because its risk classification is too low. The assumption is that such calls are very unlikely to show up in their analysis.
4. We assume that the vulnerabilities that a particular tool will flag are uniformly distributed throughout the source code.

Their analysis of `sendmail` yielded 44 "probable" vulnerabilities. Our analysis yielded 79. Adjusting their number for the 17.95% of the code they missed based on our uniform distribution assumption, their modified number of vulnerabilities for the sake of comparison would be 53.6. With this set of simplified assumptions, their results give a 32.15% reduction in false positives. In practice, we would expect to see results from their tool that give up to a 50% reduction.

7 Related work

Regular "lint" tools such as `LCLint` [7] perform similar functions, but in the context of general robustness; security features generally are not included. Also, such tools tend work on a per-build basis, and use context-free parsing.

Security experts have long proposed building simple scanners that operate on source code, looking for simple patterns that can potentially be exploited. To date, we know of three limited prototypes of such systems (other than ours), all of which process C, and possibly C++.

The first is `slint`[13], a general-purpose security scanner developed by mudge, formerly of the l0pht. While there is a public web page for this product, no technical information is public.

The second is the Bishop and Dilger race condition scanner. In [5], they detail a fairly accurate static analysis for TOCTOU problems. Their prototype is similar in functionality and power to our race condition scanning. For example, it uses regular expressions for token recognition, instead of context-free parsing.

The primary difference between the two tools is that the Bishop and Dilger scanner considers variable names on a per-function basis, whereas `ITS4` does not. If two functions each have a variable with the same name, `ITS4` will treat all variables with the same name as the same variable, even if across separate files. We believe the `ITS4` behavior to be slightly more useful, because most programmers name parameters and local variables consistently across functions. For example, consider the following code:

```
void do_it(char *fname) {
  FILE *f = fopen(fname, "w");
}
int main(int argc, char **argv) {
  char *fname = argv[1];
  if(access(fname, W_OK))
     do_it(fname);
}
```

The Bishop and Dilger scanner will miss the above race condition, because it does not support interprocedural analysis.

Bishop and Dilger's tool has never been distributed, however, a third party reimplementation has recently become available[2].

The only other tool we know about that statically scans for security vulnerabilities is presented in [17]. We discussed this tool (primarily in Sections 3 and 6), as well as its relative advantages and disadvantages compared to the `ITS4` approach.

Other forms of static analysis are possible. For example, we discussed locating the places in the code where input to the program is possible. From there, the usual goal is to follow program flow to see what damage untrusted input can do. Static language support for such an analysis is now available for a superset of the Java programming language[14].

8 Conclusion

We have presented `ITS4`, a static analysis tool for C and C++. While its parsing model makes it poorly suited for highly accurate static analysis, the same model makes the tool very practical for real world use; even with some facility for a heuristic-driven static analysis of the program, `ITS4` can scan large programs efficiently, while still achieving adequate results. The tool is also appropriate for integration into programming environments with little modification.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[2] Antonomasia. scancode.plx. http://www.notatla.demon.co.uk/SOFTWARE.

[3] B. Arkin, F. Hill, S. Marks, M. Schmidt, T. Walls, and G. McGraw. How we learned to cheat at online poker: A study in software security. *The developer.com Journal*, September 1999. http://www.developer.com/journal.

[4] M. Bishop. Writing safe setuid programs, 1998. seclab.cs.ucdavis.edu/ bishop/secprog.html.

[5] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

[6] C. Cowan et. al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the Seventh USENIX Security Symposium*, pages 63–77, San Antonio, TX, 1998.

[7] D. Evans, J. Guttag, J. Horning, and Y. Meng Tan. Lclint: A tool for using specifications to check code. In *In proceedings of the SIGSOFT Symposium on the Foundations of Software Engineering*, December 1994.

[8] S. Garfinkel and G. Spafford. *Practical Unix and Internet Security.* O'Reilly and Associates, Inc., 1996.

[9] I. Goldberg and D. Wagner. Randomness and the netscape browser: How secure is the world wide web? *Communications of the ACM*, January 1996.

[10] InterPay. I-pay product web site. http://www.ipay.com.

[11] W. Landi and B. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of Programming Language Design and Implementation*, 1992.

[12] E. Levy. The bugtraq mailing list. http://www.securityfocus.com.

[13] mudge. The slint web page. http://www.l0pht.com/slint.html.

[14] A. Myers. Practical mostly-static information flow control. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Antonio, TX, January 1999.

[15] A. Tridgell. Personal Communication.

[16] D. Wagner. Personal Communication.

[17] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Year 2000 Network and Distributed System Security Symposium (NDSS)*, pages 3–17, San Diego, CA, 2000.