Testing C Programs for Buffer Overflow Vulnerabilities

Eric Haugh
haugh@cs.ucdavis.edu

Matt Bishop*
bishop@cs.ucdavis.edu

University of California at Davis

Abstract

Security vulnerabilities often result from buffer overflows. A testing technique that instruments programs with code that keeps track of memory buffers, and checks arguments to functions to determine if they satisfy certain conditions, warns when a buffer overflow may occur. It does so when executed with "normal" test data as opposed to test data designed to trigger buffer overflows. A tool using this method was developed and evaluated by testing three widely used, open source software packages. This evaluation shows that the tool is useful for finding buffer overflow flaws, that it has a low false positive rate, and compares well with other techniques.

1 Introduction

Buffer overflow vulnerabilities are one of the most common security flaws [6]. Over the past few years, they have accounted for up to 50% of the advisories issued by CERT, demonstrating just how serious the issue is. The infamous Internet worm of 1988 exploited a buffer overflow vulnerability in the `fingerd(8)` server program[8].

A buffer overflow flaw typically results when a programmer fails to do bounds checking when writing data into a fixed length buffer, or does the bounds checking incorrectly (for example, the check is off-by-one) [15, 20].

In the classic scenario, the buffer is located on the program stack, and the value written over is the return address for the current stack frame [1, 23]. The return address is changed to point back into the buffer, and when the function in which the overflow occurred returns, the program jumps to the bogus return address and begins executing the contents of the buffer. Since the contents of the buffer were determined by the attacker, she can then execute any code that fits into the buffer, with the same privileges as the program.

Overflowable buffers allocated on the heap or the data segment also pose a threat, though they are typically harder for an attacker to exploit. The attacker must find some value in memory she can overwrite that is security critical, such as a user id or a filename. Sometimes a function pointer stored on the heap can be changed to point to an arbitrary location, so that when the pointer is dereferenced, code of the attacker's choosing will be executed. Such attacks have occurred [4, 19].

Traditional testing methods, such as statement or branch coverage, do little to find buffer overflow vulnerabilities [25]. The problem is that the program values and execution paths needed to reveal a security flaw do not show up during regular operation; hence, they are also unlikely to show up during testing. We propose using a testing method that tracks *possible* buffer lengths as the program executes. Library functions have preconditions defining when buffer overflow will not occur. If the lengths *could* cause these conditions to be violated, buffer overflow could occur (even if it does not in this particular execution). This ameliorates the problem above.

Section 2 reviews some of the many approaches to the problem of detecting buffer overflow flaws in source code. Section 3 presents an approach to aid a tester to find buffer overflow vulnerabilities. This involves instrumenting a source program, testing the program, and then using the warnings generated by the instrumentation as an indicator of likely buffer overflow conditions. Section 4 describes a tool that implements this approach. Section 5 presents the results of using this tool on three open source software packages, and section 6 discusses these results and compares the approach with previous approaches. Section 7 concludes.

Throughout this paper we focus on the C programming language. C provides little syntactic checking of bounds, and C programs and functions tend to be very terse, with (unfortunately) a minimum of error checking. Hence there are many C programs that suffer from security problems, including buffer overflows. But the

*This work was funded by the National Aeronautics and Space Administration's Jet Propulsion Laboratory through contract 1215858 to the University of California at Davis.

method works for any similar language.

2 Previous Work

Work analyzing programs for buffer overflows falls into two classes: *static analysis* and *dynamic analysis*.

2.1 Static Analysis

A number of tools examine source code for buffer overflow. `ITS4`, typical of a large class of these tools, scans C and C++ source code for known dangerous library calls [22]. It also does a small amount of checking on the arguments to these calls and reports the severity of the threat. For example, library calls that copy a fixed-length string into a buffer are rated as less severe than library calls that copy the contents of an array into a buffer (presumably, because the programmer knows the length of the string and not the number of characters in the array). It also looks for other potential problems such as race conditions. Other similar tools include `RATS`[21] and Splint[16].

Wagner *et al.* used an integer range analysis to locate potential buffer overflows [25]. They treat C strings as an abstract data type, and assume that they are only manipulated by the C Standard Library functions, such as `strcpy(3)` and `strcat(3)`. They track allocated memory and the possible length of strings, and whenever the maximum length of a string can exceed the minimum allocated space, a buffer overflow may occur. Pointer aliasing, the flow-insensitive analysis, and the way function calls are handled mean that the string length and allocated memory amount are approximations, rather than the actual values for each possible execution. Hence this method is imprecise. Dor *et al.* improved this scheme, but required annotation[7].

The problem with static analysis methods is their imprecision. Because the general problem of detecting buffer overflow vulnerabilities by scanning source code is in general undecidable, all such tools use heuristics to determine where buffer overflow *might* occur. Dynamic tools take a different approach.

2.2 Dynamic Analysis

Dynamic analysis examines program execution to determine whether buffer overflows occur during that execution. Compilers can add code to check bounds or to arrange data structures in memory to cause hardware faults if bounds are exceeded; however, this additional instrumentation is often turned off in the name of efficiency. Various tools augment, or replace, this ability. For example, Purify [13] can detect many kinds of memory errors, including accessing a buffer past its bounds. To do so, it instrumentsg the compiled program with code that performs different kinds of memory bookkeeping.

An alternate approach is to test programs. A tool called Fuzz was used to test standard UNIX utilities by giving them input consisting of large, random streams of characters[17]. 25-33% of the programs crashed or hung. The dominant causes were problems with pointers and array dereferencing, including buffer overflow flaws. Property-based testing [9, 11] checks that programs satisfy certain properties, including security related properties. For example, the property that the program is free of buffer overflow vulnerabilities is applicable to most programs and can be specified in TASPEC. During testing, violations of the specification are detected. Software fault injection testing methods make syntactic changes to the code under test. If the change can result in a violation of the system security policy, the tester has found a portion of code that must be correct in order to avoid the presence of a vulnerability. Ghosh and O'Connor use this technique to find buffer overflows [12].

The main problem with dynamic analysis is the need for test data that causes the overflows. Unless the data given to the program causes an overflow, these dynamic techniques will not detect any possible cases where buffer overflow occurs.

3 Using Buffer Sizes in Dynamic Analysis

We extend Wagner *et al.*'s method to dynamic execution. This enables us to detect *potential* buffer overflows that do not actually occur during testing, but *might have* occurred had different test data been used[14].

Consider the library function `strcpy(dst, src)`, which copies a string from `src` to `dst`. Because this library function does not do any bounds checking, if the length of `src` is longer than the space allocated to `dst`, the data at the end of `src` that didn't fit into `dst` overwrites the memory locations that follow `dst`. This is the quintessential buffer overflow [1], and if the contents of `src` are determined by user input, an attacker may be able to exploit this to execute arbitrary code or change data values.

During normal testing, if the string stored in `src` is not long enough, the potential buffer overflow will not be detected. But, suppose the space allocated to `src` is longer than the space allocated to `dst` for some execution of the program. This can be detected at runtime, and it may indicate a vulnerability. Consider the program fragment shown in table 1. This has a buffer overflow condition, because if the user enters more than 100 characters, the library function `strcpy` will overflow `dst`. This can be detected at runtime, when `strcpy` is called, by noticing that the amount of space allocated to `src` is

```
char src[200];
char dst[100];

fgets(src, 200, stdin);
strcpy(dst, src);
```

Table 1. Unsafe call to `strcpy()`

less than the amount of space allocated to `dst`.

If the program is instrumented to keep track of how much space is allocated to each buffer, this potential overflow can be detected during testing, even if the buffer overflow did not occur during testing.

This approach does not eliminate false positives. For example:

```
char src[200];
char dst[100];

fgets(src, 200, stdin);
src[99] = '\0';
strcpy(dst, src);
```

Even though the call to `fgets` can place a string of length up to 199 in `src`, the null assignment just before the call to `strcpy(3)` means that `src` always contains a string of length 99 or less. So in this case, the call to `strcpy` is safe.

If the buffers passed to `strcpy` are dynamically allocated, it may not be possible to compute their actual allocations until execution. Consider this code:

```
char src[200];
char* dst;

fgets(src, 200, stdin);
dst = malloc(sizeof(char)*(strlen(src)+1));
strcpy(dst, src);
```

If the input is 199 characters, the space allocated to `dst` will be less than the space allocated to `src`, and a warning will be issued. However, no buffer overflow can occur because the amount of space allocated to `dst` is allocated dynamically to be the length of the string in `src`.

Other common C library functions amenable to this analysis are listed below. The property to be tested is also shown. We refer to these as "interesting functions." For our purposes, we limited our analysis to these; the interested reader is encouraged to find other functions and derive the relevant property. In this list, the notation $len(\texttt{s})$ means the length of the string stored in the buffer `s`, and $alloc(\texttt{s})$ means the amount of space allocated to `s`.

1. `strcat(s, suffix);` is $alloc(\texttt{s}) < len(\texttt{s}) + alloc(\texttt{suffix})$?
2. `strncat(dst, src, n);` is $\texttt{n} + len(dst) >= alloc(\texttt{dst})$?
3. `sprintf(dst, "%s", src);` is $alloc(\texttt{dst}) < alloc(\texttt{src})$?
4. `snprintf(dst, n, "%s", src);` is $\texttt{n} > alloc(\texttt{dst})$?
5. `fgets(s, n,...);` is $alloc(\texttt{s}) < \texttt{n}$?
6. `memcpy(dst, src, n);` is $\texttt{n} > alloc(\texttt{dst})$?
7. `memccpy(dst, src, c, n);` is $\texttt{n} > alloc(\texttt{dst})$?
8. `memmove(dst, src, n);` is $\texttt{n} > alloc(\texttt{dst})$?
9. `bcopy(src, dst, n);` is $\texttt{n} > alloc(\texttt{dst})$?

When a security analyst obtains a report of possible security problems, she may wish to prioritize the order in which she investigates problems. ITS4 prioritized the list of vulnerabilities it found, and we employ a similar technique. For example, when the destination of `strcpy` is dynamically allocated, the programmer had to calculate how much space to allocate. If the allocation is too small for one case, we believe it is likely the case for all executions of the program. Then the program will misbehave during normal testing anyway. Of course, there will be exceptions, but our experience indicates that overflows involving dynamic allocation occur less often than those involving one or more buffers of static size. A similar argument applies to `strcat(3)` and `sprintf(3)`. When the destination and the source buffers are both statically allocated, a "type 0" warning is issued. When the destination is dynamically allocated, a "type 2" warning occurs (whether the source is dynamically or statically allocated). When the source is dynamically allocated but the destination is statically allocated, a "type 1" error arises. Table 2 summarizes these warnings.

The benefit of this dynamic approach is that there is no need to make approximations to deal with pointers and casts, which make C difficult to analyze statically. At

Type	Description
0	source and destination statically allocated
1	source dynamically allocated, destination statically allocated
2	destination dynamically allocated

Table 2. Warning Types

any point in the execution of the program, the value of any variable is known. Because of the necessary imprecision with a static approach for a language like C, any warning generated by a static overflow detection tool must be investigated by inspection. With the dynamic approach outlined above, the tester must still investigate each warning by inspection, as well as generate a set of test cases to satisfy some coverage metric. But because the dynamic approach will suffer from less imprecision, the number of false positives is potentially less. If the test set must be generated for functional testing of the application, this approach is more likely to result in less work for the tester, while still providing a similar level of accuracy.

4 The Tool

The tool `STOBO` (Systematic Testing Of Buffer Overflows) implements these ideas. It takes as input the source files of a program P to be tested, and generates an instrumented version of each file, which when compiled creates P'. The input files must be preprocessed before being input to `STOBO`. When executed, P' has the same behavior as P, except information about the testing coverage achieved and the warnings that were generated are emitted to a trace file. The coverage metric used by `STOBO` is called "interesting function coverage." This is a simple metric that is satisfied when every function call to one of the interesting functions is executed. Clearly, interesting function coverage is subsumed by statement coverage. This metric was chosen for its simplicity, and because it is relatively easy for a tester to satisfy.

To keep track of the buffers that the programmer may pass to an interesting function, `STOBO` creates special function calls which appear in P'. One call is added for each variable declaration that declares a buffer, and one for each C Standard Library function that manages dynamically allocated memory. Also, each call to an interesting function is replaced with a call to a wrapper function, which then invokes the interesting function.

Consider how `STOBO` would modify the following code fragment:

```
void func()
{
    char buf1[100], buf2[100], buf3[200];
    /* do stuff */
}
```

In the `STOBO` output, this will appear as:

```
void func1() {
    char buf1[100], buf2[100], buf3[200];
__STOBO_first_stack_buf(buf1,
                        sizeof(buf1));
__STOBO_stack_buf(buf2, sizeof(buf2));
__STOBO_stack_buf(buf3, sizeof(buf3));
   /* do stuff */
}
```

The new function calls record the fact that new buffers have come into scope. Each of these calls place the starting address of a buffer onto a list, along with the size of the buffer.

The function `__STOBO_first_stack_buf()` does some additional work compared to `__STOBO_stack_buf()`: it scans the list of already recorded statically allocated buffers, and removes any entries for buffers whose lifetime has ended. An alternative for removing expired buffers would be to insert instrumentation at the point each buffer expires (e.g., at all return statements within the function). However, the use of non-local jumps, such as `longjmp(3)`, means that some buffers would not be removed from the list when they are no longer in use.

To keep track of dynamic memory, each call to one of the C Standard Library functions `malloc(3)`, `calloc(3)`, `realloc(3)`, and `free(3)` are replaced with a wrapper function. The first of these three wrappers record the starting address paired with the buffer length, which is placed on a list of dynamically allocated buffers. This list is separate from the two lists used for statically allocated buffers. The wrapper for `free(3)` removes the freed buffer from the list.

Sometimes a programmer will allocate an amount of memory using `malloc(3)` or `calloc(3)` that is constant across different executions of the program. For example:

```
ptr = malloc(50);
```

It is better to treat the memory pointed to by `ptr` as statically allocated, with respect to issuing warnings. When the amount of memory allocated by a call to `malloc(3)` or `calloc(3)` is determined by a constant expression, a different wrapper function is used that tracks that memory seperately from memory dynamically allocated with a non-constant expression.

Consider the output from `STOBO` in table 3. The `strcpy(3)` wrapper first scans each buffer list, comparing `ptr1` with each entry of each list until a match is found. Each entry consists of a buffer starting address and length, so it is easy to compute whether or not `ptr1` points to an element of the buffer represented by the entry. Since `ptr1` points to the 19th element of `buf`, the

```
char buf[30];
char *ptr1, *ptr2;

__STOBO_stack_buf(buf, sizeof(buf));

ptr2 = __STOBO_const_mem_malloc(20);
...
ptr1 = buf + 20;
__STOBO_strcpy(ptr1, ptr2);
```

Table 3. Sample STOBO output

wrapper computes the effective buffer size of `ptr1` to be 10, and remembers that `ptr1` matched an entry from the list of statically allocated buffers. Then each list is scanned again for `ptr2`, which is found on the list of dynamically allocated buffers of constant size, and has a length of 20. Then the wrapper compares 20 to 10, finds that `ptr2` points to a buffer of static length that's larger than the one of static length pointed to by `ptr1`, and generates a type 0 warning.

5 Evaluation

STOBO was used to test three versions of the popluar ftp server wu-ftpd: 2.4.2-beta-18, 2.5.0, and 2.6.2. The first two were chosen to see if STOBO could uncover their known vulnerabilities, and the third to see if STOBO could uncover new ones. ITS4 was also used to analyze 2.6.2, so that ITS4 and STOBO could be compared. The net-tools-1.46 package for Linux was tested next. This package consists of several commands related to networking, along with a support library. It was chosen so that testing with STOBO could be compared to the tool developed by Wagner[25], which found a number of buffer overflow flaws in net-tools-1.46. All testing was done using Redhat 7.2 for the i386.

5.1 wu-ftpd

2.4.2-beta-18 is known to have an exploitable buffer overflow flaw due to a misuse of `strcat(3)`[2]. This call to `strcat(3)` was flagged by STOBO with a "type 0" warning. Two known overflow flaws exist in 2.5.0[3], the first of which was another misuse of `strcat(3)`, which was again uncovered with a "type 0" warning. The second flaw was caused by a series of calls to `sprintf(3)` and `strcpy(3)`. Two of the calls to `sprintf(3)` were flagged by STOBO, one with a "type 0" warning and the other with a "type 1" warning.

A number of buffer overflow flaws in wu-ftpd-2.6.2 were uncovered, but none of them appeared to result in any serious vulnerability. Nonetheless, the ability to uncover these flaws still demonstrates the usefulness of the tool.

Function	True Positives	False Positives	Total
sprintf	8	5	13
strcat	5	5	10
strcpy	20	22	42
All	33	32	65

Table 4. STOBO results for wu-ftpd-2.6.2, all warning types

Function	True Positives	False Positives	Total
sprintf	6	1	7
strcat	1	1	2
strcpy	4	3	7
All	11	5	16

Table 5. STOBO results for wu-ftpd-2.6.2, warning type 0

Function	True Positives	False Positives	Total
sprintf	2	3	5
strcat	4	4	8
strcpy	16	10	26
All	22	17	39

Table 6. STOBO results for wu-ftpd-2.6.2, warning type 1

Function	True Positives	False Positives	Total
sprintf	0	1	1
strcpy	0	9	9
All	0	10	10

Table 7. STOBO results for wu-ftpd-2.6.2, warning type 2

Tables 4, 5, 6, and 7 summarize the number and types of warnings generated by the tool. For the purpose of this paper, a "true positive" means there exists some input to the program under test that results in the function call writing data past the end of the destination buffer (even if it turns out this flaw doesn't represent a security vulnerability, for the reason stated above). "False positive" means that for no input to the program under test, does the function call write past the end of its destination buffer.

On this program, a "type 2" warning never indicated the presence of a flaw. Overall, testing with STOBO found 33 buffer overflows, while incurring 32 false positives. Ignoring "type 2" warnings, the number of false positives is 22, or 0.67 false positives for every buffer over-

Function	True Positives	False Positives	Total
bcopy	0	3	3
fgets	0	17	17
memcpy	0	5	5
snprintf	0	36	36
sprintf	8	49	57
strcat	5	10	15
strcpy	23	59	82
All	36	179	215

Table 8. ITS4 results for wu-ftpd-2.6.2

flow discovered.

Table 8 shows the results of testing with ITS4. It was run with a command line parameter that set the sensitivity cutoff to 1. At this cutoff, all vulnerabilities in the ITS4 database are reported, except ones at the level of NO_RISK. This cutoff was chosen because it was the highest that includes all of the interesting functions checked by STOBO.

5.2 net-tools

A test set was developed for each command. The union of these test sets was considered to satisfy interesting function coverage for the support library, even though different parts of the support library were covered by different test sets.

A number of overflow flaws were uncovered. The results are shown in tables 9, 10, 11, and 12. Ignoring "type 2" warnings, STOBO found 19 buffer overflow flaws and generated 3 false alarms. This is a false positive rate of 0.158 false positives per true positive.

For each of the vulnerabilities found by Wagner's tool, STOBO emitted a "type 1" warning. The first flaw found by Wagner's tool appears in inet.c, from the support library[24]:

```
if ((np = getnetbyname(name)) !=
    (struct netent *)NULL) {
  sin->sin_addr.s_addr = htonl(np->n_net);
  strcpy(name, np->n_name);
  return 1;
}
```

The call to getnetbyname(3) returns a struct whose field n_name can have an arbitrary length. This field is copied by strcpy(3) into name, which may have a length of only 64 bytes. Since the data returned by getnetbyname(3) can come from over the network, an attacker may be able to arrange for np->n_name to have a length longer than 64.

A few other flaws that appear exploitable were also found in net-tools-1.46, which are similar to the one just

Function	True Positives	False Positives	Total
sprintf	3	0	3
strcat	6	0	6
strcpy	10	3	13
All	19	3	22

Table 9. STOBO results for net-tools-1.46, all warning types

Function	True Positives	False Positives	Total
strcat	6	0	6
strcpy	4	2	6
All	10	2	12

Table 10. STOBO results for net-tools-1.46, warning type 0

Function	True Positives	False Positives	Total
sprintf	3	0	3
strcpy	6	0	6
All	9	0	9

Table 11. STOBO results for net-tools-1.46, warning type 1

Function	True Positives	False Positives	Total
strcpy	0	1	1
All	0	1	1

Table 12. STOBO results for net-tools-1.46, warning type 2

described. Unfortunately, data on the number of false positives and known false negatives generated by Wagner's tool was not available.

6 Discussion

STOBO was able to identify known security vulnerabilities in past versions of wu-ftpd, demonstrating its ability to find known problems. With respect to the limited sample of programs presented here, STOBO did a good job of keeping false positives low while still finding flaws. For wu-ftpd-2.6.2, the false positive rate was 0.67 false positives per true positives, while for net-tools, it was 0.158. Compare this to a simple tool like ITS4, which had a false positive rate of 4.97 for wu-ftpd-2.6.2. Since false positives mean more work on the part of the analyst who must manually inspect all warnings, it is important to minimize false positives. Further, the test data used for functional testing may be used for STOBO, thus reducing the time needed to test for buffer overflows.

The known false negatives with STOBO were also low, with zero for wu-ftpd-2.6.2. (Because it is difficult to know how many flaws exist in a non-trivial program, this discussion is necessarily about *known* false negatives, not all false negatives.)

STOBO requires that the buffer overflows arise from programmer misuse of library functions. A test run on sudo did not uncover a known buffer overflow flaw. Upon inspection, the known flaw arose because a variable was not properly re-initialized before use. As a result, STOBO could not detect that the (meaningless) value in the variable bore no relation to the value needed for testing the precondition to the interesting function. Such an analysis is beyond the scope of STOBO.

Some obervations about the interesting functions are pertinent. First, in all the STOBO tests, the "type 2" warnings indicated false positives. This supports the claim made earlier, that programmers usually determine the amount of dynamically allocated destination storage correctly. Secondly, all of the flaws found by STOBO involved `strcpy(3)`, `strcat(3)`, and `sprintf(3)`. STOBO found none involving functions that take a buffer length parameter. This suggests that when programmers fill buffers using functions that take a length parameter, they tend to use the length parameter correctly.

We should point out that functions like `strncpy`, which takes a length parameter, suffers from other problems. In many places in the tested programs, the authors forgot to add a terminating NULL byte to the destination. The problem is that if the length of the source string is the same as the length parameter, `strncpy` does not add a NULL byte. The result in the destination is therefore semantically not a string, so any future operations or functions that assume it is a string produce unexpected results.

The authors of [18] recommend replacing `strncpy` with `strlcpy`, which takes a length parameter whose value should be equal to the length of the destination buffer, making it harder to commit off-by-one errors. The function also guarantees that the destination will be null-terminated. None of the functions tested with STOBO that have similar semantics were found to be involved with any buffer overflow flaws. This provides indirect, empirical evidence that `strlcpy` is harder to misuse than `strncpy`.

Testing using STOBO is a special case of property-based testing[9, 11]. In property-based testing, the tester writes a specification that associates code locations with specifications in a language called TASPEC[10]. The program being tested is then instrumented according to the specification. In TASPEC, a tester can say, "instrument all calls to `malloc()`." STOBO instruments all calls to `malloc()`, but does so differently depending on whether the parameter passed in is represented as a constant expression. Also, TASPEC has no formal mechanism for describing different kinds of warnings. It reports all violations consistently. It would be straightforward to extend TASPEC to support this, though.

Aspect-oriented programming is a way to modularize cross-cutting concerns. Programming tasks like error-checking and logging, which are usually spread throughout source code in object-oriented or procedural languages, are put into their own module. AspectC[5], an aspect-oriented extension for C, could be used to partially implement the functionality of STOBO. However, AspectC does not provide a mechanism for instrumenting variable declarations, which would be required to instrument statically declared buffers. This suggests that an aspect extension for C that includes the ability to specify variable declarations would be useful.

7 Conclusion and Future Work

STOBO provides a dynamic technique to look for buffer overflows that the test data does not exercise. It compares favorably to other tools such as ITS4 and Wagner *et al.*'s static analysis tool because STOBO takes advantage of values computed during the execution of the program.

Although obvious in retrospect, STOBO failed to report several errors that ITS4 reported. Upon examination, the ITS4 reports were generated by potential buffer overflows arising from segments of code that could not *both* be compiled (because they were guarded by `#if ... #else ... #endif` preprocessor statements). These false positives did not occur with STOBO because STOBO reported problems from executing code, and the code that ITS4 flagged was never compiled. Hence, on

systems where the code was not compiled, ITS4's reports were false positives. On systems where the code was compiled, STOBO reported the same problems that ITS4 reported.

Several extensions are possible. Extending STOBO to find vulnerabilities involving direct assignment to buffer elements would allow it to uncover overflows not related to functions. This would require STOBO to determine the possible integer range of the expression used for the index into the buffer, along the lines of what Wagner's tool does. It is hard to see how this information could be obtained dynamically with "normal" user input, since such input will probably not cause the value of the index to be outside the bounds of the buffer.

Lowering the false positive rate is another interesting question. Most false positives arose when flow of control constructs properly guarded otherwise dangerous function calls. Consider this code from wu-ftpd-2.6.2:

```
if (strlen($6) + 7 <= sizeof(buf)) {
    sprintf(buf, "index %s", (char *) $6);
```

This call to sprintf was flagged by STOBO, but because it appears in the body of the if statement, it is executed only when the buffer cannot be overflowed (due to the condition in the if statement). A more complex flow analysis might detect this, allowing the (spurious) warning to be suppressed.

References

[1] AlephOne. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[2] Cert coordination center, advisory ca-1999-03. http://www.cert.org/advisories/CA-99-03.html.

[3] Cert coordination center, advisory ca-1999-13. http://www.cert.org/advisories/CA-1999-13.html.

[4] Cert coordination center, vulnerability note vu#363715. http://www.kb.cert.org/vuls/id/363715.

[5] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *9th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Vienna University of Technology, Austria*, September 2001.

[6] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of the DARPA Information Survivability Conference and Expo*, 1999.

[7] N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in c programs via integer analysis. In *Proceedings of the Eight International Static Analysis Symposium*, June 2001.

[8] M. Eichin and J. Rochlis. With microscope and tweezers: An analysis of the internet virus of november 1988. In *Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy*, 1989.

[9] G. Fink and M. Bishop. Property based testing: A new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4), July 1997.

[10] G. Fink, M. Helmke, M. Bishop, and K. Levitt. An interface language between specifications and testing. Technical Report CSE-95-15, University of California, Davis, 1995.

[11] G. Fink, C. Ko, M. Archer, and K. Levitt. Toward a property-based testing environment with application to security critical software. In *Proceedings of the 4th Irvine Software Symposium*, pages 39–48, April 1994.

[12] A. Ghosh, T. O'Connor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 104–114, May 1998. Oakland, CA.

[13] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.

[14] E. Haugh. Testing c programs for buffer overflow vulnerabilities. Master's thesis, University of California at Davis, September 2002.

[15] O. Kirch. The poisoned nul byte, post to the *bugtraq* mailing list, October 1998. http://www.securityfocus.com/archive/1/10884.

[16] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium, Washington, D. C.*, August 2001.

[17] B. Miller, L. Fredricksen, and B. So. Empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[18] T. C. Miller and T. de Raadt. strlcpy and strlcat - consistent, safe, string copy and concatenation. In *Proceedings of the 1999 USENIX Annual Technical Conference*, June 1999. Monterey, California, USA.

[19] Nsfocus security advisory, sun solaris xsun "-co" heap overflow. http://online.securityfocus.com/archive/1/265370.

[20] Openbsd developers, single-byte buffer overflow vulnerability in ftpd, December 2000. http://www.openbsd.org/advisories/ftpd_replydirname.txt.

[21] Secure software solutions, rats, the rough auditing tool for security. http://www.securesw.com/rats/.

[22] J. Viega, J. Bloch, T. Kohno, and G. McGraw. Its4: A static vulnerability scanner for c and c++ code. In *Proceedings of the 16th Annual Computer Security Applications Conference*, December 2000.

[23] J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley, Boston, 2002.

[24] D. Wagner. Personal communication, May 2002.

[25] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Symposium on Network and Distributed Systems Security (NDSS '00)*, pages 3–17, February 2000. San Diego CA.