# A Comparison of Publicly Available Tools for Static Intrusion Prevention*

John Wilander and Mariam Kamkar

Dept. of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping
Sweden
{johwi, marka}@ida.liu.se
http://www.ida.liu.se/~johwi

**Abstract.** The size and complexity of today's software systems is growing, increasing the number of bugs and thus the possibility of security vulnerabilities. Two common attacks against such vulnerabilities are buffer overflow and format string attacks. In this paper we implement a testbed of 44 function calls in C to empirically compare five publicly available tools for static analysis aiming to stop these attacks. The results show very high rates of false positives for the tools building on lexical analysis and very low rates of true positives for the tools building on syntactical and semantical analysis. ...

**Keywords:** security intrusions, intrusion prevention, static analysis, security testing, buffer overflow, format string attack

## 1 Introduction

As our software systems are growing larger and more complex the amount of bugs increase. Many of these bugs constitute security vulnerabilities. According to statistics from CERT Coordination Center at Carnegie Mellon University the number of reported security vulnerabilities in software has increased with nearly 500% in two years [5].

Now there is good news and bad news. The good news is that there is lots of information out there on how these security vulnerabilities occur, how the attacks against them work and most importantly how they can be avoided. The bad news is that this information apparently does not lead to less vulnerabilities. The same mistakes are made over and over again which for instance is shown in the statistics for the infamous *buffer overflow* vulnerability. David Wagner et al from University of California at Berkeley show that buffer overflows alone stand

* Paper published at 7th Nordic Workshop on Secure IT Systems, 2002, Karlstad, Sweden. This work has been supported by the national computer graduate school in computer science (CUGS), commissioned by the Swedish government and the board of education.

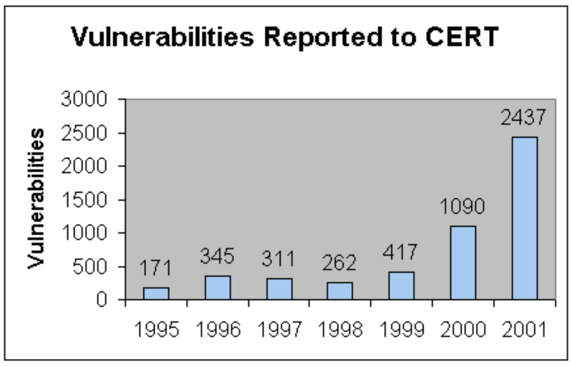

**Fig. 1.** Software vulnerabilities reported to CERT 1995–2001.

for about 50% of the vulnerabilities reported by CERT [33]. Equally dangerous is the *format string* vulnerability which was publicly unknown until 2000.

In the middle of January 2002 the discussion about responsibility for security intrusions took an interesting turn. The US National Academies released a prepublication recommending policy-makers to create laws that would hold companies accountable for security breaches resulting from vulnerable products [24] which got global media attention [3, 20]. So far, only the intruder can be charged in court. In the future software companies may be charged for not preventing intrusions. This stresses the importance of helping software engineers to produce more secure software. Automated development and testing tools aimed for security could be one of the solutions for this growing problem.

A good starting point would be tools that can be applied directly to the source code and solve or warn about security vulnerabilities. This means trying to solve the problems in the implementation and testing phase. Applying security related methodologies throughout the whole development cycle would most probably be more effective, but given the amount of existing software, the strive for modular design reusing software components, and the time it would take to educate software engineers in secure analysis and design, we argue that security tools trying to clean up vulnerable source code are necessary. A further discussion on this issue can be found in the January/February 2002 issue of IEEE Software [13].

In this paper we investigate the effectiveness of five publicly available static intrusion prevention tools—namely the security testing tools ITS4, Flawfinder, RATS, Splint and BOON. Our approach has been to first get an in-depth understanding of how buffer overflow and format string attacks work and from this knowledge build up a testbed with identified security bugs. We then make an empirical test with our testbed. This work is a follow-up of John Wilander's Master's Thesis [36].

The rest of the paper is organized as follows. Section 2 describes process memory management in UNIX and how buffer overflow and format string attacks work. Here we define our testbed of 23 vulnerable functions in C. Section 3 presents the concept of intrusion prevention and describes the techniques used in the five analyzed tools. Section 4 presents our empirical comparison of the tools' effectiveness against the previously described vulnerabilities. Related work is presented in section 5. Finally section 6 contains our conclusions.

## 2 Attacks and Vulnerabilities

The analysis of intrusions in this paper concerns a subset of all violations of security policies that would constitute a security intrusion according definitions in for example the Internet Security Glossary [27]. In our context an intrusion or a successful attack aims for *changing the flow of control*, letting the attacker execute arbitrary code. Software security bugs, or *vulnerabilities*, allowing these kind of intrusions are considered the worst possible since "arbitrary code" often means starting a new *shell*. This shell will have the same access rights to the system as the process attacked. If the process had *root access*, so will the attacker in his or her new shell, leaving the whole system open for any kind of manipulation.

### 2.1 Changing the Flow of Control

Changing the flow of control and executing arbitrary code involves two steps for an attacker:

1. Injecting *attack code* or *attack parameters* into some memory structure (e.g. a buffer) of the vulnerable process.
2. Abusing some vulnerable function writing to memory of the process to alter data that controls execution flow.

Attack code could mean assembly code for starting a shell (less than 100 bytes space will do) whereas attack parameters are used as input to code already existing in the vulnerable process, for example using the parameter `"/bin/sh"` as input to the `system()` library function would start a shell.

Our biggest concern is step two—redirecting control flow by writing to memory. That is the hard part and the possibility of changing the flow of control in this way is the most unlikely condition of the two to hold. The possibility of injecting attack code or attack parameters is higher since it does not necessarily have to violate any rules or restrictions of the program.

Changing flow of control is made by altering a *code pointer*. A code pointer is basically a value which gives the *program counter* a new memory address to start executing code at. If a code pointer can be made to point to attack code the program is vulnerable. The most popular code pointer to target is the return address on the stack. But programmer defined *function pointers*, so called *longjmp buffers*, and the old *base pointer* are equally effective targets of attack.

### 2.2 Buffer Overflow Attacks

Buffer overflow attacks are the most common security intrusion attack [33, 11] and has been extensively analyzed and described in several papers and on-line documents [22, 17, 8, 6]. Buffers, wherever they are allocated in memory, may be overflown with too much data if there is no check to ensure that the data being written into the buffer actually fits there. When too much data is written into a buffer the extra data will "spill over" into the adjacent memory structure, effectively overwriting anything that was stored there before. This can be abused to overwrite a code pointer and change the flow of control.

The most common buffer overflow attack is shown in the simplified example below. A local buffer allocated on the stack is overflown with 'A's and eventually the return address is overwritten, in this case with the address `0xbffff740`.

| | |
|---|---|
| | |
| Local buffer | `AAAAAAAA` `AAAAAAAA` |
| Old base pointer | `AAAAAAAA` |
| Return address | `0xbffff740` |
| Arguments | Arguments |

**Fig. 2.** A buffer overflow overwriting the return address.

If an attacker can supply the input to the buffer he or she can design the data to redirect the return address to his or her attack code.

### 2.3 Buffer Overflow Vulnerabilities

So how come there is no check whether the data fits into the destination buffer? The problem is that several of ANSI C's standard library functions rely on the programmer to do the checking, which they often do not. Many of these functions are powerful for handling strings and thus popular. More secure versions have in some cases been implemented but are not always know by programmers. There are lists of these dangerous C functions often involved in published buffer overflows [35, 30, 31]. From these lists we have chosen to take the fifteen functions considered most risky into our testbed:

1. `gets()`
2. `cuserid()`
3. `scanf()`
4. `fscanf()`
5. `sscanf()`
6. `vscanf()`
7. `vsscanf()`
8. `vfscanf()`
9. `sprintf()`
10. `strcat()`
11. `strcpy()`
12. `streadd()`
13. `strecpy()`
14. `vsprintf()`
15. `strtrns()`

This list is not exhaustive but should provide useful test data for our comparison of the tools.

### 2.4 Format String Attacks

22nd of June 2000 the first *format string attack* was published [29]. Comments in the exploit source code dates to the 15th of October 1999. Until then this whole category of security bugs was publicly unknown. Since then format string attacks have been acknowledged for being as dangerous as buffer overflow attacks. They are described in an extensive article by Team Teso [25] and also in a shorter article by Tim Newsham [21].

String functions in ANSI C often handle so called *format strings*. They allow for dynamic composition or formatting of strings using *conversion specifications* starting with the character `%` and ending with a conversion specifier. Each conversion specification results in fetching zero or more subsequent arguments.

Let's say a part of a program looks like this:

```
void print_function_1(char *string) {
  printf("%s", string); }
```

A call to `print_func_1()` would print the string argument passed. The same functionality could (seemingly) be achieved with somewhat simpler code:

```
void print_function_2(char *string) {
  printf(string); }
```

Using the function argument `string` directly will still print the argument passed to `print_function_2()`. But what if we call `print_function_2()` with a string containing conversion specifications, for example `print_function_2("%d %d%d%d")`? Then `printf()` will interpret the string as a format string and in this case assume that there are four integers stored on the stack and thus pop four times four bytes of stack memory and print the values stored there. So if programmers take this shortcut when using format string functions, the possibility arises for an attacker to inject conversion specifications that will be evaluated.

Now, considering the conversion specifier `%n` things get dangerous. `%n` will cause the format string function to pop four bytes of the stack and use that value as a memory pointer for storing the number of characters so far in the format string (i.e. the number of characters before `%n`.). So by injecting a format string containing `%n` an attacker can *write* data into the process' memory.

If an attacker is able to provide the format string to a an ANSI C format function in part or as a whole a format string vulnerability is present. By combining the various conversion specifications and making use of the fact that the format string itself is stored on the stack we can view and write on arbitrary memory addresses.

### 2.5 Format String Vulnerabilities

While the `scanf()`-family is involved in numerous of buffer overflow exploits [1] the format string attacks published concern the `printf()`-family of format string functions [25, 7]. For that reason our test only concerns the latter subset of the ANSI C format functions. So we add another eight function calls to our testbed (`sprintf()` and `vsprintf()` are used differently here than in the buffer overflow case):

| | |
|---|---|
| 16. `printf()` | 20. `vprintf()` |
| 17. `fprintf()` | 21. `vfprintf()` |
| 18. `sprintf()` | 22. `vsprintf()` |
| 19. `snprintf()` | 23. `vsnprintf()` |

## 3 Intrusion Prevention

There are several ways of trying to prohibit intrusions. Halme and Bauer present a taxonomy of *anti-intrusion techniques* called *AINT* [14] where they define:

**Intrusion prevention.** Precludes or severely handicaps the likelihood of a particular intrusion's success.

We divide intrusion prevention into *static intrusion prevention* and *dynamic intrusion prevention.* In this section we will first describe the differences between these two categories. Secondly, we describe five publicly available tools for static intrusion prevention, describe shortly how they work, and in the end compare their effectiveness against vulnerabilities described in section 2.2. This is not a complete survey of static intrusion prevention tools, rather a subset with the following constraints:

- Tools used in the testing phase of the software.
- Tools that require no altering of source code to detect security vulnerabilities.
- Tools that are implemented and publicly available, not system specific tools.

Our motivation for this is to evaluate and compare tools that could easily and quickly be introduced to software developers and increase software quality from a security point of view.

### 3.1 Dynamic Intrusion Prevention

The dynamic or *run-time* intrusion prevention approach is to change the run-time environment or system functionality making vulnerable programs harmless or at least less vulnerable. This means that in an ordinary environment the program would still be vulnerable (the security bugs are still there) but in the new, more secure environment those same vulnerabilities cannot be exploited in the same way—it protects *known* targets from attacks. Their general weakness lies in the fact that the protection schemes all depend on how bugs are known to be exploited today, but they do not get rid of the actual bugs. Whenever an attacker has figured out a new attack target reachable with the same security bug, these dynamic solutions often stand defenseless. On the other hand they will be effective against exploitation of any new bugs aiming for the same target.

### 3.2 Static Intrusion Prevention

Static intrusion prevention tries to prevent attacks by finding the security vulnerabilities in the source code so that the programmer can remove them. Removing all security bugs from a program is considered infeasible [16] which makes the static solution incomplete. Nevertheless, removing bugs known to be exploitable brings down the likelihood of successful attacks against all possible security targets in the software. Static intrusion prevention removes the attackers tools, the security bugs. The two main drawbacks of this approach is that someone has to keep an updated database over programming flaws to test for, and since the tools only *detect* vulnerabilities the user has to know how to fix the problem once a warning has been issued. In this paper we have chosen to focus on five publicly available tools for static intrusion prevention.

### 3.3 ITS4

In late 2000 researchers at Reliable Software Technologies, now Cigital, presented a static analysis tool for detecting security vulnerabilities in C and C++ code—*It's the Software Stupid! Security Scanner* or *ITS4* for short [30]. The tool does a lexical analysis building a token stream of the code. Then the tokens are matched with known vulnerable functions in a database. The reason for not performing a deeper analysis with the help of syntactic analysis (parsing) is that such an analysis cannot be made on the fly during programming. ITS4 is built to give developers support while coding, highlighting potential security problems as they are written. Parsing also suffers from being build dependent, not always covering the whole source code because of pre-processor conditionals.

When writing their paper the vulnerability database contained 131 potential vulnerabilities including problems with *race conditions* (not included in this paper, for reference see article by Bishop and Dilger [2]) and buffer overflows.

*Pseudo random functions* are also considered risky since they're often used wrongly in security-critical applications. An entry in the database consists of:

- A brief description of the problem
- A high-level description of how to code around the problem.
- A grading of the vulnerability on the 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.
- Whether or not the function can retrieve input from an external source such as a file or a network connection.

ITS4 has a modular design which allows for integration in various development environments by replacing its front-end or back-end. In fact that was one of the design goals for ITS4. For the moment it only supports integration with GNU Emacs.

The ITS4 security tool is available for download on the Internet.

`http://www.cigital.com/its4/`

## 3.4 Flawfinder and Rats

Two new security testing tools where released in May 2001—*Flawfinder* developed by David A. Wheeler [34] and *Rough Auditing Tool for Security* (RATS) developed by Secure Software Solutions [28]. They both scan source code on the lexical level, searching for security bugs. Their solutions are very similar to ITS4. When it was noticed that the two teams where developing similar tools they decided on a common release date and on trying to combine the two tools into one in the future.

Just as ITS4 Flawfinder works by using a built-in database of C/C++ functions with well-known problems, such as buffer overflow risks, format string problems, race conditions, and more. The tool produces a list of potential vulnerabilities sorted by risk. This risk level depends not only on the function, but on the values of the parameters of the function. For example, constant strings are considered less risky than fully variable strings. The Flawfinder 0.19 vulnerability database contains 55 C security bugs.

RATS scans not only C and C++ code but also Perl, PHP and Python source code and flags common security bugs such as buffer overflows and race conditions. Just as Flawfinder and ITS4, RATS has a database of vulnerabilities and sorts found security bugs by risk. The RATS 1.3 vulnerability database contains 102 C security bugs.

Both these security testing tools are invoked from a shell with source code as input. They traverse the code and produce output with risk grading and short descriptions of the potential problems.

The security tools Flawfinder and RATS are available for download on the Internet.

`http://www.dwheeler.com/flawfinder/`
`http://www.securesw.com/rats/`

## 3.5 Splint

The next static analysis tool we describe is *LCLint* implemented by David Evans et al [10, 23]. The name and some of its functionality originates from a popular static analysis tool for C called *Lint* released in the seventies [15]. LCLint has later been enhanced to search for security specific bugs [16] and the first of January 2002 LCLint got the name *Secure Programming Lint* or *Splint* for short.

The Splint approach is to use programmer provided semantic comments, so called *annotations*, to perform static analysis on the syntactic level, making use of the program's parse tree. This means that the tool has a much better chance of differentiating between correct and incorrect use of functions than the tools working on the lexical level.

The annotations specify function constraints in the program—what a function *requires* and *ensures*. Here is a simplified example from the annotated library `standard.h` in the Splint package:

```
char *strcpy (char *s1, char *s2)
      /*@requires maxSet(s1) >= maxRead(s2) @*/
      /*@ensures  maxRead(s1) == maxRead (s2) @*/
```

The `requires` clause specifies that buffer `s1` must be big enough to hold all characters readable from buffer `s2`. The `ensures` clause says that, upon return, the length of buffer `s1` is equal to the length of buffer `s2`. If a program contains a call to `strcpy()` with a destination buffer `s1` smaller than the source buffer `s2`, a buffer overflow vulnerability is present and Splint should report the bug.

To detect bugs the constraints in the annotations have to be resolved. Low level constraints are first *generated* at the subexpression level (i.e. they are not defined by annotations). Then statement constraints are generated by cojoining these subexpression constraints, assuming that two different subexpressions cannot change the same data. The generated constraints are then matched with the annotated constraints to determine if the latter hold. If they do not Splint issues a warning.

Note that we will not add any annotations to our test source code since that would be a violation of the second testing constraint defined in section 3. We rely fully on Splint's annotated libraries to make a fair comparison.

The Splint security tool is available for download on the Internet.
`http://www.splint.org/`

## 3.6 BOON

David Wagner et al presented a tool in 2000 describing aiming for detecting buffer overflow vulnerabilities in C code [33]. In July 2002 their tool, or rather working prototype, was publicly released under the name *BOON* which stands for *Buffer Overrun detectiON*. Under the assumption that most buffer overflows are in string buffers they model string variables (i.e. the string buffers) as two properties—the allocated size, and the number of bytes currently in use. Then all string functions are modeled in terms of their effects on these two properties of the string variable. The constraints are solved and matched to detect inconsistencies similarly to Splint.

Before analyzing the source code you have to use the C preprocessor on it to expand all macros and #include's. Then BOON parses the code and reports any detected vulnerabilities as belonging to one of three categories, namely "Almost certainly a buffer overflow", "Possibly a buffer overflow" and "Slight chance of a buffer overflow". The user needs to go check the source code by hand and see whether it is a real buffer overflow or not. Note that BOON does not detect format string vulnerabilities and is thus not tested for that.

The BOON security tool is available for download on the Internet.
`http://www.cs.berkeley.edu/~daw/boon/`

### 3.7 Other Static Solutions

There are several other approaches to static intrusion prevention. The area connects to general software testing which provides a broad range of potential methodologies.

A tool yet to be published is *Czech* by Jose Nazario [18]. Czech is a C source code checking tool that will do full out static analysis and variable tainting.

**Software Fault Injection** A technique originally used in hardware testing called *fault injection* has also been used to find errors in software [32]. This has been used for security testing. By injecting faults, the system being tested is forced into an anomalous state during execution and the effects on system security is observed and evaluated.

Anup Ghosh et al implemented a prototype tool called *Fault Injection Security Tool*, or FIST for short [12]. The tool shows promising results but preparations of the source code have to be made by hand which means that the process is not automated. Also FIST is not available for download so we have excluded it from our analysis.

Also Wenliang Du and Aditya Mathur have done research on software fault injection for security testing [9]. They inject faults from the environment of the application, i.e. anomalous user input, erroneous environment variables and so on. In their paper they describe a methodology not yet implemented. Therefore their approach is not part of our analysis.

**Constraint-Based Testing** Umesh Shankar et al from University of California at Berkeley present an interesting solution to finding format string vulnerabilities [26]. They add a new C type qualifier called *tainted* to tag data that has originated from an untrustworthy source. Then they set up typing rules so that tainted data will be propagated, keeping its tag. If tainted data is used as a format string the tester is warned of the possible vulnerability. Sadly, we did not manage to get their tool to report any vulnerabilities with the supplied annotated library functions.

|  | Flawfinder | ITS4 | RATS | Splint | BOON * |
|---|---|---|---|---|---|
| True Positives | 22 (96%) | 21 (91%) | 19 (83%) | 7 (30%) | 4 (27%) |
| False Positives | 15 (71%) | 11 (52%) | 14 (67%) | 4 (19%) | 4 (31%) |
| True Negatives | 6 (29%) | 10 (48%) | 7 (33%) | 17 (81%) | 9 (69%) |
| False Negatives | 1 (4%) | 2 (9%) | 4 (17%) | 16 (70%) | 11 (73%) |

**Table 1.** Overall effectiveness and accuracy of static intrusion prevention. "Positive" means a warning was issued, "Negative" means no warning was issued. In total 44 function calls, 23 unsafe and 21 safe. * BOON only tested with buffer overflow vulnerabilities.

## 4 Comparison of Static Intrusion Prevention Tools

Our testbed contains 20 vulnerable functions chosen from ITS4's vulnerability database (category `RISKY` to `MOST_RISKY`), Secure programming for Linux and UNIX HOWTO [35], and the whole `[fvsn]printf()`-family (see section 2.3 and 2.5 for a complete list). We do not claim that this test suite is perfectly fair, nor complete. But the sources from where we have chosen the vulnerabilities seem reasonable and the test result will at least provide us with an interesting comparison. Our 20 vulnerable functions are used in 13 safe buffer writings, 15 unsafe buffer writings, 8 safe format string calls and 8 unsafe format string calls, in total 44 function calls. We did not go into complex constructs to implement the safe function calls, rather a straight forward solution. An example of the difference between safe and unsafe calls is shown below:

```
char buffer[BUFSIZE];

if(strlen(input_string)<BUFSIZE)
  strcpy(buffer, input_string); /* Safe */
strcpy(buffer, input_string);   /* Unsafe */
```

Overall results from our tests is presented in table 1 and detailed results are presented in table 2. The source code in short form can be found in Appendix A. The exact source code and the print-outs from the various testing tools can be found on our homepage:
`http://www.ida.liu.se/~johwi`

### 4.1 Observations and Conclusions

As you would think all three lexical testing tools ITS4, Flawfinder and RATS, perform about the same on the true positive side. After all, a great part of our tested vulnerabilities where found in their databases or in publications connected to them, as stated before. But they differ considerably on the false positives where ITS4 is best.

For security aware programmers with knowledge of how buffer overflow and format string attacks work these tools can be very helpful. They will most probably get minor testing output, be able to sort out what is important and most

| **Vulnerable Function** | **Flawfinder** | | **ITS4** | | **RATS** | | **Splint** | | **BOON** | |
|---|---|---|---|---|---|---|---|---|---|---|
| | True | False | True | False | True | False | True | False | True | False |
| gets() | 1 | - | 1 | - | 1 | - | 1 | - | 1 | - |
| scanf() | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| fscanf() | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| sscanf() | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| vscanf() | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| vsscanf() | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| vfscanf() | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| cuserid() | 0 | - | 1 | - | 1 | - | 0 | - | 0 | - |
| sprintf() | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| strcat() | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| strcpy() | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| streadd() | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| strecpy() | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| vsprintf() | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| strtrns() | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| printf() | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | - | - |
| fprintf() | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | - | - |
| sprintf() | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | - | - |
| snprintf() | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | - | - |
| vprintf() | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | - | - |
| vfprintf() | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | - | - |
| vsprintf() | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | - | - |
| vsnprintf() | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | - | - |

**Table 2.** Detailed effectiveness and accuracy of intrusion prevention. True = 1 means an unsafe call was found, False = 1 means a safe function call was deemed unsafe. "-" means no such test is possible.

importantly know how to solve the reported problems. For less experienced programmers the output might be too large and since the tools give no instructions on how to solve the problems they will need some other form of help.

Quite interesting is that Splint and BOON finds so few bugs. We contacted Splint author David Larochelle concerning this and he responded that the undetected bugs where not considered a serious threat since they are known to the security community and easily found with the UNIX command `grep`. We disagree with him—why not detect as many security bugs as possible? And why not help the developers that are not aware of the security vulnerabilities coming from misuse of several C functions?

Splint is the only tool that can distinguish between safe and unsafe calls to `strcat()` and `strcpy()`. This implicates that Splint has a good possibility to accurately detect security bugs with a low rate of false positives, just as you would think considering its deeper analysis of the code.

The general feeling we get after running the constraint-based testing tools is that they are still in some kind of a prototype state. Splint has been around under the name LCLint for some time and is used for general syntactical and semantical testing. But the security part needs to be completed. BOON is published as a prototype and should of course be judged as such.

None of the tools has high enough true positives combined with low enough false positives. Our conclusion is that none of them can really give the programmer peace of mind. And combining their output would be tedious.

## 5 Related Work

We have found one comparative study made of static intrusion prevention tools—"Source Code Scanners for Better Code" [19] by Jose Nazario. He compares the result from ITS4, Flawfinder and RATS when testing a part of the source code for OpenLDAP known to be vulnerable. It only contains one call to one of our 23 vulnerable functions—`vsprintf()`. No test for false positives is done either.

A study with another focus but relating to ours is "A Comparison of Static Analysis and Fault Injection Techniques for Developing Robust System Services" by Broadwell and Ong [4]. They investigate the strengths of static analysis versus software fault injection in finding errors in several large software packages such as Apache and MySQL. In static analysis they use ITS4 to find race conditions and BOON to find buffer overflows.

## 6 Conclusions

We have shown that the current state of static intrusion prevention tools is not satisfying. Tools building on lexical analysis produce too many false positives leading to manual work, and tools building on deeper analysis on syntactical and semantical level produce too many false negatives leading to security risks. Thus the main usage for these tools would be as support during development and code auditing, not as a substitute for manual debugging and testing.

## References

1. Arash Baratloo, Navjot Singh, and Timothy Tsai. Libsafe: Protecting critical elements of stacks. White Paper `http://www.research.avayalabs.com/project/libsafe/`, December 1999.
2. Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 2(2):131–152, Spring 1996.
3. Lisa M. Bowman. Companies on the hook for security. `http://news.com.com/2100-1023-821266.html`, January 2002.
4. Pete Broadwell and Emil Ong. A comparison of static analysis and fault injection techniques for developing robust system services. Technical report, Computer Science Division, University of California, Berkeley, `http://www.cs.berkeley.edu/~pbwell/saswifi.pdf`, May 2002.
5. CERT Coordination Center. Cert/cc statistics 1988-2001. `http://www.cert.org/stats/`, February 2002.

6. Matt Conover and w00w00 Security Team. w00w00 on heap overflows. http://www.w00w00.org/files/articles/heaptut.txt, January 1999.
7. Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, Washington DC, USA, August 2001.
8. DilDog. The tao of Windows buffer overflow. http://www.cultdeadcow.com/cDc_files/cDc-351/, April 1998.
9. Wenliang Du and Aditya P. Mathur. Vulnerability testing of software system using fault injection. COAST, Purdue University, Technical Report 98-02 http://www.cerias.purdue.edu/coast/coast-library.html, April 1998.
10. David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, December 1994.
11. David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, February 2002.
12. Anup Ghosh, Tom O'Connor, and Gary McGraw. An automated approach for identifying potential vulnerabilities in software. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 104–114, May 1998.
13. Anup K. Ghosh, Chuck Howell, and James A. Whittaker. Building software securely from the ground up. *IEEE Software*, 19(1):14–16, February 2002.
14. Lawrence R. Halme and R. Kenneth Bauer. AINT misbehaving: A taxonomy of anti-intrusion techniques. http://www.sans.org/newlook/resources/IDFAQ/aint.htm, April 2000.
15. S. C. Johnson. Lint, a C program checker. AT&T Bell Laboratories: Murray Hill, NJ. http://citeseer.nj.nec.com/johnson78lint.html, July 1978.
16. David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, Washington DC, USA, August 2001.
17. Gary McGraw and John Viega. An analysis of how buffer overflow attacks work. IBM developerWorks: Security: Security articles http://www-106.ibm.com/developerworks/security/library/smash.html?dwzon%e=security, March 2000.
18. Jose Nazario. Project pedantic—source code analysis tool(s). http://pedantic.sourceforge.net/, March 2002.
19. Jose Nazario. Source code scanners for better code. The Linux Journal http://www.linuxjournal.com/article.php?sid=5673, January 2002.
20. BBC News. Software security law call. http://news.bbc.co.uk/hi/english/sci/tech/newsid_1762000/1762261.stm, January 2002.
21. Tim Newsham. Format string attacks. White Paper http://www.guardent.com/rd_whtpr_formatNewsham.html, September 2000.
22. Aleph One. Smashing the stack for fun and profit. http://immunix.org/StackGuard/profit.html, November 1996.
23. C E Pramode and C E Gopakumar. Static checking of C programs with LCLint. *Linux Gazette*, 51 http://www.linuxgazette.com/issue51/pramode.html, March 2000.
24. Computer Science and National Research Council Telecommunications Board. Cybersecurity today and tomorrow: Pay now or pay later (prepublication). Technical report, National Academies, USA, http://www.nap.edu/books/0309083125/html/, January 2002.

25. Scut and Team Teso. Exploiting format string vulnerabilities. `http://teso.scene.at/articles/formatstring/`, September 2001.
26. Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Automated detection of format-string vulnerabilities using type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, `http://www.cs.berkeley.edu/~ushankar/`, August 2001.
27. Robert W. Shirey. Request for comments: 2828, Internet security glossary. `http://www.faqs.org/rfcs/rfc2828.html`, May 2000.
28. Secure Software Soliutions. Rough auditing tool for security, RATS 1.3. `http://www.securesw.com/rats/`, September 2001.
29. tf8. Bugtraq id 1387, Wu-Ftpd remote format string stack overwrite vulnerability. `http://www.securityfocus.com/bid/1387`, June 2000.
30. John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Proceedings of the 16th Annual Computer Security Applications Conference*, December 2000.
31. John Viega and Gary McGraw. *Building Secure Software : How to Avoid Security Problems the Right Way.* Addison–Wesley, 2001.
32. Jeffrey Voas and Gary McGraw. *Software Fault Injection: Inoculating Programs Against Errors.* John Wiley & Sons, 1997.
33. David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed System Security Symposium*, pages 3–17, Catamaran Resort Hotel, San Diego, California, February 2000.
34. David A. Wheeler. Flawfinder. Web page `http://www.dwheeler.com/flawfinder/`, May 2001.
35. David A. Wheeler. Secure programming for Linux and Unix HOWTO v2.89. `http://www.dwheeler.com/secure-programs/`, October 2001.
36. John Wilander. Security intrusions and intrusion prevention. Master's thesis, Linkopings universitet, `http://www.ida.liu.se/~johwi`, April 2002.

## A Testbed for Buffer Overflow and Format String Vulnerabilities

In this appendix we have included the 44 function calls used to compare publicly available tools for static intrusion prevention. To shorten it down we have only included the interesting parts. The full code can be downloaded from our homepage http://www.ida.liu.se/~johwi.

```
#define BUFSIZE 9
static char static_global_buffer = 'A';
static char global_buffer[BUFSIZE];

/***** Buffer Overflow Vulnerabilities *****/

pointer = gets(buffer); /* Unsafe */

scanf("%8s", buffer_safe);  /* Safe */
scanf("%s", buffer_unsafe); /* Unsafe */

fscanf(fopen(file_name, "w"), "%8s", buffer_safe);  /* Safe */
fscanf(fopen(file_name, "w"), "%s", buffer_unsafe); /* Unsafe */

sscanf(input_string, "%8s", buffer_safe);  /* Safe */
sscanf(input_string, "%s", buffer_unsafe); /* Unsafe */

if(choice==0) vscanf("%8s", arglist); /* Safe */
else vscanf("%s", arglist);           /* Unsafe */

if(choice==0) vsscanf(input_string, "%8s", arglist); /* Safe */
else vsscanf(input_string, "%s", arglist);           /* Unsafe */

if(choice==0)
  vfscanf(fopen(file_name, "w"), "%8s", arglist); /* Safe */
else
  vfscanf(fopen(file_name, "w"), "%s", arglist);  /* Unsafe */

sprintf(buffer_safe, "%8s", input_string);  /* Safe */
sprintf(buffer_unsafe, "%s", input_string); /* Unsafe */

if(strlen(input_string)<BUFSIZE)
  strcat(buffer_safe, input_string);   /* Safe */
strcat(buffer_unsafe, input_string);   /* Unsafe */

if(strlen(input_string)<BUFSIZE)
  strcpy(buffer_safe, input_string);   /* Safe */
strcpy(buffer_unsafe, input_string);   /* Unsafe */

cuserid(buffer_unsafe); /* Unsafe */

if(choice==0) vsprintf (buffer_safe, "%8s", arglist); /* Safe */
```

```
else vsprintf (buffer_unsafe, "%s", arglist);         /* Unsafe */

res = streadd(buffer_safe, "a", "");            /* Safe */
res = streadd(buffer_unsafe, input_string, ""); /* Unsafe */

res = strecpy(buffer_safe, "a", "");            /* Safe */
res = strecpy(buffer_unsafe, input_string, ""); /* Unsafe */

res = strtrns("a", "a", "A", buffer_safe);            /* Safe */
res = strtrns(input_string, "a", "A", buffer_unsafe); /* Unsafe */

/***** Format String Vulnerabilities *****/

printf(&static_global_buffer); /* Safe */
printf(global_buffer);         /* Unsafe */

fprintf(stdout, &static_global_buffer); /* Safe */
fprintf(stdout, global_buffer);         /* Unsafe */

char local_buffer[BUFSIZE];
/* Safe */
sprintf(local_buffer, &static_global_buffer, input_string);
/* Unsafe */
sprintf(local_buffer, global_buffer, input_string);

char local_buffer[BUFSIZE];
/* Safe */
snprintf(local_buffer, BUFSIZE, &static_global_buffer, input_string);
/* Unsafe */
snprintf(local_buffer, BUFSIZE, global_buffer, input_string);

if(choice==0) vprintf(&static_global_buffer, arglist); /* Safe */
else vprintf(global_buffer, arglist);                  /* Unsafe */

if(choice==0) /* Safe */
  vfprintf(stdout, &static_global_buffer, arglist);
else /* Unsafe */
  vfprintf(stdout, global_buffer, arglist);

char local_buffer[BUFSIZE];
if(choice==0) /* Safe */
  vsprintf(local_buffer, &static_global_buffer, arglist);
else /* Unsafe */
  vsprintf(local_buffer, global_buffer, arglist);

char local_buffer[BUFSIZE];
if(choice==0) /* Safe */
  vsnprintf(local_buffer, BUFSIZE, &static_global_buffer, arglist);
else /* Unsafe */
  vsnprintf(local_buffer, BUFSIZE, global_buffer, arglist);
```