

Improving Security Using Extensible Lightweight Static Analysis

David Evans and David Larochele, *University of Virginia*

Building secure systems involves numerous complex and challenging problems, ranging from building strong cryptosystems and designing authentication protocols to producing a trust model and security policy. Despite these challenges, most security attacks exploit either human weaknesses—such as poorly chosen passwords and careless configuration—or software implementation flaws. Although it’s hard to do much about human frailties, some help is available through

education, better interface design, and security-conscious defaults. With software implementation flaws, however, the problems are typically both preventable and well understood.

Analyzing reports of security attacks quickly reveals that most attacks do not result from clever attackers discovering new kinds of flaws, but rather stem from repeated exploits of well-known problems. Figure 1 summarizes Mitre’s Common Vulnerabilities and Exposures list of 190 entries from 1 January 2001 through 18 September 2001.¹ Thirty-seven of these entries are standard buffer overflow vulnerabilities (including three related memory-access vulnerabilities), and 11 involve format bugs. Most of the rest also reveal common flaws detectable by static analysis, including resource leaks (11), file name problems (19), and symbolic links (20). Only four of the entries involve

cryptographic problems. Analyses of other vulnerability and incident reports reveal similar repetition. For example, David Wagner and his colleagues found that buffer overflow vulnerabilities account for approximately 50 percent of the Software Engineering Institute’s CERT advisories.²

So why do developers keep making the same mistakes? Some errors are caused by legacy code, others by programmers’ carelessness or lack of awareness about security concerns. However, the root problem is that while security vulnerabilities, such as buffer overflows, are well understood, the techniques for avoiding them are not codified into the development process. Even conscientious programmers can overlook security issues, especially those that rely on undocumented assumptions about procedures and data types. Instead of relying on programmers’ memories, we should strive to produce

Security attacks that exploit well-known implementation flaws occur with disturbing frequency because the software development process does not include techniques for preventing those flaws. The authors have developed an extensible tool that detects common flaws using lightweight static analysis.

tools that codify what is known about common security vulnerabilities and integrate it directly into the development process.

This article describes a way to codify that knowledge. We describe Splint, a tool that uses lightweight static analysis to detect likely vulnerabilities in programs. Splint's analyses are similar to those done by a compiler. Hence, they are efficient and scalable, but they can detect a wide range of implementation flaws by exploiting annotations added to programs.

Mitigating software vulnerabilities

"Our recommendation now is the same as our recommendation a month ago, if you haven't patched your software, do so now."

—Scott Culp, security program manager for Microsoft's security response center

In this quotation, Culp is commenting on the Internet Information Server's buffer overflow vulnerability that was exploited by the Code Red worm to acquire over 300,000 zombie machines for launching a denial-of-service attack on the White House Web site. The quotation suggests one way to deal with security vulnerabilities: wait until the bugs are exploited by an attacker, produce a patch that you hope fixes the problem without introducing new bugs, and whine when system administrators don't install patches quickly enough. Not surprisingly, this approach has proven largely ineffective.

We can group more promising approaches for reducing software flaw damage into two categories: mitigate the damage flaws can cause, or eliminate flaws before the software is deployed.

Limiting damage

Techniques that limit security risks from software flaws include modifying program binaries to insert runtime checks or running applications in restricted environments to limit what they may do.^{3,4} Other projects have developed safe libraries⁵ and compiler modifications⁶ specifically for addressing classes of buffer overflow vulnerabilities. These approaches all reduce the risk of security vulnerabilities while requiring only minimal extra work from application developers.

One disadvantage of runtime damage-limitation approaches is that they increase performance overhead. More importantly,

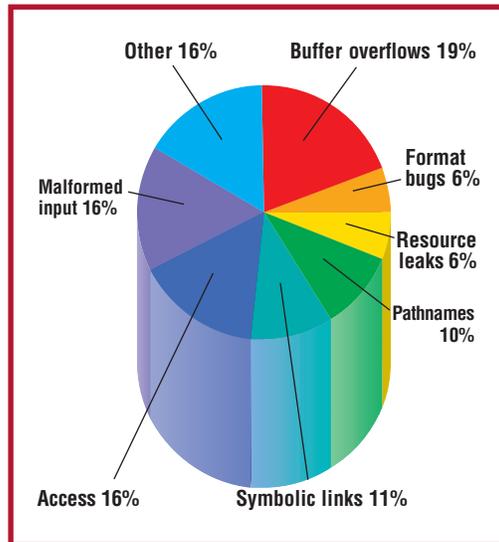


Figure 1. Common Vulnerabilities and Exposures list for the first nine months of 2001. Most of the entries are common flaws detectable by static analysis, including 37 buffer overflow vulnerabilities.

such approaches do not eliminate the flaw but simply replace it with a denial-of-service vulnerability. Recovering from a detected problem typically requires terminating the program. Hence, although security-sensitive applications should use damage-limitation techniques, the approach should not supplant techniques for eliminating flaws.

Eliminating flaws

Techniques to detect and correct software flaws include human code reviews, testing, and static analysis. Human code reviews are time-consuming and expensive but can find conceptual problems that are impossible to find automatically. However, even extraordinarily thorough people are likely to overlook more mundane problems. Code reviews depend on the expertise of the human reviewers, whereas automated techniques can benefit from expert knowledge codified in tools. Testing is typically ineffective for finding security vulnerabilities. Attackers attempt to exploit weaknesses that system designers did not consider, and standard testing is unlikely to uncover such weaknesses.

Static analysis techniques take a different approach. Rather than observe program executions, they analyze source code directly. Thus, using static analysis lets us make claims about all possible program executions rather than just the test-case execution. From a security viewpoint, this is a significant advantage.

There are a range of static analysis techniques, offering tradeoffs between the re-

Splint finds potential vulnerabilities by checking to see that source code is consistent with the properties implied by annotations.

quired effort and analysis complexity. At the low-effort end are standard compilers, which perform type checking and other simple program analyses. At the other extreme, full program verifiers attempt to prove complex properties about programs. They typically require a complete formal specification and use automated theorem provers. These techniques have been effective but are almost always too expensive and cumbersome to use on even security-critical programs.

Our approach

We use lightweight static analysis techniques that require incrementally more effort than using a compiler but a fraction of the effort required for full program verification. This requires certain compromises. In particular, we use heuristics to assist in the analysis. Our design criteria eschew theoretical claims in favor of useful results.

Detecting likely vulnerabilities in real programs depends on making compromises that increase the class of properties that can be checked while sacrificing soundness and completeness. This means that our checker will sometimes generate false warnings and sometimes miss real problems, but our goal is to create a tool that produces useful results for real programs with a reasonable effort.

Splint overview

Splint (previously known as LCLint) is a lightweight static analysis tool for ANSI C. Here, we describe Splint, version 3.0.1, which is available as source code and binaries for several platforms under GPL from www.splint.org.

We designed Splint to be as fast and easy to use as a compiler. It can do checking no compiler can do, however, by exploiting annotations added to libraries and programs that document assumptions and intents. Splint finds potential vulnerabilities by checking to see that source code is consistent with the properties implied by annotations.

Annotations

We denote annotations using stylized C comments identified by an @ character following the /* comment marker. We associate annotations syntactically with function parameters and return values, global variables, and structure fields.

The annotation `/*@nonnull@*/`, for ex-

ample, can be used syntactically like a type qualifier. In a parameter declaration, the `nonnull` annotation documents an assumption that the value passed for this parameter is not `NULL`. Given this, Splint reports a warning for any call site where the actual parameter might be `NULL`. In checking the function's implementation, Splint assumes that the `nonnull`-annotated parameter's initial value is not `NULL`. On a return value declaration, a `nonnull` annotation would indicate that the function never returns `NULL`. Splint would then report a warning for any return path that might return `NULL`, and would check the call-site assuming the function result is never `NULL`. In a global variable declaration, a `nonnull` annotation indicates that the variable's value will not be `NULL` at an interface point—that is, it might be `NULL` within the function's body but would not be `NULL` at a call site or return point. Failure to handle possible `NULL` return values is unlikely to be detected in normal testing, but is often exploited by denial of service attacks.

Annotations can also document assumptions over an object's lifetime. For example, we use the `only` annotation on a pointer reference to indicate that the reference is the sole long-lived reference to its target storage (there might also be temporary local aliases). An `only` annotation implies an obligation to release storage. The system does this either by passing the object as a parameter annotated with `only`, returning the object as a result annotated with `only` or assigning the object to an external reference annotated with `only`. Each of these options transfers the obligation to some other reference. For example, the library storage allocator `malloc` is annotated with `only` on its result, and the deallocator `free` also takes an `only` parameter. Hence, one way to satisfy the obligation to release `malloc`'s storage is to pass it to `free`.

Splint reports a warning for any code path that fails to satisfy the storage-release obligation, because it causes a memory leak. Although memory leaks do not typically constitute a direct security threat, attackers can exploit them to increase a denial-of-service attack's effectiveness. In the first half of 2001, three of the Common Vulnerabilities and Exposures entries involved memory leaks (CVE-2001-0017, CVE-2001-0041 and CVE-2001-0055). Some storage management can't be

modeled with only references, as the programs must share references across procedure and structure boundaries. To contend with this, Splint provides annotations for describing different storage management models.⁷

Analysis

There are both theoretical and practical limits to what we can analyze statically. Precise analysis of the most interesting properties of arbitrary C programs depends on several undecidable problems, including reachability and determining possible aliases.⁸ Given this, we could either limit our checking to issues like type checking, which do not depend on solving undecidable problems, or admit to some imprecision in our results. Because our goal is to do as much useful checking as possible, we allow checking that is both unsound and incomplete. Splint thus produces both false positives and false negatives. We intend the warnings to be as useful as possible to programmers but offer no guarantee that all messages indicate real bugs or that all bugs will be found. We also make it easy for users to configure checking to suppress particular messages and weaken or strengthen checking assumptions.

With static analysis, designers face a tradeoff between precision and scalability. To make our analysis fast and scalable to large programs, we made certain compromises. The most important is to limit our analysis to data flow within procedure bodies. Splint analyzes procedure calls using information from annotations that describes preconditions and postconditions. We made another compromise between flow-sensitive analysis, which considers all program paths, and flow-insensitive analysis, which ignores control flow. Splint considers control-flow paths, but, to limit analysis path blowup, it merges possible paths at branch points. It analyzes loops using heuristics to recognize common idioms. This lets Splint correctly determine the number of iterations and bounds of many loops without requiring loop invariants or abstract evaluation. Splint's simplifying assumptions are sometimes wrong; this often reveals convoluted code that is a challenge for both humans and automated tools to analyze. Hence, we provide easy ways for programmers to customize checking behavior locally and suppress spurious warnings that result from imprecise analysis.

Buffer overflows

Buffer overflow vulnerabilities are perhaps the single most important security problem of the past decade. The simplest buffer overflow attack, *stack smashing*, overwrites a buffer on the stack, replacing the return address. Thus, when the function returns, instead of jumping to the return address, control jumps to the address the attacker placed on the stack. The attacker can then execute arbitrary code. Buffer overflow attacks can also exploit buffers on the heap, but these are less common and harder to create.

C programs are particularly vulnerable to buffer overflow attacks. C was designed with an emphasis on performance and simplicity rather than security and reliability. It provides direct low-level memory access and pointer arithmetic without bounds checking. Worse, the ANSI C library provides unsafe functions—such as `gets`—that write an unbounded amount of user input into a fixed-size buffer without any bounds checking. Buffers stored on the stack are often passed to these functions. To exploit such vulnerabilities, an attacker merely enters an input larger than the buffer's size, encoding an attack program binary in the input.

Splint detects both stack and heap-based buffer overflow vulnerabilities. The simplest detection techniques just identify calls to often misused functions; more precise techniques depend on function descriptions and program-value analysis.

Use warnings

The simplest way to detect possible buffer overflows is to produce a warning whenever the code uses library functions susceptible to buffer overflow vulnerabilities. The `gets` function is always vulnerable, so it seems reasonable for a static analysis tool to report all uses of `gets`. Other library functions, such as `strcpy`, can be used safely but are often the source of buffer overflow vulnerabilities. Splint provides the annotation `warn flag-specifier message`, which precedes a declaration to indicate that declarator use should produce a warning. For example, the Splint library declares `gets` with

```
/*@warn bufferoverflowhigh
   "Use of gets leads to ... "@*/
```

Buffer overflow vulnerabilities are perhaps the single most important security problem of the past decade.

Splint resolves preconditions using postconditions from previous statements and any annotated preconditions for the function.

to indicate that Splint should issue a warning message whenever `gets` is used and the `bufferoverflowhigh` flag is set.

Several security scanning tools provide similar functionality, including Flawfinder (www.dwheeler.com/flawfinder), ITS4,⁹ and the Rough Auditing Tool for Security (www.securesw.com/rats). Unlike Splint, however, these tools use lexical analysis instead of parsing the code. This means they will report spurious warnings if the names of the vulnerable functions are used in other ways (for example, as local variables).

The main limitation of use warnings is that they are so imprecise. They alert humans to possibly dangerous code but provide no assistance in determining whether a particular use of a potentially dangerous function is safe. To improve the results, we need a more precise specification of how a function might be safely used, and a more precise analysis of program values.

Describing functions

Consider the `strcpy` function: it takes two `char *` parameters (`s1` and `s2`) and copies the string that the second parameter points to into the buffer to which the first parameter points. A call to `strcpy` will overflow the buffer pointed to by the first parameter if that buffer is not large enough to hold the string pointed to by the second parameter. This property can be described by adding a *requires* clause to the declaration of `strcpy`: `/*@requires maxSet(s1)>=maxRead(s2) @*/`.

This precondition uses two buffer attribute annotations, `maxSet` and `maxRead`. The value of `maxSet(b)` is the highest integer i such that `b[i]` can be safely used as an lvalue (that is, on the left side of an assignment expression). The value of `maxRead(b)` is the highest integer i such that `b[i]` can be safely used as an rvalue. The `s2` parameter also has a `nullterminated` annotation that indicates that it is a `nullterminated` character string. This implies that `s2[i]` must be a `NUL` character for some $i \leq \text{maxRead}(s2)$.

At a call site, Splint produces a warning if a precondition is not satisfied. Hence, a call `strcpy(s, t)` would produce a warning if Splint cannot determine that `maxSet(s) >= maxRead(t)`. The warning would indicate that the buffer allocated for `s` might be overrun by the `strcpy` call.

Analyzing program values

Splint analyzes a function body starting from the annotated preconditions and checks that the function implementation ensures the postconditions. It generates preconditions and postconditions at the expression level in the parse tree using internal rules or, in the case of function calls, annotated descriptions. For example, the declaration `char buf[MAXSIZE]` generates the postconditions `maxSet(buf) = MAXSIZE - 1` and `minSet(buf) = 0`.

Where the expression `buf[i]` is used as an lvalue, Splint generates the precondition `maxSet(buf) >= i`. All constraint variables also identify particular code locations. Because a variable's value can change, the analysis must distinguish between values at different code points.

Splint resolves preconditions using postconditions from previous statements and any annotated preconditions for the function. If it cannot resolve a generated precondition at the beginning of a function or satisfy a documented postcondition at the end, Splint issues a descriptive warning about the unsatisfied condition. Hence, for the `buf[i]` example above, Splint would produce a warning if it cannot determine that the value of i is between 0 and `MAXSIZE - 1`.

Splint propagates constraints across statements using an axiomatic semantics and simplifies constraints using constraint-specific algebraic rules, such as `maxSet(ptr + i) = maxSet(ptr) - i`.

To handle loops, we use heuristics that recognize common loop forms.¹⁰ Our experience indicates that a few heuristics can match many loops in real programs. This lets us effectively analyze loops without needing loop invariants or expensive analyses.

Extensible checking

In addition to the built-in checks, Splint provides mechanisms for defining new checks and annotations to detect new vulnerabilities or violations of application-specific properties. A large class of useful checks can be described as constraints on attributes associated with program objects or the global execution state. Unlike types, however, the values of these attributes can change along an execution path.

Splint provides a general language that lets users define attributes associated with

```

attribute taintedness
  context reference char *
  oneof untainted, tainted
  annotations
    tainted reference ==> tainted
    untainted reference ==> untainted
  transfers
    tainted as untainted ==> error "Possibly tainted storage used as untainted."
  merge
    tainted + untainted ==> tainted
  defaults
    reference ==> tainted
    literal ==> untainted
    null ==> untainted
end

```

Figure 2. Definition of the taintedness attribute.

different kinds of program objects as well as rules that both constrain the values of attributes at interface points and specify how attributes change. The limited expressiveness of user attributes means that Splint can check user-defined properties efficiently. Because user-defined attribute checking is integrated with normal checking, Splint's analysis of user-defined attributes can take advantage of other analyses, such as alias and nullness analysis.

Next, we illustrate how user-defined checks can detect new vulnerabilities using a taintedness attribute to detect format bugs. We have also used extensible checking to detect misuses of files and sockets (such as failing to close a file or to reset a read/write file between certain operations) and incompatibilities between Unix and Win32.¹¹

Detecting format bugs

In June 2000, researchers discovered a new class of vulnerability: the *format bug*.¹² If an attacker can pass hostile input as the format string for a variable arguments routine such as `printf`, the attacker can write arbitrary values to memory and gain control over the host in a manner similar to a buffer overflow attack. The `%n` directive is particularly susceptible to attack, as it treats its corresponding argument as an `int *`, and stores the number of bytes printed so far in that location.

A simple way to detect format vulnerabilities is to provide warnings for any format string that is unknown at compile time. If the `+formatconst` flag is set, Splint issues a warning at all callsites where a format string is not known at compile time. This can produce spurious messages, however, because there might be unknown format strings that are not vulnerable to hostile input.

A more precise way to detect format bugs is to only report warnings when the format string is derived from potentially malicious data (that is, when it comes from the user or external environment). Perl's taint option¹³ suggests a way to do this. The taint option, which is activated by running Perl with the `-T` flag, considers all user input as tainted and produces a runtime error, halting execution before a tainted value is used in an unsafe way. Untainted values can be derived from tainted input by using Perl's regular expression matching.

Taintedness attribute

Splint can be used to detect possibly dangerous operations with tainted values at compile time. To accomplish this, we define a *taintedness* attribute associated with `char *` objects and introduce the annotations `tainted` and `untainted` to indicate assumptions about the taintedness of a reference. Umesh Shankar and his colleagues used a similar approach.¹⁴ Instead of using attributes with explicit rules, they used type qualifiers. This lets them take advantage of type theory, and, in particular, use well-known type-inference algorithms to automatically infer the correct type qualifiers for many programs.

Splint's attributes are more flexible and expressive than type qualifiers. Figure 2 shows the complete attribute definition. The first three lines define the taintedness attribute associated with `char *` objects, which can be in one of two states: `untainted` or `tainted`. The next clause specifies rules for transferring objects between references, for example, by passing a parameter or returning a result. The `tainted as untainted ==> error` rule directs Splint to report a warning when a `tainted` object is used where an `untainted` object is expected. This would oc-

Using Splint is an iterative process.

cur if the system passed a tainted object as an untainted parameter or returned it as an untainted result. All other transfers (for example, untainted as tainted) are implicitly permitted and leave the transferred object in its original state.

Next, the merge clause indicates that combining tainted and untainted objects produces a tainted object. Thus, if a reference is tainted along one control path and untainted along another control path, checking assumes that it is tainted after the two branches merge. It is also used to merge taintedness states in function specifications (see the `strcat` example in the next section).

The annotations clause defines two annotations that programmers can use in declarations to document taintedness assumptions. In this case, the names of the annotations match the taintedness states. The final clause specifies default values used for declarators without taintedness annotations. We choose default values to make it easy to start checking an unannotated program. Here we assume unannotated references are possibly tainted and Splint will report a warning where unannotated references are passed to functions that require untainted parameters. The warnings indicate either a format bug in the code or a place where an untainted annotation should be added. Running Splint again after adding the annotation will propagate the newly documented assumption through the program.

Specifying library functions

When the source code for library code is unavailable, we cannot rely on the default annotations because Splint needs the source code to detect inconsistencies. We must therefore provide annotated declarations that document taintedness assumptions for standard library functions. We do this by providing annotated declarations in the `tainted.xh` file. For example,

```
int printf
  (/*@untainted@*/ char *fmt, ...);
```

indicates that the first argument to `printf` must be untainted. We can also use ensures clauses to indicate that a value is tainted after a call returns. For example, the first parameter to `fgets` is tainted after `fgets` returns:

```
char *fgets
  (/*@returned@*/ char *s, int n,
  FILE *stream)
  /*@ensures tainted s@*/ ;
```

The `returned` annotation on the parameter means that the return value aliases the storage passed as `s`, so the result is also tainted (Splint's alias analysis also uses this information).

We also must deal with functions that might take tainted or untainted objects, but where the final taintedness states of other parameters and results might depend on the parameters' initial taintedness states. For example, `strcat` is annotated this way:

```
char *strcat
  (/*@returned@*/ char *s1,
  char *s2)
  /*@ensures s1:taintedness =
  s1:taintedness | s2:taintedness@*/
```

Because the parameters lack annotations, they are implicitly tainted according to the default rules, and either untainted or tainted references can be passed as parameters to `strcat`. The `ensures` clause means that after `strcat` returns, the first parameter (and the result, because of the `returned` annotation on `s1`) will be tainted if either passed object was tainted. Splint merges the two taintedness states using the attribute definition rules—hence, if the `s1` parameter is untainted and the `s2` parameter is tainted, the result and first parameter will be tainted after `strcat` returns.

Experience

Using Splint is an iterative process. First, we run Splint to produce warnings and then change either the code or the annotations accordingly. Next, we run Splint again to check the changes and propagate the newly documented assumptions. We continue this process until Splint issues no warnings. Because Splint checks approximately 1,000 lines per second, running Splint repeatedly is not burdensome.

Splint's predecessor, LCLint, has been used to detect a range of problems, including data hiding¹⁵ and memory leaks, dead storage usage, and `NULL` dereferences⁷ on programs comprising hundreds of thousands of lines of code. LCLint is widely used

by working programmers, especially in the open-source development community.^{16,17}

So far, our experience with buffer overflow checking and extensible checking is limited but encouraging. We have used Splint to detect both known and previously unknown buffer overflow vulnerabilities in `wu-ftpd`, a popular ftp server, and BIND, the libraries and tools that comprise the Domain Name System's reference implementation.

Here, we summarize our experience analyzing `wu-ftpd` version 2.5.0, a 20,000 line program with known (but not known specifically to the authors when the analysis was done) format and buffer overflow bugs. We detected the known flaws as well as finding some previously unknown flaws in `wu-ftpd`. It takes Splint less than four seconds to check all of `wu-ftpd` on a 1.2-GHz Athlon machine.

Format bugs

Running Splint on `wu-ftpd` version 2.5.0 produced two warnings regarding taintedness. The first one was:

```
ftpd.c: (in function vreply)
ftpd.c:4608:69: Invalid transfer from implicitly
  tainted fmt to untainted (Possibly tainted
  storage used as untainted.):
   vsnprintf(..., fmt, ...)
ftpd.c:4586:33: fmt becomes implicitly
  tainted
```

In `tainted.xh`, `vsprintf` is declared with an `untainted` annotation on its format string parameter. The passed value, `fmt`, is a parameter to `vreply`, and hence it might be tainted according to the default rules. We added an `untainted` annotation to the `fmt` parameter declaration to document the assumption that an untainted value must be passed to `vreply`.

After adding the annotation, Splint reported three warnings for possibly tainted values passed to `vreply` in `reply` and `lreply`. We thus added three additional annotations. Running Splint again produced five warnings—three of which involved passing the global variable `globerr` as an untainted parameter. Adding an `untainted` annotation to the variable declaration directed Splint to ensure that `globerr` is never tainted at an interface point. The other warnings concerned possibly tainted values passed to `lreply` in

Table 1

False warnings checking `wu-ftpd`

Cause	Number	Percent
External assumptions	6	7.9
Arithmetic limitations	13	17.1
Alias analysis	3	3.9
Flow control	20	26.3
Loop heuristics	10	13.2
Other	24	31.6

`site_exec`. Because these values were obtained from a remote user, they constituted a serious vulnerability (CVE-2000-0573).

The second message Splint produced in the first execution reported a similar invalid transfer in `setproctitle`. After adding an annotation and rerunning Splint, we found two additional format string bugs in `wu-ftpd`. These vulnerabilities, described in CERT CA-2000-13, are easily fixed using the `%s` constant format string.

We also ran Splint on `wu-ftpd` 2.6.1, a version that fixed the known format bugs. After adding eight `untainted` annotations, Splint ran without reporting any format bug vulnerabilities.

Buffer overflow vulnerabilities

Running Splint on `wu-ftpd` 2.5 without adding annotations produced 166 warnings for potential out-of-bounds writes. After adding 66 annotations in an iterative process such as the one we described above for checking taintedness, Splint produced 101 warnings. Twenty-five of these warnings indicated real problems and 76 were false (summarized in Table 1).

Six of the false warnings resulted because Splint was unaware of assumptions external to the `wu-ftpd` code. For example, `wu-ftpd` allocates an array based on the system constant `OPEN_MAX`, which specifies the maximum number of files a process can have open. The program then writes to this buffer using the integer value of an open file stream's file descriptor as the index. This is safe because the file descriptor's value is always less than `OPEN_MAX`. Without a more detailed specification of the meaning of file descriptor values, there is no way for a static analysis tool to determine that the memory access is safe.

Ten false warnings resulted from loops that were correct but did not match the loop heuristics. To some extent, we could address this by incorporating additional loop

No tool will eliminate all security risks, but lightweight static analysis should be part of the development process for security-sensitive applications.

heuristics into Splint, but there will always be some unmatched loops. The remaining 60 spurious messages resulted from limitations in Splint's ability to reason about arithmetic, control flow, and aliases. We're optimistic that implementing known techniques into Splint will overcome many of these limitations without unacceptable sacrifices in efficiency and usability. It is impossible, though, to eliminate all spurious messages because of the general undecidability of static analysis.

Lightweight static analysis is a promising technique for detecting likely software vulnerabilities, helping programmers fix them before software is deployed rather than patch them after attackers exploit the problem.

Although static analysis is an important approach to security, it is not a panacea. It does not replace runtime access controls, systematic testing, and careful security assessments. Splint can only find problems that are revealed through inconsistencies between the code, language conventions, and assumptions documented in annotations. Occasionally, such inconsistencies reveal serious design flaws, but Splint offers no general mechanisms for detecting high-level design flaws that could lead to security vulnerabilities.

The effort involved in annotating programs is significant, however, and limits how widely these techniques will be used in the near future. Providing an annotated standard library solves part of the problem. However, it does not remove the need to add annotations to user functions where correctness depends on documenting assumptions that cross interface boundaries. Much of the work in annotating legacy programs is fairly tedious and mechanical, and we are currently working on techniques for automating this process. Techniques for combining runtime information with static analysis to automatically guess annotations also show promise.¹⁸

No tool will eliminate all security risks, but lightweight static analysis should be part of the development process for security-sensitive applications. We hope that the

security community will develop a tool suite that codifies knowledge about security vulnerabilities in a way that makes it accessible to all programmers.

Newly discovered security vulnerabilities should not lead just to a patch for a specific program's problem, but also to checking rules that detect similar problems in other programs and prevent the same mistake in future programs. Lightweight static checking will play an important part in codifying security knowledge and moving from today's penetrate-and-patch model to a penetrate-patch-and-prevent model where, once understood, a security vulnerability can be codified into tools that detect it automatically. 

Acknowledgments

David Evans' work is supported by an NSF Career Award and a NASA Langley Research Grant. David Larochelle's work is supported by a Usenix Student Research Grant.

References

1. *Common Vulnerabilities and Exposures*, version 20010918, The Mitre Corporation, 2001; <http://cve.mitre.org> (current Nov. 2001).
2. D. Wagner et al., "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," *Proc. 2000 Network and Distributed System Security Symp.*, Internet Society, Reston, Va., 2000; www.isoc.org/ndss2000/proceedings (current Nov. 2001).
3. I. Goldberg et al., "A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker," *Proc. Sixth Usenix Security Symp.*, Usenix Assoc., Berkeley, Calif., 1996; www.cs.berkeley.edu/~daw/papers/janus-usenix96.ps (current Nov. 2001).
4. D. Evans and A. Twyman, "Flexible Policy-Directed Code Safety," *IEEE Symp. Security and Privacy*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 32-45.
5. A. Baratloo, N. Singh, and T. Tsai, "Transparent Runtime Defense Against Stack-Smashing Attacks," *Proc. Ninth Usenix Security Symp.*, Usenix Assoc., Berkeley, Calif., 2000; www.usenix.org/events/usenix2000/general/baratloo.html (current Nov. 2001).
6. C. Cowan et al., "StackGuard: Automatic Adaptive Detection and Prevention of Buffer Overflow Attacks," *Proc. Seventh Usenix Security Symp.*, Usenix Assoc., Berkeley, Calif., 1998; <http://immunix.org/StackGuard/usenixsc98.pdf> (current Nov. 2001).
7. D. Evans, "Static Detection of Dynamic Memory Errors," *SIGPLAN Conf. Programming Language Design and Implementation*, ACM Press, New York, 1996, pp. 44-53.
8. G. Ramalingam, "The Undecidability of Aliasing," *ACM Trans. Programming Languages and Systems*, vol. 16, no. 5, 1994, pp. 1467-1471.
9. J. Viega et al., "ITS4: A Static Vulnerability Scanner for C and C++ Code," *Proc. Ann. Computer Security Applications Conf.*, IEEE CS Press, Los Alamitos, Calif., 2000; www.acsac.org/2000/abstracts/78.html (current Nov. 2001).
10. D. Larochelle and D. Evans, "Statically Detecting

Likely Buffer Overflow Vulnerabilities,” *Proc. 10th Usenix Security Symp.*, Usenix Assoc., Berkeley, Calif., 2001; www.usenix.org/events/sec01/larochelle.html (current Nov. 2001).

11. C. Barker, “Static Error Checking of C Applications Ported from UNIX to WIN32 Systems Using LCLint,” senior thesis, Dept. Computer Science, University of Virginia, Charlottesville, 2001.
12. C. Cowan et al., “FormatGuard: Automatic Protection From printf Format String Vulnerabilities,” *Proc. 10th Usenix Security Symp.*, Usenix Assoc., Berkeley, Calif., 2001; www.usenix.org/events/sec01/cowanbarringer.html (current Nov. 2001).
13. L. Wall, T. Christiansen, and J. Orwant, *Programming Perl*, 3rd edition, O’Reilly & Associates, Sebastopol, Calif., 2000.
14. U. Shankar et al., “Detecting Format String Vulnerabilities with Type Qualifiers,” *Proc. 10th Usenix Security Symp.*, Usenix Assoc., Berkeley, Calif., 2001; www.usenix.org/events/sec01/shankar.html (current Nov. 2001).
15. D. Evans et al., “LCLint: A Tool for Using Specifications to Check Code,” *SIGSOFT Symp. Foundations of Software Eng.*, ACM Press, New York, 1994; www.cs.virginia.edu/~evans/sigsoft94.html (current Nov. 2001).
16. D. Santo Orcero, “The Code Analyzer LCLint,” *Linux Journal*, May 2000; www.linuxjournal.com/article.php?sid=3599 (current Nov. 2001).
17. C.E. Pramode and C.E. Gopakumar, “Static Checking of C programs with LCLint,” *Linux Gazette*, Mar. 2000; www.linuxgazette.com/issue51/pramode.html (current Nov. 2001).

18. M.D. Ernst et al., “Dynamically Discovering Likely Program Invariants to Support Program Evolution,” *Proc. Int’l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 213–224.

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.

About the Authors



David Evans is an assistant professor in University of Virginia’s Department of Computer Science. His research interests include annotation-assisted static checking and programming swarms of computing devices. He received BS, MS, and PhD in computer science from the Massachusetts Institute of Technology. Contact him at the Department of Computer Science, School of Engineering & Applied Science, University of Virginia, 151 Engineer’s Way, P.O. Box 400740; Charlottesville, VA 22904-4740; devans@virginia.edu or www.cs.virginia.edu/evans.

David Larochelle is a PhD student in University of Virginia’s Department of Computer Science, where he works on lightweight static analysis with a focus on security. He has a BS in computer science from the College of William and Mary in Williamsburg, Virginia, and an MCS in computer science from the University of Virginia. Contact him at the Department of Computer Science, School of Engineering & Applied Science, University of Virginia, 151 Engineer’s Way, P.O. Box 400740, Charlottesville, VA 22904-4740; larochelle@cs.virginia.edu or www.ds.virginia.edu/larochelle.



Call for Articles

Software Engineering Education: A Focus on Practice

Publication: September/October 2002 • Submission: 1 April 2002

What do software engineering professionals need to know, according to those who hire and manage them? They must be able to produce secure and high-quality systems in a timely, predictable, and cost-effective manner.

This special issue will focus on the methods and techniques for enhancing software education programs worldwide—academic, re-education, alternative—to give graduates the knowledge and skills they need for an industrial software career.

Potential topics include

- balancing theory, technology, and practice
- experience reports with professional education
- software processes in the curriculum
- teaching software engineering practices (project management, requirements, design, construction, ...)
- quality and security practices
- team building
- software engineering in beginning courses
- computer science education vs. SE education
- undergraduate vs. graduate SE education
- nontraditional education (distance education, asynchronous learning, laboratory teaching, ...)
- innovative SE courses or curricula
- training for the workplace

For more information about the focus, contact the guest editors; for author guidelines and submission details, contact the magazine assistant at software@computer.org or go to <http://computer.org/software/author.htm>.

Submissions are due at software@computer.org on or before 1 April 2002. If you would like advance editorial comment on a proposed topic, send the guest editors an extended abstract by 1 February; they will return comments by 15 February.

Manuscripts must not exceed 5,400 words including figures and tables, which count for 200 words each. Submissions in excess of these limits may be rejected without refereeing. The articles we deem within the theme’s scope will be peer-reviewed and are subject to editing for magazine style, clarity, organization, and space. We reserve the right to edit the title of all submissions. Be sure to include the name of the theme for which you are submitting an article.

Guest Editors:

Watts S. Humphrey, Software Engineering Institute
Carnegie Mellon University, watts@sei.cmu.edu

Thomas B. Hilburn, Dept. of Computing and Mathematics
Embry-Riddle Aeronautical University hilburn@db.erau.edu