To appear in *IEEE Software*, Jan/Feb 2002.
Pre-publication version 31 October 2001. Not for distribution.

Improving Security Using Extensible Lightweight Static Analysis

David Evans and David Larochelle
University of Virginia

Abstract

Most security attacks exploit instances of well-known classes of implementation flaws. Many of these flaws could be detected and eliminated before software is deployed. These problems continue to be present with disturbing frequency, not because they are not sufficiently understood by the security community, but because techniques for preventing them have not been integrated into the software development process. This paper describes an extensible tool that uses lightweight static analysis to detect common security vulnerabilities (including buffer overflows and format string vulnerabilities) and can be readily extended to detect new vulnerabilities.

Keywords: static analysis, security vulnerabilities, checking, buffer overflows, format bugs.

1. Software and Security

Building secure systems involves a myriad of complex and challenging problems ranging from building strong cryptosystems and designing authentication protocols to producing a trust model and security policy. Despite all these hard problems, the vast preponderance of security attacks exploit either human weaknesses (e.g., poorly chosen passwords, careless configuration, social engineering) or software implementation flaws. It is hard to do much about human frailties, although education, better interface design, and security-conscious defaults can help here. Software implementation flaws, however, typically involve well-understood and preventable problems.

An analysis of any vulnerability database quickly reveals that most software vulnerabilities are not the result of clever attackers discovering new classes of software flaws. Instead, the vast preponderance of vulnerabilities constitute repetitive instances of well-known problems. The Common Vulnerabilities and Exposures list contains 190 entries from 1 January 2001 through 18 September 2001 [3], summarized in Figure 1. Of these 37 are standard buffer overflow vulnerabilities (including three related memory access vulnerabilities). Section 4 describes how Splint detects buffer overflow vulnerabilities. Eleven entries involve format bugs, described in Section 5.1. Most of the other vulnerabilities also reveal common flaws that could be detected using static analyses including resource leaks (11 vulnerabilities), problems with file names (19) and symbolic links (20). Only four of the vulnerabilities stemmed from cryptographic problems. Analyses of other vulnerability and incident reports reveal similar repetition – Wagner et. al., found that buffer overflow vulnerabilities account for approximately 50% of CERT advisories [19].

So why do developers keep making the same mistakes? Some may be put down to carelessness or lack or awareness of security concerns on the part of developers, others to legacy code, but even experienced security-aware developers make these mistakes. The problem is that although security vulnerabilities such as buffer overflows are well

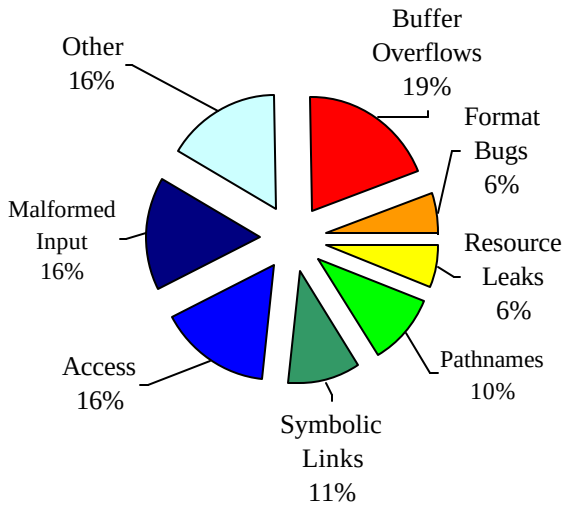

Figure 1. Entries in Common Vulnerabilities and Exposures, 1 Jan 2001 – 18 Sept 2001.

understood, the techniques for avoiding them are not codified into the development process. Even conscientious programmers can overlook security issues, especially when security issues rely on undocumented assumptions about procedures and data-types. Instead of relying on programmers' memories, we should strive to produce tools that codify what is known about common security vulnerabilities and can integrate it directly into the development process. This article describes Splint, a tool that represents a step towards those goals.

2. 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.*
>
> Microsoft's security program manager on the buffer overflow vulnerability in IIS that was exploited by the Code Red worm to acquire over 300,000 zombie machines to launch a distributed denial-of-service attack on the White House web site.

One way to deal with security vulnerabilities is suggested by the above quote – wait until the bugs are exploited by an attacker, produce a patch that you hope fixed the problem without introducing any new bugs, and then whine when system administrators don't install patches quickly enough. Not surprisingly, this approach has proven largely ineffective.

More promising approaches for reducing the damage caused by software flaws can be grouped into two categories – mitigate the damage flaws can cause or eliminate some of the flaws before the software is deployed. Techniques that limit the damage software flaws may cause include modifying program binaries to insert run-time checks or running applications in restricted environments that limit what they may do. Variations on this approach include the Janus [10] and Naccio [9]. In addition, several projects have developed safe libraries [1] and compiler modifications [5] 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 run-time damage-limitation approaches is that they require some performance overhead. A more serious weakness of damage-limitation approaches is that they do not eliminate the flaw but replace it with a denial-of-service vulnerability. It is usually not possible to recover from a detected problem without terminating the program. Hence, although damage-limitation techniques should be used in security-sensitive applications, they do not supplant techniques for eliminating the flaws.

Techniques for detecting and correcting software flaws include human code reviews, testing, and static analysis. Human code reviews are time-consuming and expensive, but can find the types of conceptual problems it would be impossible to find automatically. They are likely to miss, however, more mundane problems that even extraordinarily thorough people will overlook. Code reviews depend on the expertise of the humans involved, whereas automated techniques can benefit from expert knowledge codified in tools. Testing typically is not effective in finding security vulnerabilities. These vulnerabilities are revealed when attackers attempt to exploit weaknesses in the systems the designers did not think about; hence, they are not likely to be found through standard testing.

Static analysis techniques take a different approach. Rather than observe executions of the program, they analyze the source code directly. This enables them to make claims about all possible executions of a program instead of just the particular execution observed in a test case. From a security viewpoint, this is a significant advantage.

There is a wide range of static analysis techniques, offering a tradeoff between the

effort required to use them and the complexity of the analyses they are able to perform. Standard compilers perform type checking and other simple program analyses. This represents the low-effort end of the design spectrum. 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 nearly always too expensive and cumbersome to use on even security-critical programs.

Our approach is to 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 the use of heuristics to assist 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 – our goal is to produce a tool that produces useful results for real programs with a reasonable effort.

3. Splint

Splint[1] is a lightweight static analysis tool for ANSI C. It is designed to be as fast and easy to use as a compiler. It is able to do checking no compiler can do, however, by exploiting annotations added to libraries and programs that document assumptions and intents. Splint will check that source code is consistent with the properties implied by annotations.

3.1. Annotations

Annotations are denoted using stylized C comments identified by an `@` character following the `/*` comment marker. Annotations can be associated syntactically with function parameters and results, global variables and structure fields.

For example, the annotation `/*@notnull@*/` can be used in a pointer declaration syntactically like a type qualifier. In a parameter declaration, the notnull annotation documents an assumption that the value passed for this parameter is not `NULL`. Splint would report a warning for any call site where the actual parameter might be `NULL`. In checking the implementation of the function, Splint could assume that the initial value of the notnull-annotated parameter is not `NULL`. On a return value declaration, a notnull annotation would indicate that the function never returns `NULL`. Splint would report a warning for any return path that might return `NULL`, and check the call site assuming the function result is never `NULL`. In a global variable declaration, a notnull annotation would indicate that the value of the variable may not be `NULL` at an interface point – that is, it may be `NULL` within the body of a function, but may not be `NULL` at a call site or return point. Failure to handle possible `NULL` return values can be exploited in denial of service attacks, and is often not detected in normal testing.

Annotations can also document assumptions over the lifetime of an object. For example, the only annotation is used on a pointer reference to indicate that the reference is the sole long-lived (there may be temporary local aliases) reference to the storage it points to. An only annotation implies an obligation to release storage. This can be

[1] This paper describes Splint Version 3.0.1. The latest version is available as source code and binaries for several platforms under GPL from http://splint.cs.virginia.edu. Previous versions of Splint were known as LCLint. The name is an extraction of "SPecification Lint" and "Secure Programming Lint". The original Lint [11] was developed to overcome the lack of type checking for function calls in early versions of C.

done 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 transfers the obligation to some other reference. The library storage allocator `malloc` is annotated with only on its result, and the deallocator `free` takes an only parameter. Hence, one way to satisfy the obligation to release storage returned by `malloc`, is to pass it to `free`. Splint reports a warning for any code path on which the obligation to release storage is not satisfied since it would cause a memory leak. Memory leaks do not typically constitute a direct security threat, but they may be exploited to increase the effectiveness of a denial-of-service attack. Three of the CVE entries for the first half of 2001 involve memory leaks that can be exploited in denial of service attacks (CVE-2001-0017, CVE-2001-0041 and CVE-2001-0055). Not all storage management can be modeled with only references; sometimes programs need to share references across procedure and structure boundaries. Splint provides annotations for describing different storage management [8].

3.2. Analysis

There are both theoretical and practical limits on what can be analyzed statically. Precise analysis of most interesting properties of arbitrary C programs depends on several undecidable problems including reachability and determining possible aliases [15]. We could either limit our checking to issues that do not depend on solving undecidable problems (for example, type checking), or admit some imprecision in our results. Since our goal is to do as much useful checking as possible, we choose to allow checking that is both unsound and incomplete. This means Splint produces both false positives and false negatives. Warnings are intended to be as useful as possible to the programmer, but there is no guarantee that all messages indicate real bugs or that all bugs will be found. We make it easy for users to configure checking to suppress particular messages and weaken or strengthen checking assumptions.

Designers of static analyses face a tradeoff between precision and scalability. In order for our analysis to be fast and to scale to large programs certain compromises are made. The most important is to limit our analysis to dataflow within procedure bodies. Procedure calls are analyzed using information in annotations that describe preconditions and postconditions. Another compromise is between flow-sensitive (consider all program paths) and flow-insensitive (ignore control flow) analyses. Splint considers control flow paths, but to limit the blowup of analysis paths it merges possible paths at branch points. Loops are analyzed by using heuristics to recognize common idioms. This enables Splint to correctly determine the number of iterations and bounds of many loops without the need for loop invariants or abstract evaluation. The simplifying assumptions made by Splint are sometimes wrong; often this reveals convoluted code that is a challenge for both humans and automated tools to analyze. It is important to provide easy ways for programmers to customize checking behavior locally and suppress spurious warnings that result from imprecise analysis.

4. Buffer Overflows

As discussed in Section 1, buffer overflow vulnerabilities are perhaps the single most important security problem for the past decade. The simplest buffer overflow attack, *stack smashing*, overwrites a buffer on the stack to replace the return address. When the function returns, instead of jumping to the return address, control will jump to the address that was placed on the stack by the attacker. This gives the attacker the ability to execute arbitrary code. Buffer overflow attacks can also exploit buffers on the heap, but these are less common and harder to create. Splint detects both stack and heap-based buffer overflow vulnerabilities in the same way.

Programs written in C are particularly vulnerable to this type of attack. 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 has to enter an input larger than the size of the buffer and encode an attack program binary in that input.

4.1. Use Warnings

The simplest way to detect possible buffer overflows is to produce a warning whenever library functions susceptible to buffer overflow vulnerabilities are used. 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` may be used safely, but are often the source of buffer overflow vulnerabilities. Splint provides the annotation warn *flag-specifier message* that can be associated with a declaration to indicate that a warning should be produced whenever the declarator is used. For example, the Splint library declares `gets` with

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

to indicate that a warning message should be produced whenever `gets` is used if the bufferoverflowhigh flag is set.

Several security scanning tools provide similar functionality including Flawfinder [21], ITS4 [18], and the Rough Auditing Tool for Security (RATS) [16]. Unlike Splint, these tools use lexical analysis instead of parsing the code. This means they will report spurious warnings if the names of vulnerable functions are used in other ways (for example, as local user-defined functions). The main limitation of use warnings, however, 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 possibly dangerous function is safe or dangerous. To improve the results, we need a more precise specification of how a function may be safely used, and a more detailed analysis of program values.

4.2. Describing Functions

Consider the `strcpy (char *s1, char *s2)` function – it takes two `char *` parameters, and copies the string pointed to by the second parameter into the buffer pointed to by the first parameter. A call to `strcpy` overflows the buffer pointed to by the first parameter if it 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*] may be safely used as an lvalue (e.g., on the left side of an assignment expression). The value of maxRead (*b*) is the highest integer *i* such that *b*[*i*] may be safely used as an rvalue. The nullterminated annotation on the `s2` parameter indicates that it is a null-terminated character string. This implies that `s2[i]` must be a NUL character for some *i* <= maxRead `(s2)`.

At a call site, Splint will produce 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 reveal that the buffer allocated for `s` may be overrun by the call to `strcpy`.

4.3. Analyzing Program Values

Splint analyzes a function body and determines if the annotated preconditions are sufficient to ensure that the function is used correctly. Preconditions and postconditions are generated at the expression level in the

parse tree using internal rules or annotated descriptions in the case of function calls. 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 variables appearing in constraints also identify particular code locations. Since the value of a variable may change, it is important that the analysis can distinguish between values at different code points.

Preconditions are resolved using post-conditions from previous statements and any annotated preconditions for the function. If a generated precondition cannot be resolved at the beginning of a function, or a documented postcondition is not satisfied at the end, a descriptive warning about the unsatisfied condition is produced. Hence, Splint would produce a warning if it cannot determine that the value of `i` is between `0` and `MAXSIZE - 1`.

Constraints are propagated across statements using an axiomatic semantics. In addition, constraint-specific algebraic rules such as

$$\text{maxSet}(ptr + i) = \text{maxSet}(ptr) - i$$

are used to simplify constraints.

To handle loops, we use heuristics that recognize common loop forms [12]. Our experience indicates that a few heuristics can match a large number of loops in real programs. This allows us to effectively analyze many loops without needing loop invariants or expensive analyses.

5. 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 in terms of attributes associated with program objects or the global execution state. Unlike types, the values of these attributes may change along an execution path.

Splint provides a general language for defining attributes that may be associated with different kinds of program objects, and for defining rules that constrain the values of attributes at interface points and specify how attributes change. Because user-defined attribute checking is integrated with Splint's normal checking, it can take advantage of other analyses done by Splint such as alias and nullness analysis. The limited expressiveness of user attributes means that user-defined properties can be checked efficiently. Here, we illustrate the potential for user-defined checks to detect new vulnerabilities or application-specific constraints by showing how Splint can be extended to detect format bugs using a taintedness attribute. We have also used extensible checking to detect misuses of the files (e.g., failing to close a FILE, failing to reset a read/write file between certain operations), sockets, and incompatibilities between Unix and Win32 [2].

5.1. Taintedness

A new class of vulnerability was discovered in June 2000 known as a "format bug" [4]. 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 – 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 not known at compile time. Splint provides this checking – if the `+formatconst` flag is set, Splint will report a warning for any format strings that are not known at compile time. This will produce

spurious messages, however, since there may be format strings that are not known at compile time but 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, it came from the user or external environment). Perl's taint option [20] suggests a way of doing this. When it is used (by running Perl with the `-T` flag), Perl considers all user input to be *tainted*, and produces a run-time error (and halts execution) if a tainted value is used in an unsafe way. Untainted values can be derived from tainted input by using Perl's regular expression matching.

With Splint's extensible checking, we can detect dangerous operations with tainted values at compile time. We can define a *taintedness* attribute associated with `char *` objects. We introduce the annotations tainted and untainted to indicate assumptions about the taintedness of a reference. A similar approach was taken by Shankar, et. al., [17]. Instead of using attributes with explicit rules, they used type qualifiers. This enables them to take advantage of type theory, and in particular, to 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.

The complete attribute definition is shown in Figure 2. The first three lines define the taintedness attribute that is associated with `char *` objects and 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 indicates that a warning should be reported whenever an object with taintedness tainted is transferred to a reference declared as untainted. This would occur if a tainted object were passed as an untainted parameter or returned 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. This is used to determine that if a reference is tainted along one control path, and untainted along another control path, checking should assume that it is tainted after the two branches merge. It is also used to merge taintedness states in function specifications (as in the `strcat` example in the next section).

The annotations clause defines two annotations that can be used in declarations to document taintedness assumptions. In this case, the names of the annotations match the taintedness states. The final clause specifies default values that will be used for declarators without one of the taintedness annotations. The default values are chosen to make it easy to start checking an unannotated program. By assuming unannotated references are possibly tainted, Splint will produce a warning where these references are passed to functions that require untainted parameters. This indicates either a format bug in the code, or a place where an untainted annotation should be added to the code. Running Splint again after adding the annotation will propagate the newly documented assumption through the program.

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

Figure 2. Defining Taintedness.

5.2. Specifying Library Functions

For library code where source code is not available, we cannot rely on the default annotations since Splint would not detect inconsistencies without source code. We need to provide annotated declarations that document taintedness assumptions for standard library functions. This is done 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 the return value aliases the storage passed as `s`, so the result is also tainted (this information is also used by Splint's alias analysis.)

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

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

Since there are no annotations on the parameters, they are implicitly tainted according to the default rules, and it is acceptable to pass either untainted or tainted references as parameters to `strcat`. The ensures clause means that after `strcat` returns, the taintedness of the first parameter (and the result because of the returned annotation on `s1`) will be the result of merging the taintedness of the two parameters before the call. The merge is done using the rules in the attribute definition – hence if the actual parameter passed as `s1` is untainted and the parameter passed as `s2` is tainted, the result and first parameter will be tainted after `strcat` returns.

6. Experience

Using Splint is an iterative process. Running Splint produces warnings that lead to either changes in the code or annotations. Then, Splint is run again to check the changes and propagate the newly documented assumptions. This process continues until no warnings are produced. Since Splint checks approximately 1000 lines per second, the need to run Splint again is not burdensome.

Earlier versions of Splint have been used to detect a range of problems not specifically focused on security including data hiding [7]; memory leaks, uses of dead storage, and null dereferences [8] on programs comprising hundreds of thousands of lines of code. Splint is used by working programmers, especially in the open source development community [13, 14].

Our experience with the buffer overflow checking and extensible checking so far 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, libraries and tools that comprise the reference implementation of DNS.

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 bugs and known and unknown buffer overflows. It takes less than 4 seconds to check all of wu-ftpd on a 1.2GHz Athlon machine.

Format Bugs. Running Splint on wu-ftpd version 2.5.0 with only taintedness checking

turned on produces two warnings, the first one is:

```
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`, `vsnprintf` is declared with an untainted annotation on its format string parameter. The passed value, `fmt`, is a parameter to `vreply`, hence it is possibly tainted according to the default rules. We add an untainted annotation to the `fmt` parameter declaration of `vreply` to document the assumption that it must be passed an untainted value.

After adding the annotation, Splint reports three warnings for possibly tainted values passed to `vreply` in `reply` and `lreply`. This leads us to add three additional annotations. Running Splint again produces five warnings – three of which involve passing the global variable `globerr` as an untainted parameter. Adding an untainted annotation to the variable declaration directs Splint to check that `globerr` is never tainted at an interface point. The other warnings concern possibly tainted values passed to `lreply` in `site_exec`. Since these values are obtained from a remote user, this constitutes a serious vulnerability (CVE-2000-0573).

The second message produced by the first Splint execution reports a similar invalid transfer in `setproctitle`. After adding an annotation and re-running Splint, this leads us to two additional format string bugs in wu-ftpd. These vulnerabilities are described in CERT CA-2000-13 and can be easily fixed by using the `%s` constant format string.

We also ran Splint on wu-ftpd 2.6.1, a version that fixes the known format bugs. After adding eight untainted annotations, Splint runs without reporting any format bug vulnerabilities.[2]

Buffer Overflow Vulnerabilities. Running Splint on wu-ftpd 2.5 without adding annotations produces 166 warnings for potential out of bounds writes.

After adding 66 annotations using an iterative process like the one described for checking taintedness, we found 25 messages that indicated real problems. There were 76 messages considered spurious, summarized in Table 1. Six of these resulted from Splint being 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 may have open. This buffer is then written to using the integer value of the file descriptor of an open file stream as the index. This is safe because the value of the file descriptor 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 which were correct but which did not match

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

Table 1. False warnings checking wu-ftpd.

the loop heuristics. To some extent this could be addressed by incorporating

[2] Splint does find two legitimate buffer overflow vulnerabilities in wu-ftpd 2.6.1. They had already been corrected in the latest development version.

additional loop heuristics into Splint, however there will always be some unmatched loops. The remaining 60 spurious messages resulted from limitations of Splint's ability to reason about arithmetic, control flow and aliases. We are optimistic that many of these limitations can be overcome without unacceptably sacrificing the efficiency and usability goals using known techniques that have not yet been implemented in Split. It is impossible, though, to eliminate all spurious messages because of the general undecidability of static analysis.

An impediment to widespread adoption is the effort involved in annotating programs. Providing an annotated standard library solves part of the problem, but does not remove the need to add annotations where correct use of standard library functions depend on 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 run-time information with static analysis to automatically guess annotations show some promise [6].

7. Conclusion

The vast majority of security attacks exploit vulnerabilities in software that are well understood and can be eliminated. Lightweight static analysis is a promising technique for detecting likely vulnerabilities so they can be fixed before software is deployed, not patched after attackers have exploited them.

Although static analysis is an important approach to security, it is not a panacea. It does not take the place of run-time 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 these inconsistencies will reveal serious design flaws, but Split offers no general mechanisms for detecting high-level design flaws that could lead to security vulnerabilities.

No tool will eliminate all security risks – but lightweight static analysis should increasingly become part of the development process for security-sensitive applications. It is our 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. In future, newly discovered security vulnerabilities should not lead to just a patch to fix the problem in a particular program, 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 to a penetrate-patch-and-prevent model where once understood, a security vulnerability can be codified into tools that detect it automatically.

References

[1] Arash Baratloo, Navjot Singh and Timothy Tsai. *Transparent Run-Time Defense Against Stack-Smashing Attacks*. 9th USENIX Security Symposium, August 2000.

[2] Chris Barker. *Static Error Checking of C Applications Ported from UNIX to WIN32 Systems Using LCLint*. University of Virginia Senior Thesis, March 2001.

[3] Common Vulnerabilities and Exposures Version 20010918. Available from http://cve.mitre.org/.

[4] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. *FormatGuard: Automatic Protection From printf Format String Vulnerabilities*. 10th USENIX Security Symposium, August 2001.

[5] Crispin Cowan, Calton Pu, David Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang. *Automatic Detection and Prevention of Buffer-Overflow Attacks*. 7th USENIX Security Symposium, January 1998.

[6] Michael D. Ernst, Jake Cockrell, William G. Griswold and David Notkin. *Dynamically Discovering Likely Program Invariants to Support Program Evolution.* International Conference on Software Engineering. May 1999.

[7] David Evans, John Guttag, Jim Horning and Yang Meng Tan. *LCLint: A Tool for Using Specifications to Check Code*. SIGSOFT Symposium on the Foundations of Software Engineering. December 1994.

[8] David Evans. *Static Detection of Dynamic Memory Errors*. SIGPLAN Conference on Programming Language Design and Implementation. May 1996.

[9] David Evans and Andrew Twyman. *Flexible Policy-Directed Code Safety*. IEEE Symposium on Security and Privacy. May 1999.

[10] Ian Goldberg, David Wagner, Randi Thomas and Eric A. Brewer. *A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker*. 6th USENIX Security Symposium. July 1996.

[11] S. C. Johnson. *Lint, a C Program Checker*. Computer Science Technical Report, Bell Laboratories, Murray Hill, July 1978.

[12] David Larochelle and David Evans. *Statically Detecting Likely Buffer Overflow Vulnerabilities.* 10th USENIX Security Symposium, August 2001.

[13] David Santo Orcero. *The Code Analyzer LCLint.* Linux Journal. May 2000.

[14] Pramode C E and Gopakumar C E. *Static Checking of C programs with LCLint.* Linux Gazette Issue 51. March 2000.

[15] Ramalingam G. *The Undecidability of Aliasing*. ACM Transactions on Programming Languages and Systems, September 1994.

[16] Secure Software Solutions. *Secure Software Announces Initial Release of RATS*. http://www.securesw.com/rats/. May 2001.

[17] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. *Detecting Format String Vulnerabilities with Type Qualifiers.* 10th USENIX Security Symposium, August 2001.

[18] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. *ITS4 : A Static Vulnerability Scanner for C and C++ Code*. Annual Computer Security Applications Conference. December 2000.

[19] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. *A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities*. Network and Distributed System Security Symposium. February 2000.

[20] Larry Wall, Tom Christiansen, Jon Orwant. *Programming Perl* (3rd Edition). O'Reilly, July 2000.

[21] David Wheeler. Flawfinder Home Page. http://www.dwheeler.com/flawfinder/. 2001.

Contact Information:

David Evans
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
(434) 982-2218

David Larochelle
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
(434) 982-2291

Biographical Sketches:

David Evans

David Evans is an assistant professor at the University of Virginia Department of Computer Science. He has BS, MS and PhD degrees from the Massachusetts Institute of Technology. His research interests include annotation-assisted static checking and programming swarms of computing devices.

David Larochelle

David Larochelle is a PhD student at the University of Virginia Department of Computer Science working on lightweight static analysis with a focus on security. He has a BS in Computer Science from William and Mary.