

Improving Computer Security using Extended Static Checking

Brian V. Chess
Computer Engineering
University of California, Santa Cruz

Abstract

We describe a method for finding security flaws in source code by way of static analysis. The method is notable because it allows a user to specify a wide range of security properties while also leveraging a set of pre-defined common flaws. It works by using an automated theorem prover to analyze verification conditions generated from C source code and a set of specifications that define security properties. We demonstrate that the method can be used to identify real vulnerabilities in real programs.

1 Introduction

A large percentage of the documented vulnerabilities in computer systems have been introduced by programmers as they created or modified the source code for various system components [17]. A majority of these flaws fall into one of only a few major categories, with array bounds errors and race conditions being among the most common.

The primary contribution of this paper is to demonstrate that many common types of security flaws can be detected through the application of Extended Static Checking [7], a technique developed for the general problem of finding errors in source code at compile-time. We have created a prototype checker, named Eau Claire, capable of analyzing source code written in C. (Previous extended static checkers have analyzed programs written in Modula-3 [7] and Java [20].) In order to evaluate the source code, Eau Claire reads specifications that define security properties. Eau Claire represents the first application of Extended Static Checking to the specific problem of identifying security vulnerabilities.

Eau Claire is flexible enough that an experienced user

can specify a wide variety of properties. At the same time, a novice user could find a number of common security flaws by making use of a library of pre-written specifications. Specifications are particularly powerful when they are re-used. Information recorded in the form of specifications during a program inspection can greatly benefit future inspections.

Eau Claire analyzes programs written in C because it remains the most popular language for writing system software. Eau Claire handles C programs involving integral and aggregate types, arrays, pointers, pointer arithmetic, most control structures, and function calls. It can analyze modular programs: not all functions need to be defined in the source code being analyzed. Among the things it does not model are function pointers and bitwise operators. The presence of these constructs does not force Eau Claire to give up on a program as a whole. Instead, it gives warnings about functions that contain unmodelled constructs and suggests that the analysis of these functions may be compromised.

Eau Claire works by translating a program's source code into a series of verification conditions and presenting the verification conditions to an automatic theorem prover. Although it makes use of a theorem prover, Eau Claire is not a program verifier. The purpose of an extended static checker is only to look for certain types of errors, not to prove that a program is correct. Setting the goal to be less than a full proof of correctness is very liberating; by not promising to find all errors, the creators of a checker are free to look for a good balance between the number and importance of the errors that are found and the burden of running the checker. This approach puts practicality first. While not calling into question the undecidable nature of the underlying problems, we present evidence that it is possible to automate the process of finding many types of security flaws. Part of being practical is being willing to give up soundness.

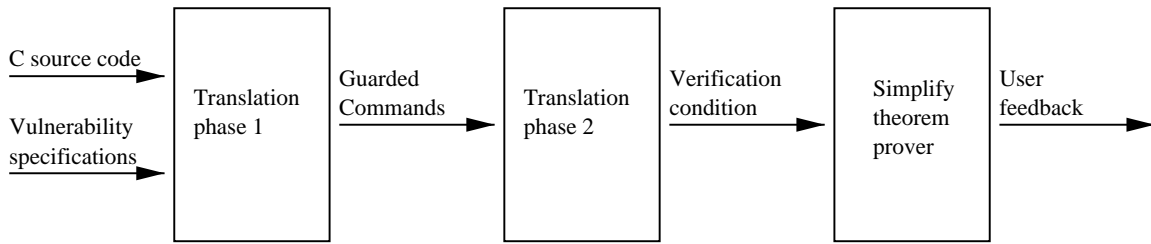


Figure 1. The checking process implemented by Eau Claire.

By not requiring a perfect answer in all cases, Eau Claire can produce a useful answer in most cases.

Since Eau Claire creates an independent verification condition for each function in the target program, its execution time is the sum of the times required to process each function. Empirical evidence suggests that on average Eau Claire is 25 times slower than the compiler gcc. That makes it too slow to be run with every compile as part of the typical development process but plenty fast enough to be a standard part of the release process. We expect that most users will begin by using pre-written specifications for C standard library functions and then begin to write their own specifications as they gain experience.

Section 2 dives into the details of the checking process that Eau Claire implements, and Section 3 describes the results of applying Eau Claire to two programs with known security vulnerabilities. In one case, Eau Claire shows that a vulnerability that was supposed to have been fixed actually still exists (a previously unknown result). Section 4 provides a context for this work by describing other recent efforts near the intersection of security and static checking.

2 Method

Figure 1 gives a top-level view of how Eau Claire works. It first translates a C function and the relevant specifications into a variation of Dijkstra’s Guarded Commands [9]. It then translates the Guarded Commands into a verification condition. Finally, it presents the verification condition to the automatic theorem prover Simplify [8, 28]. If Simplify refutes the theorem, then the associated function is in violation of one or more of the specifications. The verification conditions are generated in such a way that the counterexample provided by Simplify contains enough information that the user can track down the source of the mismatch.

The first step, the translation of C into Guarded Commands, requires a concrete interpretation of the sometimes vague semantics of the C language. It is here that Eau Claire gives up soundness in favor of ease of use. The second step, the translation of Guarded Commands into a verification condition, does not suffer thusly because the semantics of the Guarded Command language are well defined.

Sections 2.1 and 2.2 introduce Eau Claire’s Guarded Command language and specification language. Sections 2.3 and 2.4 discuss the translation of C into Guarded Commands and the translation of Guarded Commands into a verification condition.

2.1 Guarded Commands

Dijkstra created the Guarded Command programming language so that he could describe non-trivial algorithms and analyze them in a formal manner. Guarded commands are a useful midway point between a C function and a verification condition because they retain the sequential and imperative nature of C, but their semantics are defined rigorously, making it possible to construct a theorem from Guarded Commands in a straightforward and sound manner. Eau Claire’s guarded command language contains several concepts not found in Dijkstra’s original language including exceptions [21], assert and assume statements [1], and the concept of the erroneous state [19]. It also omits several major constructs that were part of the original language including restriction, conditionals, looping, and the guards that gave the language its name. Since loops are not available, the following discussion assumes that all commands terminate; there is no notion of an infinite loop.

The semantics of a command are given by its *weakest precondition*. When a machine halts after executing a guarded command the machine’s final state can be characterized as normal, exceptional, or erroneous. The

Command Example	Description	Weakest Precondition
Skip skip	Halt in a normal state, otherwise, do not modify the state of the machine. In order for P to evaluate to <i>true</i> after the execution of skip, it must evaluate to <i>true</i> before the execution of skip.	$\text{wp}(\text{skip}, P, Q) \equiv P$
Raise raise	Halt in an exceptional state, otherwise, do not modify the state of the machine. In order for Q to evaluate to <i>true</i> after the execution of raise, it must evaluate to <i>true</i> before the execution of raise.	$\text{wp}(\text{raise}, P, Q) \equiv Q$
Assignment $x := v$	Set the variable x to the value v . The notation $P(x : v)$ represents the predicate P with all occurrences of x replaced by v .	$\text{wp}(x := v, P, Q) \equiv P(x : v)$
Composition $A ; B$	Execute the command A , then execute the command B provided that A halted in a normal state.	$\text{wp}(A ; B, P, Q) \equiv \text{wp}(A, \text{wp}(B, P, Q), Q)$
Exception handling $A ! B$	Execute the command A , then execute the command B provided that A halted in an exceptional state.	$\text{wp}(A ! B, P, Q) \equiv \text{wp}(A, P, \text{wp}(B, P, Q))$
Alternation $A \square B$	Execute either command A or command B , selecting between the two in a nondeterministic manner.	$\text{wp}(A \square B) \equiv \text{wp}(A, P, Q) \wedge \text{wp}(B, P, Q)$
Assume assume(R)	If the predicate R evaluates to <i>true</i> , then halt in a normal state. If R does not evaluate to <i>true</i> , then the statement cannot be executed.	$\text{wp}(\text{assume}(R), P, Q) \equiv R \Rightarrow P$
Assert assert(R)	If the predicate R evaluates to <i>true</i> , halt in a normal state. Otherwise halt in an erroneous state (go wrong).	$\text{wp}(\text{assert}(R), P, Q) \equiv P \wedge R$

Table 1. Eau Claire's Guarded Command language.

command's weakest precondition describes the required state of the machine before the execution of the command that is necessary to achieve a desired final state. More formally, consider a command A and predicates P , Q , and W that relate to the state of the machine executing A . The weakest precondition for the command, $\text{wp}(A, P, Q, W)$, is a predicate describing the state of the machine prior to the execution of A so that P , Q , or W will evaluate to *true* subsequent to the execution of A . In particular, P will evaluate to *true* if the machine halts in a normal state, Q will evaluate to *true* if the machine halts in an exceptional state, and W will evaluate to *true* if the machine halts in an erroneous state. Table 1 gives the commands that comprise Eau Claire's Guarded Command language.

Halting in an erroneous state is also known as *going wrong*. Eau Claire uses going wrong to represent a specification violation. Since the purpose of the theorem being generated is to argue that no specification violation exists, the fourth argument to the wp function, W , will always be *false* (satisfied by no machine state). We will therefore omit it from the remaining formulas in the in-

terest of clarity.

In many languages (including C) a semicolon denotes the end of a statement, and the body of a function consists of a sequence of statements. In Guarded Commands, the semicolon represents a statement that is used to sequentially compose two other statements. The language does not need a conditional construct; a combination of alternation and assume statements can be used instead. For example, the compound C statement

```
if (x > y)
    z = 0;
else
    z = 1;
```

can be represented as

```
( assume(x > y); z := 0 ) □
( assume(¬(x > y)); z := 1 )
```

The nondeterminism of the alternation statement is tamed by fact that an assume statement cannot be executed if its predicate does not evaluate to *true*. The machine is forced to choose a path that it can execute.

```

spec (function name)(formal argument names)
{
  requires (precondition expression): "(violation message)"
  modifies (list of variables)
  ensures (postcondition expression): "(violation message)"
}

```

Figure 2. Eau Claire’s function specification syntax

2.2 Specifications

Eau Claire’s specification language is modeled after ESC/Modula3’s specification language [6]. At its heart is the concept of *procedural abstraction*, the idea that a set of operations can be grouped, named, and invoked as though they were a single operation. In order for procedural abstraction to be useful, there must be a contract between a procedure’s caller and the procedure’s implementation. The contract tells the caller what it must provide to the procedure and what to expect from the procedure in return. The contract tells the procedure what it should expect from the caller and what it must provide in return. Most programming languages enforce some aspects of this contract at compile time. For example, ISO C requires that when a function is declared with a prototype, the formal and actual function arguments must agree in both number and type [13].

Eau Claire’s function specification language allows the contract between callers and implementations to be further elaborated. As illustrated in Figure 2, a function specification may contain zero or more of the following:

Requires: *requires* R . The function precondition. Eau Claire checks that all callers ensure that the expression R evaluates to *true* before the function is invoked. If a caller does not guarantee that the precondition is met, Eau Claire will include the associated violation message in its output along with the textual location of the call. When it checks the specified function, Eau Claire assumes that R evaluates to *true*.

Modifies: *modifies* V . The list of variables that may be modified by the implementation. Eau Claire accounts for the side effects that a function call might have upon the environment of the caller by assuming that all variables on the list V will be modified in accordance with the function’s postcondition. When checking the specified function’s im-

plementation, Eau Claire would ideally check to make sure that an implementation did not alter any state not on the list, but it does not do so currently.

Ensures: *ensures* P . The function postcondition. Eau Claire checks that the implementation causes the expression P to evaluate to *true* when the function returns. If there are cases in which the postcondition may not be met, Eau Claire will include the associated violation message in its output. When evaluating callers, Eau Claire assumes that P evaluates to *true* when the function returns.

In order to make function specifications easier to read, a function may have multiple *requires* entries, in which case the function’s precondition is equivalent to the conjunction of the precondition expressions. The same is true for *ensures* and the function’s postcondition. If a specification has multiple *modifies* entries, the list of variables that may be modified is the union of the variable lists.

As much as possible, the semantics of specification expressions are the same as the semantics of C expressions. A C expression is considered *true* if it evaluates to an integer other than zero. A C expression is considered *false* if it evaluates to zero. Specification expressions may contain variables, logical operators, comparison operators, arithmetic operators, pointer indirection, structure selection, and array selection. Specification expressions may not contain function calls or operators that can change the value of a variable. Assignment is not permitted, for example. Eau Claire also supports a number of special constructs in specification expressions that do not exist in C, as described in Table 2.

In addition to function specifications, Eau Claire allows for the declaration of *specification variables*. Specification variables are not concrete: they do not affect a program’s compilation or execution, and therefore they cannot appear outside of specifications. Specification variables are useful for reasoning about program prop-

Construct	Description
<code>\$final</code>	In many cases it is useful to refer to both the initial value and the final value of a variable in a postcondition. In order to remain consistent with precondition expressions, an unadorned variable in a postcondition refers to the variable in its initial state. The expression <code>\$final(v)</code> refers to the variable <code>v</code> in its final state.
<code>\$return</code>	In a postcondition the expression <code>\$return</code> represents the return value of the function.
<code>\$valid</code>	The expression <code>\$valid(v)</code> evaluates to 1 if the pointer or array variable <code>v</code> is non-NULL and points to allocated storage. It evaluates to 0 otherwise.
<code>\$length</code>	The expression <code>\$length(v)</code> is equal to the number of allocated elements pointed to by the array or pointer variable <code>v</code> .
<code>\$string</code>	The expression <code>\$string(s)</code> evaluates to 1 if <code>\$valid(s)</code> evaluates to 1 and if a NULL terminator occupies at least one of the allocated locations pointed to by <code>s</code> . It evaluates to 0 otherwise.
<code>\$string_length</code>	The expression <code>\$string_length(s)</code> is equal to the least index of the pointer or array variable <code>s</code> occupied by the value 0.
<code>\$static</code>	An expression of the form <code>\$static(s)</code> evaluates to <i>true</i> only if <code>s</code> is a pointer to a statically allocated object such as a string constant.
<code>exists</code>	An expression of the form <code>exists (variable list) (expression)</code> evaluates to <i>true</i> if the inner expression is non-zero for at least one combination of variable values.
<code>forall</code>	An expression of the form <code>forall (variable list) (expression)</code> evaluates to <i>true</i> only if the inner expression is non-zero for all combinations of variable values.

Table 2. In addition to standard C expressions, Eau Claire provides special constructs for use in specifications.

erties that may not be explicitly represented by the program’s concrete variables.

2.3 Translating C into Guarded Commands

At first blush, translating C into Guarded Commands might appear to be challenging because C includes function calls, data structures, pointers, and side effects, none of which can be represented directly in Eau Claire’s Guarded Command language. While it is true that these topics present challenges, the process of translating C into Guarded Commands is further complicated by ambiguities regarding the definition of the C language [13, 24].

First, there is no single authoritative definition of C. Major C dialects include standard C (also known as ISO C or ANSI C), traditional C (sometimes called K&R C), and clean C (a subset of C++). Naturally, implementations vary in the degree to which they adhere to one standard or another.

Second, all definitions of C require an implementation to prescribe its own meaning for certain constructs.

For example, it is the responsibility of the implementation to define the range of values that can be represented by the integral types.

Third, the behavior of some constructs is not fully specified—the language definition does not always impose a requirement on the implementation. For example, the order of evaluation of expressions is largely unconstrained. An implementation could evaluate the expression `(x * y * z)` by first evaluating `z`, then `x`, then `y`, and then multiplying.

Finally, the behavior resulting from some operations is completely undefined. For example, accessing uninitialized memory could result in the termination of the program or it could result in no ill-effects whatsoever. The same is true of dereferencing a null pointer or indexing an array outside of its bounds.

The Extended Static Checking philosophy allows Eau Claire to give up on finding all of the potential errors in a program and instead concentrate on finding the most likely and most important errors. Letting go of soundness allows us to make the important assumption that the programmer is not intentionally trying to hide vul-

nerabilities from the tool. For that reason, we treat all translation dilemmas in the same manner: Eau Claire interprets C programs as though they were written by a competent programmer who is interested in creating a correct and portable program. Table 3 lists C language constructs and describes how Eau Claire translates them.

The purpose of Eau Claire is not to determine how closely the program in question adheres to any particular standard (other than the specified security properties) or to measure the program’s semantic correctness. For this reason, Eau Claire assumes that undefined operations do not occur. This assumption limits the types of security flaws that Eau Claire can detect. Imagine a function that modifies a variable more than once between successive sequence points. Under the definition of ISO C, the function is performing an illegal operation. An ISO C compiler would be within its rights to cause the variable take on an arbitrary value. Eau Claire will not consider this possibility, instead assuming that the modifications will take place in the order in which they appear. Missing this type of only remotely feasible flaw is a good trade-off in exchange for not tormenting the user with a list of every possible location of an undefined operation.

2.4 Translating Guarded Commands into a verification condition

After Eau Claire has translated a function into a guarded command, it augments the command using the function’s specification. Given the function F with guarded command translation $gc(F)$ and requires and ensures conditions R and P respectively, the augmented guarded command for the function is

```
assume( $R$ ) ;  
gc( $F$ ) ;  
assert( $P$ )
```

In addition to the requires expressions provided by the user (if any), Eau Claire adds to R an expression declaring that all of the fields for each type of structure or union used in the function be unique. A similar expression declares that all data types used in the function are unique. This is necessary because Eau Claire uses arrays to represent structure fields and pointer dereferencing. Without them, the theorem prover would be free to assume that any two structure field names referred to the same field within the structure or that any two data type names were equivalent.

A correct but impractical verification condition could be extracted from the augmented guarded command simply by computing its weakest precondition according to the equations given in Section 2.1, but Flanagan and Saxe point out that doing so can result in a formula that is exponentially larger than the Guarded Command program that it represents [11]. This would disallow the checking of large C functions. Instead Eau Claire implements the two-stage generation algorithm that Flanagan and Saxe recommend, which produces a formula that is in the worst case quadratic in the size of the Guarded Command program it represents.

3 Experimental Results

This section presents two real-world security vulnerabilities and shows how Eau Claire can be used to find them. The examples illustrate some of the errors that Eau Claire is able to detect, but Eau Claire is by no means limited to these types of vulnerabilities.

3.1 RSAREF buffer overflow

In 1994, RSA Data Security released RSAREF2, a reference implementation of the RSA and DES encryption algorithms, the MD2 and MD5 message digest algorithms, and a number of other widely used cryptographic tools. Because RSAREF2 is free for non-commercial purposes, it was quickly adopted for use in a number of popular Internet applications including PGP (a set of programs for securing email messages, files, and network connections), SSH (a secure remote login program), and Apache mod_ssl (a module that enables the Apache web server to communicate using the Secure Socket Layer (SSL) protocol).

In 1999, Solino and Richarte discovered buffer overflow vulnerabilities in RSAREF2 [3]. The vulnerabilities made it possible for an attacker to overwrite the call stack and thereby execute arbitrary code by exploiting RSAREF2’s RSA implementation. Specifically, an overflow can occur when the library attempts to encrypt or decrypt blocks that are larger than the maximum block size compiled into the program. The buffer overflow happens inside the standard library function `memcpy`.

With a specification for `memcpy` (given in Figure 3) that requires the destination to be large enough to hold the number of bytes being copied, Eau Claire flags 3 of the 20 calls to `memcpy` as potential problems. Two of

Construct	Description
Expressions	For the purpose of determining side-effects, Eau Claire assumes that expressions are evaluated from left to right. It supports arithmetic and logical operator expressions, negation, and conditional expressions.
Operations	Eau Claire supports the integral operations addition, subtraction, multiplication, and division. It does not support bitwise manipulation, and it assumes that arithmetic overflow does not occur.
Type sizes	Eau Claire allows the user to define the number of bytes used to represent the integral types and the number of bytes used to represent pointers.
Floating point types	Eau Claire does not model floating point values or operations. Converting a floating point number into an integral value results in the integral value being unknown.
Arrays	Eau Claire models an array as a single variable rather than as a collection of dynamically named variables. Eau Claire thereby assumes that arrays do not overlap.
Pointers	In order to account for aliasing, Eau Claire models a pointer dereference operation as an index into the pointer's type [23]. In other words, dereferencing a pointer p of type T is modeled as $T[p]$. For the purpose of evaluating Boolean expressions, Eau Claire assumes that NULL is equal to zero.
Structures and Unions	Eau Claire treats structure fields as arrays indexed by the structure variable. In other words, Eau Claire models $s.f$ as $f[s]$. This perhaps runs counter to the "conventional wisdom" that would model $s.f$ as $s[f]$, but doing so would be inconsistent with Eau Claire's treatment of pointers.
Conditionals	Eau Claire supports <code>if</code> statements and <code>switch</code> statements.
Loops	Eau Claire supports <code>do</code> , <code>while</code> , and <code>for</code> loops, but by default it assumes that the loop body is executed no more than once. The user can supply a maximum loop depth, and Eau Claire will unroll loops to the specified level.
Flow control	In Eau Claire's Guarded Command language <code>return</code> , <code>break</code> , and <code>continue</code> statements are modeled by raising and catching exceptions. A variable associated with the exception keeps track of the exception's type so that it can be treated appropriately when it is caught.
Function calls	Eau Claire replaces function calls with an assertion that the function's precondition is met and an assumption that its postcondition holds. Eau Claire does not associate specifications with function pointers. If a function does not have a specification, it is assumed to have no side-effects, and its return value is unknown.
References	Eau Claire understands variable references only as an indication that a function call argument is a variable parameter. In other contexts, a reference operation produces an unknown value. This is not an inherent limitation of the method but simply an implementation shortcut.

Table 3. A list of C language constructs, each with a description of Eau Claire's interpretation.

```
spec memcpy(dest, src, n)
{
    requires $length(dest) >= n
}
```

Figure 3. A specification for the function `memcpy`. The destination must be large enough to hold the number of bytes to be copied.

```
spec R_GenerateBytes(buffer,
    bufferLength, randomSource)
{
    requires $length(buffer) ==
        bufferLength
}
```

Figure 4. A specification for the function `R_GenerateBytes`. The length of the buffer must be equal to the `bufferLength` parameter.

these calls are the buffer overflow vulnerabilities identified by Solino and Gerardo. The third occurs in a function named `R_GenerateBytes`, the purpose of which is to fill a buffer with randomly generated data. Eau Claire flags it because it cannot guarantee that the function's buffer length parameter is truly the length of the buffer to be filled. With a specification (given in Figure 4) making that a requirement, Eau Claire no longer flags `R_GenerateBytes` as the source of a potential error. Eau Claire also verifies that the five calls to `R_GenerateBytes` in `RSAREF2` meet the given specification.

In this example Eau Claire's precision and accuracy are both excellent. Initially Eau Claire produced no false negatives and a single false positive. The false positive could have been identified as such by inspection, but with the addition a single specification, Eau Claire was able to eliminate it. In terms of execution time and memory usage, Eau Claire's performance was good but probably too slow for a programmer to run it with every compile during development. `RSAREF2` consists of 79 functions in 13 source files. Including its 11 header files, it is 4728 lines long. Eau Claire took 33 seconds to process `RSAREF2` running on a 550 MHz Pentium III

workstation. Running on the same computer, gcc took two seconds to compile `RSAREF2`. Memory usage for Eau Claire never exceeded ten megabytes.

3.2 Redhat lpr race condition

The program traditionally used to access to the print queue in UNIX systems, `lpr`, has long been a source of security problems. Because the printer is a hardware resource, the programs that control access to it must have root privileges. Of course, most users of the system need access to the printer but do not have root privileges. As is often the case, writing a program that allows only limited access to root resources has proven problematic.

A security hole in the version of `lpr` distributed with Redhat Linux versions 4.1 through 6.1 was the result of a race condition involving the way file permissions were checked [25]. As Figure 5 shows, the program first checked to see if the user had permission to read the file, then (acting as root) opened the file. If the user could substitute a legitimate file with a file they did not have permission to read (probably by changing a soft link) between the time that `lpr` checked the file permissions and the time it opened the file, the user could print files that they could not otherwise read.

Redhat applied a common technique for eliminating a race condition: the revised `lpr` now takes on the user's identity before opening the file, as shown in Figure 6. Eau Claire can identify programs where this fix has not been properly implemented. Figure 7 gives specifications that allow Eau Claire to report an error if `open` could be called without first setting the effective user id.

Not surprisingly, Eau Claire reports that the call to `open` in Figure 5 is the location of a potential problem. Of more interest is the fact that Eau Claire reports that the revised `lpr` in Figure 6 has the same potential problem: the program does not check the value returned by `seteuid` or `setegid`, so if either of the calls fail, the call to `open` will be carried out in the same insecure way as it was originally. It may seem far-fetched that `seteuid` or `setegid` would fail, but less than a year after the `lpr` vulnerability was found, a bug in the Linux kernel came to light that would allow a user to cause just such a failure [4]. This flaw in `lpr` was previously undiscovered.

It would be perfectly feasible for Eau Claire to use specification variables to keep track of the current setting of the effective user and group ID values. If it did, it would have the potential to catch errors that would

```

int fd;
for (int i=1; i < argc; i++)
{
    /* first make sure that the
       user can read the file,
       then open it */
    if (!access(argv[i], O_RDONLY))
    {
        fd = open(argv[i], O_RDONLY);
        print(fd);
    }
}

```

Figure 5. A pseudocode demonstration of the race condition in Redhat's lpr. If the user could switch the file between the time lpr checked the file permissions and the time it opened the file, the user could print a file that they could not normally read.

```

int fd;
for (int i=1; i < argc; i++)
{
    int uid = getuid();
    int gid = getgid();
    int original_euid = geteuid();
    int original_egid = getegid();

    /* set the effective user id to be that of the current
       user before opening the file */
    seteuid(uid);
    setegid(gid);

    fd = open(argv[i], O_RDONLY);

    /* reset the effective user id to it's original value */
    seteuid(original_euid);
    setegid(original_egid);

    print(fd);
}

```

Figure 6. A pseudocode version of the fix Redhat made to lpr. Now the program takes on the user's privileges before opening the file. The vulnerability is more limited now, but it is still present because the return value of the seteuid and setegid calls are not checked.

```

/* specification variables for tracking whether or not the
   effective user id and effective group id have been set */
/* spec var $euid_set */
/* spec var $egid_set */

/* seteuid returns 0 after changing the current euid to
   the requested value or returns a non-zero error
   code if the request cannot be fulfilled. */
/*
spec seteuid(euid)
{
  modifies $euid_set
  ensures ($final($euid_set) == 1) ||
          ($return != 0)
}
*/
int seteuid(int egid);

/* setegid returns 0 after changing the current egid to
   the requested value or returns a non-zero error
   code if the request cannot be fulfilled. */
/*
spec setegid(egid)
{
  modifies $egid_set
  ensures ($final($egid_set) == 1) ||
          ($return != 0)
}
*/
int setegid(int egid);

/*
spec open(filename, flags, ...)
{
  requires $euid_set:
    "euid has not been explicitly set"
}
*/
int open(char* filename, int flags, ...);

```

Figure 7. Specifications for checking lpr. Specification variables track whether or not the effective user and group IDs have been set. A race condition may exist if `open` is called without first changing the effective IDs.

cause the ID values to be set incorrectly. The actual implementation of `lpr` makes this approach less appealing because `open` is used to access a number of control files in addition to the file being printed for the user. When `lpr` opens the control files, it needs to have root permissions. That fact would complicate the specification for the `open` function; its `requires` clause would now have to take the name of the file being opened into consideration. While that too is perfectly feasible, it is application specific, whereas the specifications given in Figure 7 could be part of the standard library specifications applied to privileged utility applications.

The original `lpr` implementation occupies one source file and two header files. It is 878 lines long and consists of 10 functions. Eau Claire flags three of these functions because they make calls to `open` and there are no calls to `seteuid` or `setegid`. Eau Claire flags the same three functions in the revised version of `lpr` because the program disregards the return values from the new calls to `seteuid` and `setegid`.

In order to correct the flaws found by Eau Claire, we modified the 12 occurrences of the two statements

```
seteuid(uid);
setegid(gid);
```

to be

```
if (seteuid(uid))
    return;
if (setegid(gid))
    return;
```

The correct fix for these flaws would provide better error handling, but in this case a `return` statement is enough to avoid calling `open` after an attempt to set an ID has failed. Eau Claire only flags one call to `open` in the modified program. After adding a specification to the function that requires the IDs be set prior to the function's invocation, Eau Claire reports no errors.

In this case too Eau Claire reported no false negatives and only a single false positive. Once again the false positive was fairly easy to identify by inspection, but the addition of a single specification was all that Eau Claire needed in order to eliminate it. Although `lpr` only took 0.5 seconds to compile using `gcc`, Eau Claire took 50 seconds to analyze it, with a maximum memory usage of eight megabytes. The great majority of that time (43 seconds) was spent on a single function: `main`, which is 277 lines long. Although enormous functions are not

the epitome of good programming practice, they are not uncommon, so it is worthwhile to note that Eau Claire is up to the task.

4 Related Work

Typical methods for finding software defects are often ill-suited to the problem of uncovering security vulnerabilities. Researchers have responded with security-specific techniques for ferreting out security-specific defects.

4.1 Lexical Analysis

The most common way for a race condition to lead to a security problem is called a Time Of Check To Time of Use (TOCCTOU) flaw [2]. In turn, the most common TOCCTOU flaw is related to file access: if a privileged program checks file access permissions by referencing a file's name and then later references the file's name again in order to operate on it, an attacker has an opportunity to change the underlying filesystem object between the time of the check and the time of use. The flaw presented in Section 3.2 is of just this type. Bishop and Dilger [2] built a lexical analysis tool specifically for the purpose of unearthing file access race conditions.

Viega et al. [29] point out that quite a few common security problems are easy to identify in source code. For example, the presence of a call to the C library function `gets` almost always indicates a security problem because it is difficult to prevent buffer overflow attacks with `gets`. Their source code analysis tool, ITS4, scans C and C++ programs for vulnerabilities that can be identified purely from the lexical structure of the program—ITS4 does not take into account interaction between procedures, variable values, or flow of control. Before it begins examining a program, ITS4 reads in the set of vulnerabilities it is targeting. The authors have compiled a library of vulnerabilities that cover likely buffer overflow candidates, race conditions, and calls to poorly written pseudo-random number routines.

4.2 Run-time Checking

Most operating systems prevent programs from performing grossly aberrant behavior (illegal memory access for example), and some programming languages and environments provide features for more advanced monitoring. Perl uses a data tainting model to insure

that user-supplied commands are not executed directly in UNIX setuid scripts [31]. Javascript uses a similar approach to protect a user's privacy by preventing user-specific data from being transmitted over the network [12]. Java provides a flexible mechanism for controlling the run-time security policy enforced by the Java virtual machine [15].

Some of C's inherent weaknesses can be overcome through the use of runtime checking too. Jones and Kelly [16] propose a scheme for adding object bounds checks at compile-time, but standard libraries that are not recompiled remain vulnerable to buffer overflow attacks, and the performance of checked programs may suffer. Similarly, program instrumentation approaches like the one taken by Purify [14] are usually impractical because of performance degradation and increased memory usage. Format string vulnerabilities, where an attacker makes use of a function with a `printf`-style format string argument to perform an attack with results much like a buffer overflow, can be thwarted at runtime with minimal computational overhead [5, 26].

Necula [22] proposes a combination of formal reasoning and run-time checking with his system for packaging proofs with executable programs. Before a program is executed, its proof can be checked in order to make sure that the code will maintain the system's safety and security policies. Because the topic of the proof is assembly code, the types of properties that can be easily proven are at a correspondingly low level. Necula suggests that his approach would be useful for determining whether or not to allow a piece of code to execute in the kernel's address space by requiring a proof that the program will maintain the consistency of the kernel's data structures. This would be efficient because proof checking is much faster than proof construction, and once a piece of code had been approved, no further effort to constrain its behavior would be necessary. In Necula's view, producing proofs should be part of the function of a compiler.

4.3 Static Checking

LCLint is a C program checker [10]. Without adding specifications to the program being checked, its ability to find errors is limited to the same realm as most lint programs. With programmer-supplied specifications, it is able to perform additional checks by applying compiler flow analysis techniques. It can find abstraction violations, unannounced modifications to global vari-

ables, and the possible use-before-initialization errors. While these are all common sources of problems in C programs, none of them are direct widespread causes of security flaws.

Larochelle and Evans have modified LCLint in order to statically detect buffer overflow vulnerabilities [18]. Their method is similar to the one taken here, but limited to reasoning about minimum and maximum array indices that may be read or written. Programmer-provided preconditions and postconditions combined with built-in specifications for standard libraries are used in combination with the program itself in order to generate a set of constraints. If the constraints cannot be resolved, then a buffer overflow flaw may be present. Unlike the approach taken here, a programmer cannot write additional specifications in order to use the modified version of LCLint to find new types of vulnerabilities.

Wagner et al. have developed a static checker that uses integer range analysis in order to determine whether or not a C program contains potential buffer overflow errors [30]. While capable of finding many errors that lexical analysis tools would miss, the checker is still somewhat imprecise: it ignores statement order, it cannot model interprocedural dependencies, and it ignores pointer aliasing.

Inspired by Perl's taint mode, Shankar et al. make use of type qualifiers in order to perform a taint analysis for the purpose of statically detecting format string vulnerabilities in C programs [27]. Their system requires a programmer to annotate a small number of variables as either tainted or untainted and then uses type inference rules (along with pre-annotated system libraries) to propagate the qualifiers. Once the qualifiers have been propagated, the system can detect format string vulnerabilities by performing type checking.

5 Conclusions

While it is inconceivable that any single approach will solve all computer security problems, significant classes of vulnerabilities are caused by programming errors that can be detected using Extended Static Checking.

One form of static checking, type checking, has long been embraced by programmers. Type checking has been successful because the rewards it provides far outweigh the cost of its use. Program verification technology has not fared so well, and perhaps the reason is that

the rewards do not appear to outweigh the cost. Conventional wisdom holds that the undecidable nature of most static analysis questions implies that program verification techniques are a mirage, that there is no reward to be had. On the other side of the equation, fully specifying the behavior of a program is a daunting task.

In this case the conventional wisdom is wrong. Extended Static Checking makes use of program verification techniques and is, as we have demonstrated, capable of detecting common types of security vulnerabilities. This is a hefty reward because security problems are difficult to detect by other means and the penalty for missing them can be severe. The cost of Extended Static Checking is variable. At the low end, a programmer can make use of pre-existing specification libraries to check for common flaws. The checker can be invoked much like a compiler, and the time required to perform the checking is typically a small multiple of a compiler's execution time. A user who is willing to invest effort in writing specifications can improve the precision and accuracy of the checker and also check for new types of flaws or flaws that are specific to their problem domain.

References

- [1] R. Back and J. von Wright. *Refinement Calculus: A systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.
- [2] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):132–152, Spring 1996.
- [3] RSAREF buffer overflow. The bugtraq mailing list: <http://www.securityfocus.com>. Vulnerability 843.
- [4] Linux capabilities vulnerability. The bugtraq mailing list: <http://www.securityfocus.com>. Vulnerability 1322.
- [5] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. Formatguard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the USENIX Security Symposium*, 2001.
- [6] D. Detlefs. An overview of the extended static checking system. In *The first Formal Methods in Software Practice workshop collocated with ISSTA 96*, 1995.
- [7] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, December 1998.
- [8] D. L. Detlefs, G. Nelson, and J. B. Saxe. Simplify: the ESC theorem prover. Unpublished manuscript, November 1996.
- [9] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.
- [10] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: a tool for using specifications to check code. In *Symposium on the foundations of software engineering*. SIGSOFT, December 1994.
- [11] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Symposium on the Principles of Programming Languages*. ACM, 2001.
- [12] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly & Associates, 1996.
- [13] S. P. Harbison and G. L. Steele Jr. *C, a Reference Manual*. Prentice-Hall, Englewood Cliffs, NJ, 1995.
- [14] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136, 1992.
- [15] The Java security home page. On the web as <http://java.sun.com/security/>.
- [16] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Third International Workshop on Automated Debugging*, 1997.
- [17] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer security flaws. *ACM Computing Surveys*, 26(3):211–254, September 1994.
- [18] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *2001 USENIX Security Symposium*, August 2001.
- [19] K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995.

- [20] K. R. M. Leino, J. B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. Technical Report 1999-02, Compaq Systems Research Center, May 1999.
- [21] M. S. Manasse and C. G. Nelson. Correct compilation of control structures. Technical report, AT&T Bell Laboratories, September 1984.
- [22] G. Necula. Proof-carrying code. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, 1997.
- [23] G. Nelson. Pointers are bad, records are bad, but record-pointers are good. Technical report, Xerox, Palo Alto Research Center, November 1982.
- [24] M. Norrish. *C Formalised in HOL*. PhD thesis, University of Cambridge, 1998.
- [25] File access problems in lpr/lpd. Redhat Security Advisory, October 1999. RHSA-1999:041-01.
- [26] T. J. Robbins. libformat. On the web as <http://box3n.gumbynet.org/fyre/software/libformat.pdf>.
- [27] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.
- [28] The Simplify home page, Compaq Systems Research Center. On the web as <http://research.compaq.com/SRC/esc/Simplify.html>.
- [29] J. Viega, J. T. Bloch, T. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Proceedings of the Annual Computer Security Applications Conference*. Applied Computer Security Associates, 2000.
- [30] D. Wagner, J. S. Foster, E. A. Brewer, and A Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and distributed system security symposium*, February 2000.
- [31] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl*. O'Reilly & Associates, 1996.