AUTOMATIC VULNERABILITY DETECTION USING STATIC SOURCE CODE ANALYSIS

by

ALEXANDER IVANOV SOTIROV

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2005

Submitted by Alexander Ivanov Sotirov in partial fulfullment of the requirements for the degree of Master of Science specializing in Computer Science

Accepted on bahalf of the Faculty of the Graduate School by the thesis commitee:

LIST OF SYMBOLS

$\wedge$	Meet function in a lattice

ACKNOWLEDGEMENTS

This work would have been impossible without the help of many others. Some, among them Joel Jones and Mike Rhiney, encouraged me to aspire to greater heights, while others' lives served as an example of how not to end up. The latter shall remain nameless.

I would like to dedicate this thesis to my brother, Vladimir Sotirov.

CONTENTS

LIST OF SYMBOLS iii

ACKNOWLEDGEMENTS iv

LIST OF TABLES vii

LIST OF FIGURES viii

ABSTRACT ix

1 Introduction 1

1.1 Overview 4

1.2 Organization 4

2 Software Vulnerabilities 5

2.1 Buffer Overflows 5

2.2 Format String Bugs 6

2.3 Integer Overflows 7

3 Static Analysis for Vulnerability Detection 10

3.1 Introduction 10

3.2 Approaches to static analysis 12

3.3 Static Analysis Tools 18

4 Compiler Technology 21

4.1 Overview 21

4.2 GCC Internals 27

5 Design 35

5.1 Overview 35

5.2 Classification of Software Vulnerabilities 36

5.3 Proposed Solution 43

5.4 Implementation 49

6 Results 53

6.1 Methodology 53

6.2 Results 59

6.3 Testing Competing Tools 62

7 Future Work 68

8 Conclusion 71

REFERENCES 73

APPENDICES 76

A Vulncheck Source Code 76

B Glibc Annotations 104

LIST OF TABLES

5.1 Potentially vulnerable functions and language constructs 45

6.1 Results of running Vulncheck in taint analysis mode 60

6.2 Results of running Vulncheck in value range propagation mode . . . 61

6.3 Results of running Vulncheck with taint analysis and value range propagation 61

6.4 Results of running Flawfinder in its default configuration 63

6.5 Results of running Flawfinder with minlevel = 3 64

6.6 Results of running Flawfinder with the -F option 64

6.7 Results of running SPLINT 65

6.8 Comparison between Vulncheck, Flawfinder and SPLINT 67

LIST OF FIGURES

4.1 Abstract Syntax Tree representation of C = A + B 22

4.2 Example of a C program and its corresponding SSA form 23

4.3 The effect of constant propagation on a C program 24

4.4 Diagram of the old GCC structure (from [18]) 29

4.5 Diagram of the new GCC structure (from [18]) 31

4.6 Example of a C program and its GIMPLE representation (from [19]) 33

5.1 A screenshot of running Vulncheck 52

ABSTRACT

We present a static source analysis technique for vulnerability detection in C programs. Our approach is based on a combination of *taint analysis*, a well known vulnerability detection method, and *value range propagation*, a technique previously used for compiler optimizations.

We examine a sample set of vulnerabilities and develop a vulnerability classification based on common source code patterns. We identify three common characteristics present in most software vulnerabilities: one, data is read from an untrusted source, two, untrusted data is insufficiently validated, and three, untrusted data is used in a potentially vulnerable function or a language construct. We develop a static source analysis that is able to identify execution paths with these three characteristics and report them as potential vulnerabilities.

We present an efficient implementation of our approach as an extension to the GNU C Compiler. We discuss the benefits of integrating a vulnerability detection system in a compiler. Finally, we present experimental results indicating a high level of accuracy of our technique.

Chapter 1

Introduction

Software vulnerabilities are one of the main causes of security incidents in computer systems. In 2004, the United States Computer Emergency Readiness Team released 27 security alerts, 25 of which reported a critical software vulnerability [31].

Software vulnerabilities arise from deficiencies in the design of computer programs or mistakes in their implementation. An example of a design flaw is the Solaris sadmin service, which allows any unprivileged user to forge their security credentials and execute arbitrary commands as root. The solution to this problem is to redesign the software and enforce the use of a stronger authentication mechanism. Vulnerabilities of this kind are harder to fix, but fortunately, they are rare. Most software vulnerabilities are a result of programming mistakes, in particular the misuse of unsafe and error-prone features of the C programming language, such as pointer arithmetic, lack of a native string type and lack of array bounds checking.

Though the causes of software vulnerabilities are not much different from the causes of software defects in general, their impact is a lot more severe. A user might be willing to save their work more often in case a program crashes, but there is little they can do to lessen the consequences of a security compromise. This makes the

problem of detecting existing vulnerabilities and preventing new ones very important for all software developers.

The oldest approach to finding software vulnerabilities is manual source code auditing. As the name suggests, this method involves reading the source code of a program and looking for security problems. It is similar to the code review process common in the software engineering discipline, but it places additional requirements on the auditor. In addition to familiarity with the software architecture and source code, he or she should have considerable computer security expertise to be able to identify vulnerable code. A comprehensive source code audit requires a lot of time and its success depends entirely on the skills of the auditor. Despite these shortcomings, many vulnerabilities have been found using this method and it is still considered one of the best approaches.

A second option is the so-called fuzz testing or fuzzing. This method works by feeding invalid or malformed data to a program and monitoring its responses. A segmentation fault or an unhandled exception while processing malformed input is a good indication of a vulnerability. The fuzzing data may be randomly generated, or specifically crafted to test corner cases. It can be fed to a program as command line parameters, network packets, input files, or any other way a program accepts user input. Fuzzing frameworks such as SPIKE [1] and Peach [6] allow for rapid development of new application- or protocol-specific testing tools that can then be used for vulnerability detection. The advantage of fuzzing is that it can be easily automated and used for regression testing during software development.

Runtime checking is a technique for preventing the exploitations of vulnerabilities, but it can be used for vulnerability detection, especially in combination with fuzz

testing. This method involves inserting additional checks into a program to ensure that its behavior conforms to a certain set of restrictions. For example, Mudflap [7] is a pointer use checking extension to the GCC compiler. It adds instrumentation code to potentially unsafe pointer operations in C programs and detects errors such as NULL pointer dereferencing, buffer overflows and memory leaks. ProPolice [9] is another GCC extension that uses instrumentation to protect application from stack smashing attacks. The Microsoft Visual C++ compiler includes a similar feature. Runtime checking can help uncover vulnerabilities during fuzz testing, even if they do not result in a system crash.

A disadvantage of both fuzz testing and runtime checking is that some vulnerabilities can be triggered only under a very specific set of circumstances, which might not arise during regular use of the program. For example, exploiting an application might require sending a long sequence of specifically crafted network packets to reach a vulnerable state. Such vulnerabilities can be found during a manual source code audit, but the process is often too time-consuming. Static source analysis tools greatly increase the efficiency of source code auditing. Tools like Flawfinder [35] and SPLINT [10] examine the source code of a program and report possible vulnerabilities. Due to limitations inherent in source code analysis, these tools produce both false positives (reporting safe code as vulnerable) and false negatives (missing vulnerable code). Despite these limitations, static source analysis tools are very useful for guiding security auditors to potentially vulnerable code and significantly reduce the time it takes to perform an audit.

1.1 Overview

This thesis is a presentation of a static source analysis technique for vulnerability detection. We will develop a classification of software vulnerabilities and investigate the common patterns present in vulnerable source code. Next we will describe an approach to detecting potential vulnerabilities, using a combination of taint analysis and value range propagation. We will present Vulncheck, an implementation of our technique as an extension to the GCC compiler. We will investigate the effectiveness of our approach by testing it on a sample of vulnerable programs and comparing its results to existing techniques.

1.2 Organization

We begin Chapter 2 with an overview of software vulnerabilities and discuss their causes. In Chapter 3 we review existing static analysis techniques and vulnerability detection tools. We continue in Chapter 4 with an overview of compiler technology and discuss its implementation in the GNU C compiler. In Chapter 5 we present our classification of software vulnerabilities and describe the three common characteristics found in most vulnerabilities. We continue with a description of our approach to detecting these vulnerabilities and discuss its implementation as an extension to the GNU C compiler. In Chapter 6 we present experimental results and compare our system to two popular vulnerability detection tools. We discuss future work in Chapter 7 and conclude this thesis in Chapter 8.

Chapter 2

Software Vulnerabilities

In this section we give an overview of software vulnerabilities commonly found in computer programs and discuss their distinguishing features. The purpose of this section is to serve as an introduction for readers who might be familiar with static analysis, but are new to the field of vulnerability research. In section 5.2 we present a more detailed classification of vulnerabilities based on their distinguishing source code features.

2.1 Buffer Overflows

One of the oldest known vulnerabilities is the buffer overflow, famously used by the Morris Internet Worm in 1988 [27]. The classic buffer overflow is a result of misuse of string manipulation functions in the standard C library. The C language has no native string data type and uses arrays of characters to represent strings. When reading strings from the user or concatenating existing strings, it is the programmer's responsibility to ensure that all character arrays are large enough to accommodate the length of the strings. Unfortunately, programmers often use functions such as strcpy() and strcat() without verifying their safety. This may lead to user data overwriting

memory past the end of an array. One example of a buffer overflow resulting from misusing strcpy() is given below:

```
1   char dst[256];
2   char* s = read_string ();
3   strcpy (dst, s);
```

The string s is read from the user on line 2 and can be of arbitrarily length. The strcpy() function copies it into the dst buffer. If the length of the user string is grater than 256, the strcpy() function will write data past then end of the dst[] array.

If the array is located on the stack, a buffer overflow can be used to overwrite a function return address and execute code specified by the attacker [2]. A heap based overflow can be exploited by overwriting the internal structures used by the system memory allocator and tricking it into overwriting critical system data. However, the mechanics of exploiting software vulnerabilities are outside the scope of this work and will not be discussed further. For the purposes of vulnerability detection, it is sufficient to assume that any bug can be exploited by a determined attacker.

2.2 Format String Bugs

Format string bugs belong to a relatively recent class of attacks. The first published reports of format string bugs appeared in 2000, followed by the rapid discovery of similar vulnerabilities in most high-profile software projects. These include the Apache web server, wu-ftpd FTP server, OpenBSD kernel, IRIX telnetd server and many others. The format string bug is a relatively simple vulnerability. It arises when data received from an attacker is passed as a format string argument to one of

the output formatting functions in the standard C library, commonly known as the printf family of functions. These functions produce output as specified by directives in the format string. Some of these directives allow for writing to a memory location specified in the format string. If the format string is under the control of the attacker, the printf() function can be used to write data to an arbitrary memory location. An attacker can use this to modify the control flow of a vulnerable program and execute code of his or her choice. The following example illustrates this bug:

```
1   char *s = read_string ();
2   printf (s);
```

The string s is read from the user and passed as a format string to printf(). An attacker can use format specifier such as "%s" and "%d" to direct printf() to access memory at an arbitrary location. The correct way to use printf() in the above code would be printf("%s", s), using a static format string.

2.3 Integer Overflows

A third kind of software vulnerability is the integer overflow. These vulnerabilities are harder to exploit than buffer overflows and format string bugs, but despite this they have been discovered in OpenSSH, Internet Explorer and the Linux kernel. There are two kinds of integer issues: sign conversion bugs and arithmetic overflows.

Sign conversion bugs occur when a signed integer is converted to an unsigned integer. On most modern hardware a small negative number, when converted to an unsigned integer, will become a very large positive number. Consider the following C code:

```
1   char buf[10];
2   int n = read_int ();
3   if (n < sizeof(buf))
4       memcpy (buf, src, n);
```

On line 2 we read an integer n from the user. On line 3 we check if n is smaller than the size of a buffer and if it is, we copy n bytes into the buffer. If n is a small negative number, it will pass the check. The memcpy() function expects an unsigned integer as its third parameter, so n will be implicitly converted to an unsigned integer and become a large positive number. This leads to the program copying too much data into a buffer of insufficient size and is exploitable in similar fashion to a buffer overflow.

Arithmetic overflows occur when a value larger than the maximum integer size is stored in an integer variable. The C standard says that an arithmetic overflow causes "undefined behavior", but on most hardware the value wraps around and becomes a small positive number. For example, on a 32-bit Intel processor incrementing 0xFFFFFFFF by 1 will wrap the value around to 0. The main cause of arithmetic overflows is addition or multiplication of integers that come from an untrusted source. If the result is used to allocate memory or as an array index, an attack can overwrite data in the program. Consider the following example of a vulnerable program:

```
1   int n = read_int ();
2   int *a = malloc (n * 4);
3   a[1] = read_int ();
```

On line 1 we read an integer n from the user. On line 2 we allocate an array

of n integers. To calculate the size of the array, we multiply n by 4. If the attacker chooses the value of n to be 0x40000000, the result of the multiplication will overflow the maximum integer size of the system and will become 0. The malloc() function will allocate 0 bytes, but the program will believe that it has allocated enough memory for 0x40000000 array elements. The array access operation on line 3 will overwrite data past the end of the allocated memory block.

The next chapter describes techniques for analyzing source code and detecting vulnerabilities similar to the ones presented above.

Chapter 3

Static Analysis for Vulnerability Detection

3.1 Introduction

Static source analysis tools examine the source code of a program without executing it. The purpose of these tools is to extract some information from the source code or make judgments about it. The most common use of static analysis is in optimizing compilers. In fact, most of the high-level optimizations performed by a modern compiler depend on the results of static analyses such as control-flow and data-flow analysis. Outside of the compiler realm, static analysis techniques are used primarily in the areas of software metrics, quality assurance, program understanding and refactoring. Tools like the lint C syntax checker have used basic static analysis techniques for many years. More recently, static analysis has been used in code visualization tools [12] and has uncovered more than a hundred previously unknown bugs in the Linux kernel [8]. Our discussion will focus on the security applications of static analysis.

When used for vulnerability detection, static source analysis tools attempt to find specific vulnerable code patterns, or detect a possible control-flow path on which

the program reaches a vulnerable state. Unlike dynamic analysis, these tools are able to detect vulnerabilities in code that is rarely reached during the normal operation of a program. Of course, static source analysis also has its limitations. Most tools rely on a static rule set of predetermined code patterns that they should detect and report. These rules are developed by experienced vulnerability researchers and allow users with less security expertise to find bugs they would not otherwise be able to recognize. The drawback is that a tool will be able to report only vulnerabilities that exist in its rule set. If a tool finds no bugs in a program, this is not a guarantee that no bugs exist.

Even if a rule has been written for a particular vulnerability, a static source analysis tool might not find all instances where this rule applies. Any sufficiently complex analysis of a program is equivalent to solving the halting problem [22], which is provably undecidable [30]. Any static analysis that must return a result and do it within a reasonable amount of time will have to make approximations. This leads to false positives and false negatives in the output of the tool. One approach to minimizing the number of false negatives is to make the analysis more conservative and flag as potentially vulnerable any code that is not provably safe. For any source code with real-life complexity, this will increase the number of false positives well beyond manageability. A tool that guarantees the safety of 1% of your source, but reports the remaining 99% as vulnerable is obviousely of no use. Making the analysis less conservative will lower the number of false positives but some vulnerabilities will remain undetected. The design of static analysis tools is heavily influenced by the desire to find as many vulnerabilities as possible, while keeping the number of false positives low.

3.2 Approaches to static analysis

In this section we will review some popular approaches to static analysis and discuss their advantages and disadvantages.

3.2.1 Pattern matching

The simplest static analysis technique is pattern matching. A common source code auditing technique is to use the grep tool to find all occurrences of "strcpy" in the source code. Most of these would be calls to the strcpy() function in the standard C library, which is often misused and is a good indicator of a potential vulnerability. Further inspection of each call site is required to determine which calls to strcpy() are safe and which are not. This method is very imprecise and suffers from a number of practical problems. Pattern matching is unable to perform even trivial analysis of the source code, which makes it unsuitable for detecting complicated vulnerabilities. Another drawback of pattern matching is that the number of false positives can be very large. Lacking a proper C parser, a pattern matching tool is unable to tell apart comments from real code and is easily fooled by unexpected whitespace and macros.

3.2.2 Lexical analysis

Lexical analysis offers a slight improvement over simple pattern matching. A lexer is used to turn the source code into a stream of tokens, discarding whitespace. The tokens are matched against a database of known vulnerability patterns. This technique is used by tools like Flawfinder [35], RATS [24] and ITS4 [32]. Lexical analysis improves the accuracy of pattern matching, because a lexer can handle irreg-

ular whitespace and code formatting. Unfortunately, the benefits of lexical analysis are small and the number of false positives reported by these tools is still very high.

3.2.3 Parsing and AST analysis

The next step in improving the accuracy of static analysis is parsing the source code and building an abstract syntax tree (AST) representation of the program. This work is typically performed by the frontend of a compiler and this offers an opportunity for code reuse. One of the earliest C static source analysis tools, the lint [15] syntax checker from 1979, shared large sections of its source code with the UNIX V7 C compiler. The lint tool detected unreachable statements, unused variables and functions called with an improper number of arguments, and featured a stronger type system than the C compiler. Modern compilers are able to perform most of these checks, making lint obsolete, but more complicated static analysis tools still reuse compiler frontend code just like their predecessor.

Parsing C and especially C++ source code can be a very complex task. While most compilers are compatible with C standards such as ANSI C and C99, most programs rely on at least some nonstandard language features, compiler extensions or even compiler bugs. A good example is the following description of the difficulties experienced during cross platform development in C++:

> From the very beginning many incompatibility issues between gcc and Visual C++ compiler had to be addressed. Microsoft Visual C++ 6.0 came with a compiler full of flaws and bugs and lacked support for some important parts of the C++ ISO standard such as the absence of template

partial specialization. Microsoft Visual C++ 6 also came with other major problems like an extremely bad implementation of the C++ Standard Template Library (STL).

As for gcc, some major problems had to be faced with version 2.96. This was an unstable version of the compiler but that made it through to the RedHat 7 GNU/Linux distribution. The main flaw in this implementation of the GNU compiler was that lacked support for dynamic casting. The CLAM framework could not do without some dynamic casts that were applied on polymorphic variables at crucial points of the code. For this reason, CLAM did not (and still does not) support gcc version 2.96. When version 3.X of the compiler were published, some minor changes had to be applied to the code, especially due to the increased strictness in some syntax rules. [3]

To be able to correctly parse and analyze a wide range of programs, a static analysis tool needs a parser compatible with at least one of the major compilers. Integrating with a compiler frontend will ensure this compatibility. For this reason most of the advanced analysis tools on the UNIX platform utilize the GCC frontend, which is freely available under the GPL license. A promising new development is the MetaC++ project, which provides a library with an integrated GCC frontend and exposes the AST tree for use in source-to-source code translators and source code analysis tools [29].

The abstract syntax tree allows us to analyze not only the syntax, but also the semantics of a program. Lexical analysis tools will be confused by a variable with the

same name as a vulnerable function, but AST analysis will accurately distinguish the different kinds of identifiers. On the AST level macros and complicated expressions are expanded, which can reveal vulnerabilities hidden from lexical analysis tools. The pattern matching approach can be significantly improved by matching AST trees instead of sequences of tokens or characters. More complex analyses such as type qualifier and data-flow analysis can be built by utilizing the abstract syntax tree representation.

3.2.4 Type qualifiers

Some more advanced vulnerability detection tools are based on the type qualifier framework developed by Jeffrey Foster [11]. He describes type qualifiers as properties that "qualify" the standard types in languages such as C, C++, Java and ML. Most of these languages already have a limited number of type qualifiers (for example the const and register keywords in C programs.) Foster proposes a general purpose system for adding user-defined type qualifiers by annotating the source code and detecting type inconsistencies by type qualifier inference.

Most tools using the type qualifier approach are built on top of cqual, a lightweight type-based analysis engine for C programs. One example is the format string detection system developed by Shankar [26]. A new type qualifier, "tainted", is used to mark data from untrusted sources in the program. The tainted qualifiers are propagated through the program using type qualifier inference. If a variable marked tainted is used in a format string function, the tool reports a format string bug. The authors report that the tool is surprisingly effective and produces few false positives.

Type qualifier systems have a number of advantages. Tools like cqual require

only a few type qualifier annotations to be added to a program and the type qualifier inference is efficient even for large programs. The disadvantage of this approach is that it is applicable only to a small number of security vulnerabilities that can be expressed in terms of type inconsistencies.

3.2.5 Data-flow analysis

Unfortunately not all vulnerabilities can be described as simply as the format string bugs. Detecting buffer overflows, integer overflows, array indexing, or pointer arithmetic problems requires more complicated analyses than pattern matching or type inference. These vulnerabilities arise when variables in a program can take on values outside a certain safe range. For example a strcpy() function call is vulnerable when the size of the source string exceeds the size of the destination buffer. Detecting the conditions under which this can occur without executing the program is a hard problem. In any nontrivial program there are dependencies between the data manipulated by the code, which further complicates the task. Data-flow analysis is a traditional compiler technique for solving similar problems and can be used as a basis of vulnerability detection systems.

Precise data-flow analysis of C programs is impossible without a solution to the reachability and aliasing problems, which are unfortunately undecidable [21]. This leaves static source analysis tools with two options: use a conservative data-flow analysis, which provides accurate results in a limited number of cases, or make approximations and accept unsound or incomplete results. Generally a small number of false positives is acceptable and the ability to analyze source code without any restrictions is important, so most vulnerability checkers choose the latter option.

3.2.6 Taint Analysis

The concept of tainting refers to marking data coming from an untrusted source as "tainted" and propagating its status to all locations where the data is used. A security policy specifies what uses of untrusted data are allowed or restricted. An attempt to use tainted data in a violation of this policy is an indication of a vulnerability.

The most well known example of this technique is the taint mode provided by the Perl programming language [13]. When running in this mode, the interpreter flags all data read from files, network sockets, command line arguments, environmental variables and other untrusted sources as tainted. The language provides facilities for untainting (or "laundering") untrusted data after the programmer has verified that it is safe. Tainted data may not be used in any function which modifies files, directories and processes, or executes external programs. If this rule is violated, the interpreter will abort the execution of the program.

Tainting is particularly well suited for interpreted environments because data can be flagged and inspected at runtime. Applying tainting to a compiled programming language requires static analysis and does not guarantee the same level of precision.

The type qualifier inference used with great success by Shankar [26] to detect format string bugs is a form of taint analysis.

3.3 Static Analysis Tools

3.3.1 Flawfinder

Flawfinder [35] is a pattern matching vulnerability detection tool. It contains a large database of vulnerable patterns in C and C++ programs. During the analyzis of the source code, Flawfinder produces a list of potential security flaws, each with an associated risk level. The list of flaws reported by the tool is sorted by their risk level, with the riskiest shown first. Since Flawfinder usually reports many false positives, this sorting allows the user of the tool to focus on hits that are more likely to be real vulnerabilities.

3.3.2 ITS4

ITS4 [32] is another pattern matching tool. It is similar to Flawfinder, but it has the ability to analyze multiple languages in addition to C. Another difference is that ITS4 has special handlers for some rules in its ruleset, which perform additional processing to determine if a hit is really a vulnerability. For example, the handler for the strcpy() function marks the call as safe if the second parameter is a static string. Despite the flexibility offered by the custom handlers, the scope of the analysis in ITS4 is still limited to pattern matching and lexical analysis.

3.3.3 Cqual

Cqual [11] is a type-based analysis tool. It extends the C type system with custom type qualifiers and propagates them through type inference. Some vulnerabilities can be expressed as type inconsistencies and detected using Cqual. This approach

has been applied with great success by Shankar [26] to the problem of finding format string bugs.

3.3.4 SPLINT

SPLINT [10] is a tool for checking C programs for security vulnerabilities and programming mistakes. It uses a lightweight data-flow analysis to verify assertions about the program. Most of the checks performed by SPLINT require the programmer to add source code annotations which guide the analysis. This tool can be used to detects problems such as NULL pointer dereferences, unused variables, memory leaks and buffer overruns. Additional checks can be written in a special extension language.

3.3.5 MOPS

MOPS [4] is a tool for verifying the conformance of C programs to rules that can be expressed as temporal safety properties. It represents these properties as finite state automata and uses a model checking approach to find if any insecure state is reachable in the program. This approach is fully interprocerual and sound for some classes of vulnerabilities.

3.3.6 BOON

BOON [33] is a tool for detecting buffer overrun vulnerabilities in C code. It generates integer range constraints based on string operations in the source code and solves them, producing warnings about potential buffer overflows. The range analysis is flow-insensitive and generates a very large number of false alarms.

3.3.7 Stanford Checker

The Stanford Checker [8] is a system for building domain- and application-specific compiler extensions that check the source code for violations of specific rules. The system is built on top of the GCC frontend and uses data-flow analysis. The Stanford Checker has been used to find a large number of errors in Linux kernel, but most of them were not security related. The primary focus of the project is on programing mistakes, but its techniques are applicable to finding security vulnerabilities.

In the next chapter, we review compiler technology applicable to static analysis for vulnerability detection and describe the implementation of the GNU C Compiler.

Chapter 4

Compiler Technology

In this chapter we review compiler technology relevant to the design of our vulnerability detection tool and describe some of the implementation details of the GNU C Compiler.

4.1 Overview

4.1.1 Compiler Structure

A compiler is a program that translates programs written in a high-level language into a lower-level representation, typically machine language or object code. Modern compilers consist of multiple sequential phases, each performing an analysis or transformation of the source program and making the result available to the next phase. The main phases of a compiler are lexing, parsing, optimization, and code generation.

4.1.2 Lexing and Parsing

The first phase of a compiler performs lexical analysis of the source code. Lexing removes whitespace and comments, and transforms the source into a stream of tokens to be processed by a parser. The parser analyses the syntax of the source code and ensures that it conforms to the language grammar. Syntax errors are detected and reported at this stage of the compiler. The output of the parser is an abstract syntax tree (AST) representation of the program.

4.1.3 Abstract Syntax Trees

The abstract syntax tree representation clearly defines the syntax of a program and can be used for semantic analysis, type checking and high-level optimizations. The tree consists of tree nodes, designed to express the structure of the language. An example of an abstract syntax tree is given in figure 4.1.

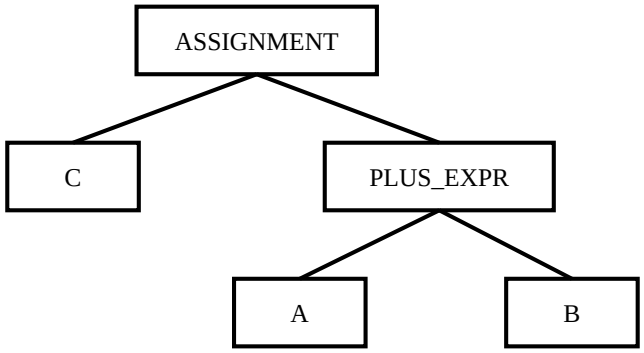

Figure 4.1: Abstract Syntax Tree representation of C = A + B

```
a = 1;
if (b > a) {
    a = b;
}
c = a;
```

(a) C source code

```
a_1 = 1;
if (b > a_1) {
    a_2 = b;
}
a_3 = PHI<a_1, a_2>;
c = a_3;
```

(b) SSA form

Figure 4.2: Example of a C program and its corresponding SSA form

4.1.4 Static Single Assignment Form

Most modern compilers transform the program into a static single assignment (SSA) form [23] before optimizing it. The SSA form has the property that each variable in the program is the target of only one assignment. This transformation is performed by creating multiple versions of a variable, one for each assignment, and introducing so-called Phi-functions at the join points of multiple versions. An example of a program and its corresponding SSA form is given in figure 4.2.

The SSA form of a program allows many optimizations to be performed more efficiently and greatly simplifies their implementation. Since unrelated definitions of each variable are assigned new variable names, data-flow analysis algorithms do not have to needlessly examine their relationships. Cytron et al. [5] present strong evidence of the benefits of static single assignment for optimization.

```
a = 1;
if (a < 0)
    foo();
b = a + 1;
c = a + b;
```

(a) Before

```
a = 1;
b = 2;
c = 3;
```

(b) After

Figure 4.3: The effect of constant propagation on a C program

4.1.5 Data-flow Analysis

Many opportunities for optimization arise from the flow of data and the dependencies between different variables in a program. Data-flow analysis allows us to determine this information without actually executing the program. One common optimization based on data-flow analysis is constant propagation. The goal of this optimization is to find variables that are assigned constants and replace all their uses with a direct use of the constant. To illustrate the effect of constant propagation, consider the program in figure 4.3 and its equivalent, but much more efficient form after constant propagation.

As an illustration of data-flow analysis, we will briefly describe a simple constant propagation algorithm that operates on the SSA form of a program [34]. The algorithm works by associating a lattice value with each variable in the program. There are three possible lattice values: UNDEFINED, VARYING and CONSTANT.

Initially, all variables are set to UNDEFINED. As more information is discovered, variables are set to CONSTANT or VARYING and this information is propagated through the program. When multiple variables meet in a Phi-function, the result is a new lattice value computed using the following "meet" function:

anything $\wedge$ UNDEFINED = anything

anything $\wedge$ VARYING = VARYING

CONSTANT $\wedge$ CONSTANT = CONSTANT (if the constants are the same)

CONSTANT $\wedge$ CONSTANT = VARYING (if the constants are different)

A description of the algorithm is given bellow:

1. Initialize all variables as UNDEFINED.

2. Examine all expressions. If an expression has a constant result, set the lattice value of the corresponding variable to CONSTANT. If an expression cannot be evaluated at compile-time, set the lattice value to VARYING. Otherwise set the lattice value to UNDEFINED.

3. Add all uses of newly discovered VARYING and CONSTANT variables to a worklist.

4. Take a use off the worklist. If it is a Phi-function, calculate a new lattice value using the meet function given above. If it is an arithmetic expression with all CONSTANT arguments, calculate its result and set the corresponding lattice value to CONSTANT. If any argument has a VARYING lattice value, set the result of the expression to VARYING.

5. Go to step 3 and repeat until a fixed point is reached.

After the data-flow analysis determines which variables are constants, another pass over the program is needed to replace all their uses with the corresponding constant values. Propagating constant values into the condition of an if statement allows us to optimize the program even further by determining which branch is taken at compile-time and eliminate unreachable code.

4.1.6 Value Range Propagation

An advanced data-flow analysis can be used to determine a range of possible values for each variable in a program. Patterson [20] describes a value range propagation algorithm in the context of branch prediction optimizations, but his approach is applicable to a wide range of problems. In section 5.3 we utilize Patterson's algorithm to improve the efficiency of our vulnerability detection system.

The value range propagation algorithm is similar in operation to the constant propagation described in the previous section. The main difference is that instead of constants, it propagates value ranges, which requires a more complicated lattice. Patterson's implementation represents the possible values of each variable as a set of weighted ranges with a lower bound, upper bound, stride and probability. The effects of arithmetic and conditional statements are simulated through range operations. For example if the range of variable a is [3, 10] and b has a range of [5,7], the algorithm will correctly infer that the range of the expression a + b is [8, 17].

Another difference is that the constant propagation algorithm evaluates an expression in a loop at most twice before its value is determined to be CONSTANT or VARYING. Loop variables do not need to be handled explicitly, because they will become VARYING as soon as the the algorithm visits the body of the loop again.

Consider the following code:

```
1   i = 0;
2   while (i < 1000)
3       i++;
```

If a value range propagation algorithm does not explicitly handle loop variables, the body of this loop will be evaluated 1000 times during the propagation. The range of i will start at [0,0] and will expand by one during each iteration of the loop. This is obviously very inefficient. Patterson's algorithm identifies variables modified in a loop and attempts to derive a range based on the operations performed in the body of the loop. Most loops in C programs match a small number of simple templates and their variables can be derived efficiently. The remaining few loops of higher complexity are still handled by "executing" the loop multiple times.

Having reviewed a number of basic compiler techniques, in the next section we describe their implementation in the GNU C Compiler.

4.2 GCC Internals

GCC, the GNU Compiler Collection, is an open source compiler suite which supports multiple languages and a large number of target architectures. The first release of GCC was in 1987 as part of the Free Software Foundation's GNU project. Since then it has become the compiler of choice for Linux and BSD environments, as well as a number of commercial operating systems, including Mac OS X, NEXTSTEP and BeOS. It is also supported as an alternative compiler on most UNIX systems and on Windows. GCC is being actively developed and as of version 4.0 it includes some

architectural changes making it more suitable for use as a research compiler. We have chosen to implement our vulnerability detection system as an extension to the GCC compiler. In this section we discuss the architecture of GCC, focusing on the internal representation and analyses passes relevant to our project.

4.2.1 GCC Structure

The GCC compiler has a simple pipeline architecture. The frontend of the compiler parses input files and builds a high-level intermediate representation (IR) of the source code. This representation is passed to the middleend, which runs a number of optimization passes on it, converts it to a lower-level intermediate representation and passes it to the backend. The backend emits assembly code for the target processor.

The decoupling of the frontend from the backend allows GCC to support multiple languages and target architectures. There are multiple frontends that provide support for C, C++, Java, Fortran, ADA and other languages. Backends exist for a large number of processors, including x86, Sparc, Alpha, ARM and PowerPC. Each of the frontends can be used with any of the backends. This flexibility provides a number of advantages. It minimizes the effort required to add support for a new language or processor, because a new frontend will work on all architectures already supported by GCC, and a new backend will work with all existing language frontends.

The architecture of GCC has changed significantly in version 4.0. In previous versions there was no common high-level IR and each frontend used its own tree-based representation. It was the frontend's job to convert this representation to the common low-level Register Transfer Language (RTL) before passing it to the backend. The

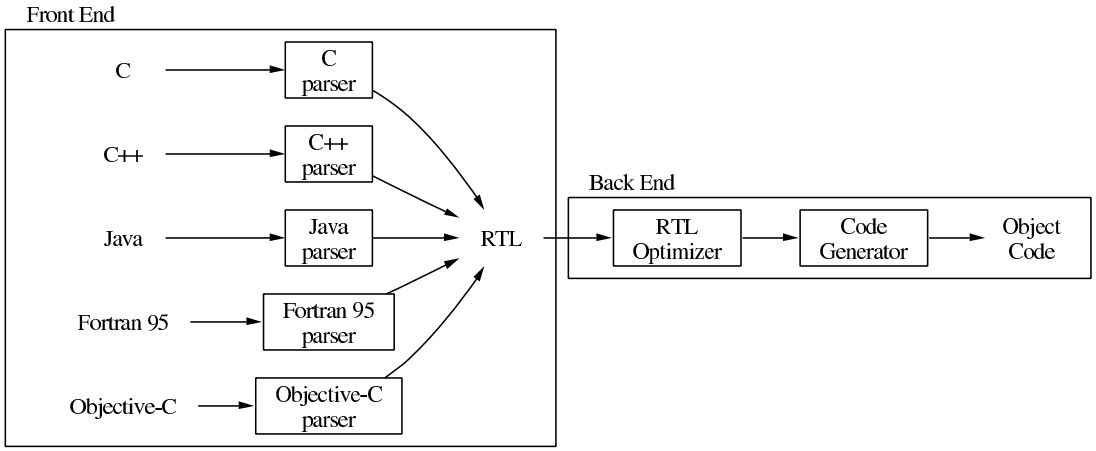

Figure 4.4: Diagram of the old GCC structure (from [18])

RTL was similar to a processor independent assembly language. High-level language features like data types, structures and variable references were not represented. This made the RTL unsuitable for high-level analyses and transformations. For many years most optimization passes of the GCC compiler were low-level optimizations on the RTL.

The desire to implement more high-level optimizations, in particular function inlining in C++ programs led to the development of high-level optimizers which used the tree representation of the C++ frontend. The drawback of this approach was that each frontend had its own IR and an optimization pass for one language would not work on another. The frontend tree representations were also very complex and relied on implicit assumptions about the language semantics. This made it very hard to implement more advanced optimizations.

4.2.2 TREE-SSA Project

The TREE-SSA project was started in late 2000 with the goal of building a language-independent framework for high-level optimization [18]. The main advantages of the new framework are the simplified intermediate representation and the common infrastructure for high-level analysis. The frontends generate a common high-level IR called GENERIC and pass it to the new language and target independent middleend. The first task of the middleend is to lower the representation and turn it into GIMPLE, a simplified subset of GENERIC. The GIMPLE representation is converted into static single assignment (SSA) form, which is used by most of the optimization passes. The TREE-SSA framework provides infrastructure for accessing the control-flow graph, use-def chains, points-to information, dominator trees and many other analyses useful in high-level optimization. The availability of these analyses greatly simplifies the task of implementing new optimization passes. After all tree optimizers have finished, the program is converted into low-level RTL, which is passed to the backend.

The TREE-SSA framework was incorporated into the development version of GCC in May 2004. The first official release of GCC to include TREE-SSA will be version 4.0, which is scheduled for release in early 2005.

4.2.3 GENERIC and GIMPLE

The GENERIC representation is a very high-level Abstract Syntax Tree (AST) representation, capable of representing all languages supported by the compiler. As such, it is complicated and contains many implicit assumptions about language se-

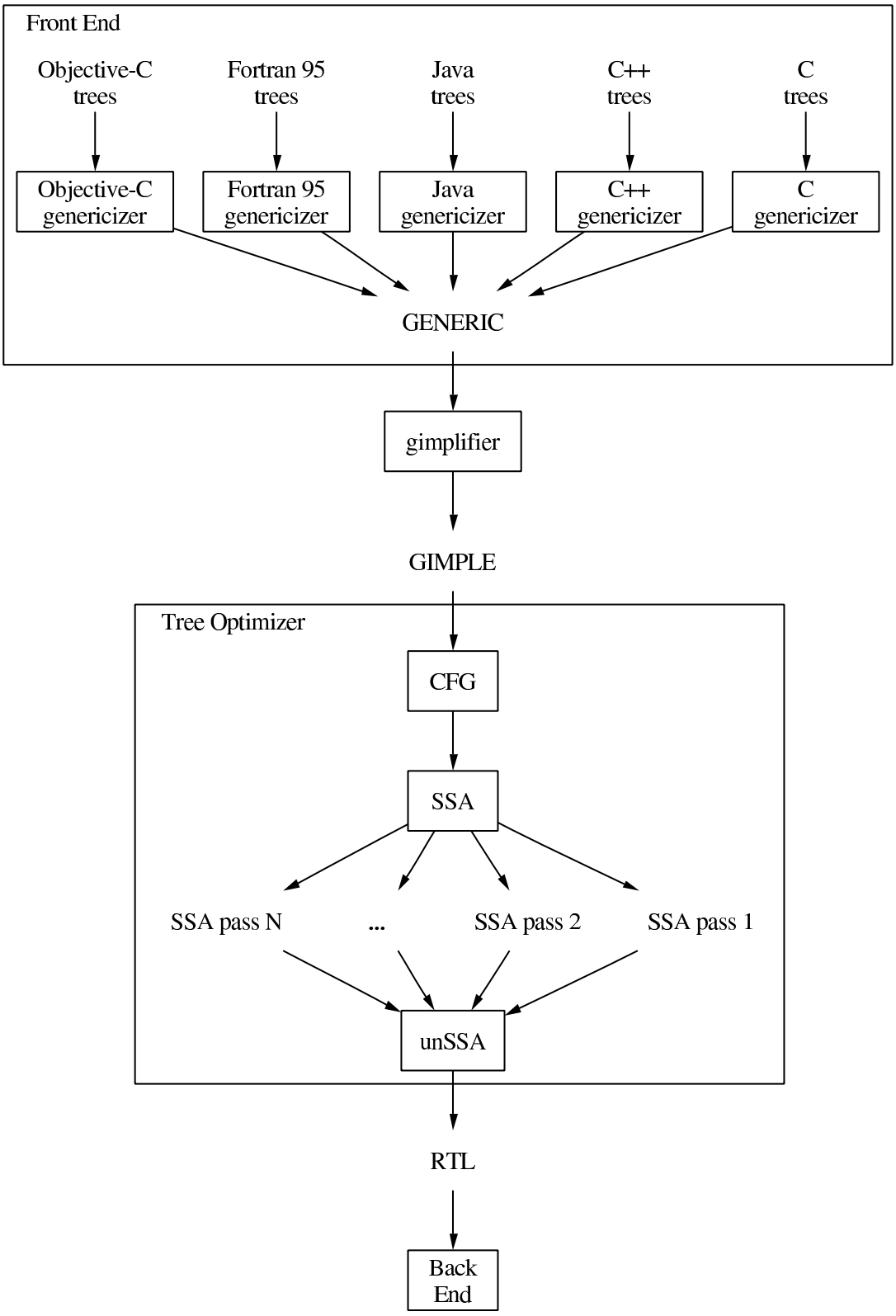

Figure 4.5: Diagram of the new GCC structure (from [18])

mantics. This representation is very well suited to representing the output of a parser, but it is hard to write an optimization pass operating on GENERIC because of its complexity.

The second representation used by GCC is the GIMPLE representation. It is based on the SIMPLE IR designed by the compiler research group at McGill University for use in the McCAT compiler framework [28]. GIMPLE is a simplified subset of the GENERIC representation. Its grammar is very small and restrictive, eliminating all hidden or implicit side effects of statements. When GENERIC is converted to GIMPLE, all complex expressions are broken down into a three-address form, adding temporary variables where necessary. The control flow of the program is also simplified and all loops are represented with IF and GOTO statements. The result is a representation that is expressive enough to represent languages like C, C++, Java and ADA, yet simple enough to use in language-independent tree level optimizers.

The goal of the design is to make the GIMPLE representation very easy to analyze and transform. Figure 4.6 shows a C program and its GIMPLE representations before and after control-flow simplification.

4.2.4 Implemented Analyses

The TREE-SSA framework implements a number of analyses and optimizations, including:

- Constant propagation
- Value range propagation
- Partial redundancy elimination

```
if (foo (a + b, c)) {
    c = b++ / a;
}
return c;
```

(a) C source code

```
t1 = a + b;
t2 = foo (t1, c);
if (t2 != 0) {
    t3 = b;
    b = b + 1;
    c = t3 / a;
}
return c;
```

(b) High GIMPLE

```
t1 = a + b;
t2 = foo (t1, c);
if (t1 != 0) <L1, L2>
L1:
t3 = b;
b = b + 1;
c = t3 / a;
goto L3;
L2:
L3:
return c;
```

(c) Low GIMPLE

Figure 4.6: Example of a C program and its GIMPLE representation (from [19])

- Dead and unreachable code elimination
- Tail recursion elimination
- Autovectorization

The above list is far from complete. There are many improvements planned for the next release of the compiler. The most important improvement will be the structure aliasing support, which will allow tree optimizers to distinguish between assignments to different fields in a structure. Since most large C programs make heavy use of structures, many opportunities for optimization and analysis are currently not realized. Another important project is the interprocedural analysis (IPA) infrastructure, enabling optimization passes to perform optimizations across multiple functions. This will bring GCC one step closer to the long-term goal of whole-program optimization.

In the next chapter, we present our static analysis for vulnerability detection and describe its implementation as an extension to the GNU C Compiler.

Chapter 5

Design

5.1 Overview

Our work was motivated by a desire to improve upon existing static analysis techniques for vulnerability detection. The combination of our experience with software vulnerabilities and compiler technology proved particularly beneficial to this effort.

We started by analyzing a sample of software vulnerabilities, looking for similarities in the source code of the affected programs. We were able to identify a small number of source code patterns that were responsible for the majority of the vulnerabilities. In section 5.2 we present a vulnerability classification based on these results.

We discovered that a large number of the vulnerabilities in our sample were a result of a program execution path with three common characteristics. We developed a data-flow analysis technique for detecting source code exhibiting these characteristics and implemented it as a vulnerability checking pass in the GNU C Compiler. The design and implementation of our technique are presented in sections 5.3 and 5.4.

5.2 Classification of Software Vulnerabilities

There are many available classifications of software vulnerabilities, but most of them focus on properties which are of little relevance to the problem of static source analysis. Information about the location of a vulnerability, versions affected by it, impact, severity, available fixes and workarounds is widely available [25] and very useful in the field of operational security. Researchers working on runtime checking have developed a classification based on the different memory locations that are overwritten during exploitation attempts [16]. Unfortunately, the surveyed classifications came up lacking when describing what the vulnerabilities look like on the source code level. The closest we came to a good classification was the rule set used by Flawfinder [35], which contains regular expressions for matching many common bugs.

We analyzed a sample of software vulnerabilities in popular UNIX software packages and developed a new classification based on common source code patterns. The sample we used was based on security advisories released by the Debian project in 2003 and 2004. Debian is a popular distribution of Linux and contains more than 8000 software packages, including the majority of open source UNIX software. Our intuition was that we would be able to detect similar vulnerabilities using similar techniques. The goal of the classification was to identify these similarities.

For every class of vulnerabilities, we have developed a short sample code illustrating the problem. We present our classification in the following sections.

5.2.1 Classic string buffer overflow

```
1   char dst[256];
2   char* s = read_string ();
3   strcpy (dst, s);
```

The string s is read from an untrusted source and its length is controlled by the attacker. The string is copied into a buffer of fixed size using the strcpy() function. If the length of the string exceeds the size of the destination buffer, the strcpy() function will overwrite data past the end of the buffer.

This type of code occurs when a program is processing user input and the programmer makes an assumption about its maximum size. This bug is becoming less common, because most programmers are aware of the danger of buffer overflows and try to avoid the strcpy() function.

One remaining issue, especially in older UNIX programs, is the assumption about the maximum filename, path or hostname length. For example, on an early version of Solaris, a 256 byte buffer would be large enough to hold any path on the system. Even if the code uses strcpy() with no length checking, the program will be safe. When the code is compiled on a modern UNIX system, where the maximum path size is 1024 or even 4096, the program will become vulnerable.

5.2.2 Memcpy buffer overflow

```
1   char src[10], dst[10];
2   int n = read_int ();
3   memcpy (dst, src, n);
```

The integer n is read from an untrusted source and its value is controlled by the attacker. No input validation is performed before n is used as the third argument of memcpy(). If the attacker chooses a value of n bigger than the size of the destination buffer, the memcpy() function will overwrite data past the end of the buffer.

There is no excuse for this code, but we have seen this vulnerability in multiple programs, including, to our great surprise, the Apache web server. It is more likely to occur when one function reads the data from the user and passes it to another one for processing. Each function expects the other one to validate the input, but neither of them does.

5.2.3 Format string bug

```
1   char* s = read_string ();
2   printf (s);
```

The string s is read from an untrusted source and its contents are controlled by the attacker. The string is used as a format string in one of the functions of the printf family. By including special format specifiers like "%n" in the format string, the attacker can trick printf() into writing to an arbitrary memory location.

This vulnerability is different than most others we have encountered, because it was not identified as a vulnerability until late 2000. This type of code was very common in C programs, including ones with an otherwise good security record. Generations of programmers had used this little shortcut, instead of the proper printf("%s", s). It was considered a harmless trick to avoid typing five extra characters.

This bug is relatively easy to find, even through manual source code inspection. The only case which makes detection more complicated is when the program uses a custom formatting function as a wrapper around printf(). This is often done in debugging and logging functions, which need to print out extra information before a warning or error message. Any calls to this custom function will have to be inspected just like the calls to the functions of the printf family.

5.2.4 Out of bounds array access

```
1   char s[256];
2   int n = read_int ();
3   s[n] = '\0';
```

The integer n is read from an untrusted source and its value is controlled by the attacker. No input validation is performed before it is used as an array index. By choosing a specific value of n, the attacker can access any memory location on the system. Depending on the program, this can give the attacker the ability to read or write to an arbitrary memory location, which is easily exploitable.

This type of code sometimes occurs when a program allows the user to choose from a number of options, each one stored as an element in an array. This example is just one of the many vulnerabilities enabled by the lack of array bounds checking in C. It is rare that a program uses user input as a direct index into an array. The more common case is when the index is computed based on a loop variable and can be influenced by user data.

5.2.5 Out of bounds array access with a negative array index

```
1   char s[256];
2   int n = read_int ();
3   if (n <= sizeof (s))
4       s[n] = '\0';
```

The signed integer n is read from an untrusted source and its value is controlled by the attacker. The program validates the integer n by making sure it is less than the size of the array and then uses it as an array index. If the attacker supplies a negative value, the condition of the if statement will be satisfied. A negative array index gives the attacker the ability to write to an arbitrary memory location, as long as it has a lower memory address than the array.

The obvious vulnerability in the example in section 5.2.4 has been fixed in this code, but it is still exploitable because of the subtler signedness issue. This is proof that fixing a vulnerability is not always an easy task.

5.2.6 Arithmetic overflow when allocating an array of objects

```
1   int n = read_int ();
2   void *ptr = malloc (sizeof (struct s) *  n);
```

The integer n is read from an untrusted source and its value is controlled by the attacker. On line 2 we multiply n by a constant. By choosing a specific large value of n, the attacker can cause an arithmetic overflow and allocate 0 bytes for the array. Any subsequent use of the array will access data which has not been allocated.

This type of code occurs when a program needs to read or process a variable number of data items. A common pattern is to read an integer indicating the number of items and allocate an array of sufficient size for all of them. The number of items is not validated because the programmer relies on the memory allocator to return an error if there is not enough memory to satisfy the allocation request.

5.2.7 Signed to unsigned conversion.

```
1   char src[10], dst[10];
2   int n = read_int ();
3   if (n <= sizeof (dst))
4       memcpy (dst, src, n);
```

The signed integer n is read from an untrusted source and its value is controlled by the attacker. If the attacker supplies a negative value, the condition of the if statement will be satisfied and the memcpy() function will be called. The third parameter to memcpy() is declared as an unsigned integer and the compiler will implicitly cast n to an unsigned integer. By choosing a specific negative value of n, the result of the cast will be a positive number greater than 10. The memcpy() function will overwrite data past the end of the destination buffer.

This type of code occurs when a program needs to read or process binary data of variable length. A common pattern is to read an integer indicating the number of bytes that follow and then copy this many bytes into an internal buffer. An attempt is made to validate the number of bytes, but it is possible to subvert this check if the variable is declared as an signed integer.

5.2.8 Scanf buffer overflow

```
1   char s[256];
2   scanf ("%s", s);
```

A function of the scanf family is used to read user input into a fixed size buffer. If the "%s" format specifier is used, scanf will read user input until either whitespace or the end of file is reached. By supplying more data than the buffer can hold, an attacker can overwrite data past the end of the array.

This type of code is not very common and is easy to detect using pattern matching. One case which complicates detection and can lead to false positives is the sscanf() function. It reads input from a string until it reaches a whitespace or a NUL character. If the size of the destination buffer is larger than the size of the input string, the function call will be safe.

5.2.9 Gets buffer overflow

```
1   char s[256];
2   gets (s);
```

The gets() function is used to read user input into a fixed size buffer. By supplying more data than the buffer can hold, an attacker can overwrite data past the end of the buffer.

This type of code is extremely rare and we were unable to find an actual example of it. It is included here because it is a classic vulnerability that every static source code analysis tool should be able to detect. There are very few programs still using this function, most of them are legacy applications. In an attempt to dissuade

programmers from using it in new code, the documentation of the GNU C Library includes the following warning:

> Warning: The gets function is very dangerous because it provides no protection against overflowing the string s. The GNU library includes it for compatibility only. You should always use fgets or getline instead. To remind you of this, the linker (if using GNU ld) will issue a warning whenever you use gets. [17]

5.2.10 Miscellaneous

There are vulnerabilities which arise from very complicated data dependencies and cannot be classified in one of the above categories. One example is the remotely exploitable vulnerability in the ProFTPD server [14]. It is caused by code that modifies the size of a buffer allocated by another function. The function proceeds to write to the buffer using its old size and overwrites unallocated memory. This kind of interaction between different modules in a program is very hard to properly verify using static analysis.

5.3 Proposed Solution

There is a common pattern present in seven out of the ten classes we have identified. All vulnerabilities in these classes are a result of an execution path with the following three characteristics:

1. Data is read from an untrusted source.

2. Untrusted data is insufficiently validated.

3. Untrusted data is used in a potentially vulnerable function or a language construct.

We define an untrusted source as any source of data that can be influenced or modified by an attacker. The list of untrusted sources is application and system specific, but for most C programs it will include command line arguments, environmental variables, network sockets, files and stdin input. Different sources give an attacker varying degrees of control over the data that reaches the program. There might be restrictions on its size, format and valid characters. For example, the domain name of a remote host on the Internet is considered untrusted data, but its size is limited and its format must conform to the requirements set forth in the DNS standard. A conservative worst-case approximation is to assume that all data coming from an untrusted source is completely arbitrary.

A system that does not allow user input of any kind would be free of software vulnerabilities, because it would always perform as designed. Short of waiting for a random failure, an attacker would have no way to influence the execution of the program. But most computer programs are designed to process user data. Since user interaction is necessary in these programs, they must be regarded as potentially vulnerable.

Not every use of untrusted data results in a vulnerability. In our research we have identified a limited number of potentially vulnerable functions and language constructs, listed in table 5.1. Some of them are insecure by design, like the gets()

Table 5.1: Potentially vulnerable functions and language constructs

Name	*Functions*	*Description*
Memory allocation	malloc(), calloc(), realloc()	Vulnerable if the value of the first parameter is a result of an arithmetic overflow during addition or multiplication involving untrusted data.
String manipulation	strcpy(), strcat()	Vulnerable if the size of the destination buffer is smaller than the size of the string to be copied into it.
Memory copying	memcpy(), memmove(), memccpy()	Vulnerable if the number of bytes to copy is greater than the size of the destination buffer.
Array access		Vulnerable if the array index is negative or greater than the size of the array.
Format string functions	printf(), sprintf(), fprintf()	Vulnerable if the format string contains untrusted user data.
Insecure by design	gets()	This function is always vulnerable.

function, while others are vulnerable in a very specific set of circumstances. Most are safe if the untrusted data is properly validated. The degree of validation required depends on the type of data and its use. For example, when we use a string as an argument to strcpy() we must check its length. If the same string is used in a printf() function, its length is irrelevant, but it should not contain any untrusted data.

A vulnerability occurs when an execution path exists between a read from an untrusted source and a potentially vulnerable use with insufficient data validation. Our approach to detecting these vulnerabilities is based on a combination of taint analysis and value range propagation. We describe these techniques in more detail in the following two sections.

5.3.1 Taint Analysis

We use interprocedural taint analysis to determine the program locations where data is read from an untrusted source and where tainted data is used as an argument to a potentially vulnerable function. This is a forward data-flow analysis problem and can be solved efficiently using an iterative algorithm. There are only two possible lattice values: TAINTED and NOT TAINTED. The meet function for the lattice is based on the idea that anything that meets a tainted value becomes tainted.

$$\begin{array}{cccccc} \text{NOT TAINTED} & \bigwedge & \text{NOT TAINTED} & = & \text{NOT TAINTED} \\ \text{anything} & \bigwedge & \text{TAINTED} & = & \text{TAINTED} \end{array}$$

A description of the algorithm is given bellow:

1. Initialize all variables as NOT TAINTED.

2. Find all calls to functions that read data from an untrusted source. Mark the values returned by these functions as TAINTED.

3. Propagate the tainted values through the program. If a tainted value is used in an expression, mark the result of the expression as TAINTED.

4. Repeat step 3 until a fixed point is reached.

5. Find all calls to potentially vulnerable functions. If one of their arguments is tainted, report this as a vulnerability.

Our taint analysis algorithm is able to detect the vulnerability in the following code:

```
1   char src[10], dst[10];
2   int n, m;
3   n = read_int();
4   m = n + 1;
5   memcpy (dst, src, m);
```

If read_int() is a tainting function, then n will be marked as tainted. The propagation algorithm will find the use of n in the expression on line 4 and will taint its result, which is stored in m. On line 5 we have a use of a tainted value in a call to memcpy(), a potentially vulnerable function. This is a vulnerability and it will be correctly identified and reported by our system.

5.3.2 Value Range Propagation

Applying taint analysis to a vulnerable program allows us to discover execution paths where data is read from an untrusted source and used improperly. These are only two of the properties we are looking for. The third characteristic of a vulnerability is that the data is not sufficiently validated. Consider the following example:

```
1   unsigned int n;
2   char src[10], dst[10];
3   n = read_int ();
4   if (n <= sizeof (dst))
5       memcpy (src, dst, n);       /* n is < sizeof (dst) */
6   else
7       memcpy (src, dst, n);       /* n is > sizeof (dst) */
```

Analogous to the previous example, n will be marked as tainted. Our taint analysis algorithm will report both uses of n in the memcpy() calls on lines 5 and 7 as vulnerable. This is correct for the call on line 7, because n can be arbitrary large. However, on line 5 we have a false positive. The if statement ensures that the value of n is less than the size of the destination buffer, which makes the memcpy() call safe.

Taint propagation alone is not sufficient to distinguish between the safe and vulnerable use of untrusted data in the above example. We need an analysis that takes into account the effect of the if statement and determines the possible values of the n variable in the then and else branches. Patterson's value range propagation algorithm [20] can be used solve this problem by calculating a range of possible values

for each variable. It operates on the SSA form of a program, which represents each definition of a variable as a separate version of it. This gives the algorithm the ability to distinguish between the different ranges of n in the two branches of the if statement.

Applying value range propagation to the example above will gives us a range of [0,10] for n in the then branch and [11, MAXINT] in the else branch. To determine whether a given use of n is vulnerable, we have to compare the upper bound of n at that location to the size of the destination buffer. If a buffer is dynamically allocated at runtime and its size is unknown during the static analysis, we assume that an upper bound equal to the maximum integer value indicates a vulnerability.

A summary of our vulnerability detection approach is given below:

1. Use the value range propagation algorithm to determine a range for each variable.

2. Find all calls to functions that read data from an untrusted source and mark their results as TAINTED.

3. Use the taint propagation algorithm to find all tainted values in the program.

4. Find all calls to potentially vulnerable functions and inspect their arguments. If an argument is TAINTED and its range is outside the safe range for the function, report this as a vulnerability.

5.4 Implementation

We have implemented our vulnerability detection system as an extension to the GNU C compiler. We used the TREE-SSA framework to implement a "Vulncheck"

analysis pass, executed during the optimization phase of the compiler. Our analysis produces additional warnings about vulnerabilities, but does not interfere with the normal operation of the compiler in any other way.

Vulncheck consists of three main parts: initialization, taint analysis and vulnerability reporting. Notable is the absence of value range propagation. An analysis pass implementing Patterson's algorithm was already present in the TREE-SSA framework and we were able to use its results without duplicating the work.

The initialization of Vulncheck is performed in the vulncheck_initialize() function. Its first job is to load an annotation file containing a list of vulnerable functions and functions returning untrusted data. The annotation file consists of function prototypes in standard C, annotated with additional information. There are four different annotations:

- The __USER_DATA__ annotation indicates a function returning untrusted data. If the data is returned as a result, the annotation must appear in the beginning of the prototype. If the untrusted data is returned in one of the function parameters, the annotation must be placed after that parameter.

- A function annotated with __VULN_USER__ is vulnerable if it is passed untrusted data as a parameter. This annotation must appear after the parameter to which it applies.

- The __VULN_RANGE__ annotation is used for functions which are vulnerable if one of their parameters is tainted and outside a safe range. This annotation must appear after the parameter to which it applies.

- Functions annotated with __VULN_FUNC__ are considered vulnerable, regardless of the results of taint analysis or value range propagation. This annotation must appear in the beginning of the function prototype.

We have written annotations for all security relevant functions in the standard C library. The full list is given in appendix B. The following excerpt contains examples of annotations for some of the more common functions:

```
ssize_t read (int filedes,void *buffer __USER_DATA__,size_t size);
void * memcpy (void *restrict to, const void *restrict from,
               size_t size __VULN_RANGE__);
int printf (const char *template __VULN_USER__, ...);
__VULN_FUNC__ char * gets (char *s __USER_DATA__);
```

After the list of vulnerable functions is loaded, vulncheck_initialize() iterates through the statements in the program and finds all function calls. If the name of a callee matches one of the functions annotated with __USER_DATA__, the corresponding variables are marked as TAINTED. The initialization is finished and taint_propagate() is called to propagate the taint information through the program. This is a straightforward implementation of the taint propagation algorithm described in section 5.3.1.

The final task of Vulncheck is to combine the results of the taint analysis and value range propagation and report all detected vulnerabilities. This is done by the vulncheck_report() function. It iterates over all function calls and compares their names to the list of annotated functions. Tainted values used as parameters to __VULN_USER__ functions are reported as vulnerabilities. Dealing with

__VULN_RANGE__ annotations requires one additional step: we report a vulnerability if a parameter is tainted and its upper bound is equal to MAXINT.

The integration with the GCC compiler allows us to use Vulncheck without any modifications to the build process used for compiling most UNIX software. Passing the -fvulncheck option to the compiler enables our vulnerability detection pass. A custom annotations file can be specified with the -fvulncheck-data option. In most build environments we have encountered, these two options can be specified on the command line or added to the CFLAGS environmental variable before invoking the make command and compiling the software. Figure 5.1 shows a screenshot of using Vulncheck.

```
$ gcc -O2 -o vuln -fvulncheck -fvulncheck-data=glibc.vuln vuln1.c
vuln1.c: In function 'main':
vuln1.c:18: warning: vulnerable call to memcpy
vuln1.c:21: warning: vulnerable call to printf
```

Figure 5.1: A screenshot of running Vulncheck

Chapter 6

Results

6.1 Methodology

We tested the effectiveness of our vulnerability detection approach on several sample programs containing buffer overflows and format string bugs. We measured the number of correctly identified vulnerabilities, as well as the number of false positives and false negatives in the results. Ideally, a static analysis tool would be able to find all vulnerabilities in a program without missing any or reporting safe code as vulnerable. The imprecise nature of static analysis makes this impossible. We can expect a certain number of false positives and false negatives from all vulnerability detection tools. False negatives indicate a failure of the analysis, because a real vulnerability was not detected. A tool which detects more vulnerabilities is better than a tool reporting false negatives instead. The number of false positives is also important, because they increase the effort required to verify the results of the analysis. When comparing different vulnerability detection approaches, the number of false positives and false negatives is a good indicator of their effectiveness.

A less important factor in vulnerability detection is the speed of the analysis.

Unfortunately we were unable to obtain reliable data about the speed of Vulncheck. Even on large programs (more than 10000 lines of C), the difference between compiling with and without our analysis pass was smaller than the margin of error in our measurements. This is not surprising considering that our taint analysis algorithm is no more complex than constant propagation, which is only one of the many optimization passes used by GCC. The most expensive part of our analysis, computing and propagating value ranges, is done in the compiler even when Vulncheck is disabled.

Due to limitations in our implementation, we are unable to detect any vulnerabilities arising from interactions of multiple functions or manipulation of C structures. This puts most major UNIX software out of our reach. For testing purposes, we developed sample programs that exhibit many of the properties found in real-world programs, but have none of the features our analysis cannot handle. The code of our sample programs is presented below.

6.1.1 vuln1.c

```
1   int main() {
2       char buf[10], dst[10];
3       int n, p;
4       fgets (buf, sizeof(buf), stdin);
5       n = buf[0];
6       memcpy (dst, buf, n);   /* vulnerable */
7       p = 10;
8       memcpy (dst, buf, p);   /* safe  */
9   }
```

Our first sample program contains a memcpy buffer overflow. It is a result of untrusted data coming from the fgets() function on line 4 being used in the memcpy() call on line 6. The value of n is controlled by the attacker and is not validated in any way before its use. If the attacker supplies a value larger than 10, the memcpy() function will overwrite data past the end of the destination buffer.

To make the analysis of this program a little bit more challenging, we introduced a potential false positive. The memcpy() call on line 8 is safe, because the value of p is a constant. We expect our tool to correctly identify it as safe, but other static analysis tools might report it as vulnerable, because it looks similar to the buffer overflow on line 7.

6.1.2 vuln2.c

```
1   int main()
2   {
3       char buf[10], dst[10];
4       int n, m;
5       fgets (buf, sizeof(buf), stdin);
6       n = buf[0];
7       m = n + 1;
8       memcpy (dst, buf, m);   /* vulnerable */
9       return 0;
10  }
```

This sample program is similar to the first one, with one minor difference. The

tainted value of n is not used directly in the memcpy() call, but an attacker can use it to influence the value of m. If m is larger than 10, the memcpy() function will overwrite data past the end of the destination buffer. This case is designed to test taint propagation in arithmetic expressions. It demonstrates the ability of our taint analysis to deal with complicated data dependencies, which are out of reach for tools based on pattern matching techniques.

6.1.3 vuln3.c

```
1   int main()
2   {
3       char buf[10];
4       fgets (buf, sizeof(buf), stdin);
5       printf (buf);            /* vulnerable */
6       printf ("%s", buf);      /* safe */
7       return 0;
8   }
```

There is a format string bug in our third sample program. A string read from the user is used as a format string argument to the printf() function on line 5. The printf() call on the next line is safe, because the format string is static.

6.1.4 vuln4.c

```
1   int main()
2   {
3       char str1[] = "First format string\n";
4       char str2[] = "Second format string\n";
5       char *p;
6       char buf[10];
7       int n;
8       fgets (buf, sizeof(buf), stdin);
9       n = buf[0];
10      if (n > 0)
11          p = str1;
12      else
13          p = str2;
14      printf (p);              /* safe */
15      return 0;
16  }
```

This program is an attempt to induce a false positive in tools using pattern matching techniques. The format string argument to the printf() call on line 14 is a pointer. The function would be vulnerable if the pointer points to data coming from an untrusted source. The program above chooses between two format strings based on whether the user input is a positive or negative number. Both format strings are static and the program is not vulnerable.

Pattern matching tools have no way to determine the set of objects a pointer can point to at runtime. They have two options for dealing with code similar to the function above. If they choose to ignore it, most format string bugs will go undetected. A better approach is to report all nonstatic format strings as potential vulnerabilities, which will result in some false positives. We expect most tools to choose the latter approach.

6.1.5 vuln5.c

```
1   int main()
2   {
3       char buf[10], dst[10];
4       int n;
5       fgets (buf, sizeof(buf), stdin);
6       n = buf[0];
7       if (n <= sizeof(buf))
8           memcpy (dst, buf, n);   /* safe */
9       else
10          memcpy (dst, buf, n);   /* vulnerable */
11      return n;
12  }
```

Our last sample program is the most challenging. It contains two potentially vulnerable calls to the memcpy() function. Both calls use the tainted value of n, which is read from an untrusted source on line 5. On line 7 we have an if statement

that ensures that n is less than the size of the destination buffer before copying data into it. This makes the memcpy() call on line 8 safe. The other memcpy() call in the else clause is vulnerable, because n is larger then the size of the buffer.

6.1.6 Test Setup

Our tests were run on a Linux server with a 2.7GHz Intel Xeon CPU and 4GB of memory. As the basis of our tool we used a prerelease snapshot of GCC 4.0 taken on February 20, 2005.

6.2 Results

We tested Vulncheck, our vulnerability detection tool, by running it on the five sample programs shown in the previous section. To investigate the relative effectiveness of taint analysis, value range propagation and the combination of both techniques, we ran three sets of tests. In the first set, we disabled the value range propagation algorithm and analyzed our sample programs using only taint analysis. In the second set of tests, we disabled taint analysis and used only value range propagation. Finally, we repeated the tests with both algorithms enabled.

When running in taint analysis mode, our tool was able to detect all four vulnerabilities in our sample programs, as shown in table 6.1. The only false positive was the first memcpy() call in vuln5.c, which is safe because of proper validation of untrusted data. This false positive is an indication of the main deficiency of the taint analysis approach, which is its inability to detect vulnerabilities occurring due to insufficient input validation.

Table 6.1: Results of running Vulncheck in taint analysis mode

Program	*Detected vulnerabilities*	*False positives*	*False negatives*
vuln1.c	1	0	0
vuln2.c	1	0	0
vuln3.c	1	0	0
vuln4.c	0	0	0
vuln5.c	1	1	0
Total	4	1	0

Running Vulncheck with value range propagation as its only analysis results in three correctly detected vulnerabilities, no false positives and one false negative. The results are listed in table 6.2. The advantage of value range propagation over taint analysis is that it can identify data which is outside a valid range for a potentially vulnerable use. However, the value range propagation algorithm cannot identify untrusted data, which is why it fails to detect the format string vulnerability in vuln3.c.

As expected, combining the two algorithms improves the accuracy of our vulnerability detection. The results are listed in table 6.3. Using both analyses, Vulncheck detects all 4 vulnerabilities with no false positives and no false negatives.

Our results indicate that the combination of taint analysis and value range propagation is an effective approach to detecting software vulnerabilities. The addition of value range propagation improves the accuracy of the taint analysis, minimizing the number of false positives.

Table 6.2: Results of running Vulncheck in value range propagation mode

Program	*Detected vulnerabilities*	*False positives*	*False negatives*
vuln1.c	1	0	0
vuln2.c	1	0	0
vuln3.c	0	0	1
vuln4.c	0	0	0
vuln5.c	1	0	0
Total	3	0	1

Table 6.3: Results of running Vulncheck with taint analysis and value range propagation

Program	*Detected vulnerabilities*	*False positives*	*False negatives*
vuln1.c	1	0	0
vuln2.c	1	0	0
vuln3.c	1	0	0
vuln4.c	0	0	0
vuln5.c	1	0	0
Total	4	0	0

6.3 Testing Competing Tools

To evaluate the effectiveness of our system, we tested two competing vulnerability detection tools and compared their results with those of Vulncheck. We chose Flawfinder [35] and SPLINT [10], two programs widely used in the security community. Flawfinder is an example of the pattern matching approach to vulnerability detection, while SPLINT uses an annotations based data-flow analysis. We performed the tests using the sample of vulnerable programs presented earlier in this chapter. We compare Vulncheck, Flawfinder and SPLINT by comparing the number of detected vulnerabilities, false positives and false negatives reported by each program.

6.3.1 Flawfinder

Flawfinder uses a pattern matching approach to detect potential security flaws. All rules in its ruleset have an associated risk level, ranging from 0, very little risk, to 5, high risk. When running Flawfinder, a user can specify a minimum risk level to limit the number of patterns detected in the source code. Potential security flaws with a risk level lower than the minimum will be excluded from the final results. Increasing the minimum risk level decreases the number of false positives, but it also lowers the number of detected vulnerabilities.

We performed three sets of tests using Flawfinder. In the first set, we ran the program in its default configuration. For the second set, we raised the minimum risk level to 3. For the third set, we used the -F command line option, which according to the documentation lowers the number of false positives.

When running in its default configuration, Flawfinder detected all four vul-

Table 6.4: Results of running Flawfinder in its default configuration

Program	*Detected vulnerabilities*	*False positives*	*False negatives*
vuln1.c	1	2	0
vuln2.c	1	1	0
vuln3.c	1	1	0
vuln4.c	0	2	0
vuln5.c	1	2	0
Total	4	8	0

nerabilities in our sample programs. There were eight false positives and no false negatives. These results are consistent with our expectations about pattern matching tools. They are able to detect most vulnerabilities because they report every language construct that can be misused. For example, Flawfinder reported all calls to the memcpy() function as suspicious, regardless of their actual use.

Raising the minimum risk level to 3 decreased the number of false positives to one, but the number of correctly identified vulnerabilities decreased significantly. In this configuration Flawfinder was able to find only the format string bug in vuln3.c and its report contained three false negatives.

Flawfinder was most successful when invoked with the -F command line option. In the documentation of the tool it is described as an option that excludes patterns likely to cause false positives. In this configuration the tool detected all four vulnerabilities with only three false positives.

Table 6.5: Results of running Flawfinder with minlevel = 3

Program	*Detected vulnerabilities*	*False positives*	*False negatives*
vuln1.c	0	0	1
vuln2.c	0	0	1
vuln3.c	1	0	0
vuln4.c	0	1	0
vuln5.c	0	0	1
Total	1	1	3

Table 6.6: Results of running Flawfinder with the -F option

Program	*Detected vulnerabilities*	*False positives*	*False negatives*
vuln1.c	1	1	0
vuln2.c	1	0	0
vuln3.c	1	0	0
vuln4.c	0	1	0
vuln5.c	1	1	0
Total	4	3	0

Table 6.7: Results of running SPLINT

Program	*Detected vulnerabilities*	*False positives*	*False negatives*
vuln1.c	0	0	1
vuln2.c	0	0	1
vuln3.c	1	0	0
vuln4.c	0	0	0
vuln5.c	0	0	1
Total	1	0	3

6.3.2 SPLINT

SPLINT uses program annotations and data-flow analysis to detect programming mistakes in C programs. It detects problems such as NULL pointer dereferences, unused variables, memory leaks and buffer overruns. One feature of SPLINT is its extension language which can be used to define new attributes and checks. A sample extension implementing taint analysis is provided in the SPLINT distribution.

When we first ran SPLINT on our sample set of vulnerable programs, it produced a large number of warnings. The majority of them were about type mismatches, functions without a return statement and discarded function results. As these checks were not related to security, we disabled them using the appropriate arguments when invoking SPLINT. The results of the remaining security checks performed by SPLINT are listed in table 6.7.

We were surprised by the results. SPLINT reported no false positives, but

found only one vulnerability, the format string bug in vuln3.c. All other problems remained undetected. After inspecting the source code of the tool, we discovered that its implementation of taint analysis was limited only to C strings. We attempted to expand the definition of SPLINT's "tainted" attribute to cover other data types as well, but soon we ran into another problem. The taint propagation algorithm in SPLINT was not propagating the taint information between different data types. We suspect that even if the taint propagation issue is fixed, SPLINT would not perform better than Vulncheck running in taint analysis mode.

6.3.3 Conclusion

A summary of our results is given in table 6.8. The effectiveness of SPLINT is similar to that of Flawfinder running at a high minimum risk level. Both tools detected the format string bug and nothing else. At a lower minimum risk level, Flawfinder was able to detect all 4 vulnerabilities in our sample set, but it produced a large number of false positives. Vulncheck detected all 4 vulnerabilities with zero false positives.

One must be careful about extrapolating our results to large real-life programs. The sample set used in these tests was small and specifically tailored to avoid structures that our implementation cannot handle. We believe this to be an implementation problem and not a deficiency in our general approach. In chapter 7 we will discuss our plans for addressing these problems and improving the effectiveness of the analysis.

Table 6.8: Comparison between Vulncheck, Flawfinder and SPLINT

Tool	*Detected vulnerabilities*	*False positives*	*False negatives*
Vulncheck	4	0	0
Flawfinder	4	8	0
Flawfinder (minlevel = 3)	1	1	3
Flawfinder (-F)	4	3	0
SPLINT	1	0	3

Chapter 7

Future Work

The main deficiency of Vulncheck is its inability to properly analyze source code containing structures. This is a significant impediment to the practical use of our prototype, because most large C program make heavy use of structures. The reason for this deficiency is that the data-flow infrastructure in the GCC compiler does not distinguish between assignments to different fields in a structure. An assignment to any field will be treated as a "virtual" definition of the entire structure. Similarly, a read from a structure field will be treated as a "virtual" use of the entire structure. This prevents our taint analysis and value range propagation algorithms from determining the taintedness and value range of structure fields.

The inability to propagate information through structures has an adverse effect not only on our system, but also on a number of optimizations performed by the compiler. As a result, the developers of GCC are planning to address this problem in version 4.1 of the compiler. One benefit of our integration with GCC is that as soon as structure aliasing information is available in the TREE-SSA optimization framework, Vulncheck should be able to use it with little or no modifications to our code.

One very common vulnerability that our implementation fails to detect is the classic strcpy() buffer overflow. Our taint analysis and value range propagation algorithms work well with integer variables, but they need to be extended to support C strings. Taint analysis might track two separate "tainted" attributes for each string: one for the contents of the string and one for its length. Value range propagation needs to propagate ranges for string lengths and take into account the effect of string manipulation functions.

The efficiency of our analysis will increase if we have better information about the effect of functions in the standard C library. SPLINT [10] supports richer annotations that can express information such as "strcat() returns a tainted value if either of its parameters is tainted." Adding support for similar annotations to Vulncheck is one possible area for improvement.

The most challenging future work on our system will be the addition of interprocedural and whole-program analysis. Many of the vulnerabilities we have encountered are a result of interactions between multiple functions in a program and cannot be detected using intraprocedural analysis. Currently GCC has a very limited infrastructure for interprocedural analysis and no support for whole-program optimization. Even if the infrastructure were available, our current algorithms for taint analysis and value range propagation might not scale when applied to large programs containing thousands of functions. One common technique for interprocedural data-flow analysis is to use intraprocedural propagation to build a summary of each function and then use these summaries instead of analyzing each function multiple times. We need to investigate if this technique is applicable to our analysis.

Whole-program analysis is very similar to interprocedural analysis, but its im-

plementation in the GCC compiler will be problematic. GCC is designed is to operate on one compilation unit at a time and the build systems of most software packages operate under this assumption. Implementing whole-program analysis will probably require significant changes in the architecture of the compiler, or an implementation of Vulncheck outside of GCC.

Chapter 8

Conclusion

This thesis introduced a static source analysis approach to vulnerability detection utilizing data-flow propagation algorithms.

The design of our vulnerability detection system was based on our practical experience with software vulnerabilities and exploits. We have examined a large number of known vulnerabilities and described the similarities between them. We have developed a classification of vulnerabilities based on their common characteristics. The discovery of three characteristics present in almost all described vulnerability classes allowed us to develop a static source analysis algorithm for detecting potential vulnerabilities.

Our approach was based on a combination of taint analysis, a known vulnerability detection technique, and value range propagation, previously used in compiler optimizations. The application of value range propagation to the problem of vulnerability detection is our main contribution to the field.

We described an implementation of our algorithm as a vulnerability checking pass in the GNU C compiler. We believe that the integration with a compiler will have a significant positive impact on the adoption of our vulnerability detection system by

the developer and security communities, as well as its future expandability.

The experimental results presented in this work support our conclusion that the combination of taint analysis and value range propagation is a superior approach to vulnerability detection. We were able to outperform existing tools on a variety of test cases.

We believe that as long as unsafe languages such as C are in use, static source analysis techniques will play an important role in securing critical software systems.

REFERENCES

[1] Aitel, Dave. SPIKE. `http://www.immunitysec.com/`.

[2] Aleph One [pseud.]. 1996. Smashing the stack for fun and profit. *Phrack* 49–14.

[3] Amatriain, Xavier. 2004. An object-oriented metamodel for digital signal processing with a focus on audio and music. Ph.D. thesis, Universitat Pompeu Fabra.

[4] Chen, H., and D. Wagner. 2002. MOPS: an infrastructure for examining security properties of software. Tech. Rep. UCB//CSD-02-1197, University of California, Berkeley.

[5] Cytron, Ron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* 13(4):451–490.

[6] Eddington, Michael. Peach fuzzer framework. `http://www.ioactive.com/v1.5/tools/`.

[7] Eigler, Frank. 2003. Mudflap: Pointer use checking for C/C++. In *Proceedings of the 2003 gcc developers summit*, 57–70. Ottawa, Canada.

[8] Engler, D., B. Chelf, A. Chou, and S. Hallem. 2000. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the fourth symposium on operating systems design and implementation*. San Diego, CA, USA.

[9] Etoh, Hiroaki, and Kunikazu Yoda. Protecting from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp/main.html`.

[10] Evans, David. SPLINT. `http://www.splint.org/`.

[11] Foster, Jeffrey. 2002. Type qualifiers: Lightweight specifications to improve software quality. Ph.D. thesis, University of California, Berkeley.

[12] GrammaTech. CodeSurfer. `http://www.grammatech.com/products/codesurfer/`.

[13] Hurst, Andrew. 2004. Analysis of Perl's taint mode. `http://hurstdog.org/papers/hurst04taint.pdf`.

[14] Internet Security Systems. 2003. ProFTPD ASCII file remote compromise vulnerability. `http://xforce.iss.net/xforce/alerts/id/154`.

[15] Johnson, S. 1978. Lint, a C program checker. In *Unix programmer's manual, AT&T Bell laboratories*.

[16] Kiriansky, Vladimir. 2003. Secure execution environment via program shepherding. Master's thesis, Massachusetts Institute of Technology.

[17] Loosemore, Sandra. 2004. *GNU C library application fundamentals*. GNU Press, Free Software Foundation.

[18] Novillo, Diego. 2003. Tree SSA - A new optimization infrastructure for GCC. In *Proceedings of the 2003 gcc developers summit*, 181–195. Ottawa, Canada.

[19] ———. 2004. Using GCC as a research compiler. Presentation slides from a seminar at North Carolina State University. `http://people.redhat.com/dnovillo/Papers/gcc-ncsu2004/`.

[20] Patterson, Jason R. C. 1995. Accurate static branch prediction by value range propagation. In *SIGPLAN conference on programming language design and implementation*, 67–78.

[21] Ramalingam, Ganesan. 1994. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems* 1467–1471.

[22] Rice, H. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society* 83.

[23] Rosen, B. K., M. N. Wegman, and F. K. Zadeck. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on principles of programming languages*, 12–27. ACM Press.

[24] Secure Software. RATS – rough auditing tool for security. `http://www.securesoftware.com/resources/tools.html`.

[25] SecurityFocus. SecurityFocus vulnerability archive. `http://www.securityfocus.com/bid`.

[26] Shankar, Umesh, Kunal Talwar, Jeffrey S. Foster, and David Wagner. 2001. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th usenix security symposium*, 201–220.

[27] Spafford, Eugene H. 1988. The internet worm program: An analysis. Tech. Rep. CSD-TR-823, Purdue University.

[28] Sridharan, B. 1992. An analysis framework for the McCAT compiler. Master's thesis, School of Computer Science, McGill University.

[29] Strasser, Stefan. MetaC++. `http://www-user.uni-bremen.de/ strasser/metacpp/`.

[30] Turing, Alan. 1937. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* 42:230–265.

[31] United States Computer Emergency Rediness Team. 2004. Technical cyber security alerts. `http://www.us-cert.gov/cas/techalerts/`.

[32] Viega, John, J. T. Bloch, Tadayoshi Kohno, and Gary McGraw. 2002. ITS4: A static vulnerability scanner for C and C++ code. *ACM Transactions on Information and System Security* 5(2).

[33] Wagner, David, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. 2000. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and distributed system security symposium*, 3–17. `San Diego, CA, USA`.

[34] Wegman, Mark N., and F. Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems* 13(2):181–210.

[35] Wheeler, David A. Flawfinder. `http://www.dwheeler.com/flawfinder/`.

Appendix A

Vulncheck Source Code

```
/* Vulnerability analysis pass for the GNU compiler.
   Copyright (C) 2005 Alexander Sotirov

This file is part of GCC.

GCC is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by the
Free Software Foundation; either version 2, or (at your option) any
later version.

GCC is distributed in the hope that it will be useful, but WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
for more details.

You should have received a copy of the GNU General Public License
along with GCC; see the file COPYING.  If not, write to the Free
Software Foundation, 59 Temple Place - Suite 330, Boston, MA
02111-1307, USA.  */

/* Vulnerability analysis.  */

#include "config.h"
#include "system.h"
#include "coretypes.h"
#include "tm.h"
#include "tree.h"
#include "flags.h"
#include "rtl.h"
#include "tm_p.h"
#include "ggc.h"
```

```
#include "basic-block.h"
#include "output.h"
#include "errors.h"
#include "expr.h"
#include "function.h"
#include "diagnostic.h"
#include "timevar.h"
#include "tree-dump.h"
#include "tree-flow.h"
#include "tree-pass.h"
#include "langhooks.h"
#include "toplev.h"

/* Use the TREE_VISITED bitflag to mark statements and PHI nodes that
   should not be simulated again.  */
#define DONT_SIMULATE_AGAIN(T)  TREE_VISITED (T)

/* Use the TREE_DEPRECATED bitflag to mark statements that have been
   added to one of the SSA edges worklists.  */
#define STMT_IN_SSA_EDGE_WORKLIST(T)    TREE_DEPRECATED (T)

/* Possible lattice values.  */
typedef enum
{
  NOT_TAINTED = 0,
  TAINTED
} latticevalue;

/* Main structure for taint analysis.  Contains the lattice value.  */
typedef struct
{
  latticevalue lattice_val;
} value;

/* This is used to track the current value of each variable.  */
static value *value_vector;

/* Worklist of SSA edges which will need reexamination as their
   definition has changed.  SSA edges are def-use edges in the SSA
   web.  For each D-U edge, we store the target statement or PHI node
   U.  */
static GTY(()) VEC(tree) *ssa_edges_worklist;

/* Parameter annotation flags.  */
#define USER_DATA   (1 << 0)   /* returns user data */
```

```
#define VULN_USER   (1 << 1)   /* vulnerable if passed user data */
#define VULN_RANGE  (1 << 2)   /* vulnerable if outside a range */
#define VULN_FUNC   (1 << 3)   /* vulnerable function by design */


/* Parameter annotations.  */
struct vuln_ann_d;

typedef struct vuln_ann_d
{
  int flags;
  int argnum;

  struct vuln_ann_d *next;
} vuln_ann;

/* Function annotations.  */
typedef struct vuln_func_d
{
  /* Function name.  */
  char *name;

  /* Parameter annotations, chained by the next field.  */
  vuln_ann *ann;

  /* The hashcode for this function name, cached for speed reasons. */
  hashval_t hashcode;
} vuln_func;

/* This hash table contains function annotations.  */
static htab_t vuln_func_table;

/* The name of the file containing vulnerable functions definitions,
   from the -fvulncheck-data argument.  */

const char *vulncheck_data_file_name = NULL;


/* Return the hash value of a vuln_func structure.  */

static hashval_t
vuln_func_hash (const void *p)
{
  const vuln_func *func = (vuln_func *) p;
  return func->hashcode;
```

```
}


/* Return true if two vuln_func structures are the same.  */

static int
vuln_func_eq (const void *p1, const void *p2)
{
  const vuln_func *func1 = (vuln_func *) p1;
  const vuln_func *func2 = (vuln_func *) p2;

  if (strcmp(func1->name, func2->name) == 0)
    return true;
  else
    return false;
}


/* Create a vuln_func_table entry.  */

static vuln_func *
create_vuln_func (char *name)
{
  vuln_func *func;
  void **slot;

  func = xmalloc (sizeof (vuln_func));
  func->name = xstrdup (name);
  func->ann = NULL;
  func->hashcode = htab_hash_string (name);

  slot = htab_find_slot_with_hash (vuln_func_table, func,
                                   func->hashcode,
                                   INSERT);

  if (*slot)
    free (*slot);
  *slot = (void *) func;

  return func;
}


/* Return the vuln_func_table entry for a FUNCTION_DECL, if found,
   NULL otherwise.  */
```

```
static vuln_func *
get_vuln_func (tree fndecl)
{
  tree decl_name;
  char *name;
  int len;
  vuln_func func;
  void **slot;

  /* Get the identifier name and length */

  gcc_assert (TREE_CODE (fndecl) == FUNCTION_DECL);

  decl_name = DECL_NAME (fndecl);
  gcc_assert (TREE_CODE (decl_name) == IDENTIFIER_NODE);

  name = (char *) IDENTIFIER_POINTER (decl_name);
  len = IDENTIFIER_LENGTH (decl_name);
  gcc_assert (strlen(name) == (size_t) len);

  /* Lookup the taint data for this identifier */

  func.name = name;
  func.hashcode = htab_hash_string (name);
  slot = htab_find_slot_with_hash (vuln_func_table, &func,
                                   func.hashcode,
                                   NO_INSERT);
  if (!slot)
    return NULL;
  else
    return *slot;
}


/* Finds the previous token in BUF, starting backwards from P.
   The length of the token is returned in LEN.  If a token is not
   found, returns NULL.  */

static char *
prev_token (char *buf, char *p, int *len)
{
  char *end;

  if (p == buf)
```

```
    return NULL;

  p--;

  while (*p == ' ' || *p == '\t' || *p == '*')
    {
      if (p == buf)
        return NULL;
      p--;
    }

  end = p;

  while (((*p >= 'a' && *p <= 'z') || (*p >= 'A' && *p <= 'Z')
          || (*p >= '0' && *p <= '9') || (*p == '_') || (*p == '.'))
         && (p >= buf))
    p--;

  *len = end - p;

  if (*len == 0)
    return NULL;
  else
    return p + 1;
}


/* Parse a file containing vulnerable functions defitions, creating
   new entries in the vuln_func_table hashtable.  */

static void
parse_vulncheck_data_file (const char *filename)
{
  FILE *file;
  char buf[1024];
  int line = 0;

  file = fopen (filename, "r");
  if (!file)
    fatal_error ("can%'t open taint data file %s: %m", filename);

  while (fgets(buf, sizeof(buf), file))
    {
      char *name, *p, *start;
      int len;
```

```
int argnum;

vuln_func *func;
vuln_ann **ann;

line++;

if ((p = strrchr(buf, '#')))
  *p = '\0';

p = buf;
while (*p && (*p == ' ' || *p == '\t'
              || *p == '\r' || *p == '\n'))
  p++;

if (*p == '\0')
  continue;

start = p;

if (!(p = strchr (buf, '(')))
  goto parse_error;

name = prev_token (buf, p, &len);
if (!name)
  goto parse_error;
name[len] = '\0';

func = create_vuln_func (name);
ann = &(func->ann);

if ((strncmp (start, "__USER_DATA__", 13) == 0)
     || (strncmp (start, "__VULN_FUNC__", 13) == 0))
  {
    *ann = xmalloc (sizeof (vuln_ann));
    (*ann)->argnum = 0;
    (*ann)->flags = 0;
    (*ann)->next = NULL;

    if (strncmp(start, "__USER_DATA__", 13) == 0)
      (*ann)->flags = USER_DATA;
    else if (strncmp(start, "__VULN_FUNC__", 13) == 0)
      (*ann)->flags = VULN_FUNC;

    ann = &((*ann)->next);
```

```
      }

    p++;
    argnum = 1;

    while (*p != ')' && *p != ';')
      {
        char *start = p;
        char *arg;

        if (!(p = strchr (start,',')) && !(p = strchr (start,')')))
          goto parse_error;

        arg = prev_token (start, p, &len);
        if (!arg)
          goto parse_error;
        arg[len] = '\0';

        if ((strcmp (arg, "__USER_DATA__") == 0)
            || (strcmp (arg, "__VULN_USER__") == 0)
            || (strcmp (arg, "__VULN_RANGE__") == 0))
          {
            *ann = xmalloc (sizeof (vuln_ann));
            (*ann)->argnum = argnum;
            (*ann)->flags = 0;
            (*ann)->next = NULL;

            if (strcmp(arg, "__USER_DATA__") == 0)
              (*ann)->flags = USER_DATA;
            else if (strcmp(arg, "__VULN_USER__") == 0)
              (*ann)->flags = VULN_USER;
            else if (strcmp(arg, "__VULN_RANGE__") == 0)
              (*ann)->flags = VULN_RANGE;

            ann = &((*ann)->next);
          }

        p++;
        argnum++;
      }
  }

fclose (file);
return;
```

```
parse_error:
    fatal_error ("parse error on line %d in taint data file %s",
                 line, filename);
  return;
}


/* Load the vulnerable function defitions into the vuln_func_table
  hashtable.  */

static void
load_vulncheck_data (void)
{
  vuln_func_table = htab_create (100, vuln_func_hash, vuln_func_eq,
                                 NULL);

  if (vulncheck_data_file_name)
    parse_vulncheck_data_file (vulncheck_data_file_name);
  else
    parse_vulncheck_data_file ("/usr/share/gcc/vulncheck.data");

  return;
}


/* Get the main expression from statement STMT.  */

static tree
get_rhs (tree stmt)
{
  enum tree_code code = TREE_CODE (stmt);

  switch (code)
    {
    case RETURN_EXPR:
      stmt = TREE_OPERAND (stmt, 0);
      if (!stmt || TREE_CODE (stmt) != MODIFY_EXPR)
        return stmt;
      /* FALLTHRU */

    case MODIFY_EXPR:
      stmt = TREE_OPERAND (stmt, 1);
      if (TREE_CODE (stmt) == WITH_SIZE_EXPR)
        return TREE_OPERAND (stmt, 0);
      else
```

```
        return stmt;

    case COND_EXPR:
      return COND_EXPR_COND (stmt);
    case SWITCH_EXPR:
      return SWITCH_COND (stmt);
    case GOTO_EXPR:
      return GOTO_DESTINATION (stmt);
    case LABEL_EXPR:
      return LABEL_EXPR_LABEL (stmt);

    default:
      return stmt;
    }
}


/* Dump all operands */

static void
dump_all_operands (tree stmt)
{
  tree def, use;
  use_operand_p use_p;
  def_operand_p def_p;
  ssa_op_iter iter;

  if (dump_file && (dump_flags & TDF_DETAILS))
    {
      fprintf(dump_file, "    DEF:");
      FOR_EACH_SSA_TREE_OPERAND (def, stmt, iter, SSA_OP_DEF)
        {
          fprintf(dump_file, " ");
          print_generic_expr (dump_file, def, 0);
          if (TREE_CODE (def) != SSA_NAME)
            fprintf(dump_file, " (%s)\n",
                    tree_code_name[(int) TREE_CODE (def)]);
        }
      fprintf(dump_file, "\n");

      fprintf(dump_file, "    USE:");
      FOR_EACH_SSA_TREE_OPERAND (use, stmt, iter, SSA_OP_USE)
        {
          fprintf(dump_file, " ");
          print_generic_expr (dump_file, use, 0);
```

```
    if (TREE_CODE (use) != SSA_NAME)
      fprintf(dump_file, " (%s)\n",
              tree_code_name[(int) TREE_CODE (use)]);
  }
fprintf(dump_file, "\n");

fprintf(dump_file, "    VUSE:");
FOR_EACH_SSA_TREE_OPERAND (use, stmt, iter, SSA_OP_VUSE)
  {
    fprintf(dump_file, " ");
    print_generic_expr (dump_file, use, 0);
    if (TREE_CODE (use) != SSA_NAME)
      fprintf(dump_file, " (%s)\n",
              tree_code_name[(int) TREE_CODE (use)]);
  }
fprintf(dump_file, "\n");

fprintf(dump_file, "    VMUSTDEF:");
FOR_EACH_SSA_TREE_OPERAND (def, stmt, iter, SSA_OP_VMUSTDEF)
  {
    fprintf(dump_file, " ");
    print_generic_expr (dump_file, def, 0);
    if (TREE_CODE (def) != SSA_NAME)
      fprintf(dump_file, " (%s)\n",
              tree_code_name[(int) TREE_CODE (def)]);
  }
fprintf(dump_file, "\n");

fprintf(dump_file, "    VMAYDEF:");
FOR_EACH_SSA_MAYDEF_OPERAND (def_p, use_p, stmt, iter)
  {
    def = DEF_FROM_PTR (def_p);
    use = USE_FROM_PTR (use_p);

    fprintf(dump_file, " ");
    print_generic_expr (dump_file, def, 0);
    if (TREE_CODE (def) != SSA_NAME)
      fprintf(dump_file, " (%s)",
              tree_code_name[(int) TREE_CODE (def)]);

    fprintf(dump_file, " = ");
    print_generic_expr (dump_file, use, 0);
    if (TREE_CODE (use) != SSA_NAME)
      fprintf(dump_file, " (%s)",
              tree_code_name[(int) TREE_CODE (use)]);
```

```
        }
      fprintf(dump_file, "\n");
    }
}


/* Dump lattice value VAL to file OUTF prefixed by PREFIX.  */

static void
dump_lattice_value (FILE *outf, const char *prefix, value val)
{
  switch (val.lattice_val)
    {
    case NOT_TAINTED:
      fprintf (outf, "%sNOT TAINTED", prefix);
      break;
    case TAINTED:
      fprintf (outf, "%sTAINTED", prefix);
      break;
    default:
      gcc_unreachable ();
    }
}


/* Get the constant value associated with variable VAR.  */

static value *
get_value (tree var)
{
  value *val;

  gcc_assert (TREE_CODE (var) == SSA_NAME);

  val = &value_vector[SSA_NAME_VERSION (var)];
  return val;
}


/* Set the lattice value for variable VAR to VAL.  Return true if VAL
   is different from VAR's previous value.  */

static bool
set_lattice_value (tree var, value val)
{
```

```
  value *old = get_value (var);

  /* TAINTED->NOT_TAINTED is not a valid state transition.  */
  gcc_assert (!(old->lattice_val == TAINTED
                && val.lattice_val == NOT_TAINTED));

  if (old->lattice_val != val.lattice_val)
    {
      if (dump_file && (dump_flags & TDF_DETAILS))
        {
          fprintf (dump_file, "Lattice value of ");
          print_generic_expr (dump_file, var, TDF_SLIM);
          dump_lattice_value (dump_file, " changed to ", val);
          fprintf (dump_file, "\n");
        }

      *old = val;
      return true;
    }

  return false;
}


/* Add all immediate uses of VAR to SSA_EDGES_WORKLIST.  */

static void
add_ssa_edge (tree var)
{
  tree stmt = SSA_NAME_DEF_STMT (var);
  dataflow_t df = get_immediate_uses (stmt);
  int num_uses = num_immediate_uses (df);
  int i;

  for (i = 0; i < num_uses; i++)
    {
      tree use_stmt = immediate_use (df, i);

      if (!DONT_SIMULATE_AGAIN (use_stmt)
          && !STMT_IN_SSA_EDGE_WORKLIST (use_stmt))
        {
          STMT_IN_SSA_EDGE_WORKLIST (use_stmt) = 1;
          VEC_safe_push (tree, ssa_edges_worklist, use_stmt);

          if (dump_file && (dump_flags & TDF_DETAILS))
```

```
            {
              fprintf(dump_file, "Adding ");
              print_generic_expr (dump_file, use_stmt, 0);
              fprintf(dump_file, " to SSA_EDGES_WORKLIST\n");
            }
        }
    }
}


/* Visit the call statement STMT.  If it is one of the predefined
   tainting functions, mark its DEFs as TAINTED.  */

static void
visit_call (tree stmt)
{
  tree rhs, callee;
  vuln_func *func;
  vuln_ann *ann;

  rhs = get_rhs (stmt);
  gcc_assert (TREE_CODE (rhs) == CALL_EXPR);

  /* Determine the called function.  */
  callee = get_callee_fndecl (rhs);
  if (!callee)
    return;

  if (dump_file && (dump_flags & TDF_DETAILS))
    {
      tree op;
      tree new;
      int i;

      fprintf (dump_file, "Visiting call: ");
      print_generic_expr (dump_file, stmt, TDF_SLIM);
      fprintf (dump_file, " (%s)\n",
               tree_code_name[(int) TREE_CODE (stmt)]);
      if (rhs != stmt)
        {
          fprintf (dump_file, "          RHS: ");
          print_generic_expr (dump_file, rhs, TDF_SLIM);
          fprintf (dump_file, " (%s)\n",
                   tree_code_name[(int) TREE_CODE (rhs)]);
        }
```

```
    dump_all_operands (stmt);

    new = get_call_return_operands (stmt);
    if (new)
      {
        fprintf(dump_file, "  ret:\n");
        dump_all_operands (new);
        ggc_free (new);
      }

    i = 0;
    for (op = TREE_OPERAND (rhs, 1); op; op = TREE_CHAIN (op))
      {
        fprintf(dump_file, "  op %d:\n", i);
        new = get_call_arg_operands (stmt, i);
        if (new)
          {
            dump_all_operands (new);

            ggc_free (new);
          }

        i++;
      }
  }

/* Get the vuln_func_table entry for the called function.  */
func = get_vuln_func (callee);
if (!func)
  return;

ann = func->ann;
while (ann)
  {
    if (ann->flags & USER_DATA)
      {
        tree ops;

        if (ann->argnum == 0)
          ops = get_call_return_operands (stmt);
        else
          ops = get_call_arg_operands (stmt, ann->argnum);

        if (ops)
          {
```

```
              ssa_op_iter iter;
              tree def;

              FOR_EACH_SSA_TREE_OPERAND (def, ops, iter,
                                         SSA_OP_ALL_DEFS)
                {
                  value val;
                  val.lattice_val = TAINTED;

                  gcc_assert (TREE_CODE (def) == SSA_NAME);

                  /* Mark the operand as tainted.  */
                  if (set_lattice_value (def, val))
                    add_ssa_edge (def);
                }
            }
        }
      ann = ann->next;
    }

  return;
}


/* Initialize local data structures for vulnerability analysis.  */

static void
vulncheck_initialize (void)
{
  basic_block bb;
  block_stmt_iterator i;

  /* Load the vulnerable functions definitions.  */
  load_vulncheck_data ();

  if (dump_file && (dump_flags & TDF_DETAILS))
    {
      dump_all_value_ranges (dump_file);
    }

  /* Initialize data structures for data flow propagation.  */
  value_vector = (value *) xmalloc (num_ssa_names * sizeof (value));
  memset (value_vector, 0, num_ssa_names * sizeof (value));

  /* Worklist of SSA edges.  */
```

```
ssa_edges_worklist = VEC_alloc (tree, 20);

/* Clear simulation flags for all statements and PHI nodes.  */
FOR_EACH_BB (bb)
  {
    tree phi;

    for (i = bsi_start (bb); !bsi_end_p (i); bsi_next (&i))
      {
        tree stmt = bsi_stmt (i);
        tree rhs = get_rhs (stmt);

        STMT_IN_SSA_EDGE_WORKLIST (stmt) = false;

        if (TREE_CODE (stmt) == MODIFY_EXPR
            || TREE_CODE (stmt) == CALL_EXPR)
          DONT_SIMULATE_AGAIN (stmt) = false;
        else
          DONT_SIMULATE_AGAIN (stmt) = true;

        /* Disable call clobbered operands.  */
        if (TREE_CODE (rhs) == CALL_EXPR)
          remove_clobbered_operands (stmt);
      }

    for (phi = phi_nodes (bb); phi; phi = PHI_CHAIN (phi))
      {
        STMT_IN_SSA_EDGE_WORKLIST (phi) = false;
        DONT_SIMULATE_AGAIN (phi) = false;
      }
  }

/* Invalidate dataflow information for the function.  */
free_df ();

/* Compute immediate uses for all variables.  */
compute_immediate_uses (TDFA_USE_OPS | TDFA_USE_VOPS, NULL);

if (dump_file && (dump_flags & TDF_DETAILS))
  dump_immediate_uses (dump_file);

/* Seed the algorithm by finding calls to vulnerable functions.  */
FOR_EACH_BB (bb)
  {
    for (i = bsi_start (bb); !bsi_end_p (i); bsi_next (&i))
```

```
        {
          tree stmt = bsi_stmt (i);
          tree rhs = get_rhs (stmt);

          if (dump_file && (dump_flags & TDF_DETAILS))
            {
              fprintf (dump_file, "Visiting statement: ");
              print_generic_expr (dump_file, stmt, TDF_SLIM);
              fprintf (dump_file, " (%s)\n",
                       tree_code_name[(int) TREE_CODE (stmt)]);
              if (rhs != stmt)
                {
                  fprintf (dump_file, "               RHS: ");
                  print_generic_expr (dump_file, rhs, TDF_SLIM);
                  fprintf (dump_file, " (%s)\n",
                           tree_code_name[(int) TREE_CODE (rhs)]);
                }
              dump_all_operands (stmt);
            }

          if (TREE_CODE (rhs) == CALL_EXPR)
            visit_call (stmt);

          if (dump_file && (dump_flags & TDF_DETAILS))
            fprintf (dump_file, "\n");
        }
    }
}


/* Evaluate statement STMT.

   If any operands of STMT are TAINTED, then return TAINTED.

   Else return NON_TAINTED.  */

static value
evaluate_stmt (tree stmt)
{
  value rval;
  tree use;
  ssa_op_iter iter;

  rval.lattice_val = NOT_TAINTED;
```

```
  get_stmt_operands (stmt);

  FOR_EACH_SSA_TREE_OPERAND (use, stmt, iter, SSA_OP_ALL_USES)
    {
      value *val = get_value (use);

      if (val->lattice_val == TAINTED) {
          rval.lattice_val = TAINTED;
          break;
        }
    }

  return rval;
}


/* Compute the meet operator between VAL1 and VAL2:

                any M TAINTED              = TAINTED
                NOT_TAINTED M NOT_TAINTED = NOT_TAINTED */
static value
taint_lattice_meet (value val1, value val2)
{
  value result;

  /* any M TAINTED = TAINTED.  */
  if (val1.lattice_val == TAINTED || val2.lattice_val == TAINTED)
    {
      result.lattice_val = TAINTED;
      return result;
    }

  /* NOT_TAINTED M NOT_TAINTED = NOT_TAINTED */
  result.lattice_val = NOT_TAINTED;
  return result;
}


/* Loop through the PHI_NODE's parameters for BLOCK and compare their
   lattice values to determine PHI_NODE's lattice value.  The value
   of a PHI node is determined calling taint_lattice_meet() with all
   the argument of the PHI node that are incoming via executable
   edges.  */

static void
```

```
taint_visit_phi_node (tree phi)
{
  value new_val, *old_val;
  int i;

  if (dump_file && (dump_flags & TDF_DETAILS))
    {
      fprintf (dump_file, "Visiting PHI node: ");
      print_generic_expr (dump_file, phi, dump_flags);
      fprintf (dump_file, "\n");
    }

  old_val = get_value (PHI_RESULT (phi));
  new_val = *old_val;

  for (i = 0; i < PHI_NUM_ARGS (phi); i++)
    {
      edge e;
      tree rdef;
      value *rdef_val, val;

      /* Compute the meet operator over all the PHI arguments.  */
      e = PHI_ARG_EDGE (phi, i);

      if (dump_file && (dump_flags & TDF_DETAILS))
        {
          fprintf (dump_file,
              "\n    Argument #%d (%d -> %d)\n",
              i, e->src->index, e->dest->index);
        }

      /* Compute the meet operator for the existing value of the PHI
         node and the current PHI argument.  */
      rdef = PHI_ARG_DEF (phi, i);

      if (is_gimple_min_invariant (rdef))
        {
          val.lattice_val = NOT_TAINTED;
          rdef_val = &val;
        }
      else
        rdef_val = get_value (rdef);

      new_val = taint_lattice_meet (new_val, *rdef_val);
```

```
      if (dump_file && (dump_flags & TDF_DETAILS))
        {
          fprintf (dump_file, "\t");
          print_generic_expr (dump_file, rdef, dump_flags);
          dump_lattice_value (dump_file, "\tValue: ", *rdef_val);
          fprintf (dump_file, "\n");
        }

      if (new_val.lattice_val == TAINTED)
        break;
    }

  if (dump_file && (dump_flags & TDF_DETAILS))
    {
      dump_lattice_value (dump_file, "    PHI node value: ", new_val);
      fprintf (dump_file, "\n");
    }

  /* Make the transition to the new value.  */
  if (set_lattice_value (PHI_RESULT (phi), new_val))
    add_ssa_edge (PHI_RESULT (phi));

  if (dump_file && (dump_flags & TDF_DETAILS))
    fprintf (dump_file, "\n");

  return;
}


/* Visit the assignment statement STMT.  If any of its operands are
   tainted, mark its LHS as TAINTED  */

static void
visit_assignment (tree stmt)
{
  tree def;
  ssa_op_iter iter;
  value val;

  /* Evaluate the statement.  */
  val = evaluate_stmt (stmt);

  FOR_EACH_SSA_TREE_OPERAND (def, stmt, iter, SSA_OP_ALL_DEFS)
    {
      gcc_assert (TREE_CODE (def) == SSA_NAME);
```

```
      /* Make the transition to the new value.  */
      if (set_lattice_value (def, val))
        add_ssa_edge (def);
    }

}

static void
taint_handle_vmaydef (tree stmt)
{
  tree def, use;
  use_operand_p use_p;
  def_operand_p def_p;
  ssa_op_iter iter;
  value* val;

  /* Handle VMAYDEF */

  FOR_EACH_SSA_MAYDEF_OPERAND (def_p, use_p, stmt, iter)
    {
      def = DEF_FROM_PTR (def_p);
      use = USE_FROM_PTR (use_p);

      val = get_value (use);

      if (val->lattice_val == TAINTED)
        {
          gcc_assert (TREE_CODE (def) == SSA_NAME);

          /* Make the transition to the new value.  */
          if (set_lattice_value (def, *val))
            add_ssa_edge (def);
        }
    }
}

/* Evaluate statement STMT.  If the statement produces an output value
   and its evaluation changes the lattice value of its output, return
   SSA_PROP_INTERESTING and set *OUTPUT_P to the SSA_NAME holding the
   output value.  */

static void
taint_visit_stmt (tree stmt)
{
```

```
  tree rhs;

  if (dump_file && (dump_flags & TDF_DETAILS))
    {
      fprintf (dump_file, "Visiting statement: ");
      print_generic_stmt (dump_file, stmt, TDF_SLIM);
      fprintf (dump_file, "\n");
      dump_all_operands (stmt);
    }

  taint_handle_vmaydef (stmt);

  rhs = get_rhs (stmt);

  if (TREE_CODE (stmt) == MODIFY_EXPR && TREE_CODE (rhs) != CALL_EXPR)
    {
      /* If the statement is an assignment evaluate its RHS to see
         if the lattice value of its output has changed.  */
      visit_assignment (stmt);
    }
  else
    {
      /* Any other kind of statement is not interesting for taint
         analysis and, therefore, not worth simulating.  */
      if (dump_file && (dump_flags & TDF_DETAILS))
        fprintf (dump_file, "Not an interesting statement.\n");
    }

  if (dump_file && (dump_flags & TDF_DETAILS))
    fprintf (dump_file, "\n");

  return;
}


static void
taint_propagate (void)
{
  if (dump_file && (dump_flags & TDF_DETAILS))
    fprintf (dump_file, "Processing SSA_EDGES_WORKLIST\n\n");

  while (VEC_length (tree, ssa_edges_worklist) > 0)
    {
      tree stmt = VEC_pop (tree, ssa_edges_worklist);
```

```
      /* STMT is no longer in a worklist.  */
      STMT_IN_SSA_EDGE_WORKLIST (stmt) = 0;

      if (DONT_SIMULATE_AGAIN (stmt))
        {
          if (dump_file && (dump_flags & TDF_DETAILS))
            {
              fprintf (dump_file, "Not visiting ");
              print_generic_stmt (dump_file, stmt, TDF_SLIM);
              fprintf (dump_file, "because of DONT_SIMULATE_AGAIN\n");
            }
          return;
        }

      if (TREE_CODE (stmt) == PHI_NODE)
        taint_visit_phi_node (stmt);
      else
        taint_visit_stmt (stmt);
    }
}


static void
report_vulnerable_call (vuln_func *func, tree stmt)
{
  vuln_ann *ann = func->ann;

  while (ann)
  {
    if (ann->flags & VULN_FUNC)
      {
        location_t *locus = EXPR_LOCUS (stmt);
        if (!locus)
          warning ("vulnerable statement at unknown location");
        else
          warning ("%Hvulnerable call to %s", locus, func->name);
      }
    else if (ann->flags & VULN_USER || ann->flags & VULN_RANGE)
      {
        tree ops;
        tree use;
        ssa_op_iter iter;

        ops = get_call_arg_operands (stmt, ann->argnum);
        if (!ops)
```

```
          continue;

        FOR_EACH_SSA_TREE_OPERAND (use, ops, iter, SSA_OP_ALL_USES)
          {
            value *val = get_value (use);

            if (val->lattice_val == TAINTED)
              {
                location_t *locus;

                if (ann->flags || VULN_RANGE)
                  {
                    value_range *vr = SSA_NAME_VALUE_RANGE (use);

                    /* If the value range of the argument is bounded,
                       do not report this call. */
                    if (vr && vr->type == VR_RANGE
                        && INTEGRAL_TYPE_P (TREE_TYPE (vr->max))
                        && vr->max != TYPE_MAX_VALUE (
                                      TREE_TYPE (vr->max)))
                      continue;
                  }

                locus = EXPR_LOCUS (stmt);
                if (!locus)
                  warning("vulnerable statement at unknown location");
                else
                  warning("%Hvulnerable call to %s", locus,
                          func->name);
              }
          }
      }
    ann = ann->next;
  }
}


static void
vulncheck_report (void)
{
  basic_block bb;

  FOR_EACH_BB (bb)
    {
      block_stmt_iterator i;
```

```
      for (i = bsi_start (bb); !bsi_end_p (i); bsi_next (&i))
        {
          tree stmt = bsi_stmt (i);
          tree rhs = get_rhs (stmt);

          if (TREE_CODE (rhs) == CALL_EXPR)
            {
              tree callee = get_callee_fndecl (rhs);
              vuln_func *func;

              /* If the called function cannot be determined,
                 skip it */
              if (!callee)
                continue;

              func = get_vuln_func (callee);
              if (func)
                report_vulnerable_call (func, stmt);
            }
        }
    }
}


/* Free allocated storage.  */

static void
vulncheck_finalize (void)
{
  basic_block bb;
  block_stmt_iterator i;

  free (value_vector);
  VEC_free (tree, ssa_edges_worklist);

  htab_delete (vuln_func_table);

  /* Invalidate dataflow information for the function.  */
  free_df ();

  /* Restore call clobbered operands.  */
  FOR_EACH_BB (bb)
    {
      for (i = bsi_start (bb); !bsi_end_p (i); bsi_next (&i))
```

```
        {
          tree stmt = bsi_stmt (i);
          tree rhs = get_rhs (stmt);

          if (TREE_CODE (rhs) == CALL_EXPR)
            restore_clobbered_operands (stmt);
        }
    }
}


/* Main entry point for the vulnerability analysis pass.  */

static void
execute_vulncheck (void)
{
  /* Initialize the vulnerability analysis.  */
  vulncheck_initialize ();

  /* Propagate the taint information.  */
  taint_propagate ();

  /* Report the results of the analysis.  */
  vulncheck_report ();

  /* Free allocated storage.  */
  vulncheck_finalize ();
}


static bool
gate_vulncheck (void)
{
  return flag_tree_vulncheck != 0;
}


struct tree_opt_pass pass_vulncheck =
{
  "vulncheck",                          /* name */
  gate_vulncheck,                       /* gate */
  execute_vulncheck,                    /* execute */
  NULL,                                 /* sub */
  NULL,                                 /* next */
  0,                                    /* static_pass_number */
```

```
  TV_TREE_VULNCHECK,                    /* tv_id */
  PROP_cfg | PROP_ssa | PROP_alias,     /* properties_required */
  0,                                    /* properties_provided */
  0,                                    /* properties_destroyed */
  0,                                    /* todo_flags_start */
  TODO_dump_func | TODO_verify_all,     /* todo_flags_finish */
  0                                     /* letter */
};
```

Appendix B

Glibc Annotations

```
#
# GNU C Library 2.2.x
#

# Based on The GNU C Library Reference Manual
# Edition 0.10, last updated 2001-07-06
# http://www.gnu.org/software/libc/manual/

#
# 3.2.2.1 Basic Memory Allocation
#

void * malloc (size_t size __VULN_RANGE__);

#
# 3.2.2.5 Allocating Cleared Space
#

void * calloc (size_t count __VULN_RANGE__,
               size_t eltsize __VULN_RANGE__);

#
# 5.4 Copying and Concatenation
#

void * memcpy (void *restrict to, const void *restrict from,
               size_t size __VULN_RANGE__);
wchar_t * wmemcpy (wchar_t *restrict wto,
                   const wchar_t *restruct wfrom,
```

```
                  size_t size __VULN_RANGE__);
void * mempcpy (void *restrict to, const void *restrict from,
                size_t size __VULN_RANGE__);
wchar_t * wmempcpy (wchar_t *restrict wto,
                    const wchar_t *restrict wfrom,
                    size_t size __VULN_RANGE__);
void * memmove (void *to, const void *from,
                size_t size __VULN_RANGE__);
wchar_t * wmemmove (wchar *wto, const wchar_t *wfrom,
                    size_t size __VULN_RANGE__);
void * memccpy (void *restrict to, const void *restrict from,
                int c, size_t size __VULN_RANGE__);
void * memset (void *block, int c, size_t size __VULN_RANGE__);
wchar_t * wmemset (wchar_t *block, wchar_t wc,
                   size_t size __VULN_RANGE__);

#
# 12.8 Character Input
#

__USER_DATA__ int fgetc (FILE *stream);
__USER_DATA__ wint_t fgetwc (FILE *stream);
__USER_DATA__ int fgetc_unlocked (FILE *stream);
__USER_DATA__ wint_t fgetwc_unlocked (FILE *stream);
__USER_DATA__ int getc (FILE *stream);
__USER_DATA__ wint_t getwc (FILE *stream);
__USER_DATA__ int getc_unlocked (FILE *stream);
__USER_DATA__ wint_t getwc_unlocked (FILE *stream);
__USER_DATA__ int getchar (void);
__USER_DATA__ wint_t getwchar (void);
__USER_DATA__ int getchar_unlocked (void);
__USER_DATA__ wint_t getwchar_unlocked (void);
__USER_DATA__ int getw (FILE *stream);

#
# 12.9 Line-Oriented Input
#

ssize_t getline (char **lineptr __USER_DATA__,size_t *n,FILE *stream);
ssize_t getdelim (char **lineptr __USER_DATA__, size_t *n,
                  int delimiter, FILE *stream);
char * fgets (char *s __USER_DATA__, int count, FILE *stream);
wchar_t * fgetws (wchar_t *ws __USER_DATA__, int count, FILE *stream);
char * fgets_unlocked (char *s __USER_DATA__,int count, FILE *stream);
wchar_t * fgetws_unlocked (wchar_t *ws __USER_DATA__, int count,
```

```
                            FILE *stream);
__VULN_FUNC__ char * gets (char *s __USER_DATA__);

#
# 12.11 Block Input/Output
#

size_t fread (void *data __USER_DATA__, size_t size, size_t count,
              FILE *stream);
size_t fread_unlocked (void *data __USER_DATA__, size_t size,
                       size_t count, FILE *stream);

#
# 12.12.17 Formatted Output Functions
#

int printf (const char *template __VULN_USER__, ...);
int wprintf (const wchar_t *template __VULN_USER__, ...);
int fprintf (FILE *stream, const char *template __VULN_USER__, ...);
int fwprintf (FILE *stream,const wchar_t *template __VULN_USER__,...);
int sprintf (char *s, const char *template __VULN_USER__, ...);
int swprintf (wchar_t *s, size_t size,
              const wchar_t *template __VULN_USER__, ...);
int snprintf (char *s, size_t size,
              const char *template __VULN_USER__, ...);

#
# 12.12.8 Dynamically Allocating Formatted Output
#

int asprintf (char **ptr, const char *template __VULN_USER__, ...);
int obstack_printf (struct obstack *obstack,
                    const char *template __VULN_USER__, ...);

#
# 12.12.9 Variable Arguments Output Functions
#

int vprintf (const char *template __VULN_USER__, va_list ap);
int vwprintf (const wchar_t *template __VULN_USER__, va_list ap);
int vfprintf (FILE *stream, const char *template __VULN_USER__,
              va_list ap);
int vfwprintf (FILE *stream, const wchar_t *template __VULN_USER__,
               va_list ap);
int vsprintf (char *s, const char *template __VULN_USER__,va_list ap);
```

```
int vswprintf (wchar_t *s, size_t size,
               const wchar_t *template __VULN_USER__, va_list ap);
int vsnprintf (char *s, size_t size,
               const char *template __VULN_USER__, va_list ap);
int vasprintf (char **ptr, const char *template __VULN_USER__,
               va_list ap);
int obstack_vprintf (struct obstack *obstack,
                     const char *template __VULN_USER__, va_list ap);

#
# 12.14.8 Formatted Input Functions
#

int scanf (const char *template, ... __USER_DATA__);
int wscanf (const wchar_t *template, ... __USER_DATA__);
int fscanf (FILE *stream, const char *template, ... __USER_DATA__);
int fwscanf (FILE *stream, const wchar_t *template,... __USER_DATA__);
int sscanf (const char *s, const char *template, ... __USER_DATA__);
int swscanf (const wchar_t *ws,
             const char *template, ... __USER_DATA__);

#
# 13.2 Input and Output Primitives
#

ssize_t read (int filedes, void *buffer __USER_DATA__, size_t size);
ssize_t pread (int filedes, void *buffer __USER_DATA__, size_t size,
               off_t offset);
ssize_t pread64 (int filedes, void *buffer __USER_DATA__, size_t size,
                 off64_t offset);

#
# 14.1 Working Directory
#

__USER_DATA__ char * getcwd (char *buffer __USER_DATA__, size_t size);
__VULN_FUNC__ __USER_DATA__ char * getwd (char *buffer __USER_DATA__);
__USER_DATA__ char * get_current_dir_name (void);

#
# 14.5 Symbolic Links
#

int readlink (const char *filename, char *buffer __USER_DATA__,
              size_t size);
```

```
#
# 18.2.2 syslog, vsyslog
#

void syslog (int facility_priority, char *format __VULN_USER__, ...);
void vsyslog (int facility_priority, char *format __VULN_USER__,
              va_list arglist);

#
# 25.4.1 Environment Access
#

__USER_DATA__ char * getenv (const char *name);
```