A Lightweight Security Analyzer inside GCC

Davide Pozza and Riccardo Sisto
Dip. di Automatica e Informatica
Politecnico di Torino
Corso Duca degli Abruzzi 24
I-10129 Torino, ITALY
Email:{davide.pozza, riccardo.sisto}@polito.it

Abstract

This paper describes the design and implementation of a lightweight static security analyzer that exploits the compilation process of the gcc compiler. The tool is aimed at giving to programmers useful and precise hints for improving the security of the developed software, while also detecting format string vulnerabilities, buffer overflows, and subtle vulnerabilities due to incorrect arithmetic and conversion on integers. The experimented technique is a combination of the taint analysis concept and of a value range propagation algorithm. The experimental results obtained by analyzing some real-world security critical programs show that the tool is only slightly heavier than pure compilation, and that it is able to detect known vulnerabilities, as well as unknown ones. Moreover, even if false positives are given, many of the warnings that do not correspond to vulnerabilities are indeed instances of unsafe programming practices, which can be avoided by applying a defensive programming style. Then, the tool can be profitably used during development, as a means that facilitates such coding practice.

1. Introduction

Software contains errors, because programmers are humans and, hence, they can easily make mistakes. The simple usage of a compiler to detect problems is insufficient. While it reports many errors with certainty, it allows many of them to pass unnoticed and to lie dormant for years before they are discovered and fixed. Unluckily, the longer a vulnerability is not patched, the more expensive it can be to fix it. Therefore, catching bugs early is better.

Current practices for bug detection apply testing, dynamic analysis and static analysis techniques. In essence, while each one has its own particular strengths, each of these techniques is not sufficient by itself to enable discovering all the errors, thus they usually need to be used in such a way that they complement each other weaknesses.

This paper focuses on static analysis techniques for the detection of security vulnerabilities.

The static program analysis problem of finding vulnerabilities, such as for example buffer overflows, is undecidable for languages like C and C++, because it is a Turing halting problem. Consequently, only approximate solutions can be obtained, unless analysis is in some way restricted to operate only on a decidable subset of programs. Depending on the conservativeness of the approximation and on how much approximation is used, there can be tools that provide more or less accurate results in that they either miss or do not miss errors or they produce more or less false positives.

Anyway, in general, better precision requires more analysis time. Therefore, besides the tool accuracy in detecting errors, another practical important issue is the time taken to perform the analysis. Tools that provide quite accurate results with few false positives typically take hours to perform the analysis. Then, they are useful for final assessment, but they are not adequate to be used regularly in the code development loops for short-term iterations. Instead, a lightweight tool that runs in minutes can be used during development just as the compiler is, and can help programmers to promptly detect errors, and to avoid bad coding practices. Of course, many of the warnings emitted by a lightweight tool may not correspond to existing bugs, but our experience shows that many times they indicate program points that can be coded in a safer and/or cleaner manner. So, by using lightweight tools during development, there is the side effect benefit of raising the reliability and security awareness of programmers, as well as to incentive the usage of defensive programming practices that lead to produce more robust code.

In the ambit of reliable and secure programs, code should obey some simple principles, such as: all inputs have to be validated before they are used, and defense has to be put in depth, e.g. there should be checks placed right before

0-7695-3102-4/08 $25.00 © 2008 IEEE
DOI 10.1109/ARES.2008.26

the most sensitive usages of values, aimed at ensuring that unwilling behaviors can never happen.

Based on the above considerations, this paper presents our experience in developing a new lightweight static analysis technique aimed at emitting warnings for the potentially dangerous usages of inputs in sensitive operations. The final aim is to facilitate the detection of some widespread vulnerabilities, such as format string vulnerabilities, buffer overflows and many integer-related errors that can be the cause of vulnerabilities. A tool prototype has been developed inside version 4.1 of the GCC suite of compilers, to target the C language. Anyway, the tool can be easily ported inside other versions, with minor modifications.

The rest of the paper is organized as follows. Next section recalls the classes of vulnerabilities that the presented analysis technique can detect and provides an explanation of their causes. Section 3 presents the analysis technique and section 4 describes some experimental results that have been obtained by running a prototype implementation of the technique on real world code. Finally, related works are described and conclusions are drawn.

2. Software Vulnerabilities

This section recalls the causes of the most commonly exploited C implementation vulnerabilities, i.e. format string vulnerabilities, buffer overflows, and integer related errors, which are more and more often used by attackers to trigger conditions that lead to vulnerabilities, such as buffer overflows.

A format string vulnerability [17] occurs when an attacker is able to specify all or part of the format string to a format function, such as: `*printf`, `syslog`, etc. By providing a format string containing unexpected format conversions an attacker can cause the function to output the contents of memory and/or to write arbitrary portions of memory with controlled values.

Most buffer overflow vulnerabilities [16][14] result from improper or missing buffer bound checks and/or misuse of some library functions (that work on buffers). However, some buffer overflow vulnerabilities are enabled by more subtle causes, such as the presence of integer errors in the code. Common causes are errors on the logic aimed at ensuring that operations on buffers operate inside their bounds and errors on size computation of buffers that have to be dynamically allocated.

There are two categories of integer operations that can lead to vulnerabilities, since they either cause a misinterpretation or a wrong computation of integer values. They are type conversions and integer underflows/overflows.

Integers can be subject to *type conversions* explicitly by means of *cast* operators or implicitly. Particular attention must be paid to implicit conversions that are introduced by compilers, since vulnerabilities are often caused by unaccounted effects due to these conversions.

Without going into the details of how and when integer conversion rules are applied, it is sufficient to note that an unsafe type conversion from `A` to `B` happens when all the values of `A` cannot be safely represented in `B`, because either it has a type with less width (i.e. less rank) or with different signedness.

When conversions are between types of the same width, but with different signs, *Signed-Unsigned errors* are possible. When a negative integer number is converted into an unsigned integer, it is interpreted as a large number, whereas when an unsigned integer is converted into a signed one and the number is too big to be representable as signed, it is interpreted as a negative number.

Indeed, for the conversions that involve types with different widths, it is possible to distinguish between two subcases: conversions from a larger type to a smaller one and viceversa.

A *truncation* or *loss of precision* cast arises when an integer is converted into a type with less width, being greater (for signed and unsigned integers) or smaller (for signed integers) than the maximum representable number. The consequence is that the most significant bits are lost.

When a conversion happens from an integer of smaller type to a larger one, the variable with less width is first converted to its equivalent value as a variable of greater width. It could seem that it is impossible to have problems with these conversions, since each number that is representable on a given type can be safely represented on a larger type. Unfortunately, combining these conversions with some other particular code can lead to vulnerabilities. For example, in the two's complement representation of negative numbers (the one used in GCC and in most compilers), the conversion implies sign extension. For example both `char c=0xFF; unsigned int i=c;` and `char c=0xFF; int i=c;` imply that `i` contains `0xFFFFFFFF`. If the programmer is not aware of the extension and uses some values (by assuming that they are not legitimate data values) of a variable (such as -1, i.e. `0xFFFFFFFF`), as special values into a conditional check, then it may happen that legitimate values will coincide with the special ones. We call *Extension vulnerabilities* the vulnerabilities that derive from this behavior. An example of a vulnerability (see CAN-2003-0161 or VU#897604 advisories) allowing to evade a security check by exploiting a cast from a char to an int and, consequently, causing a buffer overflow, has been discovered in the *Sendmail* server.

An *integer underflow/overflow* happens when the computation of an arithmetic operation results in a value that is too small or too big to be representable on the available bits. The C language standard [13] says that an integer overflow causes "undefined behavior". This means that compilers conforming to the standard can do anything, i.e. they either may completely ignore the overflow or abort the program. Most compilers (including GCC) simply ignore the overflow and silently store the result of the erroneous computation.

It is worth noting that integer overflows/underflows can be very perilous, since it is not possible to detect them after they have happened, mainly because when an integer overflow/underflow takes place, the program cannot distinguish between a correct and an incorrect computation. Therefore, safe programs should implement some extra logic to avoid such problems.

Integer underflows/overflows do not directly allow exploitation of programs. However, when for example integers are used in memory management functions (i.e. `*alloc`), exploitation conditions can arise.

3. The Static Analysis technique

All the vulnerabilities previously presented are made possible by at least one of the following conditions or a combination of them: (1) an attacker can control the content of a variable that is used into a security sensitive logic (e.g. a check) or a function call (e.g. a malloc() call); (2) data are not sufficiently validated (e.g. an integer variable is not checked to be into a range of allowed safe values); (3) integer variables are not bounded and arithmetic operations can cause them to underflow or overflow; (4) type conversions can cause misinterpretation of the value stored into variables.

Given that, a static analyzer that aims at detecting the vulnerabilities mentioned in the previous section should emit warnings founding its decision on the following information: (1) what are the variables that can be under the direct or indirect control of an attacker (taint variables), (2) what is the range of values that integer variables can take at run-time, (3) what integer variables can overflow/underflow, (4) what unsafe type conversions happen on variables.

The lightweight static analysis technique presented here is based on a dataflow analysis that computes the above information in a conservative way. Then, by exploiting the above knowledge, warnings are emitted according to some hard coded policies and some additional policies, specified by means of annotations of library function prototypes contained in a text file. We provide annotations for all the C standard library functions.

Figure 1 shows some annotations as an example. The lines following a function prototype are the annotations for that function. Consider the annotation for the `malloc` function, which starts with `@VULN`. This tag is used to describe the criteria used to decide whether a warning must be emitted. Inside each pair of round brackets there is a condition that has to be evaluated. Conditions are separated by OR (|) symbols and are ordered by importance, so they are evaluated in the specified order. Warnings are classified into five different ranks. The first condition (i.e. `(5: 1 VU & 1 VR & 1 VC)`) is ranked with level 5 (i.e. VERY HIGH) and is composed of a conjunction (&) of three sub-conditions. In this case, all the sub-conditions concern the first parameter (i.e. 1) of the function. The first sub-condition is true if the first parameter (i.e. 1) is controllable by a user (i.e. VU), the second one if the value of the variable is not bounded (i.e. VR), and the third one if an unsafe type conversion occurs along a path that reaches the function call (i.e. VC). Consider now the annotation of the `scanf` function. The `@VULN` tag here specifies that a warning with rank 4 (i.e. HIGH) must be emitted if the first parameter of the function contains a perilous format string (i.e. FS), e.g. it contains a %s that can lead the corresponding variable to overflow. The description of the `@INIT` and `@TP` tags will be illustrated later on.

```
void * malloc (size_t size);
@VULN: (5: 1 VU & 1 VR & 1 VC) |
 (4: 1 VU & 1 VR) | (3: 1 VR & 1 VC) | (2: 1 VR)

int printf (const char *template, ...);
@VULN: (5: 1 VU)

void * memcpy (void *restrict to, const void
                *restrict from, size_t size);
@VULN:(5: 2 VU & 3 VU & 3 VR & 3 VC)|(4: 2 VU & 3 VU &
        3 VR)|(3: 2 VU & 3 VU)|(3: 3 VU & 3 VR)
@TP: 2 > 1 & 2 > 0

int scanf (const char *template, ...);
@INIT: 0 & 2
@VULN: (4: 1 FS)
```

Figure 1. Annotation examples.

3.1. Determining taint variables

Determining the variables whose contents can be influenced by a potential attacker, is primarily aimed at narrowing the warnings emitted by the tool to the ones that really matter from a security perspective, as well as to reduce false positives. In fact, many potential vulnerabilities are not exploitable when the attacker has no control over certain data. For example, an attacker usually needs to have control over the data that overflows, in order to exploit a buffer overflow. The idea of determining such variables is not new and it has already been used with success in static analyzers for vulnerability detection (e.g. [18][12]).

We formulate the taint analysis problem as a dataflow problem, based on the lattice shown in figure 2. Differently form other approaches, we introduce the new value *may_be_tainted*, in addition to tainted and untainted values. This new value represents uncertainty about taintedness, so as to allow analysis to make conservative assumptions about

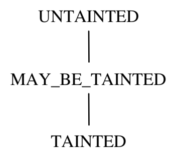

Figure 2. The lattice for taint analysis.

the fact that a variable can be tainted, but tracking this uncertain assumption to the user. For example, when there is a function call and the code of the function is not available, we can make the conservative assumption that the function returns a tainted value, however, since we cannot tell this for sure, the variable is marked as may_be_tainted instead of tainted.

It is worth noting that the lattice is unusual compared to those used in classical dataflow analysis, in that the top value, instead of representing a lack of information, represents the supposition that everything is initially untainted (i.e. not controllable by an attacker). This important particularity enables an efficient propagation of the taint property, since the propagation algorithm can start to propagate information from the input points of the program (e.g. input functions), without having to spend time to examine statements on the untainted paths of the program.

In practice, analysis starts by marking all the variables as untainted except the arguments of the main function, and those function parameters that are specified by the annotation file through the `@INIT` tag. All these are marked as tainted. Coming back to the annotation of the `scanf` function (see fig. 1), "`@INIT: 0 & 2`" means that the return parameter (0) of the function has to be marked as tainted and that the second parameter (2) of the function has to be marked as tainted. In such cases, the annotation also specifies that all the parameters after the second one have to become tainted, because of the ellipsis (`...`) in the function prototype.

Propagation is performed using an iterative worklist algorithm. All the statements that have been initially determined to use a tainted variable are pushed into the worklist. Then, propagation takes place by popping a statement at a time off the worklist and by propagating the taintedness of the variables of that statement. Hence, for each variable that has become tainted, the corresponding statements that immediately use them (they are determined by walking the definition-use chains) are added to the worklist, if not already present. Hence, propagation terminates when the worklist is empty.

We have implemented two taint analysis engines.

One analysis is performed as an inter-procedural compilation pass and allows whole program analysis on the GIMPLE form (which essentially is a three address code language with no high-level control flow structures). This analysis is performed quickly, being context-insensitive and flow-insensitive. However, the analysis is conservative and takes data aliasing into account by exploiting an implementation of the Andersen pointer analysis [2].The purpose of this intra-procedural analysis is only determining the taintedness value of all global variables and of the parameters of all program functions.

The other analysis is intra-procedural and more precise than the inter-procedural one in that it works on the SSA form and provides a limited form of context- and flow-sensitivity, while exploiting the already existing gcc implementation of an intra-procedural alias-analysis. This taint analysis can either work by exploiting the knowledge obtained by the inter-procedural one or as a stand-alone analysis.

When the intra-procedural analysis exploits the results of the inter-procedural one, the previously determined tainted parameters are marked as tainted, while the tainted global variables are marked as may_be_tainted when they were tainted. The reason for that change is that the intra-procedural analysis does not know when a global variable has become tainted (i.e. if at run-time it becomes tainted before, during, or only after the execution of the function). As a consequence, the analysis uses the *may_be_tainted* value instead of *tainted* to inform the user about the conservative but possibly imprecise assumption about the taintedness of the variable. Obviously, if the variable becomes tainted during function execution, then the global variable changes its lattice value from may_be_tainted to tainted during the intra-procedural analysis.

On the other hand, when the analysis is performed without exploiting the inter-procedural one, it works conservatively. In particular, the analysis initially assumes that all the parameters of functions and all global variables are may_be_tainted, so as to assume the worst case scenario, while reporting to the user the uncertainty about the taintedness assumptions. By considering the worst case scenario, the analysis results apply when verifying the code of libraries and of components during development, i.e when there is no knowledge or assurance on where and how functions are called (i.e. the function parameters that an attacker can control or influence are unknown). Therefore, this analysis is helpful to incentive programmers to produce robust code, by performing defensive programming.

The `@TP` annotation mechanism of function prototypes, that we define, enhances the capabilities of both taint analysis passes, because it takes the functionality of the annotated functions into account. In particular, these annotations specify how taint propagation can be performed in a stricter manner than the default conservative one. Consider the annotation `@TP: 2 > 1 & 2 > 0` of the `memcpy` function (see fig. 1). This annotation means that the taintedness of the second parameter (2) has to be propagated to the first parameter (1) and to the return parameter (0), and no other propagation must occur. Indeed, by default, the taintedness of the first parameter would be propagated to the second one and to the return value.

3.2. Determining the value ranges of integer variables

The problem of determining the range of values that integer variables can take at run-time is a well-known issue for compilers, since this knowledge can be used to predict what is the likelihood taking each particular branch of a program, which is useful to perform several optimizations.

The GCC already includes an implementation [15] of such an algorithm, called *value range propagation*. This algorithm can efficiently track the value ranges of variables through a program, by producing significantly more accurate predictions than heuristic techniques, while maintaining pretty linear runtime behavior.

As described [15], this method must be applied to dependence flow graphs where the variables have been renamed to achieve single assignment. Unluckily, the GCC implementation of this algorithm is only available for the intra-procedural optimization passes, since they work on an SSA form, while the inter-procedural passes work on GIMPLE. Therefore, our current prototype only uses intra-procedural value range propagation. It is worth noting that this implies some inability to determine that some variables have been bounded by some previous checks along the inter-procedural execution paths. By looking at vulnerability reports, it can be noted that many vulnerabilities were caused by missing checks along only few of the many execution paths that reach a given sensitive code point. Therefore, this analysis deficiency can cause some false positives, but it can be exploited to identify code points where a new safety check will make the program more resistent.

Anyway, in the future, GCC should allow inter-procedural passes to work on an SSA form, thus providing an inter-procedural value range propagation algorithm.

For our analysis, we made some slight modifications to the original GCC implementation of the value range propagation (VRP) algorithm, in order to allow considering variables as dangerous when they are unbounded, both because of lack of checks that limit their bounds and of integer underflows/overflows. It is worth noting that there is an important difference between signed and unsigned variables, since the former usually need to be lower and upper bounded, while the latter only need to be upper bounded. Thus, when a requirement about the range of a function specified by an annotation is evaluated, depending on the signedness of the parameter, a different check is performed.

The original version of the VRP algorithm represents integers in a lattice that is made of the following elements: *undefined* (nothing is known about the value range), *varying* (i.e. the integer can take any value), a *range* or an *anti-range* (the integer can only take values outside the specified range). Ranges and anti-ranges are expressed using constant bounds (e.g. [0;100]) or symbolic bounds (e.g. [x; x+1]). It has to be noted that a symbolic bound is produced when the value depends on the values of other variables with unknown bounds.

One of the modifications consists of changing the symbolic bounds containing arithmetic operators (such as x+1) into constant bounds or varying ranges. For example, we transform the range [2; x+1] into [2, MAX] where MAX is the maximum representable value in the variable type, while a range such as [x, x+1] is dropped to be varying. This conservative strategy lets us treat the variables that are unbounded, because not properly sanitized, and those that can underflow/overflow in the same way. To clarify this concept, consider what happens with ranges such as [2; x+1] and [2; x]. In both cases the range is changed into [2, MAX], since knowing that the variable can assume the maximum value is enough to evaluate the variable as dangerous (in terms of value range), despite of the overflow. Anyway, the information that the variable could overflow is maintained.

3.3. Tracking unsafe type conversions

Determining unsafe type conversions along tainted execution paths is performed by the taint analysis pass that, while tracking taintedness by following data flow information, also checks what casts are unsafe. This is done according to the casuistries presented in section 2.

Type conversion analysis emits an informative warning for each unsafe cast and maintains information about what are the types of all the unsafe casts that have been performed along the execution paths that reach each SSA variable. The types of casts, that are stored, belong to the following categories: truncation, unsigned to signed, signed to unsigned, and sign extension. Moreover, the analysis keeps a more detailed track of the last unsafe cast, in terms of code location (i.e. file and line), so as to allow cross referencing with informative warnings.

3.4. Putting analysis results together

Security warnings are emitted for library function usages that match what specified by function annotations, as well as for potentially dangerous array accesses and unsafe casts that reach conditional expressions.

Warnings for functions are heuristically ranked in such a way to have a high probability that real vulnerabilities are caught by only examining warnings ranked as VERY HIGH and HIGH, while those ranked as MID usually correspond to vulnerabilities that are induced by previous vulnerabilities, and the other ones are only useful in order to perform a quite deeper code auditing.

For example, consider the annotation for the `memcpy` function, as shown in figure 1. If only the criteria corresponding to a MID rank (i.e. 3) are met, the function is vulnerable because of some previous errors, which can be caught and ranked higher by our analyzer.

Warnings for accesses to arrays, such as *a[x] = b;* are ranked as: - VERY HIGH when x is tainted and unbounded, and b is tainted; - HIGH when x is unbounded and b is tainted; - MID when x is tainted and b is tainted, or when x is unbounded and b is untainted; - LOW when x is tainted. Here, however, for the arrays that are statically allocated, the analysis does not only look for the absence of bounds, but it performs a comparison of the bounds computed by the value range propagation algorithm with the real allocation sizes. Moreover, it is important to note that the compiler performs both an inter-procedural and an intra-procedural constant propagation algorithm [4] which can transform many variables (e.g. array indexes, and bounds of loops) into constants. Therefore, many more accurate checks concerning array accesses can be performed.

It is worth noting that the rank of warnings is not modified depending on the presence of unsafe casts that occur on index variables, but that this information is simply provided.

Warnings for conditionals are ranked using similar criteria, depending on the type of cast and on the comparison operator involved. In particular, truncation or unsigned-to-signed casts are ranked VERY HIGH when the operator is an inequality (e.g. $<$, $>$, $<=$, $>=$) and HIGH when the operator is an equality ($==$) or a difference ($!=$). Sign extension casts are ranked HIGH when the operator is an equality or a difference and MID when the operator is an inequality. Signed-to-unsigned casts are ranked HIGH when the operator is an inequality and MID when the operator is an equality.

4. Real-World Tests

The prototype of the presented analysis algorithm has been tested on 9 real-world network applications, since they are the typical kind of programs where vulnerabilities imply significant security risks.

Tab. 1 provides information about the programs that have been analyzed. For each program, the table reports the version used, the total number of lines (LOP), and the net number of non-comment, non-empty lines of code (LOC). Finally, it both reports the time taken by gcc to only perform compilation (CT) and the time taken by gcc with the integrated analysis (CAT). Tests were made with code optimization activated with option `-O2`, and were performed on a laptop equipped with a 2 Ghz Centrino and 1 GB of RAM, running Linux.

Program	Version	LOP	LOC	CT	CAT
nullhttpd	0.5.0	2254	1878	1,856s	2,365s
bftpd	1.5	4214	3666	4,693s	6,152s
cfingerd	1.4.1	5627	3702	6,908s	14,418s
muh	2.05d	5850	4278	2,981s	17,635s
dcd	0.7.0	8781	5935	4,920s	16,671s
net-tools	1.46	10878	8677	18,166s	27,741s
rsync	2.4.6	22277	15495	11,45s	26,643s

Table 1. Programs characteristics, and compilation and analysis times.

Some additional information is now provided about the programs on which the analyzer has been tested. Null-httpd is a very small, simple and multi-threaded web server for Linux and Windows. Version 0.5.0 is known to have a buffer overflow (see CVE-2002-1496). Bftpd (version 1.5 and further ones) is a small FTP server, that strives to be fast, secure and quick to install and configure, with no known vulnerabilities. Net-tools is a package that consists of several system commands related to networking, such as netstat, ifconfig, route, and so on. Version 1.46 is known to have many buffer overflow vulnerabilities. Muh is a smart IRC-bouncing tool (i.e. an IRC proxy) that remains on IRC all the time. Version 2.05d is known to have a format string vulnerability (see CVE-2000-0857). Cfingerd is aimed to be a free and secure finger daemon. Version 1.4.1 is known to have a buffer overflow (see CAN-2001-0735) and a format string vulnerability (see CAN-2001-0609). DConnect Daemon (dcd) is a Direct Connect's hub working as daemon. Version 0.7.0 is known to have a buffer overflow and multiple instances of a format string vulnerability (see SecurityTracker Alert ID: 1016641). Rsync is an utility that provides fast incremental file transfer for bringing remote files into sync. Version 2.4.6 is known to have many buffer overflows (see CVE-2003-0962, CVE-2004-2093, and CVE-2006-2083) and multiple signedness errors (see CVE-2002-0048).

Tab. 2 summarizes the experimental results we obtained by verifying the code for only warnings ranked as VERY HIGH or HIGH. Warnings have been grouped into categories, i.e. as those that involve Arrays, Functions, and Conditional expressions. For each category we distinguish among warnings that correspond to code locations that either cause or are real vulnerabilities/errors (*TP*, i.e. true positives), to code locations where we think that a security improvement should be performed (*PI*, i.e. Possible improvements), and to code locations that do not correspond

	nullhttpd	bftpd	cfingerd	muh	dcd	net-tools	rsync
Array TP	0	0	1	0	0	6	1
Array PI	0	0	0	0	3	0	1
Array FP	3	4	10	2	2	1	4
Function TP	1	7	6	2	2	14	14
Function PI	0	2	2	8	1	0	3
Function FP	2	14	7	8	7	11	8
Conditional TP	0	0	1	0	0	0	3
Conditional PI	0	3	2	1	2	1	17
Conditional FP	1	14	0	0	1	2	12
Total TP	1	7	8	2	2	20	18
Total PI	0	5	4	9	6	1	21
Total FP	6	32	17	10	10	14	24
Total warnings	7	44	29	21	18	35	63

Table 2. Summary of warnings ranked as HIGH and VERY HIGH.

to any real issue (*FP*, i.e. false positives). It is worth noting that we do not catalogue as PI the code locations that are signalled because of any analysis inaccuracy (e.g. an integer variable is rightly bounded because of checks external to the function) or because of ”unsafe” conversions and integer overflows that seem to be consciously taken into account by the programmer. We found that the major causes of false positives are these, as well as the ones due to inaccuracy when determining integer bounds. Inaccuracy may arise when arithmetic formulas are complex and/or involve the return value of functions (this is a limitation of having an intra-procedural value range propagation algorithm). In summary, the result is that, of a total number of 217 warnings, 27% correspond to TPs, 21% to PIs, and 52% to FPs. The warning rate is acceptable (a warning every 275 lines of code).

An important result is that all the known vulnerabilities have been caught by the tool prototype, and seven previously unknown vulnerabilities have been found (those of bftpd). Compilation and analysis takes, on average, approximately twice the time taken by compilation alone.

5. Related Work

Many vulnerability detection techniques for C code have been proposed in recent years. While many have focused on the detection of buffer overflows and/or format string vulnerabilities, few have taken the effects of integer overflows and of unsafe type conversions into account, both by themselves and as a trigger cause of buffer overflows.

Lexical analysis tools (e.g. ITS4 [19], VulCAn [20]) can aid guiding code auditing in order to spot vulnerabilities caused by library functions only. Moreover, since they only perform a lexical analysis and not a dataflow analysis, they leave a lot of manual work to code auditors and they can report many false positives.

Annotation driven tools (e.g. Splint [9], CSSV [8] and Eau Claire [5]) analyze program code guided by user-supplied and library-supplied annotations, which usually take the form of function pre-conditions and post-conditions that must respectively hold before and after function calls. Their main limitation is that users are usually not willing to annotate code. Anyway, depending on the underlying analysis engine, they can emit many false positives (e.g. Splint) or they can be accurate but very slow (e.g. CSSV).

Model-based tools work by extracting a model from source code and by analyzing it. Some of them (e.g. MJOLNIR [20], and ARCHER [21]) abstract away the code of programs to formulate a linear programming problem, in order to be scalable. Hence, constraint solvers are used and, when inconsistencies are detected, warnings are emitted. Our technique differs in that it is propagation based, instead of equation based, so it is typically faster. Furthermore, it has to be noted that the approaches of these model based tools are someway less conservative than the one described in this paper, since they report warnings only when there is evidence that operations happen out-of-bounds. Indeed, we always raise warnings when there is not evidence that an operation is “bounded” and we rank warnings according to risks. More accurate model-based tools (e.g. Astreé [7] and Polyspace C verifier [1]) are based on abstract interpretation techniques. Such tools provide very precise results, but they are very slow [22]. Some other model based tools (e.g. CBMC [6], UNO [11]) use model checking techniques and perform path-sensitive analysis of abstracted programs. Therefore, while they can provide precise information when they find a vulnerability, they suffer from scalability problems. Moreover, while they can typically report very few false positives, they usually detect less bugs.

Some compilers, such as gcc, already emit warnings for comparisons involving signed and unsigned variables, which inevitably lead to type conversions, but they do not provide any detection mechanism for buffer overflows and format string vulnerabilities. Moreover, detection of integer problems in gcc is quite limited, because it does not consider cases when the cast happens before comparison, and ignores truncation and extension casts. Furthermore, it does not attempt to determine if casts are safe because of the set of values that the variable cannot take, and do not reduce false positives because of untaintedness of cast variables. Besides the limited detection capability of compilers, we are not aware of other tools that emit warnings for unsafe conversions that reach conditional checks.

A compilation-based approach is Meta-level compilation [10]. Based on it, Coverity produces a commercial tool that detects input validation errors on integer variables. The tool [3] works by determining if tainted data is used in pointer arithmetic and in memory management functions, so as to report warnings. A remarkable difference is that [3] does not consider data aliasing, while the technique illustrated here does. Anyway, the analysis strategy works by turning tainted data into untainted when it is sanitized by means of any check (i.e. not necessarily the right ones), and a simple heuristic is used to only take into account integer overflows that happen on checks, i.e. data remains tainted when any arithmetic operation is performed. Indeed, our tool can better determine when arithmetic operations cause overflows, since when variables are bounded my means of constants, it is possible to know the effects of arithmetic operations by simply computing them using bound values of the variables involved in the operations. Moreover, we do not limit detection to overflows on variables involved in checks, but we consider all the variables along tainted paths. Furthermore, our tool also takes the effects of type conversions into account, and raises warnings for potential unsafe checks.

6. Conclusions

We have described the design and implementation of a lightweight static analysis technique to find vulnerabilities, incentive programmers to write defensive code, and raise their awareness on the subtle aspects of the C language. We show some of the advantages of integrating a security analyzer inside gcc, as well as how an already present compilation algorithm (i.e. value range propagation) can be exploited and coupled with taint analysis. We describe a new way of performing taint analysis, by combining an interprocedural and an intra-procedural pass and we define a new taint lattice that allows us to report uncertainty of the analysis results to the user, as well as enabling a conservative analysis of libraries. We present an annotation mechanism for library functions that improves the taint analysis precision, easily allows to customize the causes that lead a tool to emit warnings and the risk rank associated with that warnings. We propose a solution to the problem of detecting subtle integer errors, such as integer overflows and perilous type conversion errors that lead to vulnerabilities.

We showed the effectiveness of our technique by testing the tool prototype on some real-world programs. All the known vulnerabilities and some previously unknown vulnerabilities have been found by the tool. Moreover, we observed that many of the warnings raised by the tool that were not vulnerabilities corresponded to weak code points that need improvement. Compilation and analysis time is on average approximately twice the time taken by compilation only, thus indicating that this technique can indeed be used regularly during development for short term iterations.

References

[1] Polyspace C Verifier. http://www.polyspace.com/.

[2] L. O. Andersen. Program Analysis and Specialization for the C Programming Language. *PhD thesis, University of Copenhagen*, 1994.

[3] K. Ashcraft and D. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. *IEEE Security and Privacy*, 2002.

[4] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. *In Proc. of SIGPLAN symposium on Compiler construction*, 1986.

[5] B. V. Chess. Improving computer security using extended static checking. *In Proc. of SSP*, 2002.

[6] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. *In Proc. of TACAS*, 2004.

[7] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Min, D. Monniaux, and X. Rival. The ASTRÉE analyser. *LNCS 3444*, 2005.

[8] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. *In Proc. of PLDI*, 2003.

[9] D. Evans and D. Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 2002.

[10] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific Static Analyses. *In Proc. of PLDI*, 2002.

[11] G. J. Holzmann. Static source code checking for user-defined properties. *In Proc. of IDPT*, 2002.

[12] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. *In IEEE Symposium on Security and Privacy*, 2006.

[13] JTC1/SC22/WG14. ISO/IEC 9899:TC2 standard. 2005.

[14] K. Lhee and S. J. Chapin. Buffer overflow and format string overflow vulnerabilities. *SPE*, 2003.

[15] J. R. C. Patterson. Accurate Static Branch Prediction by Value Range Propagation. *In Proc. of PLDI*, 1995.

[16] J. Pincus and B. Baker. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. *IEEE Security & Privacy*, 2004.

[17] scut / team teso. Exploiting format string vulnerabilities, 2001.

[18] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format-String Vulnerabilities with Type Qualifiers. *In Proc. of USENIX Security Symposium*, 2001.

[19] J. Viega, J. T. Bloch, T. Kohno, and G. McGraw. Token-based scanning of source code for security problems. *ACM Trans. Inf. Syst. Secur.*, 2002.

[20] M. Weber, V. Shah, and C. Ren. A case study in detecting software security vulnerabilities using constraint optimization. In *In Proc. of SCAM*, 2001.

[21] Y. Xie, A. Chou, and D. Engler. ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors. *ACM SIGSOFT Software Engineering Notes*, 2003.

[22] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *In Proc. of the ACM SIGSOFT on Found. of soft. eng.*, 2004.