Large-Scale Analysis of Format String Vulnerabilities in Debian Linux

Karl Chen David Wagner

UC Berkeley

{quarl, daw}@cs.berkeley.edu

Abstract

Format-string bugs are a relatively common security vulnerability, and can lead to arbitrary code execution. In collaboration with others, we designed and implemented a system to eliminate format string vulnerabilities from an entire Linux distribution, using type-qualifier inference, a static analysis technique that can find taint violations.

We successfully analyze 66% of C/C++ source packages in the Debian 3.1 Linux distribution. Our system finds 1,533 format string taint warnings. We estimate that 85% of these are true positives, i.e., real bugs; ignoring duplicates from libraries, about 75% are real bugs.

We suggest that the technology exists to render format string vulnerabilities extinct in the near future.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection—Invasive Software

General Terms Security, Languages

Keywords Format string vulnerability, Large-scale analysis, Type-qualifier inference

1. Introduction

A *format string bug* is a software bug involving `printf(3)`-style function calls. When calling `printf`, the first parameter specifies the *format string*, a control string with which to interpret the remaining parameters. A common mistake is to call `printf` with a string containing arbitrary characters as the first parameter, as in `printf(string)`, instead of the correct `printf("%s", string)`. The former works as intended most of the time, but is a security vulnerability that can lead to arbitrary code execution. Such mistakes are easy to make, hard to notice unless one is looking for them, and dangerous when latent. In a setting where input is untrustworthy, such a bug is a *format string vulnerability*. A format string vulnerability, like other input validation bugs such as SQL injection, arises because untrustworthy input is interpreted as containing trusted control data.

Format string bugs, originally thought to be harmless, were first publicly identified as a security attack vector in 1999. Since then, researchers have proposed a number of defenses; yet hundreds of format string vulnerabilities continue to be found. As of 2007, there are over 400 entries in the Common Vulnerabilities and Exposures (CVE) database for format string vulnerabilities [35].

We want to eliminate format string vulnerabilities from the entire Debian distribution. One way to do that would be to establish a quality gate for the Debian release process based on a sound static analysis tool for format string bugs. That is, no package would be distributed until after a developer had resolved all warnings from a format string detection tool. The key question is whether the false positive rate of such a tool, and the manual effort required, are low enough that this could be an acceptable barrier to inclusion in Debian. The main finding of our work is that the answer is *yes*. We have successfully completed analysis of 3,692 of the 5,589 packages in Debian with C/C++ code (92 million lines of code), using a month of wallclock time. Our system reports that 1,533 of these packages have taint violations, and we estimate 85% of these represent real format string bugs. There is no major technological barrier to establishing "format string clean" as a prerequisite for inclusion in Debian or other large Linux distribution.

2. Contributions

In collaboration with others, we have designed and implemented a system that can eliminate unintentional format string bugs from an entire Linux distribution. The system is composed of multiple components:

1. Elkhound and Elsa
2. Oink and CQual++
3. The Platform Model
4. Vulnivore and Build-Interceptor

Elkhound [27] is a Generalized LR (GLR) parser-generator. Elsa [2] is a C and C++ lexer, parser, and type-checker generated using Elkhound. Oink [3] is a static analysis framework using Elsa; it computes a whole-program data-flow graph (data dependency graph) at the subexpression granularity. CQual++ is an Oink client providing a CQual-like type-qualifier inference analysis for both C and C++. The Platform Model is a set of annotations to the standard C library. Vulnivore is a system for automatic analysis of a large software distribution. Vulnivore uses Build-Interceptor [1] to obtain the .i files from a target build process and uses CQual++ and the Platform Model to perform the analysis. We have built or improved each of these components for the purpose of doing a large-scale format string analysis; however, each component itself is also applicable to more general problems.

2.1 Project roles

The analysis system composes the work of many people, not all of whom are authors on this paper. Jeff Foster designed and im-

PLAS '07 June 14, 2007, San Diego, California
Copyright © 2007 ACM 978-1-59593-711-7/07/0006. . . $5.00

plemented CQual, the first generation type qualifier inference analysis [15, 14, 17, 4, 23, 22, 16]. Scott McPeak designed and implemented Elkhound and Elsa [2, 27], the C and C++ front-end. Daniel S. Wilkerson designed and implemented Oink/CQual++ [3] and Build-Interceptor, and also assisted with the implementation of Elsa. Robert Johnson designed and implemented the polymorphic serializing compactifying data-flow backend [23]. David Gay wrote a regions-based memory management library [18] used by both the original CQual backend and Robert Johnson's new backend. Karl Chen designed and implemented Vulnivore and the Platform Model and also worked on Elsa, Oink, CQual++, Johnson's backend, and Build-Interceptor.

2.2 Comparison to previous work

We build on prior work using CQual for format string analysis in the following ways.

- **Soundness**. Prior work considered finding any bugs with a reasonable false-positive rate a success, without regard to soundness or false negatives. In contrast, this work focuses on sound analysis algorithms, so that no format string vulnerability goes undetected. (We use the convention that an analysis is *sound* if it has no false negatives, and *complete* if it has no false positives.)
- **Scale: number of packages**. Prior work [31, 6] reported the results of applying static analysis to 10–12 carefully selected packages. In contrast, in the experiments reported here we have applied our tools to all C/C++ code in the Debian Linux repository, and completed analysis of 3,692 packages.
- **Scale: size of individual programs**. The largest package analyzed in prior work on format string vulnerabilities had in total 220,000 lines of code after pre-processing. The largest single program we have analyzed so far has 8,000,000 lines of C code, after pre-processing.
- **C++ and code diversity**. CQual++ is re-engineered to support C++ and has support for more programming language constructs (standards-compliant or not) that are seen in the wild.

Our system can analyze thousands packages in an automated fashion; the only manual input to the system is literally a pointer to the Debian repository. No per-package annotations are required.

In addition to the analysis itself, an important but often ignored problem is obtaining the code to analyze. To scalably analyze an entire distribution, one cannot manually collect source code or modify makefiles; each package has its own idiosyncratic compilation strategy and build dependencies, and may interact with the host system. We solve the problem of obtaining code automatically as described in Section 5.2.

3. Background

Consider the following program:

```
char *getenv(char const* varname);
int printf(char const* format_string, ...);

int main() {
  char *home = getenv("HOME");
  printf(home); // Oops!
}
```

Here, the variable `home` is untrustworthy, since its contents may be set by the user. However, it is used by `printf` in a trusting way: the format string may allow an attacker to get arbitrary stack contents (using "%s", "%d", etc.), or set a memory location (using "%n"), by setting the `HOME` environment variable appropriately (for example, one may use a number of "%d" specifiers followed by "%n" specifiers to overwrite the return address on the stack). Consequently, the code above contains a format string vulnerability.

Format string bugs can be statically detected using type-qualifier inference. Type-qualifier inference is a static analysis that can be used to statically find many classes of bugs. We use type qualifiers to perform *taint analysis*, which finds security vulnerabilities such as SQL injection and format string vulnerabilities [31].

3.1 Type-qualifier inference

A *type qualifier* is an extra qualifier which can appear on any level of a variable's type. The C++ language has two built-in type qualifiers: `const` and `volatile`. CQual, a type-qualifier inference engine for C created by Foster et al. [15, 14, 17, 4, 23, 22, 16], allows one to annotate variables with additional user-defined type qualifiers. In CQual, user-defined type qualifiers are prefixed with "$" by convention. Unannotated variables' type qualifiers are *inferred*, using ML-style type inference.

Type-qualifier inference can be used to implement taint analysis by introducing the type qualifiers `$tainted` and `$untainted`. Data that comes from the environment and is untrustworthy is marked `$tainted`. Function parameters that must be trustworthy are marked `$untainted`. If data flows from `$tainted` to `$untainted`, this is a *taint violation*, or in more general type-qualifier inference terms, a *type error*. The approach of using type qualifier inference to find format string bugs was first articulated by Shankar et al. [31], and we adopt it in this work.

Continuing the above example of a format string vulnerability, to find the bug using type-qualifier inference we annotate the return value of `getenv` as `$tainted` and the input to `printf` as `$untainted`. In other words:

```
char $tainted *getenv(char const *varname);
int printf(char const $untainted
  *format_string, ...);
```

With these annotations, the format string vulnerability is detected: the type of `home` is inferred to be `char $tainted *`, which is not compatible with the type of `printf`'s first formal parameter.

Note that detecting the bug required only annotations to the prototypes of standard C library functions. This is an important property of our system, because it allows application of our tool to legacy code without annotating or changing the legacy code.

4. Related work

CQual has been used to find format string vulnerabilities [31] and other kinds of security bugs, such as user-kernel bugs [22], and for redacting crash dumps [7].

We group defenses against format string attacks as follows:

1. **Language restriction**. The language may in effect be changed by removing the "%n" specifier at the library level. This may break existing programs, and also does not protect against information disclosure attacks that dump the contents of the stack. It also does not generalize to any other kind of software bug.
2. **Lexical analysis**. Lexical analysis includes lint-like tools that search for the most common type of error (e.g., "`printf( variable)`") [34]. PScan [11] finds calls to `printf` where the format string is non-static and no arguments are given. GCC version 4 also checks for this when `-Wformat-security` is enabled. FormatGuard [10] uses the C preprocessor to detect errors in the number of arguments passed to `printf`, but does not check calls to `vprintf` or changes in the kind of format string. Though easy to implement, these are inherently less powerful than static analysis, and have false positives and false negatives; for example, a helper function that passes its parameters to `vfprintf` will hide taint errors. Indeed, many of the format

string vulnerabilities we have found are of this form and consequently would not be found by lexical analysis.

3. **Static analysis**. Static analysis can analyze program semantics, not just look for syntactic/lexical bugs, and hence can be more effective. Shankar et al. used CQual to find format string vulnerabilities in C code [31]. Later, Avots et al. showed how to find format string vulnerabilities with fewer false positives, by using a context-sensitive points-to analysis [6]. Livshits and Lam used a custom taint analysis to find input validation vulnerabilities in Java applications, with no false negatives [26]. Also, static analysis has been used for finding many other kinds of security vulnerabilities [36, 8, 13, 39, 24, 37, 28, 20, 38, 21].

4. **Dynamic analysis**. Another approach is to detect format string vulnerabilities when they are exploited, by analyzing the flow of tainted data at run time. For instance, perl includes a taint-checking mode. The libformat library [30] inserts itself into a program via the dynamic linker and aborts the program if the format string is in writable memory and contains the "%n" specifier. The libsafe library [33] checks that a "%n" specifier is not writing to a function return address. Libformat and libsafe have the advantage of applying to binary programs, but they do not protect against information disclosure. More recently, Ringenburg and Grossman use a hybrid approach combining static analysis and program transformation [29]. Purely dynamic defenses in general may defend against attacks without having to patch or recompile applications, and can more easily be sound.

The advantage of static analysis over dynamic analysis is that it can be used to find and fix bugs before software is deployed, and it incurs no run-time overhead or change.

5. Implementation of the analysis

Since we are doing static analysis, we must obtain the source code to the programs under analysis. We need the source code as seen by the compiler-proper, that is, we need the pre-processed code, in the form of .i files. For soundness, we must analyze all source files used by each program, including all static and dynamic libraries (recursively) used — we call this *whole-program analysis*.

5.1 Whole-program analysis

We analyze the *whole program*. This includes all of a program's source code, as well as the source of all libraries it uses, recursively. We chose Debian Linux to analyze because it has a large number of packages, each with appropriate package dependency information.

Debian is composed of a number of source packages. Building a source package yields corresponding binary packages, which are installed by the end-user. Source packages depend on other binary packages for their header and library files.

5.2 Getting and analyzing the code

We have created *Vulnivore*, a system that, given the version number of a Debian release and a cluster of machines to run on, automates running the entire analysis cycle. The process is entirely automated; when provided with the input source packages, Vulnivore outputs the list of packages with taint warnings with no manual intervention, and provides a user interface to view results.

The process of getting all the code to analyze for whole-program analysis is involved. We need to analyze pre-processed C and C++ files (.i files). Since each source package has its own idiosyncratic build process, we obtain the set of .i files for each program by *intercepting* the build and capturing the pre-processed files along the way using Build-Interceptor [1]; following Liblit [25] and Chen [9] this tool embeds .i files into object files as they are compiled, which travel through the linking process into the final executable. We run the builds separately by using virtual machines. Originally, we created a chroot per task, but this proved insufficient as installing the build dependencies of some packages require running network daemons and doing other things that "escape" a chroot. The second generation of Vulnivore uses User-Mode-Linux [12] as the virtual machine, which allows for easy communication between processes in the host and the client.

For each source package, we create a new virtual machine, install the required dependencies, run the build under interception, and capture the .i files for each executable. Since programs may statically link against libraries from other packages, this must be done in a particular order.

Vulnivore defines a number of stages. The stages up to the application-specific analysis include:

1. Building a source package in the virtual machine,
2. Collating sources from dynamic libraries that this source package depends on,
3. Re-compiling the collected source files to make sure they can be compiled in a vanilla environment, and checking whether special compiler flags are needed,
4. Linking to minimize the number of objects needed from dynamic libraries, and
5. Administrative tasks such as collecting statistics.

When the above stages are completed for one package, Vulnivore has produced a tar archive of all .i files needed to compile or analyze each program. (Exception: We do not include the source to libc, as our static analysis tool provides a model for the behavior of every libc function.) For example, once Vulnivore has finished its build interception, one can type essentially `g++ *.i *.ii` to compile the entire program, without needing any headers or libraries.

Once the above steps are complete, Vulnivore begins program analysis, which proceeds in 5 stages:

1. Parsing each source file using Elsa,
2. Making a data-flow graph for each source file using Oink,
3. Making a data-flow graph for the whole program using Oink,
4. Format-string taint analysis of each source file, using CQual++,
5. Format-string taint analysis of the whole program, using CQual++.

When function-granularity linking is enabled as described in section 5.6, we add additional stages after parsing:

1. Construction of function-call graph for individual source files,
2. Reachability analysis to filter out unused functions.

Each of these stages is pipelined, and the separation into multiple stages allows errors to be identified at the appropriate stage. The factoring of stages also enables an optimization: if two executable programs from different packages share source files (which will happen for example when they use the same library), we avoid repeating work by caching the results of all per-source-file stages.

Vulnivore is designed for parallel execution on a large cluster of machines. Vulnivore analyzes the dependency graph and creates a work queue with new tasks added as work is completed. The work queue and the finite state machine that governs it is based on BOINC [5]. Since a run on the entire Debian distribution takes a long time and bugs in the analysis are discovered as results are streamed out, Vulnivore's work queue allows fixes to the analysis and other stages to be made while minimizing the amount of CPU time to re-analyze the affected stages and packages.

Unfortunately, the process is complicated by cycles in the dependency graph of source packages. We solve this by building packages in layers. Suppose package A indirectly depends on package B at build time, and package B depends on package A at build time. We first do an incompletely-intercepted build of package A to use as input for building package B. Then we build package A using the fully-intercepted build of package B.

5.3 Oink and CQual++

As discussed earlier, CQual++ is a type-qualifier engine based on CQual. It has been rewritten by creating a new framework, Oink, for static analysis of C and C++ code. Oink, which uses the Elsa front-end, is designed to easily write new static analysis backends that may cooperate with each other. Other analyses have now been written. CQual++ integrates various extensions to the original CQual engine including automatic inference of polymorphic types (context-sensitivity) [22], instance sensitivity[1] [22], compaction [23, §4.4], well-formedness constraints [22], and `void*` auto-unions[23]. These techniques have been previously published, and they are summarized in Appendix B. Also, numerous internal changes have been made for scalability and performance.

CQual++ has new features to make the analysis more precise, including:

- **Handling of variable-argument (vararg) functions.** Previous work required each vararg function to be manually annotated, especially printf wrappers. In CQual++, taint automatically flows from caller arguments passed through the "..." parameter and retrieved in the callee via `va_arg`. Taint also flows through `va_copy` to allow functions that wrap around another vararg function.

  To reduce false positives, CQual++ refines the analysis by separating the data flow for each type of parameter. To prevent over-refinement, separation is actually done on equivalence classes of similar types (for example, all integer types are in the same class). In the following code:

```
void foo(long $tainted i0,
         unsigned char $untainted *s0) {
  bar("is", i0, s0);
}

void bar(char const *fmt, ...) {
  va_list va; va_start(va, fmt);
  for (; *fmt; ++fmt) {
    if (*fmt == 'i') {
      int i = va_arg(va, int); ...
    } else if (*fmt == 's') {
      char *s = va_arg(va, char *); ...
    } else { ... }
  }
}
```

  the type of `i` is correctly inferred as tainted, while the type of `s` is correctly inferred as untainted. "..." may also be annotated with qualifiers (even polymorphic ones) if needed.

- **Handling of tainted array indices.** In CQual++, taint flows from array indices and from pointers to referents. (This feature is optional.) This propagates taint through code such as:

```
extern char uppercase[256];
unsigned char c = read_from_network();
char upper = uppercase[c];
char upper2 = *(uppercase + c);
```

  Here, taint flows from `c` to `upper` and `upper2` as follows. The first array indexing case is reduced to pointer arithmetic plus a dereference as in the second case. (When this feature is on) the addition of a tainted integer to the (untainted) pointer value of `uppercase` creates a tainted pointer value. Dereferencing the tainted `char *` then returns a tainted `char` value.

[1] Previous work called this feature "field sensitivity"; however CQual++ now calls it "instance sensitivity" as the feature is characterized by the ability to allow different instances of a given struct to be analyzed separately.

5.4 Platform Model

Much of the standard C library is written in assembly, invokes syscalls, or otherwise uses C in a non-type-safe or unportable way, and thus would be difficult to analyze automatically. Therefore, we make no attempt to analyze it directly. Instead, we manually created a model of the semantics of each library call.

We assume that the C library is standards-compliant and we use the C language specification as our specification of the behavior of each library function. In practice, there are some variations in function semantics across operating systems and versions, but we model all possible behaviors that can occur on the systems we have used. (We detect differences in parameter types and signal them as errors.)

The *Platform Model* is a set of annotations to the C library (and other system libraries) that accurately models data-flow semantics in sufficient detail to ensure the correctness of CQual++'s type inference, though we do not try to model the full library semantics. For example, `strcpy(3)` might be modeled as follows:

```
char *strcpy(char *dest, char const *src) {
  dest[0] = src[0]; return dest;
}
```

While this stub is valid C code that could be compiled and run, it does not behave in the same way as the standard `strcpy`. Nonetheless, from the point of view of CQual++'s type inference analysis, it is equivalent to the real `strcpy`. We use stubs like these to concisely model how library functions affect the flow of taint through the program.

CQual++ supports another way to express the above model:

```
char $_1_2 *strcpy(char $_1_2 *dest,
                   char const $_1 *src) {}
```

This annotation indicates that the value pointed to by `src` flows, polymorphically, to the value pointed to by `dest`, and moreover that the type of the value pointed to by `dest` is the same as the type of the value pointed to by `strcpy`'s return value. The `$_1_2` syntax specifies that `dest` has a type qualifier which is higher than that of `src` in the type-qualifier lattice. In this example, if `src` is tainted, then `dest` and the return value must be tainted; if `src` is untainted, then `dest` and the return value may be untainted or tainted but must have the same type qualifier. The type-qualifier lattice and the special syntax are described by Shankar et al. [31].

To create the model, we started with GNU libc version 2.3.5 and replaced every function declaration or definition with a stub or annotation. The Platform Model has three types of annotations:

1. **Taint flow** from one parameter to another or to the return value.
2. **Trusting sinks**, which for this application are mainly the `printf(3)` family of functions and their cousins such as `syslog(3)`. Passing a tainted value to a trusting sink triggers a type error. There are currently 64 trusting sinks.
3. **Untrustworthy sources**, such as `read(2)` and `getenv(3)`. Untrustworthy sources create new tainted values. There are currently about 500 untrustworthy sources.

In all, the Platform Model contains annotations for about 2600 library functions defined by GNU libc.

The Platform Model currently is designed for finding format string vulnerabilities only. Thus, the Platform Model only annotates parameters of character type. For instance, the return value of `fread(3)` — the number of characters read — is not annotated as tainted, even though it can potentially be controlled by an attacker. Similarly, other system calls and other external mechanisms that may return untrustworthy integer data are not marked tainted. In particular, we explicitly assume that the program will not take the return value from `fread(3)`, cast it to a character type, and use it as part of the format string passed to a `printf(3)`-like function. (See the last rule in Appendix A.)

We assume that all character input read from the user, the network, the filesystem, or any other input source (such as `stdin`) is untrustworthy. We also treat all character data provided as part of the environment (such as `argv` and `environ`) as untrustworthy. This may be overly conservative. For example, a format string bug caused by user input in a local non-setuid program may not be exploitable, since the program is running in the user's own privilege domain. Nonetheless, we consider it at least a bug that should be fixed, since it may become a vulnerability when this program is composed with other systems. If another program, say a CGI script, were to invoke this one, it would not know to check certain inputs for format strings.

Our philosophy in developing the Platform Model was to err on the side of soundness, to ensure we would not miss any vulnerabilities. This has generally been effective, with only a few exceptions. Our experience with analyzing a large number of packages revealed that our initial annotations for a few functions nearly always yielded false positives:

1. `gettext(1)` and `catgets(3)` family: We initially annotated these as always returning a tainted string. However, this leads to many false positives. We changed the annotation to a polymorphic signature: the return value is tainted if the message input to `gettext` is tainted. Thus we assume that the system translation files are not under the attacker's control. We also assume that the translation files do not contain any accidental errors that would cause a mismatch between the format specifiers in the input message and the translated message. Of course, if either of these assumptions are incorrect, we might miss vulnerabilities.
2. `strerror(3)`: We initially annotated the string returned by `strerror` as tainted, because the attacker might be able to control which error message is selected. However, none of the standard error messages have "%" in them, so to reduce false positives we decided to remove this annotation.

5.5 Linking

When a program contains multiple translation units (files), uses of undefined symbols (functions and variables) must be linked to their definitions in other files.

We wish to map use of a symbol to the appropriate definition. Naively one would create a one-to-one map from symbol names to definitions, and compose this with the many-to-one map from uses of symbols to symbol names. Missing symbol definitions are an error, and by default we stop analysis if any symbol is used but has no definition. (Even if unsound, it may still be useful to find as many bugs as possible, so this can be downgraded to a warning.) C++ names are mangled in a namespace separate from C names, and orthogonally, functions have a namespace separate from variables. This resolves the difference clashes with symbols such as `clog` (both a C++ iostream variable and a C math function).

Unfortunately, things get complicated in the presence of multiple symbol definitions with the same name in the same class; in reality there is a one-to-many relation between symbol names and definitions of symbols. Composing the many-to-one mapping from use of symbols to names of symbols, and the one-to-many relation from symbol name to definitions, would lead to a many-to-many relation from uses of symbols to definitions. At link time, there is indeed a many-to-one mapping directly from the use of a symbol to the definition of a symbol, defined by operating system linker semantics. The semantics differ across operating systems, depend on the order of linker command-line arguments, and can even depend on run-time settings. For example, in Linux, the run-time environment variable `LD_BIND_NOW` selects whether the dynamic linker resolves all symbols immediately, or lazily upon first reference of each symbol, potentially affecting which definition gets used. In either case, symbol resolution depends on the order in which libraries are loaded. Library symbols may also have flags specifying whether their scope is global or local and versioning information. Some uses of a name may refer to one definition, while other uses refer to a different definition with the same name. These issues present a problem for a static analysis that must link uses of symbols to definitions.

We have considered the following approaches to handling multiple symbol definitions with the same name.

1. **Disallow**: All symbol definitions must have unique names; finding multiple definitions with the same name triggers a link error. This was the behavior of the original Oink/CQual++ linker. However, this is too restrictive since many real-world programs do use multiple symbols with the same name.
2. **Link all**: For each symbol name `n`, we find all uses of `n` and all definitions of `n`, and then proceed as though any use of `n` might potentially refer to any definition of `n`. For instance, a function call `n()` is treated as though it may execute any of the definitions (as though the program chose one non-deterministically). This was the behavior of the second version of linker. This strategy is conservative against false negatives, but resulted in false positives. It worked well when the user had defined his own definition of a function also provided by a library, typically with the same semantics. However, we discovered that this approach introduces problems when the semantics of the two definitions are widely different. For example, the GNU C library defines the `error(3)` function (which takes a format string), and some user programs use it, but the name `error` is also commonly used by user programs with unrelated semantics.
3. **Imitate linker**: Emulate the dynamic linker of a specific operating system and version with a specific set of run-time flags. This would accurately model the behavior under specific conditions. In many programs, multiple function definitions may in fact be identical, such as a user defining `strlcpy` in case the system library doesn't have it. However, when the choice matters, imitating the linker exactly is fragile because subtle changes in makefile order, operating system versions, and run-time settings may affect which definition gets used.

   In Linux, this strategy could be implemented by instrumenting the Linux dynamic linker, `ld.so`, forcing all symbols to be resolved greedily, and observing how it resolves each symbol. However, we have not yet implemented this.
4. **Hybrid**: Partially model the behavior of the system linker. The current version of the Oink linker behaves as follows: We label each symbol definition as either "strong" or "weak". This is modeled after the ELF "strong"/"weak" notion, but we do not look at actual ELF symbol attributes. If there is more than one strong definition for any name, this is a link error. If there is a strong definition, any weak definitions are ignored. If there is no strong definition but at least one weak definition, we link all weak definitions. (Normally there will only be at most one weak definition per symbol.)

In the current system, all symbols in the Platform Model are labeled “weak”, while all others are “strong”. This strategy handles the common cases where the user has redefined a symbol already defined in glibc: for portability, accidentally, or to intentionally change its behavior. It also works well with the way our build interception flattens the link structure of user code, excluding the Platform Model. The order of files listed does not matter to the Oink linker; this is desirable because the order is a fragile property of the source program.

We have not found the “one strong definition” rule too restrictive, and have seen zero false positives due to it. There is, in theory, a potential for false negatives if some uses of a function use the definition from libc, while others, due to library order, use a user-defined one with different semantics.

5.6 Performance and scalability

As we began to apply our infrastructure to thousands of applications from Debian Linux, we encountered two scalability challenges. The first is the large number of programs to be analyzed and the heterogeneity in their build systems. Many packages come with their own ad-hoc process and scripts for customizing and compiling the code, and the diversity is considerable. To address this challenge, we finally settled on a design that uses virtual machines to collect the code and compiler flags that are provided to the compiler. See Section 5.2 for details.

The second scalability challenge is the large size of individual programs to be analyzed. In our experience, the key limiting factor to scalability is the amount of RAM used by our algorithms. To address this challenge, CQual++ incorporates recent algorithmic advances that reduce the memory consumption of type-qualifier inference, including techniques for matched-parenthesis CFL reachability [16], special treatment of global variables [16], and modular analysis using graph compaction [23]. In the first phase of modular analysis, we analyze each source file by itself to generate a graph of type-qualifier constraints. We compact the type-qualifier graph, removing unnecessary internal nodes, and serialize the state. After this phase, all nodes that correspond to globally visible variables remain, but many of the other nodes have been removed. Then to analyze the whole program, we de-serialize the graph corresponding to each source file, link these graphs together, and process the result.

Oink provides a feature to optimize space consumption by filtering out unused functions before performing type-qualifier inference. Since dynamic libraries linked into a program[2] often contain many functions not used by the program, this reduces the amount of the code that we must analyze. Uncalled functions are filtered out as follows:

1. For each translation unit, emit a data-flow graph at the whole-function granularity; that is, entire function bodies are nodes.
2. Find the set of all functions that are reachable from `main()` or the constructor of any global object.
3. Run the format string analysis, ignoring functions not in the set.

Our implementation correctly recognizes signal handler functions as used: if their address is passed to the `signal(2)` system call, they will be reachable.

With these scalability improvements, the largest program we can currently analyze on a 32-bit 4 GB machine consists of 8,000,000 lines of pre-processed C. Because our 32-bit machines cannot address more than 3 GB of virtual memory per process, adding more than 4 GB of RAM does not help.

[2] For static libraries, we are able to tell which translation units are entirely unused, but not for dynamic libraries.

A complete run from scratch, including building and analyzing all Debian Linux packages, takes a month of wallclock time on a shared 64-node 2 GHz IA-32 Linux cluster, with on average half the nodes devoted to this task. (At the package granularity, analyzing all packages is very parallelizable once the common dependencies are analyzed.)

6. Experimental results

We applied our tool to all packages distributed with Debian Linux 3.1. Each package provides the source for a single application or for several related programs. Debian Linux 3.1 is composed of 8,624 source packages (which produce 15,197 binary packages), of which 5,589 contain C/C++ programs, with 261 million lines of C/C++ code before pre-processing. (Line numbers specified “before pre-processing” include comments and blank lines. The 2,733 packages without C/C++ programs are considered trivially analyzed and excluded from discussion below.) Our counts of lines of code include source code in libraries used by these packages, but we are careful to avoid double-counting library code: if two packages use the same library, we only count the code in the library once.

We are able to build 5,404 packages (211 million lines of C/C++ before pre-processing) of the 5,589 using the virtual machine process described in Section 5.2. Vulnivore succeeds at creating compilable and linkable whole-program tar archives of all relevant source files for 5,123 of these packages (201 million lines).

We succeed at parsing the whole program in 4,527 of these packages (164 million lines). The format string taint analysis can individually analyze the source files in 4,514 of these packages (151 million lines). We succeed in whole-program analysis of 3,692 of these packages. This comprises 92 million lines before pre-processing, including blank lines, and not repeat-counting library code; these correspond to 2,112 million lines of code after pre-processing, repeat-counting library code.

The failure to successfully complete analysis of a package is typically caused either by scalability limits or by implementation deficiencies that prevent us from parsing or analyzing certain rare code constructs. Developing a complete implementation that can handle all of the code idiosyncrasies found in the wild has proven beyond the resources of our research group, but we are confident that a commercial-scale implementation could eliminate these shortcomings.

Of these 3,692 packages we analyzed, 1,533 packages have taint warnings. It was beyond our capabilities to manually inspect each of these 1,533 packages. Therefore, we selected 100 of these packages (in an ad-hoc, semi-random way), and we manually inspected the warnings our tool emits for these 100 packages. Of these 100 packages, we determined that 87 have true format string bugs, and 13 are false positives. The 87 packages account for 40 unique bugs. The other 47 are duplicates: a single bug in one particular commonly-used library, `ncurses`, makes many packages susceptible to format string attacks. Incidentally, we feel that the `ncurses` bug validates our decision to analyze library source code together with the program source code.

Based on this experiment, if this experience is representative, we estimate that between 80% and 94% of packages with taint warnings contain true format string bugs (assuming a binomial distribution, and using a 95% confidence interval). Note that this counts the number of packages affected, not the number of packages whose own source code contains bugs. This suggests that over a thousand Debian packages contain format string bugs, and that our techniques could help find them.

6.1 True positives

The most common source of format string vulnerabilities we found were the regular `printf` kind, where the programmer had forgotten that the function takes a format string. These include library functions such as `syslog(3)` and `error(3)`, as well as custom wrappers written by the programmer. For example, `syslog(0, danger)` should instead be `syslog(0, "%s", danger)`.

The second most common mistake arises from calling `sprintf` to construct a string that is then used as a format string. For example, one package has the following code:

```
int Error(const char *format, ...)
{
  va_list ap; va_start(ap, format);
  char msgbuf[256];
  vsnprintf(msgbuf, sizeof(msgbuf), format, ap);
  syslog(3, msgbuf);
}
```

The call to `vsnprintf` dynamically constructs a string from untrusted input and thus may contain a `'%'` character. To fix this bug, the `vsnprintf` and `syslog` calls should be replaced by `vsyslog(3, format, ap)`.

Other common sources of bugs include:

- Using a flag or other control flow to indicate whether a character buffer contains a format string or not. For example, one package contains code of the form:

  ```
  void myerror(int err, char *p, ...)
  {
    va_list args; va_start(args, p);
    if (err==0) {
      fprintf(stderr, "%s:unknown error!\n", p);
    } else {
      vfprintf(stderr, p, args);
      fprintf(stderr, ":%s\n", strerror(err));
    }
    exit(err);
  }
  ```

  Here, if and only if `err == 0`, then it is safe to pass a tainted string as `param`. Even if used correctly, we consider this bad design. As with other format string bugs, such code mostly works for as long as it is not under attack, insidiously hiding a potential vulnerability.

- Using program arguments (`argv`) directly as format string specifiers. If the caller provides an argument that contains a `'%'` character, the program may crash, or worse.

  For non-setuid programs, these bugs may not be directly exploitable. Nonetheless, they are a security risk, if these programs are invoked by network daemons or automatic scripts, as discussed in Section 5.4. For instance, if a CGI script invokes one of these programs, this may create a remotely exploitable vulnerability. If a temporary file cleaner that is run automatically once a day invokes one of these programs with the name of each file under `/tmp`, this may create a locally exploitable vulnerability where a local user can create a filename containing a `'%'` character and escalate their privileges.

  If these bugs are considered uninteresting, it is straightforward to re-run the analysis without the taint annotations on argv.

- Blatant user-supplied format strings. The programmer expects the user to supply a format string and trusts it without validation. We consider this to be a design error. For example:

  ```
  char *fmt_string = getenv("FOO_CONFIG", "%s");
  printf(fmt_string, data);
  ```

6.2 False positives

In the warning reports we looked at, the false positives fell into a few general categories.

- **Flow sensitivity**. Some programs re-use variables, for example:

  ```
  strcpy(buf, tainted_string);
  // ...
  strcpy(buf, untainted_string);
  printf(buf);
  ```

  The workaround is to use new variables for new data. In most cases this can be done without any extra run-time overhead, and may be desirable in any case for software engineering reasons.

- **Input validation**. Some programs do manual input validation on a user-supplied format string or otherwise propagate data flow in a safe way. Example:

  ```
  char fmt[3] = "%x";
  switch (tainted_char) {
  case 'o': case 'd':
    fmt[1] = tainted_char; break;
  //...
  }
  printf(fmt, number);
  ```

  Here, the programmer could alter the code to not copy any characters from the tainted string (as a limited form of taint laundering).

  In the general input validation case, the programmer must be cognizant of the analysis and "cast" the taintedness away:

  ```
  char *str = read_from_user();
  if (format_string_validate(str)) {
    char *safe_str = (char $untainted*) str;
    printf(safe_str, data);
  } else {
    abort();
  }
  ```

  This is a current limitation with our system and there is no general automated solution; however, this situation is not very common.

- Program logic that is outside of type-qualifier inference. The most common false positive in this category takes this form:

  ```
  snprintf(fmt, 10, "%%%ds", tainted_integer);
  printf(fmt, str);
  ```

  Here, CQual++ considers `fmt` tainted, but in fact the programmer knows it is safe. To avoid the taint warning, one could replace the above with:

  ```
  printf("%*s", tainted_integer, str);
  ```

6.3 Soundness

Our goal was to analyze source code as accurately and soundly as possible, and secondarily as completely as possible. We believe we have achieved this under certain assumptions, as follows.

We define a set of rules that programmers should follow in writing new C code. We conjecture that if all of these rules are followed, then the analysis will be sound. In practice it is unavoidable to intentionally violate some rules in industrial code, but in many cases it will be possible for the programmer to be convinced that the intent of the rule has been followed. An abbreviated listing of the rules is in Appendix A.

We make no claims about the soundness of our analysis of Debian Linux: existing Linux packages were not written with these rules in mind, so there is no guarantee that our analysis will find all format string bugs in Debian Linux. Nonetheless, we feel that the shift in perspective from finding bugs to trying to prove their absence has paid significant benefits: our experience has been that the push towards soundness has enabled us to find many more bugs than we otherwise would have found. Anecdotally, as we added features that brought CQual++ closer to full soundness, we found many additional bugs.

6.4 Limitations

There are issues limiting our system from universal applicability.

1. **Scalability**. The primary obstacle to analyzing the rest of Debian is the virtual memory usage when analyzing extremely large programs. We are currently limited to 3 GB virtual memory on each machine of our IA32 cluster.
2. **Dynamic linking**. If the program uses `dlopen` to load dynamic shared objects at run-time, all bets are off, as we ignore functions called through pointers returned by `dlsym`. Currently, Oink emits an unsoundness warning during analysis if `dlopen` or `dlsym` (and related) functions are used. In the worst case, analyzing such code would require solving the halting problem. However, in practice, many uses of dynamic shared objects use a standardized "plugin" architecture. An interesting research direction is to devise a solution to allow analysis of such code.
3. **Implicit flows**. CQual++ currently assumes there are no implicit flows. This means taint may be laundered through the program counter, for example:

```
char translate_char(char c) {
  switch(c) { case 'a': return 'A';
              //...
              case 'z': return 'Z'; }
}
```

   Tainting the program counter based on conditional branches on tainted data is possible, but would lead to an unreasonable number of false positives: taint would quickly propagate to almost all data via conditional branches on tainted data. A promising direction is to heuristically recognize specific constructs, such as switch statements, to identify common idioms.

Promising directions for further scalability improvements include:

1. An idea of Daniel S. Wilkerson: instead of de-serializing all files of the program at once, aggregate them in a particular order. For example, link all modules of a single library together, before linking it to the program. This might reduce the memory consumption of CQual++. For this to work well, a new concept of "externally-visible" is defined so that functions which are not used outside of this library can be compacted.
2. Identify additional common libraries to manually annotate, for example the X libraries, instead of analyzing them.
3. Port and optimize Oink and CQual++ for 64-bit architectures. Currently the system requires nearly twice as much memory on 64-bit architectures due to the use of pointers.

7. Conclusions and Future Work

We have analyzed a large fraction of the Debian Linux distribution, which contains 8,600 open source packages. Analyzing the remaining packages would require scalability and incremental improvements to CQual++. If this system is properly integrated into an OS distribution release cycle such as Debian's, such that all taint warnings are removed, it is reasonable to forecast victory over format string vulnerabilities.

The success of our system is also promising for ridding other kinds of bugs amenable to taint analysis, such as SQL injection attacks. It would be necessary to extend the Platform Model to annotate the appropriate trusting sinks. Code that performs input validation for SQL statements may require manual annotation, or preferably, be rewritten using SQL prepared statements.

This approach can also apply scalably to languages other than C/C++, for example JQual [19] can find bugs using taint analysis of Java code. Thus, entire classes of security bugs can in the future be eliminated from widely deployed software.

Oink and CQual++ are available under the BSD license at `http://www.cubewano.org/oink/`. The release of the Vulnivore analysis framework is pending.

Acknowledgments

The authors thank Alex Aiken, Tim Fraser, Robert Johnson, Scott McPeak, Alan Roytman, Jeff Foster, Umesh Shankar, Daniel S. Wilkerson, and the anonymous reviewers. We thank Tim Fraser for finding a number of soundness holes in the Platform Model and in our rules. Marco Barreno, Chris Karlof, Adrian Mettler, David Molnar, and Russell Sears provided useful feedback on drafts.

References

[1] Build-Interceptor. Website, 2007. `http://freshmeat.net/projects/build-interceptor/`.

[2] Elsa. Website, 2007. `http://www.cs.berkeley.edu/~smcpeak/elkhound/sources/elsa/`.

[3] Oink. Website, 2007. `http://freshmeat.net/projects/oink/`.

[4] Alex Aiken, Jeffrey Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *Proc. of the Conference on Programming Language Design and Implementation*, 2003.

[5] David P. Anderson. BOINC: A system for public-resource computing and storage. In *Proc. of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID)*, 2004.

[6] Dzintars Avots, Michael Dalton, Benjamin Livshits, and Monica Lam. Improving software security a C pointer analysis. In *Proc. of the 27th International Conference on Software Engineering*, 2005.

[7] Pete Broadwell, Matt Harren, and Naveen Sastry. Scrash: A system for generating secure crash information. In *Proc. of the 12th USENIX Security Symposium*, pages 273–284, August 2003.

[8] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, 2000.

[9] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of c code. In *Proc. of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, 2004.

[10] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proc. of the 10th USENIX Security Symposium*, pages 191–200, 2001.

[11] Alan DeKok. PScan: A limited problem scanner for C. Website, 2000. `http://packages.debian.org/pscan`.

[12] Jeff Dike. User-mode Linux. In *Proc. of the 5th Annual Linux Showcase & Conference (ALS)*. Usenix, November 2001.

[13] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1), 2002.

[14] Jeffrey S. Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD thesis, University of California, Berkeley, December 2002.

[15] Jeffrey S. Foster, Manuel Fahndrich, and Alexander Aiken. A theory of type qualifiers. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[16] Jeffrey S. Foster, Robert T. Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Transactions on Programming Languages and Systems*, pages 1035–1086, November 2006.

[17] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proc. of the SIGPLAN 2002 Conference on Programming language design and implementation (PLDI)*, 2002.

[18] David Gay and Alex Aiken. Memory management with explicit regions. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, 1998.

[19] David Greenfieldboyce and Jeffrey S. Foster. Type qualifiers for Java. Technical report, University of Maryland, August 2007. `http://www.cs.umd.edu/projects/PL/jqual/`.

[20] Samuel Guyer, Emery Berger, and Calvin Lin. Detecting errors with configurable whole-program dataflow analysis. Technical report, University of Texas at Austin, 2002. `ftp://ftp.cs.utexas.edu/pub/emery/papers/detecting-errors.pdf`.

[21] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proc. of the SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[22] Rob T. Johnson and David Wagner. Finding User/Kernel pointer bugs with type inference. In *Proc. of the 13th USENIX Security Symposium*, 2004.

[23] Robert T. Johnson. *Verifying Security Properties using Type-Qualifier Inference*. PhD thesis, EECS Department, University of California, Berkeley, 2007.

[24] Nenad Jovanovic, Christopher Krügel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy (Oakland 2006)*, pages 258–263. IEEE Computer Society, 2006.

[25] Ben Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, 2005.

[26] V. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proc. of the 14th USENIX Security Symposium*, 2005.

[27] Scott McPeak and George C. Necula. Elkhound: A fast, practical GLR parser generator. In *Proc. of the 13th International Conference on Compiler Constructor (CC)*, 2004.

[28] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *20th IFIP International Information Security Conference*, 2005.

[29] Michael F. Ringenburg and Dan Grossman. Preventing format-string attacks via automatic and efficient dynamic checking. In *Proc. of the 12th Conference on Computer and Communications Security*, 2005.

[30] Tim Robbins. Libformat, 2000. `http://archives.neohapsis.com/archives/linux/lsap/2000-q3/0444.html`.

[31] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proc. of the 10th USENIX Security Symposium*, 2001.

[32] Saurabh Srivastava, Michael Hicks, and Jeffrey S. Foster. Modular information hiding and type-safe linking for C. In *Proc. of the 2007 SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, pages 3–14. ACM Press, 2007.

[33] Timothy Tsai and Navjot Singh. Libsafe 2.0: Detection of format string vulnerability exploits. Technical report, Avaya Labs, February 2001. `http://pubs.research.avayalabs.com/pdfs/ALR-2001-018-whpaper.pdf`.

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

[35] Common Vulnerabilities and Exposures. Format string vulnerabilities. Website, 2007. `http://www.cve.mitre.org/cgi-bin/cvekey.cgi?keyword=format+string`.

[36] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. of the 7th Network and Distributed System Security Symposium (NDSS)*, 2000.

[37] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proc. of the 15th USENIX Security Symposium*, 2006.

[38] Junfeng Yang, Ted Kremenek, Yichen Xie, and Dawson Engler. MECA: an extensible, expressive system and language for statically checking security properties. In *Proc. of the 10th ACM Conference on Computer and Communications Security (CCS)*, 2003.

[39] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proc. of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, 2004.

A. Appendix A: Programming rules to avoid unsoundness

This document is intended for programmers writing C code and using CQual++ to search for format string vulnerabilities in that code. This version is abbreviated; for the full version see `http://www.cubewano.org/cqual++_rules`. We describe a set of programming rules that we recommend such C code should follow. The goal is that if you, the programmer, follow all of these rules, then CQual++ will be guaranteed to detect any format string vulnerabilities in your code[3].

More specifically, the claim is as follows: If your C program obeys all of these rules and if CQual++ issues no warnings (i.e., CQual++ says the code type-checks), then your program is guaranteed to be free of format string vulnerabilities. Conversely, if your code contains any format string vulnerabilities, then CQual++ is guaranteed to detect this fact so long as your code obeys all of the rules below. Disclaimer: if any of the rules below are violated, then CQual++ might find format string bugs that are in your code, but there are no guarantees. Moreover, CQual++ does not attempt to detect that you have violated the rules below. Rules:

- Your program must be written entirely in C. (We currently posit that our analysis is sound in C, but do not make this claim for C++ due to limitations in CQual++'s current implementation.) Your program must not contain any assembly code, any code in any other language, or self-generating or self-modifying code.
- Your program must be C99 compliant and compile successfully with gcc. (CQual++ does not duplicate all of the checks that gcc performs; it assumes that the program is valid C code.)
- You must provide all of the source code to your program. This includes the source code to all libraries and any other code linked into your program, except the C library.
- Do not use `dlopen()` to dynamically load and link libraries into the program at runtime. (Static and dynamic linking managed by the linker is ok.)
- Annotate all taint sources and trusting sinks with `$tainted` and `$untainted`, respectively. This has been done for the C library.
- Avoid undefined behavior. The C language standard defines a number of constructions as having "undefined behavior," meaning the compiler implementation may choose how to handle it. Dangerous undefined behavior includes but is not limited to:

[3] Note that the authors of his paper make this claim of soundness, and not the implementor of CQual++, Daniel S. Wilkerson.

  - Buffer-overruns and array bounds violations
  - Casts that can violate memory safety and other undefined behavior
  - Using deallocated memory or uninitialized memory.

- The CQual++ linker assumes certain things are consistent across files (translation units), after pre-processing. Generally, declare everything in a header file that is included in all source files that need those declarations. In particular:

  - Use the same declaration for any type that appears in multiple files, including structs, unions, typedefs.
  - Declare functions with prototypes before use, and use the same prototype for all function declarations and definitions in all files.
  - Symbols which link together in different translation units should have the same names, except that Oink also honors the gcc feature `__attribute__((alias))`.

  A formal set of rules for proper header file usage is enumerated by Srivastava et al., whose CMod tool checks these rules. [32]

- Avoid laundering taint through the program counter (implicit flows). This includes all control-flow, including but not limited to: for, while, and do-while loops; if and switch statements; computed goto; the ternary "`?:`" operator; short-circuiting operators, like "`&&`" and "`||`", and also any operating system calls that affect control, such as the use of signals.

- Be type-safe; avoid unsafe casts. CQual++ uses as its model of the program the C type system itself; the primary rule is therefore: honor the type system. Unfortunately many casts by definition break the type-system and the use of such casts is frequent in C. Casts can cause a problem for CQual++ if potentially untrustworthy data can flow across them. (In general, casts can also violate memory safety.) For example, casting the return value of `malloc()` is safe as `malloc()` generates data internally and is not annotated as returning untrustworthy data. We support a subset of casts as safe.

  Formal rules are defined in the full version of this rules document; examples of specific behavior that is allowed include casting between scalars, such as int to short, long to unsigned and casting a `Foo*` to a `void*` and then back to a `Foo*`. Examples of disallowed behavior including using a union to perform a hidden cast or passing a parameter of the wrong type to a variable-argument function.

- Do not cast or convert integer data into character data, because we make no attempt to track the taintedness of integer data. This especially applies to external operating system calls returning tainted integer data and to implicit flow via the program counter.

B. Appendix B: CQual extensions

We summarize some recent extensions to both CQual and CQual++ that made their analyses more precise.

B.1 Context sensitivity

The original CQual assigned a single type qualifier to the parameters of a function, which connected the arguments of all callers of the function. For example, in the following code:

```
int identity(int x) { return x; }
int foo($tainted t) { return identity(t); }
int $untainted bar($untainted u)
{ return identity(u); }
```

CQual would find a type violation at the return value of `identity`, yielding a false positive. Context sensitivity adds automatic inference of polymorphic types so that the type of `identity`'s parameters depends on the call site. [22]

B.2 Instance sensitivity

The original CQual treated struct members as regular variables. This yielded false positives if some instances of a struct had tainted data but others had untainted data, for example in:

```
struct S { char buf[100]; }
int foo(char $tainted *t, char $untainted *u) {
  struct S s1, s2;
  strcpy(s1.buf, t);
  strcpy(s2.buf, u);
}
```

Instance sensitivity (previously known as field sensitivity) treats the fields of each instance of a struct as separate variables. [22] An important optimization is to create the in-memory data structures for the instance members lazily.

B.3 Graph compaction

To reduce the memory footprint, Johnson's graph compaction techniques allow CQual++ to analyze each translation unit separately and produce a summary to be used when linking. This type of modular analysis can allow much larger programs to be analyzed. [23, §4.4]. Compaction does not affect the false positive or false negative rate, but it does significantly reduce memory usage.

B.4 Well-formedness constraints

CQual++ uses well-formedness constraints to capture the flow of taint from structs to their fields:

```
struct S { char data[100]; } s1;
read(socket, &s1, sizeof(struct S));
printf(s1.data);
```

Here, taint flows from `s1` (which became tainted by the `read()` call) to its member, `s1.data`. This technique has been previously used for finding user/kernel vulnerabilities [22, 16] and in our experience, it is necessary for finding many of the format string vulnerabilities we have seen in practice.

B.5 `void*` auto-unions

While C does not explicitly have polymorphic types, `void*` is commonly used to represent an "unknown type". CQual++ handles `void*` similarly to how it handles unions: In a union, each type has its own bucket. So if a union contained an int and a pointer, flow into the union instance as a pointer would connect to flows out as a pointer, but not as an integer. CQual++ treats `void*` as a pointer to an object whose type is the union of all possible types; a cast to a type *t* lazily adds *t* as a type. Thus, in the following example:

```
int foo(int $tainted i,
        char $untainted *s, int flag) {
  void *p; int i1; char *s1;
  if (flag) p = (void*) i;
  else      p = (void*) s;
  if (flag) i1 = (int) p;
  else      s1 = (char*) p;
}
```

`i1` is correctly inferred as tainted, and `s1` as untainted. If the rule for use of `void*` as described in Appendix A is followed, the analysis is sound. Another extension allows pointer-length integer types to also behave as `void*`, since this is a commonly used idiom. [23].