FormatGuard: Automatic Protection From `printf` Format String Vulnerabilities

Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman

WireX Communications, Inc. `http://wirex.com/`

Mike Frantzen
Purdue University

Jamie Lokier
CERN

Abstract

In June 2000, a major new class of vulnerabilities called "format bugs" was discovered when an vulnerability in WU-FTP appeared that acted *almost* like a buffer overflow, but wasn't. Since then, dozens of format string vulnerabilities have appeared. This paper describes the format bug problem, and presents FormatGuard: our proposed solution. FormatGuard is a small patch to `glibc` that provides general protection against format bugs. We show that FormatGuard is effective in protecting several real programs with format vulnerabilities against live exploits, and we show that FormatGuard imposes minimal compatibility and performance costs.

1 Introduction

In June 2000, a major new class of vulnerabilities called "format bugs" was discovered when an interesting vulnerability in WU-FTP appeared that acted *almost* like a buffer overflow, but wasn't [22]. Rather, the problem was the sudden realization that it is unsafe to allow potentially hostile input to be passed directly as the format string for calls to `printf`-like functions. The danger is that creative inclusion of % directives in the format string coupled with the lack of any effective type or argument counting in C's varargs facility allows the attacker to induce unexpected behavior in programs.

This vulnerability is made particularly dangerous by the `%n` directive, which assumes that the corresponding argument to `printf` is of type "`int *`", and writes back the number of bytes formatted so far. If the attacker crafts the format string, then they can use the `%n` directive to write an arbitrary value to an arbitrary word in the program's memory. This makes format bugs every bit as dangerous as buffer overflows [8]: the attacker can send a single packet of data to a vulnerable program, and obtain a remote (possibly `root`) shell prompt for their trouble. Since June 2000, format bugs have eclipsed buffer overflow vulnerabilities for the most common form of remote penetration vulnerability.

There are several obvious solutions to this problem, which unfortunately don't work:

Remove the `%n` feature: The `printf` `%n` directive is the most dangerous, because it induces `printf` to write data back to the argument list. It has been proposed that the `%n` feature simply be removed from the `printf` family of functions. Unfortunately, there exist real programs that actually use the `%n` feature (which is in the ANSI C specification [12]) so this would break an undesirable amount of software.

Permit Only Static Format Strings: Format bugs occur because the `printf` tolerates dynamic format strings. It has been proposed that `printf` be modified to insist that the format string be static. This approach fails because a large number of programs, especially those using the GNU internationalization library, generate format strings dynamically, so this too would break an undesirable amount of software.

Count the Arguments to `printf`: Because `%n` treats the corresponding argument as an `int *` an effective format bug attack must walk back up the stack to find a word that points to the right place, and/or output a sufficient number of bytes to affect the `%n` value. Thus the attacker nearly always must provide a format string that does not match the actual number of arguments presented to `printf`. If it can be done, this approach is effective in stopping format bug attacks. Unfortunately, the varargs mechanism that C employs to permit a variable number of arguments to a given function does not permit any kind of checking of either the type or count of the arguments with-

This work supported in part by DARPA contract N66001-00-C-8032.

out breaking the standard ABI for `printf`. Varargs permits the receiving functions to "pop" an arbitrary number and type of arguments off the stack, relying on the function itself to correctly interpret the contents of the stack. A "safe varargs" that passes either an argument count or an argument terminator could be built. However, this modified varargs protocol would not be compatible with any existing dynamic or static libraries and programs.

FormatGuard, our proposed solution to the format bug problem, uses a variation on argument counting. Instead of trying to do argument counting on varargs, FormatGuard uses particular properties of GNU CPP (the C PreProcessor) macro handling of variable arguments to extract the count of actual arguments. The actual count of arguments is then passed to a safe `printf` wrapper. The wrapper parses the format string to determine how many arguments to expect, and if the format string calls for more arguments than the actual number of arguments, it raises an intrusion alert and kills the process.

The rest of this paper is organized as follows. Section 2 elaborates on the `printf` format string vulnerability. Section 3 describes FormatGuard; our solution to this problem. We present security testing in Section 4, compatibility testing in Section 5, and performance testing in Section 6. Section 7 relates FormatGuard to other defenses for `printf` format string vulnerabilities. Section 8 presents our conclusions.

2 `printf` Format String Vulnerabilities

The first known discovery of format bugs was by Tymm Twillman while auditing the source code for ProFTPD 1.2.0pre6. Basic details were released to the ProFTPD maintainers and a Linux security mailing list in early September 1999, and then publicly released via BugTraq [23] later that month. Other individuals then wrote a few other format bug exploits, but they were not immediately released to the public. It wasn't until June 2000 [22] that format bugs became widely recognized, when numerous exploits for various common software packages started to surface on security mailing lists.

Format bugs occur fundamentally because C's varargs mechanism is type unsafe. Varargs provides a set of primitives for "popping" arguments off the stack. The number of bytes "popped" depends on the type of the *expected* argument. At no time is either the type or the existence of the argument checked: the function receiving the arguments is entirely responsible for popping the correct number, type, and sequence of arguments.

The `printf` family of functions (`syslog`, `printf`, `fprintf`, `sprintf`, and `snprintf`) use varargs to support the ability to output a variable number of arguments. The format string tells the function the type and sequence of arguments to pop and then format for output. The vulnerability occurs if the format string is bogus, as is the case when the format string is actually provided by the attacker.

An example of this situation occurs when a programmer writes "`printf(str)`" as a short-hand for "`printf("%s", str)`". Because this idiom is perfectly functional, and easier to type, it has been used for many years. Unfortunately, it is also vulnerable if the attacker inserts spurious % directives in the `str` string.

The `%n` directive is particularly dangerous: it assumes that the corresponding argument to `printf` is of type "`int *`", and writes back the number of bytes formatted so far into the storage pointed to by the `int *`. The result of spurious `%n` directives in `printf` format strings is that the attacker can "walk" back up the stack some number of words by inserting some number of `%d` directives, until they reach a suitable word on the stack, and treating that word as an `int *`, use a `%n` to overwrite a word *nearly anywhere* in the victim program's address space, creating substantial security problems. If buffers are of appropriate size, the attacker can also use the buffer itself as a source of words to use as the `int *` pointer, making it even easier for the attacker to use %n to modify an arbitrary word of memory.

Thus the essential features that create format vulnerabilities are the basic lack of type safety in the C programming language, the `%n` directive that induces unexpected side-effects in `printf` calls, and the casual use of un-filtered user-input as a `printf` format string due to the common assumption that this is a safe practice. Detailed descriptions of the exploitation of `printf` vulnerabilities have been written by Bouchareine [3, 4] and Newsham [14].

3 FormatGuard: Protection from Funny Format Strings

An essential part of the format string attack described in Section 2 is that the attacker provides some number of spurious `%` directives in user-input that is subsequently used as a format string for a `printf` call. FormatGuard defends against format bug attacks by comparing the number of actual arguments presented to `printf` against the number of arguments called for by the format string. If the actual number of arguments is less than the number of arguments the format string calls for, then

```
#define printf                    mikes_print(&cnt, print0

#define print0(x, args...) x ,print1(## args)
#define print1(x, args...) x+(++cnt-cnt) ,print2(## args)
#define print2(x, args...) x+(++cnt-cnt) ,print3(## args)
...
void mikes_print(int *args, char *format, ...);
```

Figure 1 Frantzen's Argument Counter

FormatGuard deems this call to be an attack, `syslog`'s the attempt, and aborts the program. As previously discussed, C's varargs mechanism does not permit argument counting, and so the trick is to find a way to count the arguments. Section 3.1 describes how FormatGuard implements argument counting with GNU CPP, and Section 3.2 describes how FormatGuard uses the argument count to implement a protected `printf` wrapper.

3.1 Counting Arguments

Frantzen first proposed the CPP method on July 25, 2000 [10]. This method exploits the way that CPP (the C PreProcessor) handles variable argument lists. Using the macro production shown in Figure 1, CPP can count the arguments by stripping the leading argument away in each production, similar to the Lisp CAR/CDDR idiom.

On September 25, 2000 Lokier [13] proposed an improved method of using CPP variable argument syntax for argument counting. Lokier's method allowed WireX to develop argument counting for FormatGuard that is recursive, reentrant, and thus thread safe, shown in Figure 2. This code function as follows:

1. The `__formatguard_counter` production serves to capture the zero-case, so that calls to `printf` containing only a null argument list are handled correctly.

2. The `__formatguard_count1` production appends a sequence of counter place holding arguments 5, 4, 3, 2, and 1. It does so by compresses the variable argument list from `__formatguard_counter` into a single token `y`.

3. Finally, `__formatguard_count2` re-expands the compressed variable argument group `y` from `__formatguard_count1`, but in doing so maps the trailing counter place holding arguments to another series of place holders, such that the first place holder from `__formatguard_count1` is mapped to the argument `n`, which in turn is the sole output of this sequence of productions.

The result of the above three productions is that place holding counter arguments are shifted to the right in proportion to the number of arguments presented to `printf` in the first place, and therefore `__formatguard_counter()` returns the count of the number of arguments presented.

The "-1" is a kludge factor to accommodate the existence of the format string itself. The `__PRETTY_FUNCTION__` macro is inserted to allow meaningful error reporting. Figure 3 presents an example, expanding an argument list of two elements: (a, b) to return a value of 2.

3.2 Protected `printf`

Figure 2 shows a definition for a `printf` macro that includes a call to the argument counter described in Section 3.1, and passes this count to a `__protected_printf` function. The purpose is to prevent the attacker from injecting spurious % directives into an un-filtered format statements, by ensuring that the number of % directives is less than or equal to the actual number of arguments provided.

Parsing `printf` format strings can be difficult. FormatGuard determines the number of % directives in a

```
#define __formatguard_counter(y...) __formatguard_count1 ( , ##y)
#define __formatguard_count1(y...) \
   __formatguard_count2 (y, 5,4,3,2,1,0)
#define __formatguard_count2(_,x0,x1,x2,x3,x4,n,ys...)  n

#define printf(x...)     \
    __protected_printf (__PRETTY_FUNCTION__, \
        __formatguard_counter(x) - 1 , ## x)
```

Figure 2 FormatGuard Implementation, Simplifed to Handle 5 or Fewer Arguments

```
formatguard_counter (a, b)
```
which gets expanded to

```
__formatguard_count1 ( , a, b)
```
which the second macro expands to

```
__formatguard_count2 (  , a, b, 5, 4, 3, 2, 1, 0)
```
The arguments to match the __formatguard_count2 rule in the following way:

```
__formatguard_count2 (  ,  a,  b , 5,  4,  3,  2,  1, 0)
                       ^   ^   ^   ^   ^   ^   ^   ^  ^
                       |   |   |   |   |   |   |   |  |
                       _  x0  x1  x2  x3  x4   n   ys...
```
Thus n gets matched to 2, which is what is returned.

Figure 3 Example Expanding the FormatGuard Macro

format string accurately (i.e. getting the same answer that `printf` will get) by borrowing the `parse_printf_format` function from the `glibc` library itself, which conveniently enough, returns exactly the number of arguments to be formatted.

If the number of % directives exceeds the number of arguments provided to `printf`, then `__protected_printf` deems a format attack to be under way. Note that the attack is *mid-way* through: the attacker has not corrupted any significant program state, but the attacker has put the victim program in an untenable position; at the very least, it is not possible to successfully complete the `printf` call. FormatGuard responds by `syslog`'ing the intrusion attempt with an entry similar to:

```
Feb  4 04:54:40 groo foo[13128]: ImmunixOS format error - mismatch of 2 in printf called by main
```
where "`foo`" is the name of the victim program, "`printf`" is one of the FormatGuard-wrapped functions (`syslog`, `printf`, `fprintf`, `sprintf`, and `snprintf`) and "`main`" is the function that `printf` was called from. FormatGuard then aborts the process to prevent the attacker from taking control, similar to the way StackGuard handles buffer overflow attacks [8, 6].

3.3 FormatGuard Packaging: Modified `glibc`

In Linux-like systems, the `printf` family of functions is provided by the `glibc` library. The `__formatguard_count` macros shown in Figure 2 are inserted into the `/usr/include/stdio.h` file and the `__protected_printf` function is inserted into the `glibc` library itself. Thus FormatGuard is packaged as a modified implementation of `glibc 2.2`.

Note that, despite the packaging of FormatGuard with a library package, programs that are to benefit from FormatGuard protection must be re-compiled from source, using the FormatGuard version of `stdio.h`. In many cases, this imposes a substantial workload on people wishing to protect an entire system with FormatGuard. However, WireX has included both FormatGuard and StackGuard [8, 6] in the latest edition of Immunix Linux. Both the Immunix system and the FormatGuard implementation of `glibc` are available for download from `http://immunix.org/`

4 Security Effectiveness

FormatGuard presents several security limitations in the form of various cases that FormatGuard does not protect against, which we present in Section 4.1. Section 4.2 presents our testing of live exploits against actual vulnerabilities found in widely used software.

4.1 Security Limitations

FormatGuard fails to protect against format bugs under several circumstances. The first is if the attacker's format string undercounts or matches the actual argument count to the printf-like function, then FormatGuard will fail to detect the attack. In theory, it is possible for the attacker to employ such an attack by creatively mis-typing the arguments, e.g. treating an int argument as `double` argument. In practice, no such attacks have been constructed, and would likely be brittle.Insisting on an exact match of arguments and % directives would induce false-positives: it is quite common for code to provide more arguments than the format string specifies. There is even an example within the `glibc` code itself.

Table 1: FormatGuard Security Testing Against Live Exploits

Program	Result Without FormatGuard	Result With FormatGuard
`wu-ftpd` [22]	`root` shell	`root` shell
`cfengine` [20]	`root` shell	FormatGuard alert
`rpc.statd` [19]	`root` shell	FormatGuard alert
`LPRng` [24]	`root` shell	FormatGuard alert
`PHP 3.0.16` [17]	`httpd` shell	FormatGuard alert
`Bitchx` [26]	user shell	FormatGuard alert
`xlock` [2]	`root` shell	FormatGuard alert
`gftp`	user shell	user shell

The second limitation is that a program may take the address of printf, store it in a function pointer variable, and then call via the variable later. This sequence of events disables FormatGuard protection, because taking the address of printf does not generate an error, and the subsequent indirect call through the function pointer does not expand the macro. Fortunately, this is not a common thing to do with a printf-like function.

The third limitation is that FormatGuard cannot provide protection for programs that manually construct stacks of varargs arguments and then make direct calls to `vsprintf` (and friends). Because such programs can dynamically construct a variable list of arguments, it is not possible to count the arguments presented through static analysis.

A variation on this problem is libraries that present printf-like functions.These libraries in turn call vsprintf directly, and thus do not get FormatGuard protection. For example the `GLib` library (part of GTK+, not to be confused with `glibc`) provides a rich family of printf-like string manipulation functions. To address this class of problems, we are considering expanding FormatGuard protection beyond glibc into other libraries that provide printf-like functionality, such as GLib.

In practice, the only limitations that we have encountered are the direct calls to vsprintf and the non-glibc library calls to vsprintf, as we show in Section 4.2.

4.2 Security Testing

To test the security value of FormatGuard, we tested it against real vulnerable programs and real live exploit programs collected from the wild. The test procedure is to run the attack exploit against the vulnerable version of the program, to verify that the vulnerability is legitimate and the attack program is functional. We then recompile the vulnerable program from source, including FormatGuard protection, *without* repairing the vulnerability, and re-run the attack against the vulnerable program. Because of the level of integration effort required to deploy FormatGuard, we consider only the Immunix system, and thus consider only the vulnerabilities for the Linux/x86 platform. The results are shown in Table 1.

We note (with some irony) that wu-ftpd was the catalyst for the format string vulnerability problem [22, 5] and yet is one of the few format bugs that we found that FormatGuard does *not* stop. Investigation revealed that this is because wu-ftpd completely re-implements its own `printf` functions (as described in Section 4.1) and thus does not use the hardened `printf` functions that FormatGuard supplies. In similar fashion, FormatGuard failed to protect gftp, which uses the family `printf`-like functions found in the `GLib` library.

While this is unfortunate for wu-ftpd and for FormatGuard, it also provides interesting additional evidence that synthetic "biodiversity" in the form of n-version programming (re-implementing the same functionality by different people) does not necessarily provide resistance against common security failure modes [7]. In this case, biodiversity seems to have actually degraded security, because the semantic failure was replicated across implementations, necessitating the replication of FormatGuard protection across these implementations.

We also note (with further irony) that the PHP vulnerability [17] is only manifest in an unusual configuration that involves *extra* logging. The cause is unsafe format string handling in the call to `syslog`. The interesting factor to note is that security-conscious administrators often increase the level of logging on their systems to provide enhanced security. If, as these vulnerabilities tend to indicate, it is the case that format bugs often result from unsafe format string handling in `syslog` calls, then increasing logging levels may occasionally have the opposite from intended effect, and actually open the host to new vulnerabilities, further increasing the need for protection against format bugs.

5 Compatibility Testing

FormatGuard is intended to be highly transparent: FormatGuard protection should not cause programs to fail to compile or run, and the "false positive" rate (legitimate computation reported as format string attacks) should be asymtopic to zero. To be effective, FormatGuard needs to compile and run literally millions of lines of production C code. In this section, we describe the extent to which we have achieved these goals.

For the most part, we have succeeded. FormatGuard has been used to build the Immunix Linux distribution, which includes 500+ RPM packages, comprising millions of lines of C code. These Immunix systems have been running in production on assorted WireX servers and workstations since October 2000. These systems function normally, being not noticabley different from non-FormatGuard machines. To date, the observed false positive rate is zero. The experience has been similar to the StackGuard "eat our own dog food" experience [6].

However, FormatGuard is also less transparent than StackGuard: of the approximately 500 packages that we compiled with FormatGuard in the construction of the Immunix system, two required modification to accommodate StackGuard protection, while approximately 70 required modification to accommodate FormatGuard protection. These modifications were required to treat C programming idioms that break when CPP directives (macros and #ifdef statements) are included inside the arguments to a macro[1], as in the following C programming idiom:

```
printf("Hello world"
#ifdef X
" is X enabled"
#endif
"\n");
```

CPP expands the above code into either

```
printf("Hello world" " is X enabled"
"\n");
```

or

```
printf("Hello world" "\n");
```

which is a convenient way of conditionally compiling strings. This creates problems for FormatGuard, because FormatGuard makes `printf` a macro instead of a pure function, and CPP does not support `#ifdef` (or other CPP directives) as argument to macros, and so the above code will not work.

The work-around is to put the printf call in parentheses, which disables macro expansion, e.g. write `(printf)("Hello world")` instead of `printf("Hello world")`. This disables FormatGuard protection for this call *only*. Thus the developer must ensure that the resulting naked call to printf is safe. However, the problematic cases almost always involve static strings being conditionally compiled, so this is rarely a difficult problem.

Once code has been compiled with FormatGuard, there are additional limitations:

- Non-FormatGuard programs can link to FormatGuard libraries without problems. However, these programs to not get the benefit of FormatGuard protection, and are still vulnerable to format bugs.
- FormatGuard programs *cannot* link to non-FormatGuard libraries unless the FormatGuard version of `glibc` is present.

Thus the Immunix platform easily hosts foreign programs, but FormatGuard-protected programs do not run on foreign platforms without some intervention.

6 Performance Testing

Any run-time security defense will impose performance costs, due to additional run-time checks that it is performing. However, a security enhancement must be efficient enough that these overhead costs are minimal with respect to the defense they provide. Ideally, the cost should be below noticability for the intended user base.

FormatGuard achieves this level of performance. Overhead is only imposed on the run-time cost of calling `*printf` and `syslog` functions. Section 6.1 presents microbenchmarks that show the precise overhead imposed on calling these functions. Section 6.2 shows macrobenchmarks that measure the imposed overhead on (fairly) `printf`-intensive programs.

1. Rumor has it that the ANSI C standard [1] mandates that `printf` is not a macro. This is not true [16].

```
int main(void) {
    int i = 0;
    int counter = 100000000;

    while (i != counter) {
        printf(“%s %s %s\n”, “a”, “b”, “c”);
        i++;
    }
    printf(“%d\n”, i); // force compiler to retain the loop
    exit(0);           // & not optimize it away
}
```

Figure 4 Microbenchmark

6.1 Microbenchmarks

We measure the marginal overhead of FormatGuard protection on `printf` calls with a tight loop as shown in Figure 4. We measured the performance of this loop in single-user mode with and without FormatGuard protection, subtract out the run time of a loop executed without the `printf` to eliminate the loop overhead, and then divide to get the %overhead. The run time with FormatGuard was 19.09 seconds, without FormatGuard was 13.97 seconds, and the loop overhead was 0.032 seconds. Thus FormatGuard imposed a marginal overhead of 37% on a trivial `printf` call.

We then repeated the above experiment, but replaced the `printf` call with one that formats a through z, rather than just three letters. The FormatGuard run time was 134.7 seconds, without FormatGuard 99 seconds, and 0.032 second loop overhead has become negligible. Thus FormatGuard imposed a marginal slowdown of 36% on a more complex `printf` call, and we conclude that FormatGuard imposes a fairly consistent 37% marginal overhead on most `printf` calls.

6.2 Macrobenchmarks

Most programs do not spend much time running the `printf` function; `printf` is an I/O function, and even programs that are I/O intensive tend to format their own data rather than using `printf`. The `printf` function is mostly used to format error-handling code. So we had some difficulty finding programs that would show measurable degradation under FormatGuard. We found such a program in man2html [25], which uses `printf` extensively to output HTML-formatted man pages.

Our test was to batch translate 79 man pages through man2html, which is 596 KB of input. The test was run multiple times in single-user mode on a system with 256 MB of RAM, so I/O overhead was minimal. The result is that the batch takes 0.685 seconds without FormatGuard, and 0.698 seconds with FormatGuard. Thus in an arguably near worst-case application scenario, FormatGuard imposes 1.3% run-time overhead. In most cases, overhead is considerably lower, often negligible.

7 Related Work

Work related to FormatGuard is divided into analysis of format string vulnerabilities, which we described in Section 2, and work to protect programs against such vulnerabilities, which we describe here.

Fundamentally, format bugs exist because of the tension between strong type checking, and convenient polymorphism. C and Pascal made opposite choices in this regard: Pascal chose the safe route of strict type checking, which means that Pascal functions can never be spoofed with this kind of attack, but also means that it is difficult to write a convenient generic I/O function like `printf` in Pascal [11]. Conversely, C chose a completely type-unsafe `varargs` mechanism that makes it impossible to statically type check a polymorphic function call.

More recent programming languages such as ML have solved this tension with *type inference*, but these techniques are difficult to apply to C programs [15, 27]. Wagner et al [21] present a compromise solution in which a “taint” *type qualifier* is added to the C language, allowing programmers to designate data as “tainted” (provided by the adversary) and the compiler tracks the data usage through the program as tainted. If tainted data is presented to printf-like functions as the format string, the compiler flags an error. The main advantage to this approach is that it detects potential vulnerabilities at compile time, rather than when the attacker tries to exploit them. The main limitation of this approach is that it is not transparent: functions that collect user-input must be manually annotated as “tainted”.

Since it is problematic to properly type check C programs, more pragmatic means have emerged to deal specifically with format bugs. Alan DeKok wrote PScan [9] to scan C source code looking for potential format bugs by looking for the simple/common case of a `printf`-like function in which the *last* parameter is also the format string, *and* the format string is not static.

GCC itself has an un-documented feature where "`-Wformat=2`" will cause GCC to complain about non-static format strings. This is over-general, in that it complains about legitimate code, such as internationalization support, which uses functions to generate format strings. However, Joseph Myers has implemented an enhancement to `-Wformat` that unconditionally complains about the "`printf(foo)`" case. The functionality is essentially similar to PScan, with the advantage that it is built into the compiler, and the disadvantage that it is only available in a pre-release version of the GCC compiler.

Both PScan and the -Wformat enhancement offer the advantage that they provide static warnings, so the developer knows at compile time that there is a problem, providing an opportunity to fix the problem before the code ships. However, because these static analysis methods are heuristics, they are subject to both false negatives (missing vulnerabilities) and false positives (mis-identifying non-vulnerabilities) and thus they present an additional burden on developers. The additional burden, in turn, is problematic because developers are never actually required to use those tools, and thus may choose to omit them if they prove troublesome.

In contrast, runtime techniques present a low burden on developers (see Section 5) and uniformly improves the security assurance of applications. `libformat` [18] is a library that aborts programs if they call printf-like functions with a format string that is writable and contains a `%n` directive. This technique is often effective, but because both writable format strings and `%n` directives are legal, it can be subject to false positives.

libsafe cite{libsafe} is a library approach to defending against buffer overflow attacks. In version 2.0, libsafe has added protection against format bugs by applying their technique of the library inspecting the call stack for plausible arguments, in this instance rejecting `%n` directives that try to write to the function's return address on the stack. The strength of this approach is that, like libformat, it affords protection to binary programs, and protects against format bugs in direct calls to `vsprintf` (see Section 4.1). The limitations of libsafe are that it cannot protect code compiled with the "`no_frame_pointer`" optimization, and that it only protects against format string attacks aimed at the activation record.

FormatGuard tries to achieve some of the benefits of both static and run-time techniques. By using a source-code re-compilation technique, FormatGuard achieves high precision, resulting in few false negatives, and no false positive, presenting a very low burden on developers. Even if the original developer chose not to do anything about format vulnerabilities, an end-user of an open source product can re-compile the product with FormatGuard and gain protection from format bugs the developer failed to discover.

8 Conclusions

Format bugs are a dangerous and pervasive security problem that appeared suddenly in June 2000, and continues to be a major cause of software vulnerabilities. FormatGuard protects vulnerable programs against this problem. We have shown that FormatGuard is effective in stopping format bug attacks, imposes minimal compatibility, problems, and has a practical performance penalty of less than 2%. FormatGuard is incorporated into WireX's Immunix linux distribution and server products, and is available as a GPL'd patch to `glibc` at `http://immunix.org`

References

[1] American National Standards Institute, Inc. *Programming Language – C, ANSI Standard X3.159*. American National Standards Institute, Inc., 1989.

[2] "bind". xlock (exec) Input Validation Error. Bugtraq mailing list, `http://www.securityfocus.com/vdb/bottom.html?vid=1585`, August 15 2000.

[3] Kalou/Pascal Bouchareine. Format String Vulnerability. `http://plan9.hert.org/papers/format.html`, July 18 2000.

[4] Pascal Bouchareine. User Supplied Format String Bug. `http://julianor.tripod.com/usfs.html`, July 2000.

[5] Crispin Cowan. Format Bugs in Windows Code. Vuln-dev mailing list, `http://www.securityfocus.com/archive/82/81455`, September 10 2000.

[6] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting Systems from Stack Smashing Attacks with StackGuard. In *Linux Expo*, Raleigh, NC, May 1999.

[7] Crispin Cowan, Heather Hinton, Calton Pu, and Jonathan Walpole. The Cracker Patch Choice: An Analysis of Post Hoc Security Techniques. In *Proceedings of the 19th National Information Systems Security Conference (NISSC 2000)*, Baltimore, MD, October 2000.

[8] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Conference*, pages 63–77, San Antonio, TX, January 1998.

[9] Alan DeKok. PScan: A limited problem scanner for C source files. Bugtraq mailing list, `http://www.securityfocus.com/archive/1/68688` and the web `http://www.striker.ottawa.on.ca/ aland/pscan/`, July 7 2000.

[10] Mike Frantzen. Poor Man's Solution to Format Bugs. Vuln-dev mailing list, `http://www.securityfocus.com/archive/1/72118`, July 25 2000.

[11] Brian Kernighan. Why Pascal is not my Favorite Programming Language. Report 100, AT&T Bell Labs, Murry Hill, NJ, July 1981. submitted for publication.

[12] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1988.

[13] Jamie Lokier. Varargs macros subtly broken. GCC mailing list, `http://gcc.gnu.org/ml/gcc/2000-09/msg00604.html`, September 25 2000.

[14] Tim Newsham. Format String Attacks. Bugtraq mailing list, `http://www.securityfocus.com/archive/1/81565`, September 9 2000.

[15] Robert O'Callahan and Daniel Jackson. Lackwit: A Program Understanding Tool Based on Type Inference. In *Proceedings of International Conference on Software Engineering (ICSE'97)*, Boston, MA, May 1997.

[16] P.J. Plauger. *Standard C Library*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[17] "Weld Pond". @stake Advisory: PHP3/PHP4 Logging Format String Vulnerability (A 101200-1). Bugtraq mailing list, `http://www.securityfocus.com/archive/1/139259`, October 12 2000.

[18] Tim J. Robbins. libformat. `http://the.wiretapped.net/security/host-security/libformat/`, November 2001.

[19] "ron1n". statdx2 - linux rpc.statd revisited. Bugtraq mailing list, `http://marc.theaimsgroup.com/?l=bugtraq&m=97123424719960&w=2`, October 11 2000.

[20] Pekka Savola. Very probable remote root vulnerability in cfengine. Bugtraq mailing list, `http://marc.theaimsgroup.com/?l=bugtraq&m=97050677208267&w=2`, October 2 2000.

[21] Umesh Shankar, Kunal Talwar, Jeff Foster, and David Wagner. Automated Detection of Format-String Vulnerabilities. In *USENIX Security Symposium*, Washington, DC, August 2001.

[22] "tf8". Wu-Ftpd Remote Format String Stack Overwrite Vulnerability. `http://www.securityfocus.com/bid/1387`, June 22 2000.

[23] Tymm Twillman. Exploit for proftpd 1.2.0pre6. Bugtraq mailing list, `http://www.securityfocus.com/templates/archive.pike?list=1&mid=28143`, September 1999.

[24] "venomous". LPRng remote root exploit. Bugtraq mailing list, `http://marc.theaimsgroup.com/?l=bugtraq&m=97683900820267&w=2`, December 14 2000.

[25] Richard Verhoeven. man2html. `http://wsinwp01.win.tue.nl:1234/`, February 10 2000.

[26] "Zinx Verituse". BitchX - more on format bugs? Bugtraq mailing list, `http://www.securityfocus.com/archive/1/68256`, July 3 2000.

[27] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *NDSS (Network and Distributed System Security)*, San Diego, CA, February 2000.