Preventing Variadic Function Attacks Through Argument Width Counting

Brennan Ward
University of South Florida

Follow this and additional works at: https://digitalcommons.usf.edu/etd

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Scholar Commons Citation

This Thesis is brought to you for free and open access by the USF Graduate Theses and Dissertations at Digital Commons @ University of South Florida. It has been accepted for inclusion in USF Tampa Graduate Theses and Dissertations by an authorized administrator of Digital Commons @ University of South Florida. For more information, please contact digitalcommons@usf.edu.
Preventing Variadic Function Attacks

Through Argument Width Counting

by

Brennan Ward

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering Department of Computer Science and Engineering College of Engineering University of South Florida

Major Professor: Jay Ligatti, Ph.D.
Mehran Mozaffari Kermani, Ph.D.
Yao Liu, Ph.D.

Date of Approval: October 28, 2022

Keywords: Format String Attacks, Computer Security, Buffer Overflows

Copyright © 2022, Brennan Ward
# Table of Contents

List of Tables ................................................................................................................... iii

List of Figures ................................................................................................................... iv

Abstract .............................................................................................................................. v

Chapter 1: Introduction ...................................................................................................... 1
  1.1 Problem Statement ...................................................................................................... 1
  1.2 Background ................................................................................................................ 1
      1.2.1 Variadic Functions ................................................................................................. 2
      1.2.2 The x86 Stack ....................................................................................................... 2
      1.2.3 Calling Conventions ............................................................................................ 3
      1.2.4 Format-String Attacks ........................................................................................ 4
  1.3 Thesis Overview ......................................................................................................... 5

Chapter 2: Literature Review .......................................................................................... 7
  2.1 FormatGuard ............................................................................................................. 7
  2.2 HexVASAN ............................................................................................................... 8
  2.3 LibSafe 2.0 .............................................................................................................. 10
  2.4 Lisbon ...................................................................................................................... 11
  2.5 White-Listing .......................................................................................................... 13

Chapter 3: Proposed Solution ......................................................................................... 16
  3.1 Argument Width Counting ....................................................................................... 16
  3.2 Alternative Solutions ............................................................................................... 18

Chapter 4: Implementations ........................................................................................... 20
  4.1 In x86 Assembly ....................................................................................................... 20
  4.2 Under GCC ............................................................................................................... 23
  4.3 Under ChibiCC ........................................................................................................ 24

Chapter 5: Empirical Analysis ....................................................................................... 28
  5.1 Test Methodology ..................................................................................................... 28
  5.2 Microbenchmark Results ......................................................................................... 29
  5.3 Discussion ................................................................................................................. 30
  5.4 Comparison ............................................................................................................... 30

Chapter 6: Conclusion ..................................................................................................... 32
References .................................................................................................................................36
List of Tables

Table 1 An overview of the C format specifiers (% directives) including data types and functionality ................................................................. 5

Table 2 A summary of prior works .......................................................................................................................... 15

Table 3 Microbenchmark results ....................................................................................................................... 29

Table 4 A replication of Table 2 including AWC .............................................................................................. 35
List of Figures

Figure 1  An example of a stack frame for a function .................................................................4
Figure 2  Three views of the stack ..................................................................................................18
Figure 3  An example of a variadic C function call (left) ...............................................................21
Figure 4  An instrumented variadic function body .........................................................................22
Figure 5  A diff showing code added to ChibiCC that emits two x86 instructions ......................26
Figure 6  A diff showing changes made to va_start .................................................................26
Figure 7  Assembly instructions corresponding to the new code added to va_start ..............26
Figure 8  A diff showing the error-handling code added to va_arg .............................................27
Figure 9  A variadic summation function, used for benchmarking AWC .................................28
Abstract

Format String attacks, first noted in June 2000 [1], are a type of attack in which an adversary has control of the string argument (the format string) passed to a string format function (such as `printf`). Such control allows the attacker to read and write arbitrary program memory. To prevent these attacks, various methodologies have been proposed, each with their own costs and benefits. I present a novel solution to this problem through argument width counting, ensuring that such format functions cannot access stack memory beyond the space where arguments were placed. Additionally, I show how this approach can be expanded to all variadic functions, and demonstrate an implementation of this approach within a C compiler.
Chapter 1: Introduction

Variadic function attacks refer to a class of attacks in which an attacker can force a running program to access memory that would be inaccessible to a function during normal execution [2]. The consequences of this memory access vary, based on the function that is being exploited. For example, a common subset of these attacks, known as format string attacks, allow an attacker to read or write from arbitrary program memory [1]. For this reason, variadic function attacks have the potential to be exceptionally dangerous.

1.1 Problem Statement

While many solutions have been proposed to prevent variadic function attacks and format-string attacks, none have seen widespread adoption [1, 2, 3, 4, 5]. Widely adopted features are often weaker (meaning they are preventative measures, not protections) than the aforementioned solutions, such as the “-Wformat” warning option included in the GNU Compiler Collection (GCC) [6]. The problem, then, is creating a solution which can easily be widely adopted (by requiring no effort from the programmer) and will yield a significant or total mitigation of variadic function attacks.

1.2 Background

A full understanding of the severity of this problem (and the motivation for resolving it) can only come with an understanding of the details surrounding it. Specifically, one must know what a variadic function is, how they are most commonly implemented, and how the most common attack vector (format string attacks) is exploited. Additionally, some knowledge about the x86 call stack and calling conventions is required.
1.2.1 Variadic Functions

A variadic function is a function that takes a variable number of arguments. In C, as defined by the ISO C standard, these functions always take at least one argument, and then an optional number of supplementary arguments known as variadic arguments or varargs [7]. The ISO C standard does not provide a mechanism for accessing the varargs of a variadic function, which means that it is up to the program (or more often the compiler) [7]. In most C compilers (including GCC and Clang), varargs are accessed through a series of macros: va_start, va_arg, va_end, and va_copy. A struct called va_list is used to record data about the next vararg to be accessed, the specifications of which are internal to the particular compiler being used. In general, va_start is responsible for populating the initial state of the va_list, va_arg retrieves the next argument (of a type specified by the caller), va_copy makes a clone of the va_list, and va_end performs cleanup work. The problematic part of this lies in how va_arg is most commonly implemented, as it performs no verification that the next argument being accessed actually exists. This lack of validation is what allows attackers to gain access to program memory that should be inaccessible, so long as they can trigger additional invocations of va_arg.

1.2.2 The x86 Stack

In x86 assembly (a common compile target for C programs), the stack is a portion of program memory where local data (relevant to the current running function) is allocated. This includes things such as function parameters, local variables, return addresses, and copies of registers from before the current running function was called. Every time a function is called, a certain amount of space is allocated on the stack to hold this necessary information. This space is known as a stack frame, and each function is (by convention) restricted to accessing the data in its own stack frame. Unfortunately, there is no standard runtime mechanism that ensures a
function is accessing its own frame, and certain out-of-frame data is of high value to an attacker. If an attacker can view the address of a piece of stack memory, they can defeat Address Space Layout Randomization (ASLR), and if an attacker can write to the stack, they can take control of the program by manipulating return addresses [9]. With this in mind, ensuring safety of the stack is critical to ensuring program security.

1.2.3 Calling Conventions

Knowing that the stack holds execution-critical data is one part of a whole; the other is knowing where that data is, relative to the stack pointer. If an attacker gains access to the stack through a variadic function attack, they still need to know what calling convention was used when the exploited function was called. A function’s calling convention specifies the layout of the stack frame that the function will expect when it is called, and also specifies who is responsible for cleaning up the stack (the caller or the callee).

In the cdecl and stdcall function call conventions, function arguments are passed right to left on the stack, a scratch space for holding dynamically allocated memory [8]. In the fastcall convention, arguments are passed in registers until there is no space remaining, and then the stack is used, right to left. For the purposes of this paper, only the cdecl calling convention will be used. Omission of the alternative calling conventions is done for brevity, and not for any other reason. The reader can infer what would change if another convention were used, based on the definition of the other convention.

That said, it is now possible to provide a full explanation of what happens during a function call (variadic or not) under the assertions made earlier (C source, x86 target, cdecl convention).
The first step of calling a function is preparation of the stack. This is done by the caller, before the function call instruction is executed. The caller must push, in order, the value of EBP (a register holding the stack base pointer, to be restored after the function returns), the return address (a pointer to the instruction after the call), and the function arguments, in reverse order (right to left, if looking at the source of a function). After this step, the prepared stack frame will look like the one shown below.

![Stack Frame Diagram](image)

Figure 1 An example of a stack frame for a function. Data that may be accessed by va_arg is in green, while data that should not be accessed is in red. Local variables are omitted, but would be below the arguments. The caller’s EBP is the stack base pointer of the prior frame for restoration during the return process.

After this setup, execution is transferred to the callee. The callee can then access this stack data as necessary to perform whatever task it is designed to do. Finally, execution will return to the specified return address (set by the caller), and this stack frame will be discarded.

1.2.4 Format-String Attacks

Formatting functions (printf, fprintf, ..., etc.) are variadic functions. Their required (first) argument is the format string, passed as a char*, and the remaining arguments are data that is to be inserted into the format string. To specify what data should be inserted, the format string contains a number of % directives (or format specifiers), each one corresponding to a specific variadic argument. Most format specifiers simply expect a certain type of argument to be present, and injects a string representation of that argument into the format string (replacing the
specifier). One particular specifier (\%n) allows for writing data. An overview of the specifiers is shown below.

Table 1  An overview of the C format specifiers (\% directives) including data types and functionality [10].

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Data Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c</td>
<td>char</td>
<td>Inserts a character</td>
</tr>
<tr>
<td>%d</td>
<td>short/int/long</td>
<td>Inserts the string value of a integer</td>
</tr>
<tr>
<td>%e</td>
<td>float/double</td>
<td>Inserts the exponential form of a float</td>
</tr>
<tr>
<td>%f</td>
<td>float</td>
<td>Inserts the string form of a float</td>
</tr>
<tr>
<td>%g</td>
<td>float/double</td>
<td>Inserts the shortest of %e and %f</td>
</tr>
<tr>
<td>%hi</td>
<td>short</td>
<td>Inserts the string value of a signed short</td>
</tr>
<tr>
<td>%hu</td>
<td>ushort</td>
<td>Inserts the string value of an unsigned short</td>
</tr>
<tr>
<td>%i</td>
<td>short/int/long</td>
<td>Same as %d for format functions</td>
</tr>
<tr>
<td>%l</td>
<td>long</td>
<td>Inserts the string value of a signed long</td>
</tr>
<tr>
<td>%lf</td>
<td>double</td>
<td>Same as %f for format functions</td>
</tr>
<tr>
<td>%lu</td>
<td>uint/ulong</td>
<td>Inserts the string value of an unsigned integer</td>
</tr>
<tr>
<td>%lld</td>
<td>long long</td>
<td>Inserts the string value of a 64-bit integer</td>
</tr>
<tr>
<td>%llu</td>
<td>unsigned long long</td>
<td>Inserts the string value of an unsigned 64-bit integer</td>
</tr>
<tr>
<td>%o</td>
<td>short/int/long</td>
<td>Inserts the string value of an integer in octal</td>
</tr>
<tr>
<td>%p</td>
<td>void *</td>
<td>Inserts the string value of a pointer</td>
</tr>
<tr>
<td>%s</td>
<td>char *</td>
<td>Inserts a string specified by the argument</td>
</tr>
<tr>
<td>%u</td>
<td>uint/ulong</td>
<td>Inserts the string value of an unsigned integer</td>
</tr>
<tr>
<td>%x</td>
<td>short/int/long</td>
<td>Inserts the string value of an unsigned integer in hexadecimal</td>
</tr>
<tr>
<td>%n</td>
<td>int *</td>
<td>Writes the number of characters written so far to the int *</td>
</tr>
</tbody>
</table>

Unfortunately, these functions have no verification, and if an argument is of an incorrect type, or if an argument simply does not exist, data will still be pulled from the stack and used in accordance with the rules set by the specifier. Aside from the \%n specifier, this only allows an attacker to leak stack addresses or data. However, the \%n specifier is very real, and it allows an attacker to write arbitrary data to the stack [1, 3, 4, 5].

1.3 Thesis Overview

The remaining sections of this thesis are organized as follows: Chapter 2 reviews literature that poses alternative solutions. Chapter 3 describes my proposed solution, and the
process of determining that solution, as well as alternative solutions that were not selected. Chapter 4 describes implementation attempts of AWC. Chapter 5 provides benchmark results of running AWC then analyzes the findings, and Chapter 7 summarizes this work.
Chapter 2: Literature Review

Prior work to prevent variadic function attacks or format string attacks have a breadth of ideas. Attempts range from simple counting of arguments to an implementation of type-checking for variadic functions. Each solution makes certain trade-offs, some offering lesser security in exchange for higher performance. This chapter will discuss those attempts, their implementations, performance costs, and shortcomings. The sections are titled based on the name of the solution that the work being discussed presents, and not based on the name of the work.

2.1 FormatGuard

In August 2001, Crispin Cowan et. al. introduced a product called FormatGuard that claimed to provide automatic prevention of format string attacks [1]. The paper opens with a few simple solutions to format-string attacks that are not viable, given the constraints of the C language. The first two solutions, removal of the %n specifier, and only allowing static format strings, are not viable as they would break an “undesirable” amount of software [1]. How much software constitutes an undesirable amount was not specified. The third solution provided was to count the arguments provided to the printf-like function, which is noted to be invalid based on the assumption that C’s varargs mechanism would not permit such activities without breaking the Application Binary Interface (ABI). Despite this claim, FormatGuard’s implementation is done through argument counting. At compile time, it utilizes the C preprocessor to statically count the passed arguments. At runtime, it parses the format string and compares the number of
specifiers to the number of arguments, aborting the program if there are not enough real arguments.

The authors note three drawbacks with this approach. First, it fails if an attacker can achieve their goal without needing to exceed the passed number of arguments, though it is noted this has never been seen in practice [1]. Second, FormatGuard fails in the case of an indirect call, as the detection it uses does not operate on indirect calls. Finally, it can fail in the case of direct calls to vsprintf (or similar functions) that take a va_list directly, as it cannot determine the passed number of arguments in this case, similar to the second case. These drawbacks yield two failures amongst the tested programs; format attacks against wu-ftp and gftp still succeed even when protected by FormatGuard.

On the note of performance, FormatGuard achieves a metric it refers to as “below noticability”. In microbenchmarks, it was concluded that a consistent 37% overhead is applied to most printf calls [1]. In macrobenchmarks, the real overhead is significantly lower. Time was spent searching for a program that made an adequate number of printf calls to achieve a notable slowdown, and eventually the authors settled on man2html, which rigorously uses printf to create html files. When translating 79 pages worth of input (596 KB), the overhead of using FormatGuard was 1.3% compared to default. This is a worst-case scenario, and these results mean that implementing FormatGuard pays a very low performance cost to mitigate a reasonable portion of format string attacks.

2.2 HexVASAN

HexVASAN, introduced in August 2017, is a compiler-based sanitizer for variadic functions [2]. It provides a feature that is effectively type-checking for variadic functions, with a static and dynamic component. The core feature involves recording the real argument types
passed to a function call at compile time, and then ensuring calls to va_arg in variadic functions match these types exactly. One issue not addressed by the authors is that this strict type checking invalidates two rules set by the ISO C standard. First, an object of integer type may be retrieved by specifying either the corresponding integer type or the unsigned integer type, if the value of said integer is representable in both types. Second, an object of type void* may be retrieved by specifying char*, and vice versa [10]. If we disregard these issues, HexVASAN provides an effective solution to mitigating variadic function attacks on both direct and indirect function calls with a high degree of correctness.

To achieve the goal of type-checking, HexVASAN relies on a compiler-level component and a runtime mechanism. The compiler-level component is responsible for recording type data at variadic function call sites, and performing instrumentation so that the runtime mechanism can execute. At every call to a variadic function, instrumentation is done to add code that pushes the statically recorded argument types into a stack (before the call) and subsequently cleans them up (after the call returns). In the body of each variadic function, additional instrumentation is done to augment calls to the variadic macros (va_start, va_arg, va_copy, and va_end) which allow HexVASAN to track the creation, manipulation, and destruction of va_list objects for sanity-checking.

The runtime mechanism is responsible for doing the actual type-checking work. When call sites are executed, the statically determined argument types (stored in a struct called vcsd_t) are pushed to a stack. When the function is called, and va_start is invoked, a map entry is created that tracks the arguments consumed by that va_list. Each time va_arg is invoked, the map entry is updated to record that a new argument was requested, and the requested argument type is
validated against the call site’s argument type. If the expected type does not match exactly, an error message is logged and the program is aborted [2].

Overall, HexVASAN is a good solution to this problem. It does not raise false alarms, catches every error case the authors could think of, and invokes small overall performance costs. The overheads on whole programs were typically in the 1% or lower range, except for the libquantum benchmark in the SPECint CPU2006 suite. That particular benchmark performs an extremely high amount of variadic calls (880 million over one run) compared to most programs. The cause for this discrepancy is HexVASAN’s microbenchmark overhead. On a single variadic function call, the execution time is increased by a factor of four to six times, which is substantial on the micro-scale.

2.3 LibSafe 2.0

LibSafe, developed by Avaya Labs, is a software package that intercepts unsafe calls and replaces them with calls to safe alternatives [3]. It accomplishes this by being a library that loads before the standard library, which causes calls to unsafe standard library functions to be redirected to libsafe. Version 2.0 is notable here as it adds new protections against format string attacks, a feature not present in version 1.3. These protections are the result of intercepting calls to print-family functions (printf, sprintf, etc) and injecting safety checks. If a safety check fails, there are various violation handling steps that can be taken.

The safety checks are specific to each particular function, but we can analyze the checks for _IO_vfprintf() as an example. In this case, LibSafe performs two checks: the return address / frame pointer check, and the frame span check. The return address / frame pointer check validates that for every %n specifier, the target address is not a return address or frame pointer. The frame span check validates that for all specifiers, any stack memory accessed must be within
the current stack frame. This code relies on a gcc builtin (__builtin_frame_pointer(0)), and can fail if compiled by other compilers or on gcc with the -fomit-frame-pointer option [3].

In the event that either of the two checks fails, libsafe triggers an error handling sequence. The default operation is to terminate the process, but there are alternative options. In the case of a return address/frame pointer check error, libsafe may perform any of the following operations as long as the %n write was blocked:

- Add an entry to /var/log/secure using syslog().
- Print a warning to the stderr stream.
- Dump the stack’s contents (as hex) to a file.
- Send an email to a predefined list of recipients.
- Call the abort() function, producing a core dump.

Reasons for selecting these particular predefined options are not discussed.

Finally, the paper introducing LibSafe 2.0 provides no information on performance, so the impact it will have on real programs cannot be determined. As far as compatibility is concerned, LibSafe must be run on programs that are linked with glibc, and not with other standard library replacements such as libc5. Programs that utilize libc5 must be recompiled with glibc in order to be used with LibSafe.

2.4 Lisbon

Lisbon is a tool that rewrites Win32 binaries to harden them against format string attacks [4]. It claims to be the first tool which can add such mitigations to legacy Win32 binaries, as it does not require source, and can patch existing executable files. The patch logic utilizes a binary analysis / instrumentation library called BIRD, which can effectively find all variadic call sites unlike other disassemblers [4]. While Lisbon was designed to tackle format string attacks, it can
be applied to all variadic functions due to the nature of the implementation, since it targets variadic call sites / functions rather than strictly printf-like functions.

With regards to the implementation, Lisbon has separate techniques for the two families of variadic functions. Functions that have the ellipses “…” in their prototype have list-bounds checking code inserted into them. Functions that do not have the ellipses and instead take a preconstructed va_list object require that a call graph be constructed so the point of origin for the va_list can be determined, which is then instrumented. The instrumentation takes the same form in both cases, where Lisbon creates a canary word immediately after the end of the argument list, and then checks if this word is accessed. The word is introduced by wrapping variadic function calls with a stub method that can insert this word, and then calling the variadic function from the wrapper.

After the instrumentation pass, Lisbon’s dynamic execution utilizes novel techniques to pass the information from call site to callee. Specifically, Lisbon utilizes a software interrupt to set canary values in the debug registers, which are not used for other purposes during standard execution. Lisbon also introduces a management component to keep the debug register values local to the program, rather than global as they are by default. After the values are set in the debug registers, they can be checked against to ensure they are not read from the stack.

Lisbon’s performance costs stay in line with other solutions to format string attacks, which is to say that the overhead is relatively low. In real-world programs, it incurs throughput and latency costs ranging from 0.3% to 2%. In microbenchmarks, Lisbon’s latency penalty is much higher when compared to FormatGuard (217.7% for Lisbon vs. 7.5% for FormatGuard), but it does not scale with the number of arguments, which means this gap will narrow for cases
with additional arguments. This performance is better than HexVASAN, but it is significantly more expensive than FormatGuard on low argument counts.

2.5 White-Listing

White-Listing is a technique that prevents writing to addresses unless it has been explicitly permitted via a white-list of legal addresses [5]. This can prevent format string attacks that utilize the %n specifier, but it cannot prevent a format string attack from leaking data via other specifiers. For this reason, this approach is less powerful than most, but it can effectively stop all attacks that utilize the %n specifier with a low degree of complexity. The basic principles can be summarized in a four-step process:

1. A white-list must be made available at runtime to hold valid addresses.
2. Callers to print functions must register addresses to the white-list.
3. Print functions must check against the white-list when executing a %n specifier.
4. After execution, callers should unregister any registered address.

This technique can also be expanded beyond print functions, and could be applied to any function, creating a validating system that ensures a function does not write to an illegal memory region.

A basic API for registering and unregistering addresses is provided so that callers to printf-like functions can inform the white-list of legal %n usage. If no registrations occur, the mechanism can operate in an even simpler mode where it disables the %n specifier entirely. An automatic registration scheme has also been developed, which can transform existing C programs and insert the corresponding register and unregister calls as necessary. It does that by scanning function calls and finding printf-like calls that have int* arguments. Those arguments are then marked for registration, as they are the standard target for the %n specifier. This may
have false-positives, as int* methods that would not actually be used for the %n specifier could be registered. In practice, the technique applied here validates that the function will eventually propagate down to a printf-like function before automatic registration, which reduces the likelihood of false-positives.

The efficacy of this method is fairly high, as it was able to block more vulnerabilities than FormatGuard on a test set of four programs. This is because White-Listing is capable of hardening vprintf-like functions (which take the va_list as a parameter) whereas FormatGuard can only harden printf-like functions (with the ellipses). In microbenchmarks, White-Listing had a higher cost than FormatGuard in all cases. Both techniques scale with the number of format specifiers in the string, and White-Listing incurs a 30% higher penalty than FormatGuard for most specifiers, except %n, where it incurs a 50% higher penalty. Compared to reference, White-Listing incurs a 60% penalty on printf with two %n specifiers, and a 75% penalty on vsprintf with the same specifiers. The additional cost could be argued to be justified, since as mentioned in the other sections, even significant micro-benchmark penalties translate to low overall performance overhead.
Table 2  A summary of prior works. Direct performance comparisons of microbenchmarks are inadvisable since the results vary based on many conditions. Macrobenchmark performance can be directly compared, but care should be taken to ensure comparisons are being done for the same program.

<table>
<thead>
<tr>
<th>Product</th>
<th>Attack Type</th>
<th>Injects Via</th>
<th>Failure Conditions</th>
<th>C Spec</th>
<th>Micro Perf</th>
<th>Macro Perf</th>
</tr>
</thead>
<tbody>
<tr>
<td>FormatGuard</td>
<td>Format String</td>
<td>Recompilation</td>
<td>Use of vprintf-like functions</td>
<td>Compliant</td>
<td>37%</td>
<td>&lt;= 1.3%</td>
</tr>
<tr>
<td>HexVASAN</td>
<td>Variadic</td>
<td>Recompilation</td>
<td>None known</td>
<td>Incompliant</td>
<td>400-600%</td>
<td>0.1-1.2%</td>
</tr>
<tr>
<td>LibSafe 2.0</td>
<td>Format String</td>
<td>Library Replacement</td>
<td>Programs do not use glibc</td>
<td>Compliant</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
<tr>
<td>Lisbon</td>
<td>Format String</td>
<td>Instrumentation</td>
<td>Debug registers are unavailable</td>
<td>Compliant</td>
<td>217.70%</td>
<td>0.3-2%</td>
</tr>
<tr>
<td>White-Listing</td>
<td>Format String</td>
<td>Recompilation</td>
<td>Attacks do not use %n</td>
<td>Compliant</td>
<td>10-75%</td>
<td>0.3-1.6%</td>
</tr>
</tbody>
</table>
Chapter 3: Proposed Solution

While many attempts have already been made attempting to solve variadic function attacks (or format string attacks), format string attacks continue to be reported in the CVE, with ten new issues this year [12]. These attacks lead to privilege escalation, remote code execution, and unauthorized information disclosure, which equates them in severity to buffer overflows [1, 2, 3, 4, 5, 12]. Since these attacks continue to occur, more work on prevention is necessary. In this chapter I will detail my solution, Argument Width Counting, a compiler modification that can protect against variadic function attacks that requires no changes to a program’s source code and incurs a very low overhead.

3.1 Argument Width Counting

The basis of Argument Width Counting (AWC) is counting the width of all passed arguments, and limiting memory accesses to this range, relative to the first argument. The width of an argument is defined as the number of bytes the argument takes up in memory. By counting the width of arguments that were placed on the stack at the call site, and then tracking the number of bytes consumed by the callee, we can effectively track if there has been an illegal access. This works because function call sites give away the information about how much data is passed as arguments, and invocations to va_arg give away the information regarding how much data is accessed.

Specifically, at the call site, every argument is pushed to the stack, in reverse order. Before the final argument is pushed to the stack, we can record the current stack pointer, and after the second-to-first argument (the last variadic argument) is pushed, we can subtract that
recorded value from the current stack pointer. The result of that operation gives us the sum of the widths of all variadic arguments. This width can be tracked and decremented as va_arg is called, and an error can be emitted if this width is bypassed. It is often unnecessary to perform this subtraction, as the widths of the arguments are likely to be known statically by the compiler. In this case, we can just record the statically known sum in a dynamic mechanism of our choosing.

There are multiple mechanisms by which AWC could be implemented. One such mechanism is similar to HexVASAN’s, where we could keep the recorded width sums in a thread-local map keyed by va_list pointers, placing the sum into the map via va_start, and decrementing the sum via va_arg. Depending on the level of control the compiler has, it is possible to implement a new field on the va_list struct. This is possible as this struct is not well-defined by the C standard, and as such is implementation dependent [7, 13]. Fortunately, the macros that manage va_list are also implementation dependent, and we can add checks that abort the program if it is invoked invalidly without breaking from the C specification, as it is undefined behavior [13].

AWC is capable of mitigating all attacks in which the attacker must escape the stack space of the arguments to gain access to the program, which composes most attacks. It is less comprehensive than HexVASAN, as it does not enforce that the arguments are of a specific type, but this should not matter. No attacks have been seen that mistype the arguments to attack a program [1]. Additionally, this lack of strict type enforcement allows for full compliance with the C standard, a feat HexVASAN does not accomplish. Unless such an attack is developed, AWC is effective against the same set of attacks as HexVASAN, which is the intersection of all attempts before.
3.2 Alternative Solutions

The development of AWC led me to consider alternative solutions that would have also been effective at mitigating variadic attacks. The first of these was a system where only the number of arguments was counted, similar to FormatGuard. However, if we count the number of arguments, rather than the width, we run into one particular error case that is undesirable. Seen below is an image of the stack that shows how, given the presence of many small arguments, an attacker could still reach the return address without exceeding the argument count.

![Figure 2](image-url) Three views of the stack. On the left is a view before any arguments are consumed. In the middle is a view where two char arguments have been consumed by asking for a single long long, and on the right is a view where four char arguments have been consumed by asking for two long longs, leaving the return address vulnerable in an argument-counting scheme. Green shows valid space, red shows illegal space, blue shows traversed space.

A succession of small arguments can allow an attacker to request large arguments via specifiers, reaching into further memory ranges without using the allotted number of arguments. As mentioned, this mistyping attack has (as far as I am aware) never been seen in the wild, but it is necessary to consider it.

Another potential solution was the implementation of AWC via instrumentation / binary rewriting instead of as a compiler modification. This approach has some merit, and could likely achieve similar results, albeit with a higher overhead. Utilizing instrumentation rather than a compiler modification requires that we forgo direct changes to the va_list struct or the macros, and instead utilize adjustments at the call sites. It would require similar data structures to
HexVASAN (a threadlocal map and stack) to store width sums and track used memory. Overall, an instrumentation variant does not have significant contributions when compared to HexVASAN due to how similar it would be. It is likely the overhead would be smaller, but since HexVASAN’s whole-program overhead is already very small, any benefits are going to be even smaller.
Chapter 4: Implementations

Many implementations of AWC were created or attempted during the process of searching for a suitable medium to develop a working product. This chapter details those attempts, their results, and challenges encountered along the way.

4.1 In x86 Assembly

Implementing AWC in x86 Assembly requires utilizing the alternative method as described in Section 3.2. This is because the x86 implementation is not done on the compiler level, and is instead done using MSVC’s __asm directive, which allows for insertion of intel-syntax x86 assembly into C++ source code. This makes this implementation most similar to HexVASAN, as it uses the threadlocal stack/map setup instead of modifying va_list (since it cannot). Nonetheless, this implementation was an important precursor to deploying AWC on a real compiler, and is worth writing about.

During a variadic function call, the caller’s EBP (a register holding the stack base pointer) is placed on the stack, then the return address, and then the arguments, starting with the rightmost (last) argument. The assembly code that generates this stack state is fairly simple, and provides an exceptionally useful access point for counting the arguments provided to a function at the call site. Figure 3 shows the underlying assembly code, and code that has been instrumented to count the width of the arguments. The instrumented version on the right shows how to compute the total width of the variadic arguments, storing them in register EBX.
By replacing the original code with the instrumented version, we obtain the sum of the widths of the variadic arguments that have been passed to the function. With this information, we can augment the macros va_start, va_arg, va_end, and va_copy to check if an amount of stack space exceeding this width is accessed, and throw an exception, causing the program to halt.

For width-counting to work, va_start must retrieve the computed width from the register it is stored in, and store it somewhere it will not be erroneously overwritten, and is accessible by va_arg. In the simplest case, where the variadic function makes no variadic calls and has a free register, this data can be left in the register it was present in, but this case is not guaranteed. For all other cases, we must make the data available based on a pointer to the va_list object. As mentioned, we would ideally modify va_list to hold this data, but it is outside the scope of this implementation. Given that restriction, we can store this data in a map, keyed on the pointer to the va_list. Similarly, va_copy is responsible for recording the traversed space of the list it copies. From this point, execution of the width-counting measure is then reliant on a change in va_arg to check the currently traversed stack space, and throw an exception if a function attempts
to exceed the “safe” region (as denoted in Figure 1). Finally, va_end is responsible for removing mappings to lists that no longer exist in memory. An example is shown below.

```c
int sumN_asm(int n, ...) {
    va_list args;
    uint32_t size = 0;

    __asm {
        mov size, ebx
    }

    vargsizes.push(size);
    va_start(args, n);
    vargleft[&args] = vargsizes.top();

    int sum = 0;

    for (int i = 0; i < n; i++) {
        if (((vargleft[&args] - 4) < 0) throw new runtime_error("variadic overflow");
        sum += va_arg(args, int);
    }

    vargleft.erase(&args);
    va_end(args);
    vargsizes.pop();

    return sum;
}
```

Figure 4 An instrumented variadic function body. The variables vargsizes and vargleft are a thread local stack and map of widths, respectively. This function has the necessary changes made such that AWC is enabled, provided that the argument width is passed by the caller.

While true that most of the added code is C++ source code and not x86 assembly, this is still considered the x86 implementation due to the callee/caller passthrough requiring its use. Once it became clear that pursuing instrumentation variant would fall victim to the same drawbacks as its predecessors, attempts were made to implement AWC on a compiler, such that its protection would be automatically applied to any compiled programs.
4.2 Under GCC

Before attempting to implement AWC on a compiler, it was first necessary to develop a set of requirements that could determine if the compiler had the necessary features to accommodate AWC. To that end, four criteria were determined:

1. An entry point for when function calls are turned into code
2. The ability to determine if a function call is variadic or not.
3. The number of bytes that function parameters take up on the stack.
4. The ability to modify the variadic macros, and the struct va_list.

If a target compiler meets these four criteria, it is possible to implement AWC. If any one criterion is not met, then it will be difficult or impossible to successfully implement AWC.

For GCC, the first criterion is satisfied through the function "expand_call" in "calls.cc", which is responsible for doing codegen for function call expressions [14]. This function is called whenever a C/C++ function call is expanded into target machine code, and can be co-opted as an injection point for adding additional code to function calls, as is required to implement the first step of AWC. The second criterion is satisfied by the helper function "stdarg_p" defined in "tree.h", which returns true if a given function call corresponds to a variadic function [15]. Without this functionality, it would be difficult to determine when to implement the necessary AWC pre-checks, and implementing them on all functions would be wasteful at best. The third requirement is satisfied by the helper function "initialize_argument_information" in "calls.cc", which is called for every function during "expand_call" [14]. This function is responsible for gathering data about function parameters, including the size of those parameters on the stack.

With the first three components available, the bulk of the work relies on modifications to the macros and types supporting variadic functions. These macros, as well as the va_list type, are
a special compiler object called a builtin. A builtin object is not written in C/C++ code, and is instead stored directly as pre-generated intermediate code (machine-independent target code) [16]. Unfortunately, due to this, modification of these macros (and even the va_list struct) is nontrivial. An engineer wishing to edit them would need to know where they are defined inside GCC and understand the machine-independent language called Register Transfer Language Expressions (RTX). While RTX is documented, and could be learned with time, GCC considers the variadic macros internal. This designation means that GCC provides no documentation on them, their operations, or their locations [17]. As such, an engineer wanting to manipulate them would need someone already familiar with GCC to direct them to the correct location, or have the time to comb GCC’s codebase to locate the definitions, which would be a monumental task (GCC has millions of lines of code).

Due to the knowledge barrier, it is unlikely that I could learn significant details about GCC internals, locate the builtin definitions, and implement the changes as necessary to see AWC fully implemented in GCC. Based on my findings, however, it would be possible for someone with the requisite knowledge to implement AWC in GCC. But the challenge there lies in finding someone with the requisite knowledge.

4.3 Under ChibiCC

Despite the difficulties with implementing AWC over GCC, an attempt was made to implement it on a more basic C compiler, which proved fruitful. The compiler chosen was ChibiCC, a simple C compiler which supports most C11 features and is capable of compiling real programs such as Git or libpng [18]. The first task was to determine if ChibiCC satisfied the four criteria necessary for AWC.
With prior knowledge from the GCC attempt, finding injection points became easier, as the class organization of ChibiCC is similar to GCC. The first criterion was satisfied via a location in codegen.c where function call expressions from the parse tree are emitted as assembly instructions. The second was satisfied by a field in the struct “Type” named "is_variadic" which, when on a struct “Node” (which holds part of the abstract syntax tree), dictates if a function call node is a variadic call or not. Third, we found the function push_args, which is responsible for placing function args in the appropriate place (registers or the stack) as necessary. This function was retrofitted with an additional parameter "unsigned int* argsize" that allowed us to have this function report back the total size of all function arguments, successfully satisfying requirement 3. Finally, we approached the difficult task of satisfying requirement 4 that made implementing AWC on GCC impossible. Luckily, ChibiCC does not keep the variadic macros stored as pregenerated code, and instead compiles this code as needed, meaning the variadic macros were implemented in plain C code in the file "include/stdarg.h". This meant that modification of these macros was possible, satisfying requirement 4.

With all requirements satisfied, we were able to proceed with an implementation. The implementation requires minimal changes to the compiler, needing a single additional field on the va_list struct, two instructions inserted before variadic function calls, a single global variable, and check logic added to the va_arg call. Addition of the field to va_list was not too difficult, but it did require carefully updating code that performed raw memory management of the struct on the stack, as initialization of the struct cannot be done in C code, and is instead done via a builtin. After the field was added, we created a global variable __global_argsize__ in "include/stdarg.h" which allows us to pass data over the function call boundary. A global was used instead of a register since it is guaranteed that the value of the global does not need to be saved/restored.
Setting this global variable from each variadic function call incurs a cost of two x86 instructions, shown below:

```c
+ if(node->func_type->is_variadic) {
+   println(" lea __global__ argsize__(%rip), %rax");
+   println(" movl $%d, (%rax)", argsize);
+ }
```

Figure 5 A diff showing code added to ChibiCC that emits two x86 instructions. These are responsible for setting the `__global__ argsize__` variable to the sum of the sizes of the variadic arguments.

The data is retrieved from `va_start` through the following C code:

```c
-  do { *(ap) = *(__va_elem *)__va_area__; } while (0)
+  do { *(ap) = *(__va_elem *)__va_area__; ap->remaining = __global__ argsize__; } while (0)
```

Figure 6 A diff showing changes made to `va_start`. This sets the computed sum of widths to the newly created `va_list`’s “remaining” field.

The additional code added to `va_start` emits the following x86 assembly when compiled:

```
push %rax
.loc 2 17
.loc 2 17
lea __global__ argsize__(%rip), %rax
movsx d (%rax), %rax
pop %rdi
mov %eax, (%rdi)
```

Figure 7 Assembly instructions corresponding to the new code added to `va_start`. In total, this leaves us with seven new instructions added as a baseline to any variadic function call.
Additionally, we then need to perform certain checks whenever va_arg is invoked to ensure that we accurately count how much memory has been accessed by the callee, and verify that it does not exceed the real arguments provided by the caller. This check takes the form of the following C code:

```c
+   ap->remaining -= sz;
+   if((ap->remaining & 0x80000000) { 
+       printf("Program aborting due to variadic overflow!\n");
+       exit(-1);
+   }
```

Figure 8 A diff showing the error-handling code added to va_arg. The check “ap->remaining & 0x80000000” is equivalent to “ap->remaining < 0” but is more performant due to the speed at which the binary and operation can execute.

This block must be called once by each invocation of va_arg. Unfortunately, since this is implemented in C instead of assembly, and ChibiCC is a non-optimizing compiler, this operation (excluding the printf and exit calls) takes a total of 30 instructions. This is sizable, but in microbenchmarks did not incur a significant performance cost, and should not be an issue for an optimizing compiler. Alternatively, one could write this block in assembly, allowing the check to be done with fewer instructions. Nonetheless, ChibiCC proved a useful medium for creating a prototype implementation, incurring a total instruction cost of 7 + 30 * invocations of va_arg, yielding a cost ceiling of 7 + 30 * (arguments - 1).
Chapter 5: Empirical Analysis

This chapter details the results of benchmarking the ChibiCC implementation of AWC. The initial assumption was that it should be able to run faster than HexVASAN due to the checks being less rigorous. It should also be faster than implementations like FormatGuard since it does not need to utilize wrapper functions, as function calls are more expensive than inline code.

5.1 Test Methodology

Testing AWC required writing a small test program that executed a variadic function in a loop. The first portion of this is designing a variadic function that is simple enough to capture overhead without it being overshadowed by the function’s complexity. To this end, I created a simple variadic function that calculates the sum of integers, with the number of integers being the first parameter, and each integer to be summed passed as a vararg. This function has one invocation of va_start and va_end, and n invocations of va_arg (where the expensive check is performed).

```c
int sumN(int n, ...) {
    va_list args;
    va_start(args, n);
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += va_arg(args, int);
    }
    return sum;
}
```

Figure 9 A variadic summation function, used for benchmarking AWC. Specifically, this was used for the microbenchmark tests detailed in Chapter 5.
To test this function, the C standard function `clock()` was used. A start time was recorded, then one million invocations of the function `sumN()` were performed, and then the end time was recorded. The CPU time used was calculated by dividing the difference between the end and start times and dividing it by the C standard `CLOCKS_PER_SEC` constant. The test program was compiled and run two times, once with three arguments, and once with sixteen arguments, to determine how the performance scaled with the number of arguments provided to a variadic function. The benchmarking program was run 100 times per case, and then the average of those runs was taken, yielding the cost of 1M invocations of `sumN` with a target number of arguments. For both cases, the system was an Ubuntu 20.04.3 LTS (under the Windows Subsystem for Linux) desktop utilizing a Ryzen 5900x processor and 32GB of RAM.

5.2 Microbenchmark Results

Table 3  Microbenchmark results. This data is from the prototype implementation of AWC on ChibiCC. The times presented are the runtime for the full 1M execution loop, and not divided by 1M to estimate a single call.

<table>
<thead>
<tr>
<th># of Args</th>
<th>Time (ms)</th>
<th>Time w/ AWC (ms)</th>
<th>Overhead (ms)</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17.498</td>
<td>22.198</td>
<td>4.7</td>
<td>26.8602126</td>
</tr>
<tr>
<td>3</td>
<td>77.308</td>
<td>86.768</td>
<td>9.46</td>
<td>12.23676722</td>
</tr>
<tr>
<td>16</td>
<td>342.858</td>
<td>368.388</td>
<td>25.53</td>
<td>7.446231384</td>
</tr>
<tr>
<td>100</td>
<td>2070.558</td>
<td>2173.978</td>
<td>103.42</td>
<td>4.994788844</td>
</tr>
</tbody>
</table>

These benchmark results show the performance impacts of AWC on a simple variadic function. We can see that with an increasing number of arguments, the real-time overhead increases, while the percentage overhead decreases. The percentage decrease is likely due to the fact that ChibiCC is a non-optimizing compiler, and thus the static cost of the function is likely to be high when compared to a program compiled with an optimizing compiler. These results do account for loop overhead, which was measured to be 2.282ms. Additionally, it is important to note that these results are per 1M executions, not per single execution. To convert the times to
per single execution, we would simply divide them by 1M. The percentages would remain the same, but the times would be in nanoseconds.

5.3 Discussion

The AWC microbenchmark results are promising and show that AWC can be implemented with minimal overhead. While the cost may seem high on a zero-argument variadic call (nearly a 25% increase), this cost is known to drop significantly when applied in terms of whole-program overhead [1, 2, 3, 4, 5]. It is unusual that the percent overhead decreases with the number of arguments, but the reasons for this likely lie within the internals of ChibiCC. The most obvious culprit is the lack of compiler optimizations. When the test program is compiled with GCC (instead of ChibiCC), it executes 10x to 20x faster. With this in mind, we can look to the absolute time increase instead of the percentage increase to see the real cost of AWC as more arguments are introduced.

At zero arguments, we see a marginal increase of about 5ms per 1M calls (~5ns per call), nearly doubling when we approach 3 arguments. At 16 arguments, the absolute time increase is roughly 2.5x that of 3 arguments, and at 100 arguments, the increase is roughly 4x that of 16 arguments. This yields a nonlinear time increase with respect to the number of variadic arguments, the rate of change decreasing as argument count increases.

5.4 Comparsion

Argument Width Counting shows promising performance results when compared to prior solutions. FormatGuard imposes a consistent 37% overhead on calls to printf, which is higher than the cost incurred by AWC, though it does not scale with the number of arguments [1]. Interestingly, FormatGuard was re-benchmarked and shown to have argument-scaling performance costs on printf, hitting 38% at just two %n specifiers (vs. 7.5% with no arguments)
HexVASAN incurs a 4-6x overhead on variadic function calls, which is an order of magnitude higher than AWC’s performance impact [2]. Libsafe does not provide any performance metrics, so it is impossible to compare the cost of that implementation with AWC [3]. Lisbon shows performance costs in the 1.5x to 3x range, incurring the highest percentage overhead when zero arguments are present. White-Listing incurred a performance cost ranging from 10% to 75%, based on the specific function being called, the number of arguments, and the type of format specifiers present.

While a table would nicely summarize these comparisons, it would presume to make a substantial direct comparison between the mediums, which is impossible (except between AWC and HexVASAN). FormatGuard, Lisbon, and White-Listing all have different performance impacts based on the number of arguments, types of specifiers, and even the underlying function being called. AWC and HexVASAN do not experience such variation, since they are designed to natively work on all variadic functions, instead of requiring different handling for different types of functions.
Chapter 6: Conclusion

Variadic function attacks (and by extension, format string attacks) are just as dangerous as buffer overflow attacks, and continue to plague software applications to this day. Many solutions have been proposed, from simple argument counting via wrapper functions to a full type checking system for variadic functions, yet none have seen widespread adoption. This work introduced Argument Width Counting (AWC), a novel solution that ensures variadic functions cannot misuse invalid memory as function arguments. This solution is effective, low cost, and can be applied to compilers instead of individual programs. Once applied to a compiler, all emitted programs will benefit from variadic attack mitigations. As discussed in Chapter 3 and Chapter 5, AWC is (empirically) at least as effective as the best-known attempt in stopping real-world attacks (HexVASAN) and manages to have better performance than even the most lightweight solutions (FormatGuard / White-Listing).

This significant performance uplift is achieved by leveraging the fact that the C specification does not define how a program should access varadic arguments, allowing for changes to be made without becoming incompliant with the spec. Prior attempts have not made direct changes to this mechanism, instead applying modifications to usage sites, if at all [1, 2, 3, 4, 5]. The change made by AWC does change the size of the compiler-internal struct va_list and changes the operations done by the compiler-internal macros va_start, va_arg, and va_end, which could potentially be a problem if a program was relying on implementation details. However, these are builtins, and as such are not public API, so programs that are relying on
implementation details of these builtins are doing so at their own risk, as they could change at any time [17].

This work also produced a prototype implementation on a simple C compiler known as ChibiCC. This prototype allowed us to see how AWC could function in real terms, as many of the changes are conceptually applicable to any compiler. Benchmarks done with the prototype implementation were promising, showing overheads in the 5-27% range on microbenchmarks, which is lower than all previous works. The prototype was unable to be run on real-world programs since recompilation of all used programs (including the standard library) would be necessary for full protection. It is possible to compile a program for partial protection, but this protection only applies if the call sites and target functions are within the program. Standard library functions, such as printf, require recompilation of the standard library and the calling program to receive full benefits.

Future work could be done implementing AWC on GCC (or a similar optimizing compiler), and then attempting to have this feature merged into the upstream branch, enabling it for all future releases. Some tuning would be needed, as the prototype does not support running mixed AWC/non-AWC programs, but this level of compatibility could be achieved with minimal difficulty (although security would be weakened if a program in the environment was not AWC-enabled). Alternatively, future work could focus on determining if the methods utilized here are the best ones for achieving the goal of AWC. There are alternatives, including how best to pass the argument width through the call site, how best to track the consumption of arguments, and other changes. There may also be opportunities for optimization of the check done from within va_arg, which would be best written in assembly to ensure it is as fast as possible.
Finally, we close this chapter and this work by replicating Table 2 and including AWC in a final row. This allows for some comparison with the prior works, intended as a quick summary of the attempts at solving variadic/format string attacks addressed by this work.
Table 4 A replication of Table 2 including AWC. All notes from Table 2 should be adhered to.

<table>
<thead>
<tr>
<th>Product</th>
<th>Attack Type</th>
<th>Injects Via</th>
<th>Failure Conditions</th>
<th>C Spec</th>
<th>Micro Perf</th>
<th>Macro Perf</th>
</tr>
</thead>
<tbody>
<tr>
<td>FormatGuard</td>
<td>Format String</td>
<td>Recompilation</td>
<td>Use of vprintf-like functions</td>
<td>Compliant</td>
<td>37%</td>
<td>&lt;= 1.3%</td>
</tr>
<tr>
<td>HexVASAN</td>
<td>Variadic</td>
<td>Recompilation</td>
<td>None known</td>
<td>Incompliant</td>
<td>400-600%</td>
<td>0.1-1.2%</td>
</tr>
<tr>
<td>LibSafe 2.0</td>
<td>Format String</td>
<td>Library Replacement</td>
<td>Programs do not use glibc</td>
<td>Compliant</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
<tr>
<td>Lisbon</td>
<td>Format String</td>
<td>Instrumentation</td>
<td>Debug registers are unavailable</td>
<td>Compliant</td>
<td>217.70%</td>
<td>0.3-2%</td>
</tr>
<tr>
<td>White-Listing</td>
<td>Format String</td>
<td>Recompilation</td>
<td>Attacks do not use %n</td>
<td>Compliant</td>
<td>10-75%</td>
<td>0.3-1.6%</td>
</tr>
<tr>
<td>AWC</td>
<td>Variadic</td>
<td>Recompilation</td>
<td>Mistyping Attacks</td>
<td>Compliant</td>
<td>5-27%</td>
<td>Unknown</td>
</tr>
</tbody>
</table>
References


