Automated Format String Attack Prevention for Win32/X86 Binaries

Wei Li Tzi-cker Chiueh

Department of Computer Science
Stony Brook University
Stony Brook, NY 11794-4400

Abstract

A format string attack exploits the fact that variadic functions determine the exact number of input arguments based on the format string argument, and compromises the victim application's address space by accessing data areas beyond the original input argument list the caller prepares. This paper describes the design, implementation and evaluation of a Win32 binary transformation tool called Lisbon, which transparently inserts into Win32 binaries additional checks that protect them from format string vulnerabilities. Lisbon casts the format string attack prevention problem as an input argument list bound checking problem. To reduce the run-time checking overhead, Lisbon exploits the debug register hardware, which is available in most mainstream CPUs including Intel's X86 architecture, to detect if a callee accesses data outside the input argument list. Moreover, Lisbon is able to detect format string attacks without interpreting their format strings and is thus potentially applicable to similar attacks against other functions that access input arguments in the same way as `printf()`*. The runtime throughput penalty of the first Lisbon prototype is under 2% for a set of test network applications that are known to be vulnerable to format string attacks.*

1. Introduction

A format string attack exploits the fact that functions such as printf() can accept a variable number of input arguments and the exact number of input arguments is implicitly specified in the first argument, the *format string* argument. These functions are often referred to as *variadic* functions. Given a call to a variadic function, the callee has no way of determining if the input format string specifies more input arguments than is set up by the caller; only the caller can check if the number of arguments it prepares is consistent with that specified in the format string. If an attacker can control the format string argument of a variadic function in a victim application, it is possible for the attacker to read or write the application's address space. If the caller of a variadic function can explicitly specify the number of input arguments it prepares as another input argument, it would have made format string attack much more difficult, if not impossible. Unfortunately the interface to existing variadic functions such as printf() does not permit this extension.

Although there have been several attempts to solve the format string attack problem, all of them share two deficiencies. First, they did not stop all known variants of format string attacks, for example, vulnerable vprintf functions that are encapsulated by multiple layers of wrapper functions. Second, they could not be directly applied to commercially distributed Win32 binaries because some information they need is inherently lost in binary programs, for example, the number of arguments of a printf call as specified in the source code.

This paper presents the design, implementation and evaluation of a format string attack prevention system called *Lisbon*, which to the best of our knowledge is the first tool that can protect legacy Win32 binaries. There are two key challenges in adding format string attack prevention logic to Win32 binaries. First, state-of-the-art disassemblers such as IDAPro cannot achieve 100% accuracy and coverage for commercially distributed Win32 binaries. *Lisbon* solves this problem by leveraging a novel binary analysis and instrumentation infrastructure called BIRD [17].

Second, unlike source code, an executable binary generally does not contain enough information to determine the number of input arguments in each function call. Therefore, it is not always possible to determine each variadic function call's exact list of input arguments in a binary program, let alone compare its format string with its actual input arguments. *Lisbon* solves this problem by ensuring that when a variadic function accesses its input argument list, it never steps beyond the list's bound. More specifically, because it is more important to protect the local variables of a caller of a variadic function than to stop the variadic function from stepping outside its input argument list, *Lisbon* actually checks a variadic function's accesses against the lower bound of its caller's local variable region, rather

than the upper bound of its input argument list. By casting format string attack prevention into a bounds checking problem, *Lisbon* does not need to determine the exact input arguments of every variadic function call, and thus can successfully apply this technique directly to binary programs.

Another advantage of casting the format string attack prevention problem as an input argument list bound checking problem is that the resulting solution is potentially applicable to *all* variadic functions, not just printf() and its variants. More concretely, unlike other solutions, *Lisbon* does not require interpretation of the format string argument. However, this generality comes with an implementation challenge: given an input argument list associated with a function call, how to detect any overflow attempts by the callee or its descendants in the call chain. *Lisbon* solves this problem through a novel application of the debug register facility supported in most modern processors including X86, SPARC, MIPS, ARM, etc. By setting the bound to be checked into a debug register, *Lisbon* is able to perform list bound checking without requiring any software checks, thus reducing the associated performance overhead to the minimum.

Section 2 reviews previous work on format string attack prevention. Section 3 describes the format string attack problem in more detail, and the solution used in *Lisbon*. The implementation issues of the first *Lisbon* prototype on the Windows platform are discussed in Section 4. In Section 5 we present the evaluation results of the effectiveness of the *Lisbon* prototype and its performance overhead. The limitations of *Lisbon* are discussed in Section 6. Section 7 concludes this paper with a summary of research contributions and an outline of future work.

2. Related Work and Comparison

Successfully launching a format string attack requires multiple conditions to be present, and previous efforts on preventing/detecting format string attacks focused on elimination of one or multiple of these conditions.

One proposal to thwart format string attacks is to force all format strings to be constant. This is not an unreasonable approach because many format string vulnerabilities arise because of inadvertent missing of simple format string such as "%s". Pscan [9] is a simple lexical analysis tool that searches for all calls to printf-like functions, and raises a warning whenever their format string is the last argument and is not a constant. The GCC compiler [12] incorporates a similar feature. When the flag "-Wformat=2" is enabled, a compiler warning is issued whenever a non-constant format string is found. Libformat [19] is a customized library wrapping around printf-like functions for Linux applications. It intercepts every call to a variadic function and flags an error whenever its format string is modifiable and contains the "%n" specifier. Unfortunately, disallowing non-constant format string may break some applications; for example, those applications that use the GNU internationalization library tend to generate format strings dynamically [6] and thus trigger false alarms under these tools.

Instead of forcing all format strings to be constant, some systems enforce this constraint only for "tainted" format strings. Shankar et al. [24] implemented such a tainted format string analysis system. By introducing a new type qualifier called "tainted" into the C language, they are able to track the data usage and propagate the type information associated with the data. When a format string is derived from an external input, it would be marked as "tainted" and triggers a check upon its use. This approach is superior because of reduced false alarms. However, it still generates false positives. Moreover, it imposes additional work on developers in the form of extra annotation and re-compilation of the source code.

FormatGuard [6] modifies the Gnu C Preprocessor [11] to count the number of input arguments of variadic function calls statically, and checks this count against the conversion specifiers in the format string dynamically through a wrapper. If they don't match, it generates a warning. There are several limitations to FormatGuard. First, FormatGuard assumes the number of input arguments in variadic function calls is known statically, and therefore cannot prevent format string attacks through vprintf-like functions or their wrapper functions. Unfortunately, many well-known format string vulnerabilities in software such as Wu-Ftpd [28] and Proftpd [26] are indeed due to vprintf-like functions.

Libsafe [27] is a general approach to thwarting local buffer overflow attacks by ensuring that local variable accesses within a function, including accesses to input arguments, should never exceed its stack frame. As in libformat, libsafe is implemented as a wrapper library that intercepts each variadic function call and checks the format string. Libsafe can protect binary programs that make calls to variadic functions it intercepts and checks. The major limitation of libsafe is its dependence on the usage of frame pointers. As a result, it simply cannot handle programs that are compiled without frame pointer such as the "-fomit-frame-pointer" flag in GCC or the "/Oy (frame-pointer omission)" option in VC++. Unfortunately, many Win32 binaries are indeed distributed without using frame pointer.

Most known format string attacks [20] rely on the conversion specifier "%n" in printf-like functions to modify the victim application's address space. Accordingly, several systems attempt to stop format string attacks by preventing attackers from abusing the "%n" specifier. Libsafe [27] checks if the target of a "%n" specifier points to a return address or a frame pointer on the stack to prevent them from being tampered. But it does not prevent other critical data such as function pointers and sensitive data variables from

Table 1. Comparison between Lisbon and other dynamic checking approaches to preventing format string attacks

Feature	Techniques				
	Libformat	FormatGuard	Libsafe	White-listing	Lisbon
Source code not required	Y		Y		Y
Applicable to Win32 binaries					Y
No false positives		Y	Y	Y	Y
Handle vprintf-like functions	Y		Y	Y	Y
Handle wrapper functions			Y*	Y	Y
Stop read attacks		Y	Y		Y
Not format string-specific	Y		Y**		Y
Protect current stack frame	Y	Y		Y	Y

: only for direct call case* *: when frame bound checking is turned on*

being altered. Ringenburg and Grossman proposed a white-listing method [18] that constructs a list of addresses that are passed into a variadic function as integer-pointer-type arguments, and applies sanity checks only on addresses in this list. The white-list method requires interpretation of conversion specifiers and doesn't stop format string attacks that allow attackers to read memory locations in victim applications.

Compared with previous works, *Lisbon* is the first tool that can effectively protect commercially distributed Win32 binaries without breaking legitimate applications or interpreting the format strings. Table 1 shows a detailed comparison between *Lisbon* and other solutions to the format string attack problem that take the dynamic checking approach.

3. Bound Checking for Input Argument List

3.1. Format String Vulnerabilities and Attacks

Format string vulnerability mainly stems from a programming error that leads the format string of a variadic function such as printf to be directly controllable by an external input. Consider the following two calls to printf: (1) printf ("%s", user_string) and (2) printf (user_string), where user_string is derived from an external input. The results of these two calls are exactly the same if user_string is just a simple character string. However, if user_string contains conversion specifiers, each of which corresponds to a separate input argument and represents a command that controls how printf operates on its corresponding argument, the second call could trick printf into believing there are more than one input argument in this call. As a result, even though printf (user_string) contains only one argument, the fact that user_string contains K conversion specifiers is enough to convince printf to access additional memory locations on the stack that are beyond user_string. If each of these K conversion specifiers in user_string denotes an integer, then the attacker can trick printf into displaying the next K integers on the stack.

To modify the victim application's address space, format string attacks exploit a special conversion specifier "%n", which counts the number of characters written so far and writes the result into the address given by its corresponding argument. By carefully crafting a format string that prints out a pre-computed number of characters before the "%n" specifier, the attacker can write a chosen value to some memory location. Moreover, because the additional arguments required by the conversion specifiers come from the stack frame of the caller to printf (user_string), as does user_string, they are likely to be controllable by the attacker as well. With the "%n" specifier and the ability to manipulate its corresponding argument, the attacker now has the ability to write an arbitrary value to a chosen memory location. If the memory location chosen to be overwritten is a control-sensitive data structure that contains a return address or function address, the attacker could hijack the control of the victim program.

3.2. Identification of Input Argument List

The format string vulnerability is rooted in the fact that a printf-like function can be tricked by the format string argument into believing it has more arguments than the caller supplies and accessing memory locations lying beyond the bound of its input argument list. Therefore we propose a list bound checking approach to eliminate the format string vulnerability. There are two families of printf-like functions, each with a different function template and both susceptible to the format string attack. Table 2 lists members of each family. From now on, we refer to the first group as fprintf-like function, and to the second group as vprintf-like function.

The major difference between fprintf-like functions and vprintf-like functions lies in the way they take the input arguments. The prototypes of fprintf and vprintf are `int fprintf(FILE *stream, const`

Table 2. A list of variadic functions using a format string argument

Function Type	Member Library Functions
Input arguments explicitly passed through copying	*printf, fprintf, sprintf, snprintf, syslog, err, warn*
Input argument list implicitly passed through a pointer	*vprintf, vfprintf, vsprintf, vsnprintf, vsyslog, verr, vwarn*

`char * format, ...)` and `int vprintf(const char * format, va_list ap)`, respectively. Figure 1 shows the stack layouts of an fprintf-like function and a vprintf-like function called by a wrapper function. In fprintf-like functions, the input arguments are explicitly listed in the function call. The `...` notation in fprintf's function prototype means that it could be substituted by an arbitrary number of arguments when fprintf() is called. These arguments are placed on the stack right next to the first or format string argument. When a fprintf-like function is encapsulated by layers of wrapper functions, its arguments are explicitly passed through these wrapper functions. Therefore, *Lisbon* can put list bound checking code in any of the wrapper functions that eventually call the fprintf-like function.

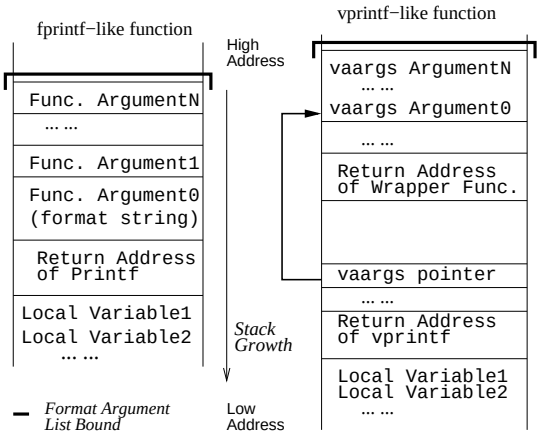

Figure 1. The stack layouts of fprintf-like and vprintf-like function calls

In contrast, vprintf-like functions have only two arguments, the first being the format string and the second being a pointer to a special data structure called `va_list`, which correspond to a list of user-defined data items that are placed consecutively on the stack but may not be adjacent to the two original input arguments. Because vprintf-like functions use a separate pointer to refer to the input argument list, this pointer can be passed through layers of wrapper functions before it is actually de-referenced. As a result, the function in which a `va_list` structure is prepared may not directly call the function in which the arguments in the `va_list` structure are accessed. Accordingly, *Lisbon* needs to identify the function whose stack frame actually holds the list of input arguments, and puts bound checking into the function it calls. Let's use the following example to illustrate *Lisbon*'s list bound checking mechanism. Assume an application calls a function `user_wrapper`, which prepares a `va_list` structure and calls another function `vprintf_wrapper`, which in turn calls vprintf. *Lisbon* needs to identify the function in which the `va_list` data structure is prepared, in this example, the `user_wrapper` function, and puts the list bound check into the function that `user_wrapper` calls, in this case `vprintf_wrapper`.

```
void user_wrapper(
  const char *fmt, ...
)
{
  va_list ap;
  ... ...
  va_start(ap, fmt);
  vprintf_wrapper(fmt, ap);
  va_end (ap);
  ... ...
}
```

```
void vprintf_wrapper (
  const char *fmt,
  va_list ap
)
{
  ... ...
  vprintf (fmt, ap);
  ... ...
}
```

To identify the function in whose stack frame the `va_list` data structure of a vprintf-like function call is stored, *Lisbon* first constructs a function call graph of the given binary [16], and traces backward from the vprintf-like function being called. In this trace-back process, *Lisbon* performs a backward *def-use* data flow analysis starting from the vprintf-like function's pointer argument, to identify where its value is first created. As a heuristic, *Lisbon* searches for a specific instruction sequence corresponding to the macro ``va_start(ap, fmt)".

3.3. Detecting List Bound Violation

One way to detect argument list bound violation for fprintf-like functions is to count the number of arguments at the call site, compare that count with the number of conversion specifiers in the format string, and declare a violation if the latter is larger than the former. This method requires one to interpret the format string in the same way as the fprintf-like functions. Although it is conceptually simple, there are two disadvantages. First, this approach requires the same format string handling engine in the C runtime library. Otherwise, any discrepancies might leave a loophole for attackers to exploit. Second and more importantly, it is not always possible to get an accurate count of the number of arguments for every function call in a binary program. Therefore, this approach cannot easily be applied to binary programs.

Lisbon takes a more general approach to checking argument list bound violation. More concretely, for each fprintf-

like or vprintf-like function call, it identifies its input argument list, reserves a canary word [7] immediately after the list's end, and checks if the called function accesses the canary word at run time. Even though a format string attack could use the "%n" directive to tamper with any arbitrary memory location in the victim application, such an attack needs to put the target memory location's address in the victim variadic function call's input argument list first. However, the proposed input argument list bound checking approach could detect and stop accesses to the memory address arguments associated with the "%n" directives, and thus effectively prevent such tampering.

The proposed scheme enforces the following invariant: the canary word associated with the input argument list of an fprintf-like or vprintf-like function call should never be accessed by the called function and its descendants in the function call graph. Because a called fprintf-like or vprintf-like function may access the caller's stack frame through a pointer passed in as an input argument, it is not safe to mark an existing word in the caller's stack frame as a canary word. Instead, *Lisbon* creates a stub wrapper function that copies the original arguments, reserves a canary word above the argument list, then calls the callee function, and finally cleans up and returns after the callee returns. Figure 2 shows how the stack layout associated with a printf() call changes as a result of the introduction of this stub wrapper function.

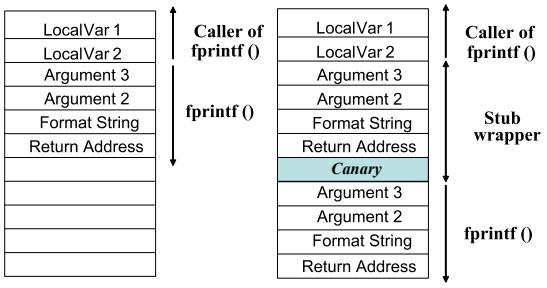

Figure 2. The stack frames before and after Lisbon's stub wrapper function

This stub wrapper function is introduced immediately above the fprintf-like function in the case of an fprintf-like function call, and immediately below the function in which the `va_list` data structure is created in the case of a vprintf-like function call.

A key challenge in protecting executable binaries from format string attacks is to infer the number of arguments in each call to a printf-like function. *Lisbon* solves this problem by shifting the focus to the protection of callers of printf-like functions, as it is more important to protect the local variables of a caller of a printf-like function than to stop the printf-like function from stepping outside its input argument list. So the research question becomes how to identify the local variable region of a caller to a printf-like function so that *Lisbon* can copy everything between the end of the caller's local variable region and the current stack pointer to the stub wrapper function and set up the canary word accordingly. The Visual C++ compiler on the Windows platform follows the convention that when a function is called, it first allocates its local variable region on the stack, and then grows the stack further when the function actually calls other functions. This convention allows *Lisbon* to treat the stack region that is immediately allocated after a function's prolog to be its local variable region. Using this approach, which is applicable to both fprintf-like and vprintf-like functions, *Lisbon* does not need to know the exact number of arguments used in each printf-like function call, and still can effectively protect the caller's stack from being overflowed by a format string attack.

4. Implementation

4.1. Binary Analysis and Instrumentation Infrastructure

Because the design goal of *Lisbon* is to protect Win32/X86 binaries from format string attacks, it is built on top of a Win32/X86 binary analysis and instrumentation infrastructure called BIRD [17], which is specifically designed as a platform for developing systems that enhance software security. BIRD supports DLLs, exception handlers, and multi-thread programming. The overall system architecture of BIRD is shown in Figure 3.

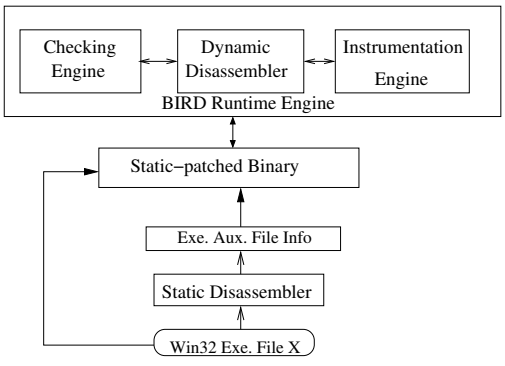

Figure 3. The system architecture of BIRD

BIRD targets at Win32 binaries on the Intel X86 architecture, which state-of-the-art disassemblers such as IDAPro cannot achieve both 100% disassembly coverage and accuracy. However, to safely instrument executable binaries requires 100% disassembly accuracy. To achieve 100% disassembly accuracy, BIRD performs both static and dynamic disassembly. Although BIRD may not

achieve 100% coverage, it does guarantee that every instruction in an executable binary will be properly examined/transformed before it is executed. That is, BIRD may miss those instructions that never get executed in a particular run, but these instructions do not matter exactly because they are not executed.

More concretely, BIRD first statically disassembles a binary file as much as it can, and defers the part that it cannot disassemble statically to a dynamic disassembler at run time. The part that can be statically disassembled is called the known area, while the rest is the unknown area. A program's control starts in the known area and can only be transferred to the unknown area through indirect branch or call instructions. By checking if the target address of every indirect branch and call lies in the unknown area, BIRD can determine whether it should invoke dynamic disassembling on the target instruction. As a program runs, its known area grows and its unknown area shrinks.

BIRD also supports binary instrumentation when the user specifies an instrumentation point specification (e.g., function prolog) and an instrumentation routine. More specifically, BIRD adds a new section to the input program that contains the instrumentation code, and replaces the instruction at each instrumentation point with a jump to the corresponding instrumentation instruction sequence. Because static disassembling cannot achieve 100% code coverage, BIRD performs both static and dynamic instrumentation.

The essential functionalities of BIRD's run-time disassembling and instrumentation are implemented as a dynamic linked library called `dyncheck.dll` which is functionally independent of the applications to be instrumented. The instrumented application's import table is modified to include `dyncheck.dll` so that it is always loaded automatically at start-up time. In order to get the control of the program when the program is executed, BIRD also instruments the program's entry point [3] so that any instrumented binary program will start with BIRD's initialization and monitoring engine first.

Normally, only functions without explicit call sites are left to dynamic disassembling and analysis. For format string attack detection, most of the interesting interception and analysis is actually done statically because callers to printf-like functions can be recognized statically. However, BIRD's dynamic interception capability is still useful because it allows *Lisbon* to detect callers to printf-like functions through indirect calls such as function pointers.

4.2. Identification of Printf-like Functions

To identify call sites to printf-like functions, *Lisbon* first needs to identify the entry point of such functions. This step is required for any systems that aim to protect Win32 binaries from format string attacks, and is largely independent of the internal working of *Lisbon*. There are two cases. If a printf-like function is statically linked into an application, its instructions are contained in the application's binary. If a printf-like function is dynamically linked, the application's binary only contains a link to its instructions, which are stored in a separate file.

When the instruction sequence of a printf-like function is already contained in an application's binary file, it is not easy to recognize its entry point because the binary file typically does not include any symbolic name associated with the printf-like function. To solve this problem, we extract the body of each printf-like function from the standard library. For example, in Visual C++, the C runtime library functions come from libraries such as `LIBC.LIB`, which maintains each library function's name, body and relocation information. Therefore, one can search `LIBC.LIB` with a printf-like function's name to retrieve its function body, derive a function signature from its body, and use this function-specific signature to search an application binary for the entry point of this printf-like function.

If a printf-like function is dynamically linked into an application at load-time, the application's binary contains a link to the function's body in its import table. A caller to a dynamically linked function uses an indirect call to transfer control to the target function, whose target address is stored in the import table entries. Therefore, to identify callers to dynamically linked printf-like functions in an application binary, *Lisbon* first locates the import table entries corresponding to these printf-like functions, and then searches the binary for functions that use indirect calls whose operand matches one of these import table entries.

4.3. Bound Checking Implementation

A key innovation in *Lisbon* is its use of debug register in detecting bound violation of input argument accesses. Debug register hardware is universally supported by most if not all mainstream CPUs such as Intel's 32-bit and 64-bit X86, ARM, SPARC, MIPS, PowerPC, etc. In addition, the interfaces these CPUs expose to the software are largely the same. In this paper, we will focus only on the Intel X86 processor [13]. However, the technique described below is equally applicable to other processors without much modification.

The original purpose of having debug registers is to support instruction and data breakpointing functions required by software debuggers. In the X86 architecture, there are totally eight debug registers (DB0 through DB7) and two model-specific registers (MSRs). Among them, DB4 and DB5 are reserved. DB6 keeps the debugger status while DB7 is for control/configuration. DB0 to DB3 are used to hold memory addresses or I/O locations that the debug-

ger wants to monitor. If being enabled, whenever a memory or instruction address matches the contents of one of these four registers, the processor raises a debug exception. With this support, the debugger does not need to perform expensive intercept-and-check in software. The hardware compares every instruction/data memory address with these breakpoint addresses in parallel with the normal virtual to physical address translation, thus it incurs no additional performance overhead. Please refer to the IA32 architectural manual [13] for the detailed layout of these DR registers and their usage.

Previously, only debuggers such as GDB, KDB and embedded debugger in Visual C++ take advantage of these debug registers. They allow programmers to monitor the accesses to specific instruction/memory locations in the programs being debugged. Whenever these locations are executed, read or written to, a debug exception is generated and the debugger is invoked to inspect the program state and handles the exception accordingly.

Accessing debug registers requires ring0 privilege, so a user program can manipulate them only by making a system call. The Windows operating system does not provide any system call that can directly modify the debug registers [15]. However, it does provide two Win32 library functions, `SetThreadContext` and `GetThreadContext`, For debuggers to set instruction/data breakpoints, which are the only known ways to access debug registers from the user space. The templates of these two functions are as follows:

```
BOOL SetThreadContext(
  HANDLE hThread,
  const CONTEXT* lpContext
);
```

```
BOOL GetThreadContext(
  HANDLE hThread,
  const CONTEXT* lpContext
);
```

Both functions take a thread handle and a pointer to CONTEXT data structure as arguments. A thread handle specifies the thread whose context is going to be set or retrieved. The CONTEXT data structure includes almost all the thread's state that is visible to user-level applications, including those debug registers. `SetThreadContext` delivers a user-specified thread context to the kernel while `GetThreadContext` copies the designated thread's context from the kernel back to the user space. Although these two functions allow *Lisbon* to read/write debug registers, unfortunately they carry serious performance penalty, because whenever a debug register is enabled, the kernel will set the application in the debug mode and the application is slowed down significantly due to extra checking and interception whenever the kernel gets control. The same performance problem exists in Linux as well [7].

Because the exact implementation of these two functions in the Windows operating system is not known, it is not possible to modify it to eliminate this performance problem. Consequently, we decide to implement a separate debug register management mechanism that directly reads/writes the debug registers in a way that is independent of SetThreadContext/GetThreadContext. This debug register management scheme consists of two components: (1) a low-overhead system call that sets or clears the debug registers according to user specification, and (2) a kernel routine that is invoked at every context switch to properly maintain the state of the debug registers across context switches. In terms of implementation complexity, these two components together take fewer than 200 lines of C code.

The first component is implemented as an interrupt handler for a software interrupt instruction `int 20`. Because this interrupt handler does not invoke the usual argument check and state saving/restoration procedure associated with system calls, it incurs a lower overhead than system calls. In particular, *Lisbon* communicates the addresses of canary words to the interrupt handler through general-purpose registers.

Without the second component, the debug registers set by the first component for one process will be effective for all processes. As the Windows kernel is not aware of *Lisbon*'s modification of these debug registers, it will not save/restore them across context switches. To solve this problem, *Lisbon* intercepts Windows' context switching function, `SwapContext`, which is responsible for switching the processor to a new thread. *Lisbon* tricks the Windows kernel to call a *Lisbon* routine called `SwapDebugContext` whenever it calls `SwapContext`. `SwapDebugContext` saves the debug registers of the old thread that is going to be switched out and restores the debug registers of the new thread that is going to be switched in. Consequently, the debug registers become an integral part of a thread's context, and properly get saved/restored during context switch.

5. Evaluation

The project goal of *Lisbon* is to provide an effective, transparent and efficient way to protect commercially distributed Win32 binaries from all known forms of format string attacks. In this section, we evaluate the effectiveness of *Lisbon* in protecting real Windows applications that are known to be vulnerable to format string attack, and then the efficiency of *Lisbon* in terms of run-time overheads.

5.1. Effectiveness

We used the following four Window binaries that are known to be vulnerable to format string attack in the effectiveness and efficiency evaluation of *Lisbon*: SHOUTcast[4] (a popular audio streaming server), ProFTPD[26] (a free FTP daemon program), Peercast[25] (a popular peer-to-peer streaming media server), Pfinger

client[1] (a free finger client). All of them have format string vulnerabilities [5, 23, 21, 22].

For each of these test programs, we either used an existing exploit or wrote a new one, and then created a *Lisbon* version from its binary. Then we ran each binary twice, once with the original binary and the other with the *Lisbon* version. When each test program ran, we launched the corresponding exploit code against it to check if it can detect the attack.

For each test program, its *Lisbon* version is able to detect the exploit sent to it and raise an alert successfully. Since applications such as SHOUTcast and peercast are distributed only in binary format, approaches such as White-listing and FormatGuard cannot protect them. In addition, these binaries are generated in such a way that the frame pointer is omitted. As a result, Libsafe would not work for them, either.

In addition to the ability to deal with binaries directly, *Lisbon* is also more complete than existing approaches. Unlike *white-listing*, *Lisbon* can detect read accesses to memory locations beyond the input argument list. This could prevent attackers from getting sensitive information embedded in the victim applications. Unlike FormatGuard, *Lisbon* can also protect vprintf-like functions even when they are encapsulated by multiple layers of wrapper functions. For example, Proftpd's vulnerability happens within a wrapper function that eventually calls `vsnprintf`.

5.2. Performance Overhead

Macro-benchmark Testing. We used two machines in performance overhead measurements. The machine used for micro-benchmark evaluation is used as the server for macro-benchmark tests. This machine runs Windows XP SP1 with PentiumIII 1.2GHz CPU and 256MB memory. The client machine for macro-benchmarking is a 1.0GHz Celeron machine with 128MB memory and runs RedHat Linux 7.2.

To measure the performance overhead of *Lisbon* when it is applied to real applications, we used the same set of test programs as in the effectiveness test. We set up an isolated network that connects the server and the client directly using a 100Mbps Ethernet link. The client machine continuously sends 1000 requests to the server, on which the test applications run. The throughput of a server application is calculated by dividing 1000 by the time interval between when the first request is issued and when the response of the last request is received. The latency is calculated by taking the average of the response times of these 1000 requests.

To test the streaming and ftp servers, the client machine continuously fetches a 1-KByte file from the server. For the Pfinger client, it repeatedly fingers a user on the server. All these requests were sent back-to-back. During these tests, only legitimate requests were used. For comparison, we ran each test application twice, once with the original binary and the other with the *Lisbon* version.

Table 3. The run-time performance overhead of Lisbon for macro-benchmark programs

Application	Throughput Penalty	Latency Penalty
SHOUTcast	99.8%	0.3%
ProFTPD	98.0%	1.6%
Peercast	99.6%	0.4%
Pfinger	N/A	0.0%

Table 3 shows that *Lisbon*'s performance overhead for these test applications is so small that it is almost negligible. This overhead includes the overhead due to BIRD. There are several reasons why *Lisbon*'s overhead is so low. First, because BIRD can statically disassemble most of the instructions in these test applications, the performance overhead due to dynamic disassembling is very small. Second, printf-like functions are invoked infrequently, and *Lisbon*'s transformation only takes effects when these functions are called. As stated in [18, 6], applications using printf-like functions heavily are rare in practice. Third, for the test applications the file system or network I/O overheads typically dominate the CPU overhead. Finally, *Lisbon* exploits the debug register hardware in the X86 architecture to minimize the list bound checking cost. These factors together suggest that the run-time overhead of *Lisbon* , even for printf-intensive applications such as man2html, is unlikely to be an issue in practice. To be fair, the performance overhead of most other previously proposed approaches to format string attack prevention is also quite small.

Micro-benchmark Testing. We also use the micro-benchmark programs in [18] to evaluate the absolute performance cost of *Lisbon*. These programs print a six-character string without any specifier, with two %d specifiers, or with two %n specifiers. The first three use the `sprintf` function, which is the lightest-weight among printf-like functions. The latter three use the `vsprintf` function. The latency measurements for these six programs under FormatGuard, White-listing and *Lisbon* are also shown in Table 4. The numbers for FormatGuard and White-listing are taken from their respective papers.

Because *Lisbon* needs to make a system call to set up the debug registers, the user/kernel mode transition is very expensive when compared with the overhead associated with the `sprintf` and `vsprintf` function. That is why *Lisbon*'s overhead is much higher than the other two. Because *Lisbon* does not require interpretation of the format string, its *absolute* performance overhead is independent of the length of and the number of conversion specifiers in the format string. On the other hand, for the *absolute* per-

formance overhead of both FormatGuard and White-listing grows proportionally with the length of and the number of conversion specifiers in the format string because they need to interpret the format string. This is why the percentage overhead of the "sprintf with no specifiers" case is lower than that of the "sprintf with 2 %d specifiers" case for both FormatGuard and White-listing, but is actually much higher for *Lisbon*.

Table 4. Latency penalty comparison for FormatGuard, White-listing and Lisbon on micro-benchmarks

Benchmark	Format-Guard	White-listing	Lisbon
sprintf with no specifiers	7.5%	10.2%	217.7%
sprintf with 2 %d specifiers	20.9%	28.6%	67.9%
sprintf with 2 %n specifiers	38.1%	60.0%	142.3%
vsprintf with no specifiers	cannot handle	26.4%	223.4%
vsprintf with 2 %d specifiers	cannot handle	39.8%	63.2%
vsprintf with 2 %n specifiers	cannot handle	74.7%	154.7%

Because *Lisbon*'s absolute performance overhead is fixed, its percentage overhead decreases as the baseline overhead increases. Since `sprintf` itself takes less time than `printf` and `fprintf`, the relative overhead of *Lisbon* should be smaller for other printf-like functions. To evaluate the relative overhead of *Lisbon* when different printf-like functions are used, we try different printf-like functions using the same input: a ten-character string with two %d specifiers as the format string. The latency results are shown in Table 5. As expected, the relative overhead of *Lisbon* decreases with the increase in the inherent amount of processing inside the printf-like function it protects. Among these printf-like functions, printf and vprintf is the most time-consuming because they both need to make system calls to interact with the video hardware.

Table 5. Latency measurements for different variadic functions with the same format string and input arguments.

Benchmark	Lisbon Overhead
printf with 10 chars and 2 %d specifiers	2.0%
sprintf with 10 chars and 2 %d specifiers	52.5%
fprintf with 10 chars and 2 %d specifiers	34.3%
vprintf with 10 chars and 2 %d specifiers	2.1%
vsprintf with 10 chars and 2 %d specifiers	57.3%
vfprintf with 10 chars and 2 %d specifiers	44.0%

Although micro-benchmarking results provide a better understanding of the source of performance overhead, they do not reflect the real performance impacts of *Lisbon*, which should be measured by macro-benchmarking results from real network applications.

6. Limitations

One major restriction of *Lisbon* is that it assumes that the protected variadic function never skips any of its input arguments. This assumption holds for all known printf-like functions on the Windows platform because they follow the C99 standard [10]. However, for arbitrary variadic functions, there is no guarantee that they will not skip any of the input arguments. In addition, *Lisbon* assumes the `va_list` structure used in a vprintf-like function resides in the stack frame of some upstream caller function. This assumption is true for all known vprintf-like functions that are vulnerable to format string attack. However, in general, a `va_list` structure could also reside in the heap. In that case, *Lisbon* needs to traverse the `va_list` to identify its bound, and therefore may become application-specific. Finally, the current BIRD prototype cannot instrument Windows binaries employing self-modifying code, for example, those protected by such packers as UPX or ASprotect. Therefore, *Lisbon* cannot protect them either. Despite the above limitations, we believe *Lisbon* still represents a significant advance in protecting Win32/X86 binaries from published format string attacks.

7. Conclusion

Format string attack is possible because the attacker tricks a printf-like function into accessing more arguments than is prepared by its caller, and eventually is able to read or modify the victim application's address space. This paper describes the design, implementation and evaluation of the first known tool that can protect Win32 binaries from format string vulnerabilities that have been discovered in real programs so far. This tool, called *Lisbon*, features a novel input argument list bound checking mechanism to detect format string attacks. Experiments on the first *Lisbon* prototype, which is built for the Windows XP/2000 platform, show that *Lisbon* can indeed stop known format string attacks, and its run-time performance overhead is negligible, less than 2%. Compared with previous format string attack prevention systems, *Lisbon* is novel in the following ways:

- By casting format string attack prevention into an argument list bound checking problem, *Lisbon* does not require interpretation of format string argument and thus can potentially be applied to similar attacks against functions that access input arguments in the same way as printf.
- By exploiting the debug register hardware in modern processors, *Lisbon* reduces the run-time performance overhead associated with list bound checking to the minimum.

- *Lisbon* leverages the BIRD infrastructure for binary analysis and transformation and is the first known system that can directly protect Windows binaries from format string attacks without accessing their source code.

Although there are already numerous approaches to thwarting memory corruption attacks, perhaps the most effective approach is to stop the attacker from tampering the victim application's address space in the first place. Array bound checking stops buffer overflow attacks, argument list bound checking stops format string attacks, and arithmetic overflow checking stops integer overflow attacks. Our long-term goal is to develop a comprehensive security-enhancing compiler that can immunize network applications from memory corruption attacks by transparently inserting these checks to their source code and/or binary code. This overflow-preventing compiler will be built on the CASH compiler [2] and the *Lisbon* technology presented in this paper.

References

[1] M. Baumer. The pfinger server and client. http://www.xelia.ch/unix/pfinger/about.

[2] L. chung Lam and T. cker Chiueh. Checking array bound violation using segmentation hardware. In *International Conference on Dependable Systems and Networks (DSN'05)*, pages 388–397, 2005.

[3] M. Corporation. Microsoft portable executable and common object file format specification. http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx.

[4] N. Corporation. Shoutcast: Free internet radio. http://www.shoutcast.com.

[5] S. Corporation. Shoutcast remote format string vulnerability. http://www.symantec.com/avcenter/attack_sigs/s21253.html.

[6] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. Formatguard: Automatic protection from printf format string vulnerabilities. In *Proc. of the 10th Usenix Security Symposium*, Aug 2001.

[7] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Conference*, pages 63–77, San Antonio, TX, January 1998.

[8] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition (DISCEX)*, pages 119–129, Jan 2000.

[9] A. DeKok. Pscan: A limited problem scanner for c source files. http://www.striker.ottawa.on.ca/aland/pscan/, July 2000.

[10] I. O. for Standardization. Iso/iec 9899: Programming languages - c (c99 standard). http://www.open-std.org/JTC1/SC22/WG14/www/standards.

[11] F. S. Foundation. The c preprocessor. http://www.cs.utah.edu/dept/old/texinfo/cpp/cpp_toc.html.

[12] F. S. Foundation. Gcc: the gnu compiler collection. http://gcc.gnu.org/.

[13] Intel. Intel architecture software developer's manual. http://developer.intel.com/design/pentium/manuals/.

[14] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[15] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, 2nd edition*. Prentice-Hall, 1988.

[16] MSDN. Microsoft msdn library. http://msdn.microsoft.com/library/.

[17] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[18] S. Nanda, W. Li, L. chung Lam, and T. cker Chiueh. Bird: Binary interpretation using runtime disassembly. In *Proceedings of the 4th IEEE/ACM Conference on Code Generation and Optimization (CGO'06)*, March 2006.

[19] M. F. Ringenburg and D. Grossman. Preventing format-string attacks via automatic and efficient dynamic checking. In *Proceedings of the 12th ACM conference on Computer and communication security*. ACM Press, 2005.

[20] T. Robbins. Libformat: A linux library that prevents format string attacks. http://www.securityfocus.com/tools/1818, November 2001.

[21] Scut and T. Teso. Exploiting format string vulnerabilities. http://julianor.tripod.com/teso-fs1-1.pdf, March 2001.

[22] SecuriTeam. Peercast format string vulnerability. http://www.securiteam.com/securitynews/5KP0U0AFQA.html.

[23] SecuriTeam. Pfinger format string vulnerability. http://www.securiteam.com/unixfocus/6K00N1P3FQ.html.

[24] SecurityFocus. Proftpd shutdown message format string vulnerability. http://www.securityfocus.com/bid/14381/info.

[25] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *In Proccedings of the 10th USENIX Security Symposium*, pages 201–220, Aug 2001.

[26] T. P. P. Team. Peercast p2p radio. http://www.peercast.org.

[27] T. P. P. Team. Proftpd: Highly configurable gpl-licensed ftp server software. http://www.proftpd.org.

[28] T. Tsai and N. Singh. Libsafe 2.0: Detection of format string vulnerability exploits. http://www.research.avayalabs.com/project/libsafe/doc/whitepaper-20.pdf%, Feb 2001.

[29] US-CERT. Format string input validation error in wu-ftpd site_exec() function. http://www.kb.cert.org/vuls/id/29823.