國立交通大學

資訊科學與工程研究所

碩士論文

動態格式化字串攻擊偵測方法之研究

Run-Time Detection of Format String Attacks

研究生：洪慧蘭

指導教授：黃世昆　教授

中華民國　九十七　年四月

動態格式化字串攻擊偵測方法之研究

Run-Time Detection of Format String Attacks

研 究 生：洪慧蘭　　Student：Hui-Lan Hung

指導教授：黃世昆　　Advisor：Shih-Kun Huang

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

April 2008

Hsinchu, Taiwan, Republic of China

中華民國九十七年四月

動態格式化字串攻擊偵測方法之研究

學生：洪慧蘭　　指導教授：黃世昆　老師

國立交通大學資訊科學與工程學系（研究所）碩士班

摘要

為了避免發生格式化字串弱點所引起的問題，相關字串處理函式的行為必須有所規範，不應有超越界限存取參數的行為。若攻擊者有能力控制格式化字串函式的字串參數，便能利用此弱點，提供超出參數數量的轉換符號，進行不同型態的攻擊。

在本論文中，我們提出一個對 printf 與 vprintf 系列函式的偵測攻擊方法，發展一檢查函式存取參數是否超出界限的工具，稱為 FormatDefense。此工具對格式化字串函式進行完整保護，藉由在記憶體上定義防禦線的方式，判斷此型態函式存取參數的合法性；若存取參數超越此防禦線則視為攻擊行為。我們將此方法實作在 UNIX 環境下，建立一個共享函式庫，並分析除錯資訊與追蹤堆疊變化，取得函式存取參數的界限，當程式執行時，只要連結此函式庫，便能保護格式化字串函式的運作。

我們考慮格式化字串儲存位置的變化，共歸納出六種可能的情境，評估偵測方法的有效性。實驗結果顯示，此工具皆能成功偵測到所有情境中的違法存取行為，且超越其他現存的偵測工具，證明我們的方法有更高的精確度；此外，我們將偵測方法應用在已知含有弱點的程式，皆能成功偵測到相關攻擊，驗證此方法的有效性；最後我們利用微型基準測試與巨型基準測試，評估工具所引起的執行負擔，實驗結果發現其負擔是可忽略的，顯示此工具能夠實際應用在真實程式中，防止格式化字串攻擊。

Run-Time Detection of Format String Attacks

Student：Hui-Lan Hung　　Advisor：Dr. Shih-Kun Huang

Department of Computer Science and Engineering
National Chiao Tung University

Abstract

In order to prevent format string vulnerabilities, the behavior of accessing arguments for variadic functions should be regulated. A format string attack occurs because variadic functions rely on the format string argument to determine the number of arguments. Therefore, if an attacker has ability to control the format string argument, he can exploit format string vulnerabilities to attack programs by providing more conversion specifiers than needed. In this thesis, we develop an attack-detecting tool called *FormatDefense* for `printf`-like and `vprintf`-like functions to check if a variadic function accesses arguments outside its argument list. FormatDefense defines the access bound in the memory via offline analysis of debugging information and runtime tracking of the stack. It is implemented as a shared library in the UNIX environment. We consider six scenarios based on format string locations to evaluate the effectiveness of FormatDefense. The result shows that FormatDefense surpasses several existing detection tools in detecting invalid memory access in the six scenarios. Furthermore, FormatDefense can detect exploits successfully on several programs with known format string vulnerabilities. Eventually, we use various microbenchmarks and macrobenchmarks to evaluate the performance overhead. The overhead is negligible so that FormatDefense can be applied to real programs practically to avoid format string attacks.

誌謝

首先我要感謝我的家人，謝謝他們一路支持我的選擇與堅持，謝謝他們對於我這個長期不在家的成員依然給予最多的關懷。

我要感謝我的指導教授，黃世昆老師，謝謝他在研究上的指導與生活上的叮嚀，謝謝他提供我足夠的資源，讓我能專心於課業與研究上。

我要感謝師母，宋定懿老師，謝謝她給予論文寫作上的意見。

我要感謝口試委員，孔崇旭老師與馮立琪老師，謝謝他們在口試時的建議與肯定。

我要感謝實驗室的大家，昌憲學長、泳毅學長、友祥學長、立文學長、spanky、琨翰、士瑜學弟，謝謝他們在交大的陪伴，對我的耐心指導讓我學習很多，大家都對我很好，我很喜歡 SQLab。

我要感謝我的大學同學，阿亮、live、毓雯、于瑄、蕙今、妙純、kaddy、moomin，謝謝他們一路從大學到研究所的陪伴，讓我在新竹依然能感受到他們友誼的關懷。

我要感謝我的高中同學，佩儀，謝謝她在交大的陪伴，一起完成的許多事與給我的關懷都讓我一再覺得溫暖。

最後我要感謝所有曾給予我支持與鼓勵的人，能如願進入交大且完成畢業論文是來自大家給我的力量，我很幸運，也很幸福，謝謝大家！

Table of Contents

List of Tables

List of Figures

1. Introduction

1.1 Problem Description

In 2000, the format string vulnerability was discovered in WU-FTP that acts like a buffer overflow vulnerability. Format string vulnerabilities [1] occur because attackers take advantage of a program's trust in format strings. If an attacker can control a program's format string, he can access extra arguments, thereby viewing or overwriting data in the program's memory. Format string vulnerabilities are not easy to be detected by manual testing because they usually occur in corner codes, such as those used to log errors. As of January 2008, MITRE's CVE project listed nearly 500 programs with format string vulnerabilities [2] and ranked the format string vulnerability as the ninth most-reported type of vulnerability [3] between 2001 and 2006. Thus in recent years, researchers have developed various static and dynamic tools to detect format string vulnerabilities. Many tools track the data flow of a program to determine whether a format string is tainted (i.e., it can be modified by an outside user) statically [4, 5] or dynamically [6, 7, 8]. Although these tools are capable of finding new bugs, they generate false positives or incur high overheads. Fine-grained and low-overhead dynamic detection is thus indispensable for protecting programs against format string attacks.

1.2 Background

1.2.1 Attack Models of the Format String Vulnerability

Format string vulnerabilities arise because the aim between strong type checking and convenient passing arguments method is opposite. The decision of C language is the unsafe varargs mechanism for convenience.

Format string functions in the C Standard Library, such as `printf` and `vprintf`, are variadic functions, which take an arbitrary number of arguments. These functions rely solely on the format string argument to determine the number and types of other arguments. When parsing the format string, they rely on the conversion specifier to retrieve the argument on the stack.

A format string usually contains ordinary text and conversion specifiers. The format string function prints the format string whose conversion specifiers have been replaced with the corresponding arguments. The function maintains two internal pointers: (1) FMTPTR, which points to the next character of the format string; and (2) ARGPTR, which points to the next argument in the stack. FMTPTR initially points to the beginning of the format string and advances with the function until the end of the string. When FMTPTR points to a normal character, the format string function simply prints the character into the destination stream. However, when it points to a conversion specifier (starting with a `%` sign), the function prints the argument

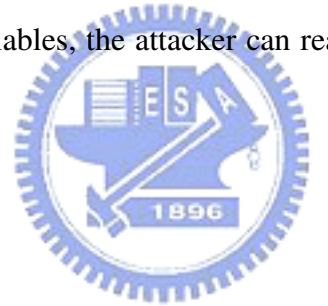
indicated by ARGPTR. The type of the conversion specifier determines how many bytes ARGPTR will advance to the next argument. For example, with the `%d` specifier, the format string function retrieves the word indicated by ARGPTR, prints the word as a signed decimal notation, and then advances ARGPTR by four bytes. Table 1 describes the output corresponding to each conversion specifier.

Table 1: Conversion specifiers

Conversion Specifier	Output
`%d` or `%i`	Signed decimal integer
`%o`	Signed octal
`%u`	Unsigned decimal integer
`%x`	Unsigned hexadecimal integer
`%c`	Character
`%s`	String of characters
`%p`	Pointer address
`%n`	Nothing printed. The argument must be a pointer to a signed `int`, where the number of characters written so far is stored

Although convenient for programmers, the above formatted output mechanism permits two types of bugs. First, the format string function accesses arguments without type checking. If the conversion type does not match the corresponding argument, the format string function will misinterpret the argument, e.g., it will regard an argument of type `int` as one of type `float`. However, it is difficult, if not impossible, to exploit this kind of bug. Second, the format string function retrieves arguments without bound checking so that it is possible to access more arguments than those passed to the function.

A typical example is that programmers tend to use `printf(str)` as shorthand for `printf("%s", str)`. Although the two function calls are almost equivalent, the former triggers undefined behavior when the `str` contains a conversion specifier. It is also common to find that programmers, even experienced ones, implement a wrapper function for `vprintf()` and allow users to provide the format string. If an attacker can control the format string passed to `printf` or `vprintf`, the program contains a format string vulnerability. By directing the format string function to access extra arguments, the attacker can view the stack or crash the program; by directing the function to access tainted variables, the attacker can read or overwrite any address in the program's memory.



1.2.1.1 Accessing Arguments outside the Real Argument List

If an attacker provides too many specifiers in a format string, he will be able to direct the format string function to view the stack or crash the program.

Viewing the Stack

For example, as shown in Figure 1, the call to `printf("%x%x%x")`, which does not provide corresponding arguments to those specified in the format string, outputs three integers in the stack in unsigned hexadecimal notations. As a result, an attacker can view the content above the stack frame of `printf` in the stack and

possibly obtain important data, e.g., the password.

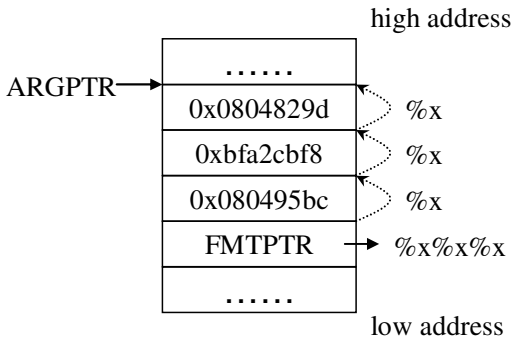


Figure 1: Using `printf("%x%x%x")` to view the stack

Crashing a Program

Denial of service is another type of trivial attack. For instance, the call to `printf("%s%s%n%n")` crashes nearly every program. The `%s` specifier reads a pointer in the stack and displays a string in the address, whereas the `%n` specifier reads a pointer in the stack and writes the current number of characters output in the address. Without corresponding arguments, the format string can easily trigger invalid memory access via arbitrary pointers in the stack.

1.2.1.2 Accessing Tainted Variables

In addition to the attacks described above, if an attacker can direct the format string function to access tainted data (i.e., data that is manipulated by the attacker), he can either view (by the `%s` specifier) or overwrite (by the `%n` specifier) any address in the memory. For example, assume a function `foo` calls `printf(fmt)`, as shown in

Figure 2. The array `fmt` contains the format string, "`\x3c\xe2\x88\xbf%n`", and the return address of the function `foo` is stored in the address `0xbf88e23c`. When `printf` handles the `%n` specifier, ARGPTR points to the beginning of the format string, which is `0xbf88e23c` in little-endian format, and the return address of `foo` will be overwritten by the `%n` specifier. An attacker can write any desired value via multiple overwrite operations. A typical exploit overwrites an entry in the Global Offset Table (GOT) to gain control of the program.

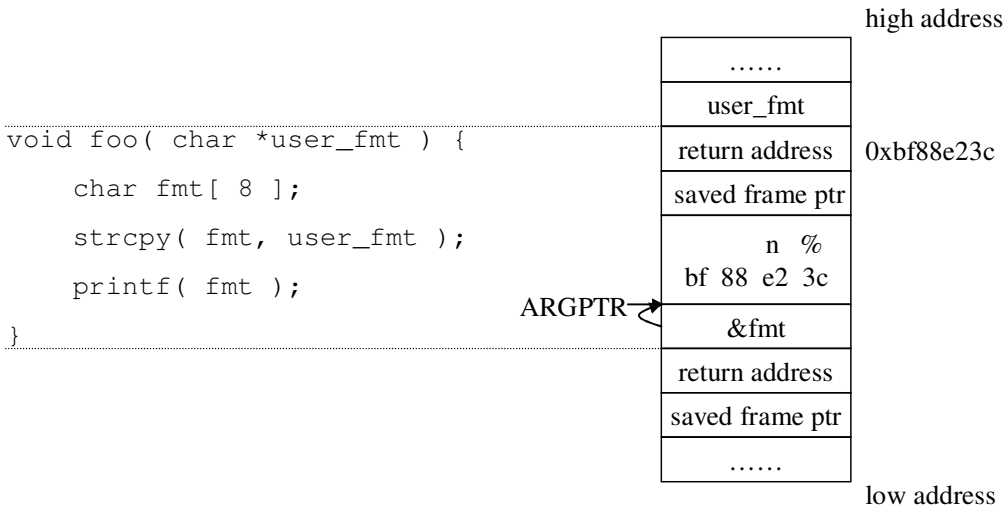


Figure 2: Overwriting the return address of the function `foo`

1.3 Motivation

Current fine-grained dynamic detection tools, such as FormatGuard [9], Libsafe 2.0 [10], and Kimchi [11], cannot detect every kind of format string attack, especially those against `vprintf`-like functions. In fact, most fine-grained tools actually enable successful attacks because they are inaccurate. We present a dynamic detection tool

called FormatDefense, which tries to detect out-of-bound access of arguments in a format string function in order to prevent format string attacks.

1.4 Objective

We implement FormatDefense in the UNIX environment because format string vulnerabilities appear more frequently in open source software [3] and we need to analyze some information contained in the source code of a target program. The tool intercepts every format string function and finds the function (e.g., `foo`) that holds the argument list for the intercepted function. To determine the access bound, FormatDefense uses the frame pointer of the function `foo` less the size of its local variable region as the bound. The intercepted format string function should not be able to access arguments outside this bound; otherwise, it will be deemed an attack. The main advantage of FormatDefense over other bound checking tools is that it is capable of protecting the local variables above the variable-length argument list.

FormatDefense has the following features:

1. **Full protection**: FormatDefense can protect both `printf`-like and `vprintf`-like functions against format string attacks.
2. **High accuracy**: FormatDefense can prevent format string functions from accessing arguments outside the argument list so that typical format string attacks

fail.

3. **Light weight**: The runtime overhead of FormatDefense is lower than that of Lisbon, which provides the best protection for the Win32 platform.

1.5 Synopsis

In Section 2, we present a review of related work. We present our proposed method, FormatDefense, in Section 3, and discuss its implementation in Section 4. A detailed evaluation of FormatDefense is given in Section 5. Then, in Section 6, we present some concluding remarks.

2. Related Work

A number of tools can protect `printf`-like functions against format string attacks, but only a few provide protection for `vprintf`-like functions. Some tools detect the non-static format string, while others specifically protect against attacks on the format string with the `%n` specifier. Other tools resolve vulnerabilities by applying bound checking methods.

2.1 Detection of a Non-static Format String

If a format string function is called with a static string, e.g., "`hello %s`", it is immune to the format string vulnerability because an attacker has no way of controlling the format string. On the other hand, if the format string is non-static, the call to the format string function *may* be vulnerable.

PScan [12] and the GNU Compiler Collection (GCC) can warn programmers about non-static format strings. PScan scans the C source code to search the call to the `printf`-like function whose last argument is a non-static format string. GCC provides a similar capability with a compiler flag, "`-Wformat=2`". These two tools are effective in detecting the `printf(fmt)` vulnerability, but they generate a false positive if an attacker cannot control the format string.

Rao [13] uses the following macro to check whether the format string is a

pointer:

```
#define printf(fmt, ...) \
    ((sizeof(fmt)==4) \
    ?incorrect_printf():correct_printf())
```

The method can trigger a false positive when the size of a static format string is 4 bytes, e.g., `printf("abc")`. In this case, the format string is mistaken as a pointer.

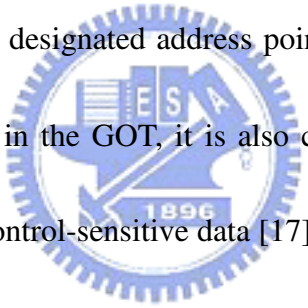
2.2 Protection of a Format String Containing a `%n` Specifier

The format string vulnerability is particularly dangerous when the format string contains a `%n` specifier, which can be used to overwrite the memory. Corrupting important control data, e.g., the GOT, will very likely lead to a successful exploit. Several tools can detect such vulnerabilities. For example, Libformat [14] intercepts calls to `printf`-like functions and prevents them from using a potentially malicious format string that is non-static and contains a `%n` specifier. Although the method is effective in preventing format string attacks, it generates false positives for normal format strings containing a `%n` specifier.

Libsafe [10] protects unsafe functions in the C library. It uses a black-list approach to protect format string functions, i.e., the designated address of the `%n` specifier cannot point to a return address or a frame pointer in the stack. In contrast,

Ringenburg and Grossman [15] propose a white-list approach that registers all valid integers at runtime. If the designated address of the `%n` specifier does *not* point to the valid integers, it is considered an attack.

FASTV [16] checks whether a tainted format string contains a `%n` specifier by using a dynamic taint technique to gather information about untrustworthy data. To detect attacks, FASTV provides two policies, a default policy and a fine-grained policy. The former deems that a tainted format string with a `%n` specifier is an attack, whereas the latter examines an untainted format string further. If the format string contains a `%n` specifier whose designated address points to a return address, a frame pointer, a DTOR, or an entry in the GOT, it is also considered an attack. However, FASTV cannot protect other control-sensitive data [17] against a `%n` specifier attack.



2.3 Bound Checking Method

In contrast to the above-mentioned tools, which either generate many false positives or do not prevent an attacker from viewing the program's memory, the bound checking approach tries to detect all kinds of format string attacks at runtime. Essentially, this approach defines a line of defense in the stack. If ARGPTR accesses arguments beyond this line, it is considered a format string attack. There is some prior work on keeping the conversion specifiers from accessing superfluous arguments. We

review three types of defense lines considered in the literature and compare their accuracy.

2.3.1 The Saved Frame Pointer as the Defense Line (line-Sfp)

In addition to verifying the safety of the `%n` specifier, it is claimed that Libsafe 2.0 [18] checks whether the argument list is contained within the same stack frame to avoid accessing arguments exceeding the current stack frame. However, we cannot find the related implementation in the source package.

Kimchi [11] is a binary-rewriting tool that protects format string functions. It uses the stack frame address of the parent function as the bound for `printf`-like functions and defines the stack frame depth as an offset relative to the stack frame of the parent function. If a program uses the frame pointer, the depth of the stack frame is calculated from the frame pointer and the stack pointer; otherwise, it is calculated via static analysis of the change in the stack pointer. Kimchi implements a `safe_printf` function that replaces the `printf` call. The depth of the stack frame is passed to `safe_printf` to verify whether the format string is safe. If the string is deemed safe, the tool calls the real `printf`; otherwise, it is defined as an attack.

The major drawback of using line-Sfp as the defense line is that it cannot protect the local variables below the saved frame pointer. If an attacker can control those

local variables, an exploit is possible.

2.3.2 The Format String as the Defense Line (line-Fmt)

Unlike tools that protect against format string bugs, the tool developed by Ganapathy *et al.* [19] analyzes whether a `printf` call is exploitable by considering the relationship among ARGPTR, FMTPTR, and the length of the format string. If ARGPTR can reach the beginning of the format string before FMTPTR reaches the end of the same string, then an exploit is possible. Thus the beginning of the format string can be used as the defense line to thwart exploits.

The limitation of this approach is that it cannot discover exploits when ARGPTR reaches other variables controlled by the attacker before it reaches the format string. In addition, the method cannot resolve cases where the format string is stored in the heap or other areas below the stack.

2.3.3 The Argument List as the Defense Line (line-Arg)

FormatGuard [9] is a modified implementation of glibc that checks whether the number of arguments specified by the format string matches the number of actual arguments passed to a `printf`-like function, and uses a macro production to provide a safe wrapper to obtain the number of arguments required by the format string. When

the number of actual arguments is less than the number of arguments specified by the format string, it is considered a format string attack. In other words, FormatGuard attempts to discover a bound that prevents `printf` from accessing arguments beyond the region of actual arguments. Nevertheless, if a program calls `printf` indirectly via a function pointer, FormatGuard cannot protect `printf` because it does not trigger a safely wrapped `printf` to expand the macro.

Lisbon [20] is a Win32 binary-rewriting tool used to detect whether a format string function accesses arguments outside the argument list. For `printf`-like and `vprintf`-like functions, Lisbon attempts to recognize the argument list and insert a "canary" above the list as a sentinel. For argument-list bound checking, the tool provides a wrapper, called `stub wrapper`, which duplicates the original arguments, puts the canary above the argument list, and then calls the format string function. If the canary, which is monitored by the *debug register*, is accessed at runtime, it is regarded as an argument list bound violation.

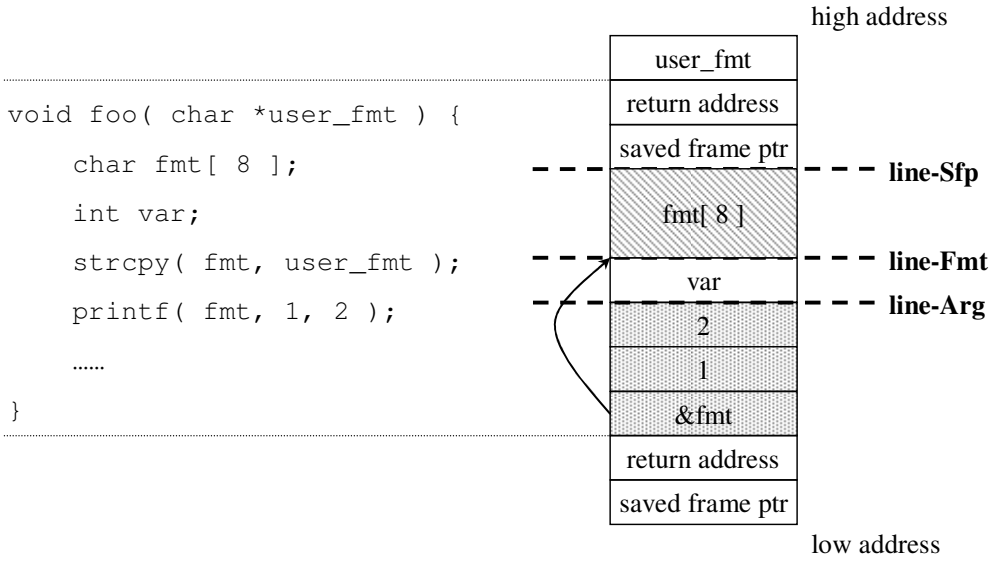


Figure 3: Illustration of the three types of defense lines

2.3.4 The Accuracy of the Defense Lines

The effectiveness of the bound checking method depends on whether the defense line is accurate. The closer the defense line is to the end of the argument list downward, the better the defense line will be, because ARGPTR advances toward higher addresses in the stack. If the defense line is not close to the end of the argument list, the gap between the line and the end of the list becomes an attack space; the defense line cannot protect the variables in that space. Hence if an attacker can control any variable in the space, he can launch the type of attack described in Section 1.2.1.

In Figure 3, we present a typical scenario to illustrate the attack spaces defined by the above three defense lines. The function `foo` takes a user-supplied format string

and copies it into the array `fmt`. Then, `printf` is called with a format string `fmt`, an integer 1, and an integer 2. In this example, line-Arg is the best of the three defense lines because it allows the smallest attack space.

2.4 Comparison

Table 2 describes detailed comparisons between FormatDefense and other bound checking tools. Even though several tools have been developed for handling the format string vulnerabilities, few of them provide protection for `vprintf`-like functions. FormatGuard and Kimchi are unable to count the variable argument list that is dynamically constructed. Some tools are format string content-based protection, e.g., Libformat and FASTV. They deal with the format string with %n specifiers, but they have no concern with the problem of accessing extra arguments by `vprintf`-like functions. For `vprintf`-like functions, Libsafe performs two checks, %n specifiers and the range of the argument list. In spite of checking the argument list, it incurs false negatives because of the lower accuracy of the defense line method. Besides, a program may store the address of a format string function in a function pointer variable, and then call the format string function later. FormatDefense is able to protect such format string function calls because of the interception of vulnerable functions.

For Lisbon, it assumes that the variadic function never skips any argument in its argument list. In other words, the variadic function does not allow itself to access one of arguments directly. If there is a varadic function that can skip any argument in its argument list, FormatDefense can prevent this kind of attack.

Table 2: Comparisons between FormatDefense and other bound-checking tools

	Tool				
Feature	Kimchi	Libsafe	FormatGuard	Lisbon	FormatDefense
protection of vprintf-like functions		Y		Y*	Y
indirect calls				Y	Y
no false positives		Y	Y	Y	Y
prevention of read attacks		Y**	Y	Y	Y
protection of current stack frame	Y		Y	Y	Y
availability of '%n' specifiers	Y		Y	Y	Y
read-only format string support	Y			Y	Y

Y*: depend on the precision of the heuristic Y**: cannot find the related implementation

3. Method

FormatDefense adopts the bound checking approach to protect both `printf`-like and `vprintf`-like functions. Based on the comparison of the defense lines shown in Figure 3, FormatDefense uses line-Arg as the bound, similar to the FormatGuard approach. However, unlike FormatGuard, which works for `printf`-like functions, but not `vprintf`-like functions, our proposed approach employs a novel means for finding line-Arg.

To determine a stack's defense line, one can traverse the stack in either the upward or the downward direction. FormatGuard uses the upward approach to calculate how many arguments a format string function accepts. However, it cannot calculate the number of arguments in `vprintf`-like functions because they take a `va_list` pointer to the real arguments passed through many layers of wrapper functions. FormatDefense follows the downward direction to locate the local variable region of the stack frame that holds the argument list. Although FormatDefense does not know the actual number of arguments passed to the format string function, it can still set the correct bound.

3.1 Stack Walking

We first perform stack walking to find the target function, i.e., the function that accepts the real argument list. When a `printf`-like function is called, the function itself is the target function (e.g., the `printf` function in Figure 4). On the other hand, when a `vprintf`-like function is called, the target function is one of its wrapper functions (e.g., the `vprintf_wrapper` function in Figure 5), because the argument list is not directly next to the stack frame of the format string function. In a typical stack layout of a program without optimization, every stack frame is identified by a frame pointer that points upward to the previous stack frame. Therefore, we need to perform the frame pointer backtrace operation repeatedly until we find the target function.

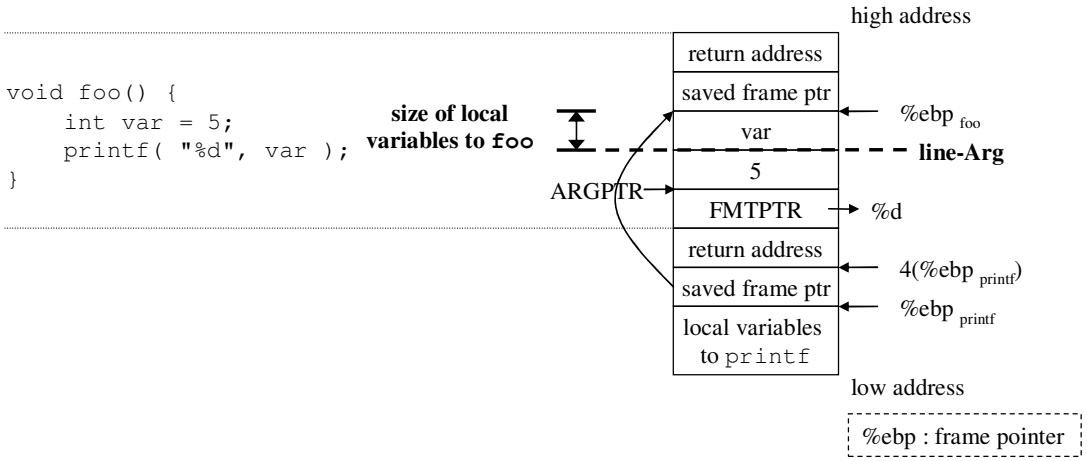


Figure 4: Determining line-Arg for `printf`-like functions

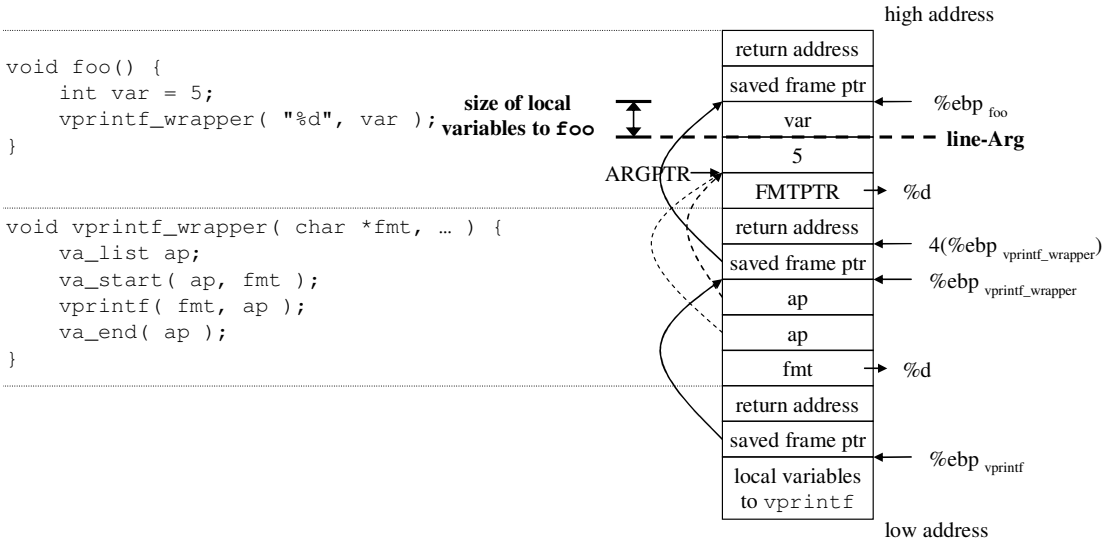


Figure 5: Determining line-Arg for `vprintf`-like functions

3.2 Determining Line-Arg and Detecting Attacks

After locating the target function, we use its return address to find its parent function, which holds the real argument list (e.g., the function `foo` in Figure 4 and Figure 5). Then, we can determine line-Arg by using the frame pointer and the size of the local variable region of the function `foo`.

We can use the debugging symbols of the target program to obtain the size of the local variable region of each function. By subtracting the size of the local variable region from `foo`'s frame pointer, we can derive line-Arg, which defines the bound for accessing arguments by format string functions. If the intercepted format string function accesses arguments outside line-Arg, the event is considered an attack.

3.3 Examples for Illustrating How to Find Line-Arg

In this section, we provide two examples, calls to `printf` and `vprintf`, and illustrate how to find line-Arg.

The mechanism of finding line-Arg for `printf`-like functions is illustrated by the example as shown in Figure 4. In the function `foo`, a local variable `var` of type `int` is declared and initialized to 5. Next, `printf` is then called with a format string, "`%d`", and the variable `var` with value 5 as its arguments. We can refer to 4(%ebp $_{printf}$) as the return address of `printf` and recognizes the function which holds the actual argument list for `printf`, i.e., the function `foo`. FormatDefense uses the size of local variable region of `foo` and %ebp $_{foo}$ to figure out line-Arg.

The process of finding line-Arg for `vprintf`-like functions is more complicated than `printf`-like ones due to the different kinds of input arguments. In Figure 5, we show an example involving a `vprintf` function and its mechanism to find line-Arg. In this example, `vprintf` accepts an object of type `va_list`, which points to a variable-length argument list. The object is initialized by the macro `va_start` which adjusts ARGPTR to point to the first argument in the argument list. In addition, the object of type `va_list` can be passed among several layers of wrapper functions. Therefore, the argument list is not adjacent to the stack frame of the intercepted format string function. Consider the code segment in Figure 5, the

function `foo` passes a format string, "`%d`", and the variable `var` with value 5 to `vprintf_wrapper`. The function `vprintf_wrapper` declares a local object `ap` of type `va_list`, which is prepared for the argument list of `vprintf`. After invoking the macro `va_start` to initialize `ap` for use, `vprintf` is called with `ap` as one of its arguments. Now ARGPTR is equivalent to `ap`.

We traverse the frame pointer, %ebp, until we find the stack frame of `foo` where ARGPTR points to, in order to locate a certain frame pointer that points to a higher address than ARGPTR. The procedure proceeds as follows. As shown in Figure 5, we refer to %ebp $_{\text{vprintf}}$ to realize where %ebp $_{\text{vprintf\_wrapper}}$ points to, and then we compare the location where %ebp $_{\text{vprintf\_wrapper}}$ points to with the location where ARGPTR points to. Because the address is not higher than ARGPTR, the process continues. Finally, we compare the location where %ebp $_{\text{main}}$ points to with the location where ARGPTR points to and the former is higher than the latter. Subsequently, we use the return address of `vprintf_wrapper` through 4(%ebp $_{\text{vprintf\_wrapper}}$) to find the function `foo` which holds the actual argument list, the integer with value 5, and to determine line-Arg similar to the procedure for `printf`-like functions.

3.4 An Algorithm for Detecting Attacks

We summarize the above-mentioned process in the DETECT_ATTACK algorithm shown in Figure 6. For input, the algorithm takes the frame pointer of a format string function, represented as `framePtr`, as well as the format string `format` and the `ap` of type `va_list`. The **while** loop in lines 1-2 traverses the frame pointers in the stack by comparing them with the `ap` until the location that the dereferenced `framePtr` points to is higher than the `ap`. Then, we obtain the frame pointer that indicates the stack frame that the `ap` points to. Line 3 takes the return address of the function that prepares the argument list that the `ap` points to. In line 4, the algorithm uses the return address to query the reference table about the size of the local variable region. Line 5 calculates the address of line-Arg. Line 6 determines how many bytes the `ap` will advance according to the format string. Line 7 obtains the final address of the `ap`. Line 8 then checks whether the address is higher than line-Arg. If it is higher, the function LOG_ATTACK is called in line 9 to log the attack. Then, in line 10, DETECT_ATTACK returns TRUE to indicate that an attack has been detected.

```
DETECT_ATTACK( framePtr, format, ap )
 1  while *framePtr < ap
 2        do framePtr ← *( framePtr )
 3  returnAddress ← *( framePtr + 1 )
 4  sizeOfLocalVars ← QUERY-LOCAL-SIZE( returnAddress )
 5  addressOfLine-Arg ← framePtr – sizeOfLocalVars
 6  sizeOfArgs ← PARSE( format )
 7  finalARGPTR ← ap + sizeOfArgs
 8  if finalARGPTR > addressOfLine-Arg
 9        then LOG_ATTACK()
10             return TRUE
11  return FALSE
```

Figure 6: The DETECT_ATTACK algorithm

4. Implementation

We now present the architecture of FormatDefense and explain how we obtain the required debugging information, i.e., relevant information regarding the size of the local variable region. We also consider the optimization for the static format string.

4.1 The Architecture of FormatDefense

The architecture of FormatDefense is illustrated in Figure 7. Given a target program, we use GCC with the –g flag to compile the program and then use *dwarfdump* to extract the debugging information from the executable file. In addition, we implement FormatDefense as a shared library, `formatDefense.so`, and use the `LD_PRELOAD` mechanism to intercept format string functions. After a program has been loaded, FormatDefense uses the debugging information to create a reference table.

When a format string function is called, it is intercepted with a corresponding wrapper function. Subsequently, FormatDefense uses the DETECT_ATTACK algorithm to identify format string attacks. If a format string function is suspected of accessing an argument beyond line-Arg, the function LOG-ATTACK will be triggered.

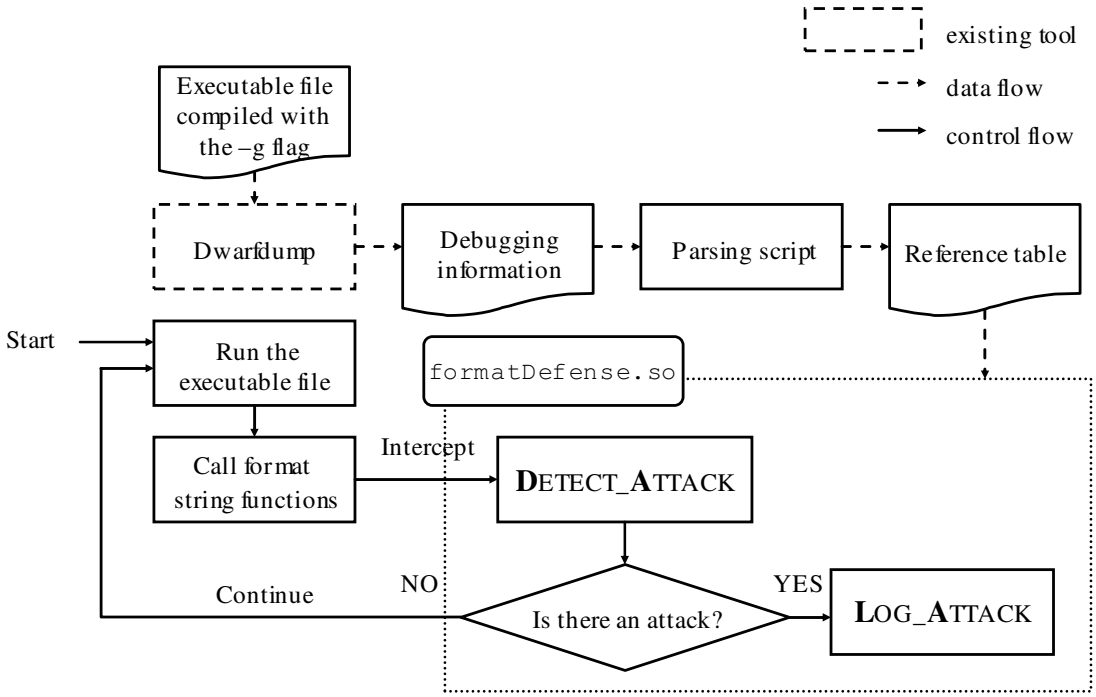
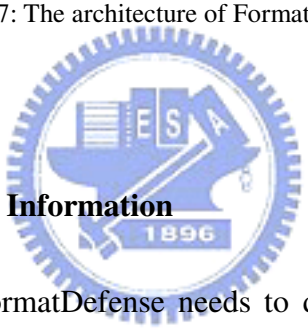


Figure 7: The architecture of FormatDefense



4.2 Obtaining the Debugging Information

As mentioned earlier, FormatDefense needs to determine the size of the local variable region of the function that holds the actual argument list. We use dwarfdump to dump the debugging information contained in the target program's executable file. The dwarfdump tool dumps the various elements of DWARF debugging information in ELF object files. DWARF is a debugging format to support source level debugging. It defines the format for the information generated by compilers, assemblers and linkage editors including the information content of the debugging entries and the way the debugging information is encoded and represented in an object file. For each program, the DWARF description is a tree structure with nodes which represent types,

variables, or functions. It is a brief representation because only the information which is required to characterize a viewpoint of a program is contained. In DWARF, the basic descriptive element is the Debugging Information Entry (DIE). A DIE has a tag which indicates what the DIE describes and a list of attributes which further describes the element.

Figure 8 shows a sample program and its corresponding DWARF description [21]. The attributes, *DW_AT_low_pc* and *DW_AT_high_pc*, indicate the range of code addresses for `main`. Variables `var1` and `var2` are at offset -20 and -16 relative to the stack frame of `main`, respectively. By subtracting the size of the return address and the saved frame pointer from the absolute value of the largest offset, we obtain the size of the local variable region. We then parse the output of dwarfdump and create a reference table, as shown in Table 3.

```
main.c
int main() {
    int var1, var2;
    printf( "%d%d", 1, 2 );
    return 0;
}
```

```
<1>< 297>    DW_TAG_subprogram
             ……
             DW_AT_name            main
             ……
             DW_AT_low_pc          0x8048354
             DW_AT_high_pc         0x804838f
             ……
<2>< 322>    DW_TAG_variable
             DW_AT_name            var1
             ……
             DW_AT_location        DW_OP_fbreg -20
<2>< 334>    DW_TAG_variable
             DW_AT_name            var2
             ……
             DW_AT_location        DW_OP_fbreg -16
```

Figure 8: The DWARF description for main.c

Table 3: The reference table for main.c

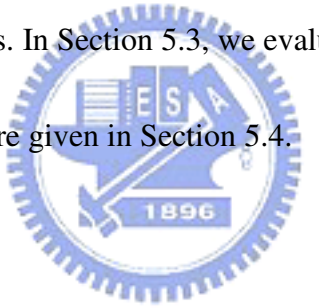
function name	low_pc	high_pc	size of local variable region (byte)
main	0x8048354	0x804838f	12

4.3 Skipping Read-only Format Strings

If a format string is static (i.e., it resides in the read-only section), an attacker cannot control it and no attack is possible. Therefore, we adopt the optimization process proposed in [11, 15] to avoid bound checking for static format strings. In FormatDefense, if FMTPTR points to the *.rodata* section, the intercepted function is immune to attacks, and we can call the real format string function directly without checking further.

5. Effectiveness and Performance Evaluation

In Section 5.1, we consider six possible scenarios of format string locations to evaluate the effectiveness of our line-Arg bound. There are three types of defense lines. We compare line-Arg, which is used by FormatDefense, with line-Sfp and line-Fmt. In the six scenarios, only the following three kinds of address relationship hold: (1) Arg < Fmt < Sfp for Scenarios 1 and 2, (2) Arg < Sfp < Fmt for Scenarios 3 and 4, and (3) Fmt < Arg < Sfp for Scenarios 5 and 6. In Section 5.2, we consider programs with known format string vulnerabilities to compare the attack space between the three defense lines. In Section 5.3, we evaluate the performance overhead. Some evaluation discussions are given in Section 5.4.



5.1 All Possible Scenarios

Scenario 1: The caller of the `printf`-like function contains the format string.

In this case, line-Fmt sits between line-Arg and line-Sfp. Figure 9 shows an example where the user provides "`%d%d%d`" as the input `user_fmt`. ARGPTR can access the local variable `var` when FMTPTR meets the third `%d` specifier. As shown in the figure, line-Fmt cannot protect `var` and line-Sfp cannot protect either `var` or `fmt`. Both variables open an attack space, hence an exploit is possible if an attacker controls these variables.

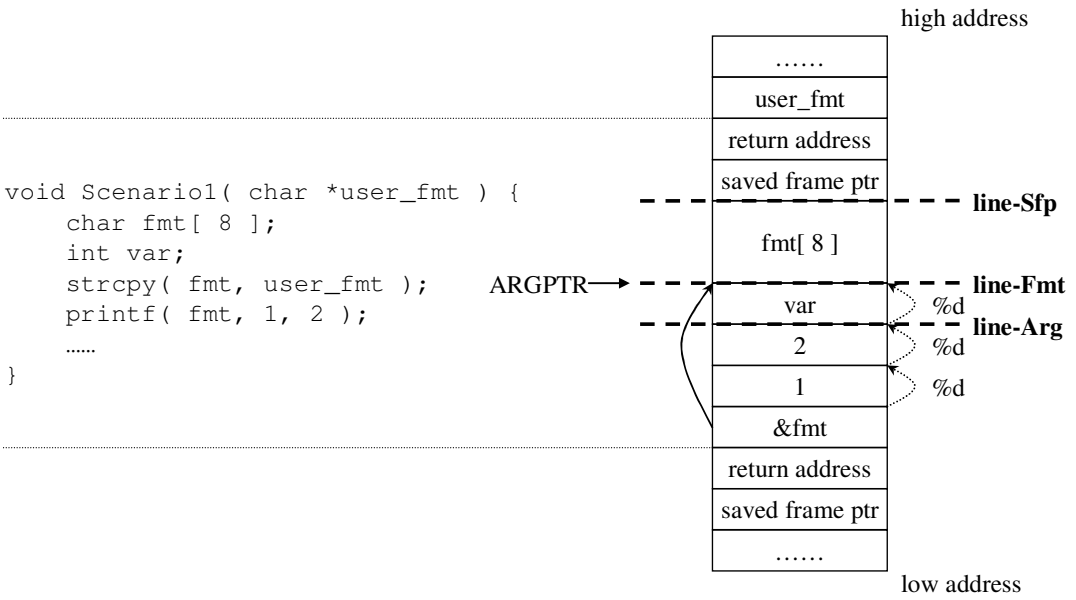


Figure 9: Arg < Fmt < Sfp for `printf`

Scenario 2: The format string and real argument list of the `vprintf`-like function reside in the same stack frame.

This case is similar to Scenario 1, but the function is `vprintf`-like. Figure 10 shows an example where the argument list for `vprintf` is composed of two integers: 1 and 2. If the pointer `user_fmt` is "`%d%d%d`", ARGPTR will access an argument outside the region of the argument list; that is, it will access the local variable `var`. The consequence is the same as that of Scenario 1.

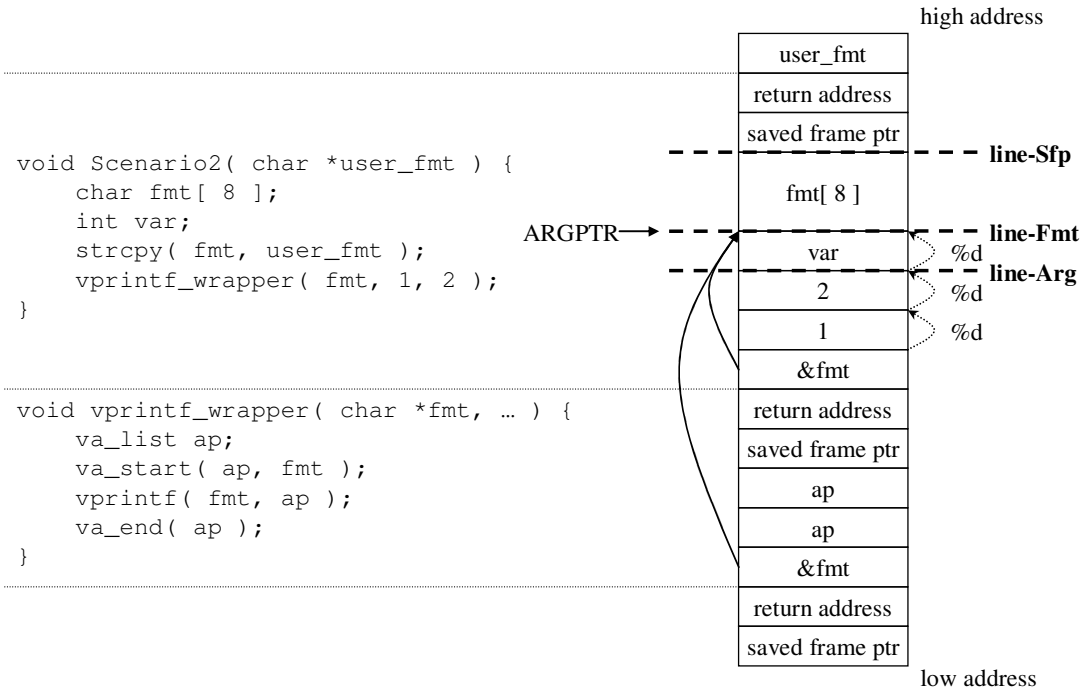


Figure 10: Arg < Fmt < Sfp for `vprintf`

Scenario 3: One of the ancestors of the `printf`-like function in the call chain contains the format string.

In this case, line-Sfp sits between line-Arg and line-Fmt. Figure 11 shows an example, where the content of `fmt` is "`%d%d%d`." ARGPTR can access the local variable `var` when FMTPTR is at the third `%d` specifier. The figure shows that line-Sfp cannot protect `var` and line-Fmt allows an even wider attack space.

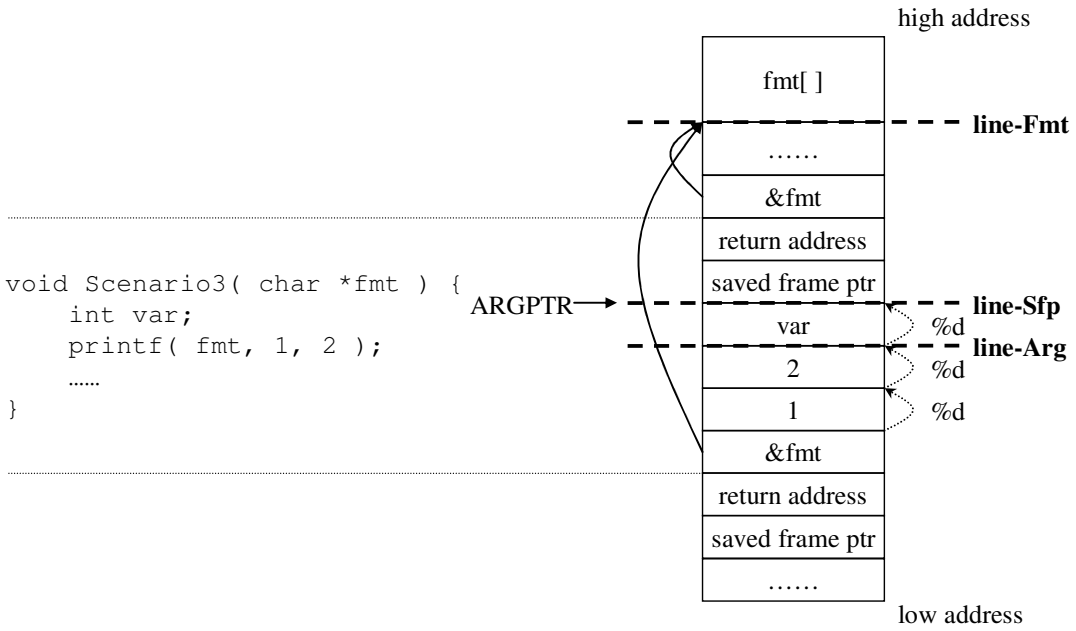


Figure 11: Arg < Sfp < Fmt for `printf`

Scenario 4: The format string and real argument list of the `vprintf`-like function reside in different stack frames.

As this case is similar to Scenario 3, we show a `vprintf`-like version of Scenario 3 in Figure 12. When the content of `fmt` is "`%d%d%d`", the last argument accessed by ARGPTR is the local variable `var`. The analysis follows that for Scenario 3.

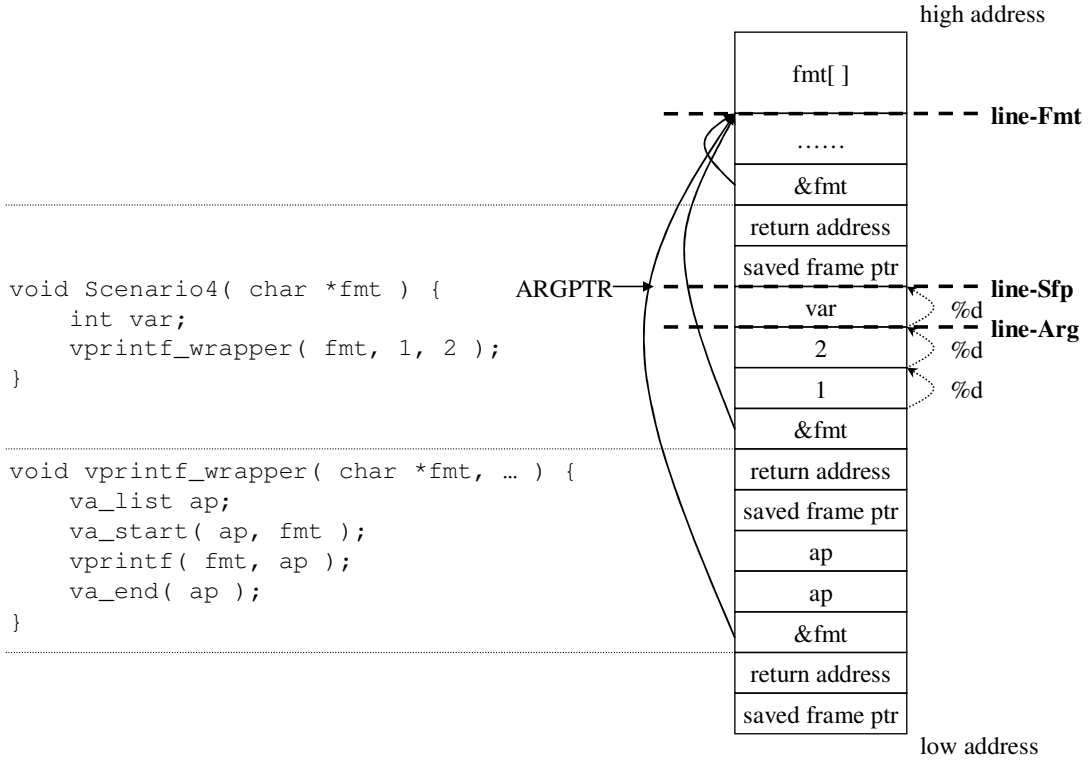


Figure 12: Arg < Sfp < Fmt for `vprintf`

Scenarios 5: The format string of the `printf` function is stored in the Block Started by Symbol (BSS) segment, the data segment, or the heap.

In this case, line-Arg sits between line-Sfp and line-Fmt. Figure 13 shows an example where the input `user_fmt`, "`%d%d%d`", is copied into the static array `fmt` located in the BSS segment. ARGPTR never reaches the format string itself because the BSS segment is below the stack segment and ARGPTR moves towards higher addresses in the memory. In this scenario, line-Fmt fails completely. Thus an exploit is possible if an attacker can control the variables in the space between `var` and the environment variables.

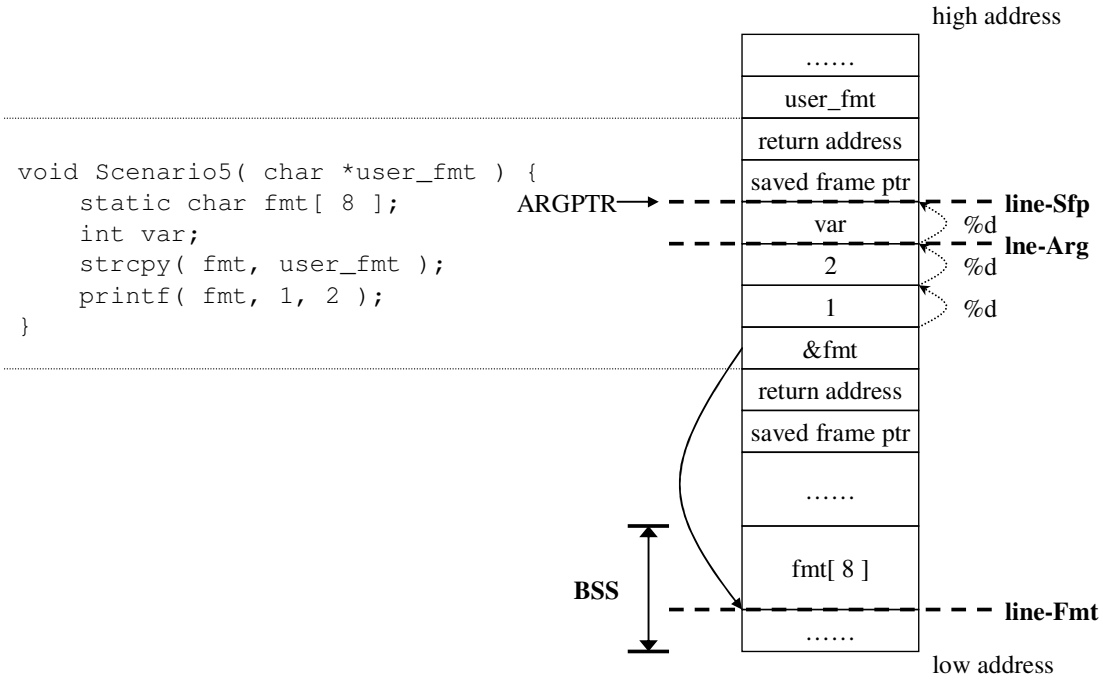


Figure 13: Fmt < Arg < Sfp for `printf`

Scenarios 6: The format string of the `vprintf` function is stored in the Block Started by Symbol (BSS) segment, the data segment, or the heap.

As this case is similar to Scenario 5, we show a `vprintf`-like version of Scenario 5 in Figure 14. ARGPTR also never reaches the format string itself, so line-Fmt fails completely. The consequence is the same as that of Scenario 5.

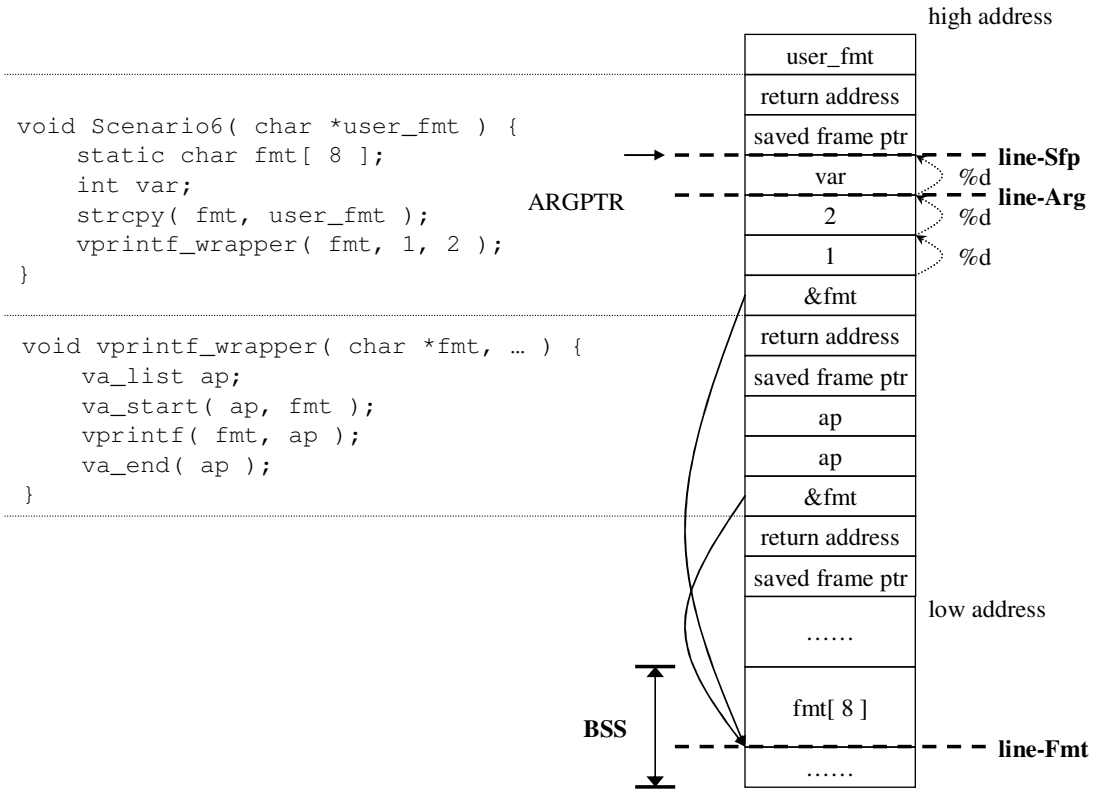


Figure 14: Fmt < Arg < Sfp for `vprintf`

Table 4 compares the different detection tools used in the above six scenarios. The detection level of each tool is rated from good to bad and denoted by A, B, or C. The tools are ranked in decreasing order; e.g., a tool rated A can detect all the attacks that can be detected by the tools rated C, but the latter cannot detect all the attacks detected by the tools rated A. Kimchi and Libsafe 2.0 use line-Sfp as the defense line and cannot protect the local variables between line-Arg and line-Sfp. The defense line line-Fmt provides better protection in Scenarios 1 and 2, but fails when the format string does not reside in the stack. The major weakness of FormatGuard is that it cannot handle `vprintf`-like functions. The only drawback of Lisbon is that it searches for the macro "`va_start(ap, fmt)`" as a heuristic to recognize the

function where the object `va_list` is created, hence it may generate false negatives for protecting `vprintf`-like functions.

Table 4: Comparisons of various detection tools in the six scenarios

	Tool					
Scenario	Kimchi	Libsafe	Line-Fmt	FormatGuard	Lisbon	FormatDefense
1	C	C	B	A	A	A
2	C	C	B	X	A*	A
3	B	B	C	A	A	A
4	B	B	C	X	A*	A
5	B	B	X	A	A	A
6	B	B	X	A	A	A

A-C: the level of detection; A*: depend on the precision of the heuristic; X: not detected

5.2 Effectiveness of FormatDefense's Protection against Known Format String Attacks

To evaluate FormatDefense, we use the following programs, which are all susceptible to known format string attacks: Splitvt-1.6.4 [22], Pfinger-0.7.5 [23], and Tcpflow-0.20 [24]. Splitvt-1.6.4 is a VT100 window splitter that is susceptible to a format string attack when a malicious format string is passed to `sprintf` via the "–rcfile" command line flag. Pfinger-0.7.5 is a daemon for the standard finger protocol. It contains a format string vulnerability that can be exploited by a malicious format string in a `.plan` file passed to `printf`. Tcpflow-0.20 is a network debugging tool that is susceptible to a format string attack when it opens an undefined device that triggers the error message passed to `vfprintf`. The vulnerabilities of

these programs belong to Scenario 5 and Scenario 6, in which relationship Fmt < Arg < Sfp holds. Figure 15 shows the vulnerable code segments in above programs.

Splitvt-1.6.4

```
char *startupfile = “%s/.splitvtrc";  //store in the data segment
char *rcfile_buf;

void splitvtrc() {
    ……
    sprintf( rcfile_buf, startupfile, home );
    puts( rcfile_buf );
    ……
}
```

Pfinger-0.7.5

```
void DoFinger1( char *hostname, char *query ) {
    static char buf[ 80*20 + 1 ];  //store in the BSS segment
    ……
    while( read( s, buf, 80*20 ) > 0 ) {
        buf[ 80*20 ] = ‘\0';
        printf( buf );
    }
}
```

Tcpflow-0.20

```
char error[ PCAP_ERRBUF_SIZE ];  //store in the BSS segment

void print_debug_message( char *fmt, va_list ap ) {
    ……
    vfprintf( stderr, fmt, ap );
    ……
}
```

Figure 15: The vulnerable code segments of Splitvt, Pfinger, and Tcpflow

FormatDefense succeeds in detecting attacks that try to exploit the vulnerability of each type of software. Furthermore, as shown in Table 5 and Table 6, line-Arg minimizes the possibility of format string attacks. The two tables detail the range between the end of the argument list and each type of defense line in Linux and BSD platforms, i.e., the attack space. Clearly, line-Arg yields the smallest attack space among the three defense lines. In addition, the attack space in BSD platform is smaller than the one in Linux platform because the ways of code generation across platforms

are different. The attack space of line-Arg in BSD platform is zero-byte in particular. It means that FormatDefense can find the exact defense line in BSD platform.

Table 5: The attack space in Linux platform

Attack Space (word)	Software		
	Splitvt-1.6.4	Pfinger-0.7.5	Tcpflow-0.20
Line-Arg	12	6	7
Line-Sfp	6484	10	20
Line-Fmt	8629	52	22

Table 6: The attack space in BSD platform

Attack Space (word)	Software		
	Splitvt-1.6.4	Pfinger-0.7.5	Tcpflow-0.20
Line-Arg	0	0	0
Line-Sfp	1089	4	8
Line-Fmt	1394	46	10

5.3 Performance Benchmark

5.3.1 Microbenchmarks

To evaluate the performance overhead of FormatDefense, we apply a series of microbenchmark programs with a loop involving one format string function call, as shown in Figure 16(a) for `printf`-like functions and Figure 16(b) for `vprintf`-like functions. The programs compiled with GCC 4.1.2 were run on a 1.86 GHz Intel Core 2 with 1.5GB of RAM. We ran each microbenchmark program twice in single-user mode, with and without FormatDefense. The performance overheads of the other tools were obtained from the literature. Table 7 shows the performance of

FormatGuard, White-listing, Lisbon, FormatDefense, and Optimized-FormatDefense on the six microbenchmarks. The overhead of the "`sprintf` with no specifiers" program is 0% for FormatDefense because the `sprintf` is compiled into a series of "`mov`" instructions for optimization. However, because FormatGuard and White-listing use a wrapper for `sprintf`, GCC cannot apply the above optimization. Lisbon performs poorly because it needs to make a system call to set the *debug registers*. The static-string optimization in FormatDefense outperforms that of each benchmark.

```
int main() {
    int i;
    char buf[ 32 ];
    for( i = 0; i < 10000000; i++ )
        sprintf( buf, "abcdef%d%d", 1, 2 );
    return 0;
}
```

(a)

```
int main() {
    vsprintf_wrapper( "abcdef%d%d", 1, 2 );
    return 0;
}
void vsprintf_wrapper( const char *fmt, … ) {
    va_list ap;
    va_start( ap, fmt );
    int i;
    char buf[ 32 ];
    for( i = 0; i < 10000000; i++ )
        vsprintf( buf, fmt, ap );
}
```

(b)

Figure 16: Microbenchmark programs

Table 7: Performance Overhead of Applying Different Detection Tools on the Microbenchmarks

Benchmark	FormatGuard	White-listing	Lisbon	FormatDefense	Optimized-FormatDefense
sprintf with no specifiers	7.5%	10.2%	217.7%	0%	0%
sprintf with 2 %d specifiers	20.9%	28.6%	67.9%	64.7%	0.63%
sprintf with 2 %n specifiers	38.1%	60.0%	142.3%	95%	1.3%
vsprintf with no specifiers	cannot handle	26.4%	223.4%	68.2%	3.6%
vsprintf with 2 %d specifiers	cannot handle	39.8%	63.2%	69.1%	1.2%
vsprintf with 2 %n specifiers	cannot handle	74.7%	154.7%	100%	2.4%

5.3.2 Macrobenchmarks

The machine employed for macrobenchmarks is the same as that used for microbenchmarks. The performance overhead for FormatDefense is measured without read-only string optimization. We use Man2html-1.6, Pfinger-0.7.5 and Splitvt-1.6.4 to evaluate the performance of FormatDefense. Man2html is a `printf`-intensive software that converts UNIX man page files to HTML web pages. We ran Man2html 79 times to translate a 596 KB man page file. For Pfinger, we fingered a user whose `.plan` file was 9784 KB. In addition, we tested Splitvt by executing the shell command "`ls -l`" in each split window 100 times. As shown in Table 8, the overhead of FormatDefense is so low that it is negligible, even without read-only string optimization.

Table 8: FormatDefense's overhead on the macrobenchmarks

Software	Latency Penalty
Man2html-1.6	0.91%
Pfinger-0.7.5	0.7%
Splitvt-1.6.4	0.6%

5.4 Evaluation Discussion

Stack alignment and the implications of sibling functions generate unused spaces in the stack for Linux platform. In this section, we show that such spaces are not critical to FormatDefense because it tries to protect the local variables behind the

argument list, and attackers find it hard to control unused spaces. Moreover, we discuss the attack space of line-Fmt in Fmt < Arg < Sfp relationship.

5.4.1 Stack Alignment

The stack layout of a program complied by the current GCC in Linux platform is shown in Figure 17. (Note that it is different from the layout in Figure 4.) GCC 4.1.2 enforces two kinds of stack alignments [25], thereby causing us to find an approximate line-Arg, not a precise one, as shown in Figure 17. The unused space between the local variable region of a function and the arguments of its callee results from the codes generated by GCC 4.1.2.

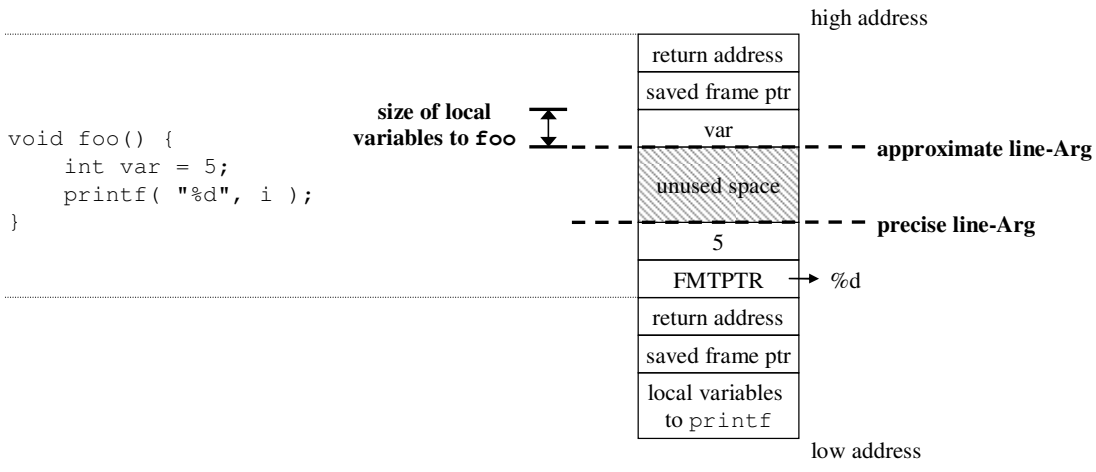


Figure 17: Stack alignment of the current GCC

In the first kind of stack alignment enforced by GCC, the stack is aligned at the program entry point, as exemplified by the assembly code for `main` shown in Figure 18. At <main+4>, the program adjusts the stack pointer to 16-alignment by default to

ensure that the stack at the program entry point, i.e., at `main`, is aligned. Under the second kind of stack alignment, the stack frame size of each non-leaf function is aligned, as exemplified by the assembly code for the function `foo` shown in Figure 19. At <foo+3>, the program allocates enough space for the local variables of `foo` and the arguments passed to `printf` to ensure that the stack frame size of the non-leaf function, i.e., the function `foo`, is aligned.

```
int main() {
    int a = 2;
    foo( a );
    return 0;
}
void foo( int b ) {
    char buf[ 4 ];
    printf( "%d", b );
}
```

```
Dump of assembler code for function main:
<main+0>:   lea   0x4(%esp),%ecx
<main+4>:   and   $0xfffffff0,%esp
<main+7>:   pushl 0xfffffffc(%ecx)
<main+10>:  push  %ebp
<main+11>:  mov   %esp,%ebp
                  ……
<main+17>:  movl  $0x2,(%esp)
<main+24>:  call  0x8048354 <foo>
                  ……
<main+42>:  ret
End of assembler dump.
```

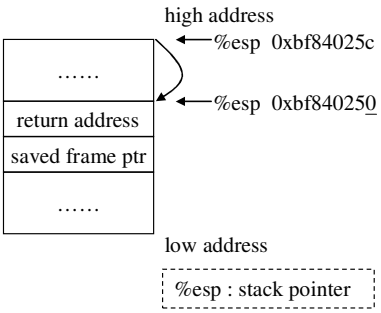


Figure 18: Assembly code for the function `main`

```
Dump of assembler code for function foo:
<foo+0>:    push  %ebp
<foo+1>:    mov   %esp,%ebp
<foo+3>:    sub   $0x18,%esp
<foo+6>:    mov   0x8(%ebp),%eax
<foo+9>:    mov   %eax,0x4(%esp)
<foo+13>:   movl  $0x804846c,(%esp)
<foo+20>:   call  0x8048290 <printf@plt>
<foo+25>:   leave
<foo+26>:   ret
End of assembler dump.
```

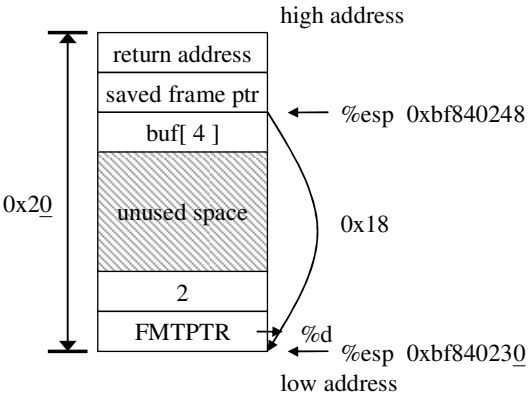


Figure 19: Assembly code for the function `foo`

However, the compiler flag "`-mpreferred-stack-boundary=num`" in GCC keeps the stack boundary aligned to a 2 raised to *num* byte boundary. In other

words, the stack frame size of non-leaf functions is aligned. The default is 4 (16-alignment), unless otherwise specified. Consequently, FormatDefense uses the "`-mpreferred-stack-boundary=2`" flag to avoid creating unused spaces. If there are no unused spaces, we can find the precise line-Arg, as shown in Figure 4.

5.4.2 Interaction between Sibling Functions

If the format string function has sibling functions in Linux platform, there may be unused spaces. For example, `main` calls two functions, `sibling_func` and `printf`; the former accepts more arguments than the latter, as shown in Figure 20. As noted in Section 5.4.1, a function should immediately allocate enough space for its local variables and the arguments of all the callees. Hence the amount of space is determined by the callee with the most arguments, i.e., the function `sibling_func` in this case. Therefore, `main` allocates the exact amount of space for its local variables and the arguments passed to `sibling_func`, as shown in Figure 20(a).

When `printf` is called after `sibling_func` returns, FormatDefense obtains the calculated line-Arg instead of the real line, as shown in Figure 20(b). Even so, the unused space is not a critical issue for FormatDefense unless an attacker can control the arguments passed to `sibling_func`. FormatDefense thus makes it more difficult for attackers to implement successful format string attacks.

```
int main() {
    int a;
    a = sibling_func( 1, 2, 3 );
    printf( "%d%d", a );
    return 0;
}
int sibling_func( int n1, int n2, int n3 ) {
    return n1 + n2 + n3;
}
```

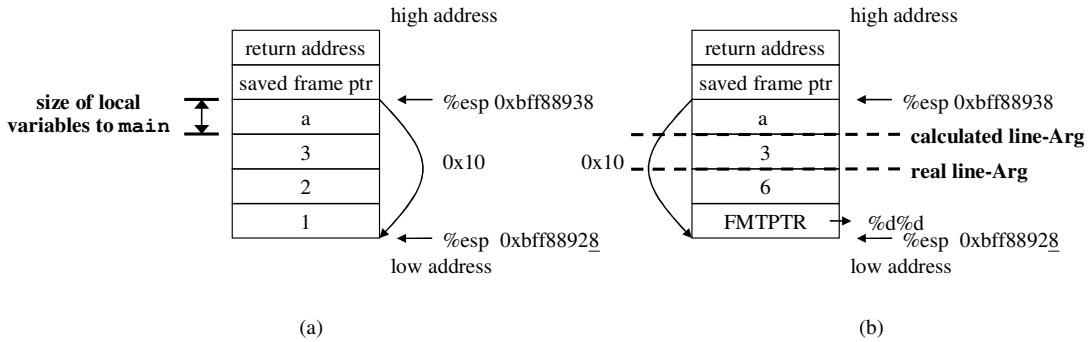


Figure 20: A format string function with sibling functions

5.4.3 Attack Space of Line-Fmt in Fmt < Arg < Sfp Relationship

In previous section, we define an attack space as the gap between the defense line and the end of the argument list. In addition, ARGPTR advances toward higher addresses in the stack. However, in Scenarios 5, 6 of Section 5.1, line-Fmt is under the stack, namely, under ARGPTR. In this situation, the attack space is defined specifically.

Figure 21 shows the detailed view of a program's stack layout [26]. There is a block of zero-filled padding whose size is determined randomly, so the locations of parameters of the function `main` and environment variables are not guessed easily in the stack. We consider that attackers are hard to control those data in this way. Therefore, we define the upper bound of the attack space in Scenarios 5, 6 of Section

5.1 as the program entry point as shown in Figure 22.

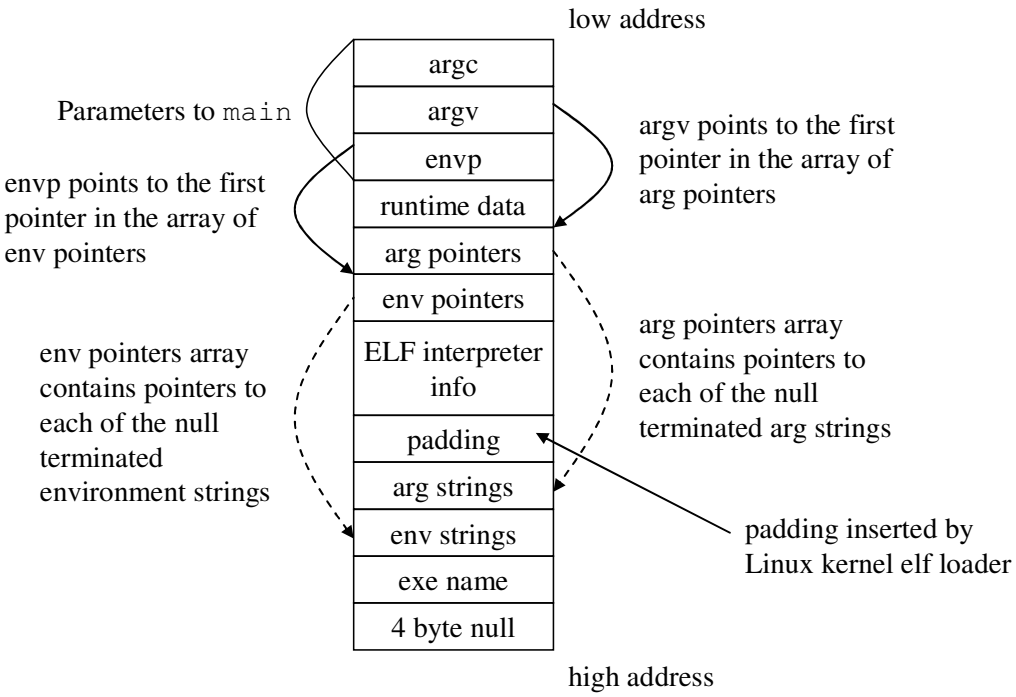


Figure 21: Detailed view of a program's stack layout

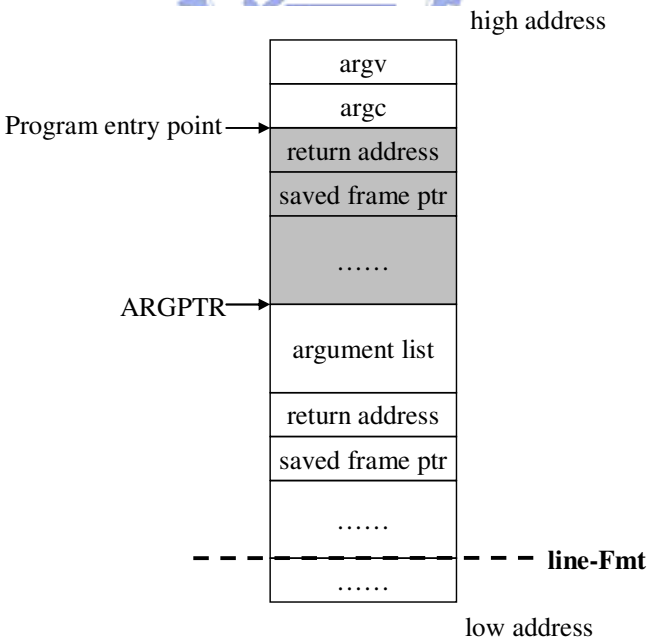


Figure 22: The attack space in Scenarios 5,6 of Section 5.1

6. Conclusion

In this thesis, we propose an approach called FormatDefense to prevent format string attacks at runtime. At first, FormatDefense can locate the stack frame that holds the argument list via tracking the stack. Then, FormatDefense can determine the access bound, line-Arg, for format string functions by analyzing the debugging symbols. If a format string function accesses arguments outside line-Arg, it is considered an attack. We evaluate FormatDefense on six possible scenarios and real programs vulnerable to known format string attacks to prove its effectiveness by comparing with other tools and measuring the attack space. In conclusion, FormatDefense is able to detect all format string exploits and provides the most accurate approach of avoiding format string attacks. Therefore, FormatDefense becomes a practical tool for preventing format string attacks due to high accuracy, low overhead and easy deployment.

References

1. Lhee KS, Chapin SJ. Buffer overflow and format string overflow vulnerabilities. *Software–Practice and Experience*, 2003; **33** (5): 423-460.
2. Common Vulnerabilities and Exposures. Search Results for format. http://www.cve.mitre.org/cgi-bin/cvekey.cgi?keyword=format+string, January 28, 2008. [28 January 2008]
3. Christey S, Martin RA. Vulnerability Type Distributions in CVE. http://cwe.mitre.org/documents/vuln-trends/index.html, May 22, 2007. [3 Mar 2008]
4. Shankar U, Talwar K, Foster JS, Wagner D. Detecting Format String Vulnerabilities with Type Qualifiers. *Proceedings of the 10th conference on USENIX Security Symposium*, 2001; 201–218.
5. Chen K, Wagner D. Large-Scale Analysis of Format String Vulnerabilities in Debian Linux. *Proceedings of the 2007 workshop on Programming Languages and Analysis for Security*, 2007; 75-84.
6. Newsome J, Song D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.
7. Cheng W, Zhao Q, Yu B, Hiroshige S. TaintTrace: Efficient Flow Tracing with

Dynamic Binary Rewriting. *Proceedings of 11th IEEE Symposium on Computers and Communications*, 2006; 749-754.

8. Xu W, Bhatkar S, Sekar R. Taint-enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. *Proceedings of the 15th conference on USENIX Security Symposium*, 2006; 121–136.
9. Cowan C, Barringer M, Beattie S, Kroah-Hartman G, Frantzen M, Lokier J. FormatGuard: Automatic Protection from printf Format String Vulnerabilities. *Proceedings of the 10th conference on USENIX Security Symposium*, 2001.
10. Baratloo A, Tsai T, Singh N. Libsafe: Protecting Critical Elements of Stacks. December, 1999.
11. You JH, Seo SC, Kim YD, Choi JY, Lee SJ, Kim BK. Kimchi: A Binary Rewriting Defense against Format String Attacks. *The 6th International Workshop on Information Security Applications*, Jeju Island, Korea, August 22-24, 2005; 179-193.
12. DeKok A. PScan: A Limited Problem Scanner for C Source Files. 2000.
13. Rao DTVR. Detection of Bugs by Compiler Optimizer Using Macro Expansion of Functions. *Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007)*, 2007; 855-862.

14. Robbins TJ. Libformat–Protection against Format String Attacks. 2001.

15. Ringenburg MF, Grossman D. Preventing Format-String Attacks via Automatic and Efficient Dynamic Checking. *Proceedings of the 12th ACM conference on Computer and Communications Security*, 2005; 354-363.

16. Lin Z, Xia N, Li G, Mao B, Xie L. Transparent Run-Time Prevention of Format-String Attacks Via Dynamic Taint and Flexible Validation. *Proceedings of the 9th International Conference*, Samos Island, Greece, 2006. (*Lecture Notes in Computer Science*, vol. 4176), 2006; 17-31.

17. Chen S, Xu J, Sezer EC, Gauriar P, Iyer RK. Non-control-data attacks are realistic threats. *Proceedings of the 14th conference on USENIX Security Symposium*, 2005; 177–192.

18. Tsai T, Singh N. Libsafe 2.0: Detection of Format String Vulnerability Exploits. White Paper, Avaya Labs, February, 2001.

19. Ganapathy V, Seshia SA, Jha S, Reps TW, Bryant RE. Automatic Discovery of API-Level Exploits. *Proceedings of the 27th International Conference on Software Engineering*, 2005; 312-321.

20. Li W, Chiueh T. Automated Format String Attack Prevention for Win32/X86 Binaries. *Proceedings of the 23th Computer Security Applications Conference(ACSAC'07),* 2007; 398-409.

21. Eager M, Eager Consulting. Introduction to the DWARF Debugging Format. February, 2007.

22. SecuriTeam. Multiple vulnerabilities in splitvt (Exploit Code). http://www.securiteam.com/unixfocus/5GP0J2A35C.html, 15 Jan. 2001. [3 Mar 2008]

23. SecuriTeam. PFinger Format String Vulnerability. http://www.securiteam.com/unixfocus/6K00N1P3FQ.html, 27 Dec. 2001. [3 Mar 2008]

24. SecuriTeam. tcpflow Format String Vulnerability. http://www.securiteam.com/unixfocus/5FP0H00AUO.html, 10 Aug. 2003. [3 Mar 2008]

25. Ye J. A proposal to align GCC stack – update. http://gcc.gnu.org/ml/gcc/2007-12/msg00567.html, 19 Dec 2007. [3 Mar 2008]

26. Shon H, Allen H, Chris E, Jonathan N, Michael L. Gray Hat Hacking; 382-384