Transparent Run-Time Prevention of Format-String Attacks Via Dynamic Taint and Flexible Validation

Zhiqiang Lin, Nai Xia, Guole Li, Bing Mao, and Li Xie

State Key Laboratory for Novel Software Technology
Nanjing University, 210093, Nanjing, China
{linzq, xianai}@dislab.nju.edu.cn

Abstract. Format-string attack is one of the few truly threats to software security. Many previous methods for addressing this problem rely on program source code analysis or special recompilation, and hence exhibit limitations when applied to protect the source code unavailable software. In this paper, we present a transparent run-time approach to the defense against format-string attacks via dynamic taint and flexible validation. By leveraging library interposition and ELF binary analysis, we taint all the untrusted user-supplied data as well as their propagations during program execution, and add a security validation layer to the `printf`-family functions in C Standard Library in order to enforce a flexible policy to detect the format string attack on the basis of whether the format string has been tainted and contains dangerous format specifiers. Compared with other existing methods, our approach offers several benefits. It does not require the knowledge of the application or any modification to the program source code, and can therefore also be used with legacy applications. Moreover, as shown in our experiment, it is highly effective against the most types of format-string attacks and incurs low performance overhead.

1 Introduction

Because of some innate features of C programming language such as lack of memory safety and function argument checking, ever since it became the mainstream programming languages of choice, there have been problems with the programs produced using it. Format string vulnerability, discovered about six years ago [1], is a case of such problems. It applies to all format-string functions in C Standard Library and still exists in many software (e.g., a recent last-3-months search in the NIST National Vulnerability Database had returned 28 matching records of this vulnerability [2]).

Due to the ability to write anything anywhere [3,4], attacks exploiting format-string vulnerabilities are extremely dangerous: they can lead to the program denial of service (e.g., to crash the program by using multiple instances of `%s`-directive), or can read sensitive data from nearly any readable location in the process memory (e.g., information leakage attack by using some `%x`-directives), or can write arbitrary integers to the area the attacker desires to with carefully crafting the format-string (e.g., the most dangerous `%n`-directive attack). Like buffer overflows, format string attacks are well-recognized as one of the few severe and truly threats to software security [5,15].

S.K. Katsikas et al. (Eds.): ISC 2006, LNCS 4176, pp. 17–31, 2006.

Many defensive methods against format-string attacks have been investigated in the past several years, including static checking (e.g., [22]), compiler extensions and run-time guarding (e.g., [15,16]), safe library functions (e.g., [13,14]), execution monitoring (e.g., [31]), and so forth. As discussed in Section 6 in details, these approaches are all valuable and they can catch real attacks. However, some of them such as FormatGuard [16], TypeQualifiers [17] and White-listing [15], require access to program source code for special analysis or recompilation, and hence exhibit limitations when applied to protect the source code unavailable software; and some of them although do not rely on program source code and appear almost transparent, they either tend to restrict code for the protection (e.g., preventing `%n`-directive in non-static format string [13]), or just provide small scope checking (e.g., only detecting malicious write of `%n`-directive associated with the return address or frame pointer [14]).

In this paper, we present an improved *dynamic* and *transparent* approach to the detection and prevention of format-string attacks. Specifically, we employ library interposition technique to intercept the `printf`-family functions in C Standard Library, i.e., `glibc` (we consider Linux platform in this paper), to do a security validation against the format string; intercept the I/O as well as string related functions to taint all the untrusted data originating from untrusted sources and their propagations during program execution. In order to get a good tradeoff between false positive and false negative, we provide two security policies, a default policy and a fine-grained policy. In our default policy, we validate the format string on the basis of whether it has been tainted and contains dangerous format specifiers. If so, a format-string attack is detected and we either signal an input/output error or kill the program to prevent it. With our fine-grained policy, we validate not only the tainted format string but also the untainted non-static one. For these untainted non-static, we check the `%n` corresponding argument whether or not points to those security critical areas such as return address, GOT (Global Offset Table) and DTORs (Destructor function table) [18]. We have developed a practical tool, FASTV (FormAt String Taint and Validation), to demonstrate this idea.

Compared with other existing methods, our approach offers several advantages:

- *Practical in application.* Our approach operates on a normally compiled binary program, and appears transparent to the protected software. This makes it practical to be used for a wide variety of programs, including proprietary and commodity programs whose source code is unavailable.
- *Effective against "real-world" attacks.* We collected 6 notable format-string attacks published at securityfocus.com [1,7,8,9,10,11]. Our approach successfully prevented all of them.
- *Easy to use.* For protections, users only need to set the environment variable and restart the program. Moreover, it can be simply set for the protection of either specific program or all processes on the machine.
- *Low run-time overheads.* As the experiment indicated, our approach only imposes about 10% run-time overhead on the protected software.

Our work makes the following contributions. We propose a novel dynamical taint and flexible validation approach to the detection and prevention of format-string attacks. In general, it is a *practical* and *efficient* solution in defending against these attacks.

Besides, we have implemented the prototype, and made an empirical evaluation showing the feasibility of our approach. In addition, perhaps more importantly, we believe our approach is also applicable for the prevention of other attacks such as SQL injection [6].

The rest of this paper is organized as follows. Section 2 presents a technical description of our approach. Section 3 describes the design and implementation of our prototype. Section 4 provides the experimental results on the effectiveness and performance evaluation. The security analysis and limitations are discussed in Section 5, followed by the related work in Section 6. Finally, Section 7 concludes the paper and outlines future work.

2 Approach Description

Since the root cause of format-string vulnerability lies in the format string, which is an argument to the `printf`-family functions, trusting the user-supplied input [3,4,14,15,16,17], the format-string argument (essentially, it is a string pointer) becomes our focus. If we can ensure the format-string argument is trustworthy or contains no dangerous format specifier when it is untrustworthy, we could hence prevent the format-string attacks. This is the key idea of our approach.

Upon the observation, we find out that the format-string argument passed to `printf`-family functions often falls into three categories:

I. *Format-string argument pointing to a static string.*
II. *Format-string argument pointing to a program dynamically generated string.*
III. *Format-string argument pointing to a user-supplied input string.*

For the static string to act as a format-string argument, since it is constant and attackers cannot modify such strings (we do not cover static binary patching attack before program running in this paper), it is trustworthy and secure. We can distinguish it by ELF [18] binary analysis from the other two kinds, because static string resides in the program read-only data area and its address space is different from other program variables.

The second kind of format-string argument is internally maintained by the program itself, and in most cases is trustworthy. However, if attackers can influence the dynamically generated string (e.g., by buffer overflow attacks) or programmers carelessly deal with these data (e.g., forgetting to pass the corresponding argument), then it can also become untrustworthy. Therefore, we need to validate the format-string argument if tough checking required. Yet, we should mark this kind of format-string argument as trustworthy if no buffer overflow like attack occurs, because the bugs caused by carelessness should be eliminated before code ships.

The last kind of format-string argument is the common form of format-string vulnerability and undoubtedly the most dangerous. Our security validation mainly aims to find out this kind of format-string argument, which comes from user-supplied input and contains dangerous `%n`-directive (we currently focus on this specifier). It is an important part in our approach of how to identify and taint the untrusted user supplied data.

Here, suppose we have tainted all the untrusted data (its detail explanation is provided in the next section).

To detect the format-string attacks, we add a security validation layer to those `printf`-family functions in `glibc`. The security validation firstly determines whether the format string is static, since static format string is much more frequently used than that of other two kinds. (1) If the format string is static, we believe it trustworthy, and the function continues its original functionality (either to call the original function or execute code that is functionally equivalent). (2) Otherwise, the format string would be either dynamic generated or user-supplied (i.e., the tainted), and next we distinguish them based on whether it is tainted. (2.1) If the format string is tainted, then it is untrustworthy and we parse it to check whether it contains dangerous format specifiers. If it does, we report the format-string attack detected, and set the running `printf`-family function error; otherwise the function also continues its original functionality. (2.2) If the format string is untainted (this is the case for dynamically generated string), as stated it may be untrustworthy, and then if tough checking required we need to check it too. This is why we provide flexible policy. The default policy does not check it. In our fine-grained policy, the checking operation is to determine whether the `%n` corresponding argument points to return address, frame pointer, GOT (containing addresses of shared library functions) and DTORS (containing address of special destructor functions), because these areas are easily exploited as the attack target [4].

We employ ELF binary analysis and library interposition technique to achieve our goals. ELF binary analysis is used to identify the read-only static string and those attacker's target address, such as GOT and DTORS. Library interposition enables us to intercept those I/O and string propagation related functions in C Standard Library, so as to taint the user-supplied input. Another reason for using library interposition is that this technique does not require access to program source code, which makes our approach a more wider application.

3 Design and Implementation

We have developed a prototype, FASTV, to demonstrate our approach. The internal architecture of FASTV is illustrated in Figure 1. As shown in this figure, its core components are the dynamic taint of user-supplied input and flexible validation of format-string argument. We describe in detail these two parts design and implementation in this section, and additionally present the reactions when detected the format-string attacks. We provide the techniques to taint the user-supplied input in Section 3.1, and discuss how to validate the format-string argument according to different security policy in Section 3.2. The security reaction is provided in Section 3.3.

3.1 Dynamic Tainting Untrusted Data

Previous work [31] has suggested the use of taint analysis to track the input data that may lead to malicious attacks. However, their approach makes program run in an emulation environment and adds instrumentation to every data movement or arithmetic instruction, thereby imposing significant runtime overheads. In contrast, based on the

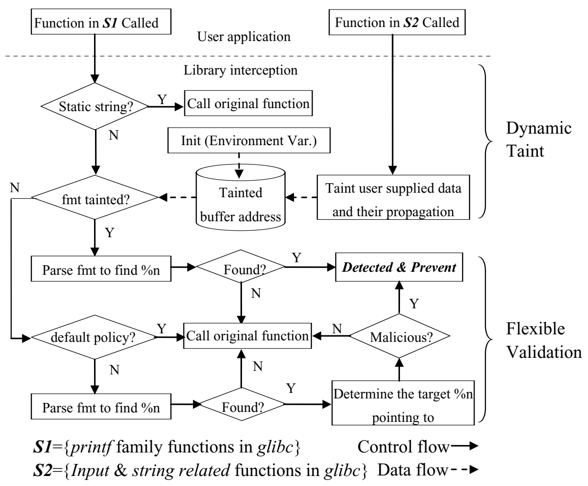


Fig. 1. Internal architecture of FASTV

observation that format string attack is usually caused by user-supplied input and these data are often string related, we could hence just intercept those *I/O* and *string* related functions in `glibc` instead of using hardware or software emulation based approach to track and taint the untrusted data.

Run-Time Representation of Tainted Data. In our approach, the taint operation is just to track the starting address and size information of those untrusted data in memory. We use a red-black tree [19] to store the tainted address and size in order to promote the performance. By associating taint operation with the starting address and size rather than other information, we can make our approach space cost little, and ensure the correctness of tainting in the presence of pointer aliasing. Why we do not store the content of every tainted data, the reason is when parsing the string content, we can get it from the tainted address.

Tainting Original Untrusted Data. For these directly user-supplied data, whenever the *I/O* related `glibc` functions are called, we intercept them and taint these input by inserting their buffer's starting address and size into our red-black tree. Here we should ensure that only one copy of a given buffer address exists in the tree. Thus, if the buffer has been tainted (by first searching the red-black to determine), we would not taint it again.

Note for the environment variable (e.g., user-supplied command-line data such as `argv`), we also need to taint them by inserting their associated address and size into the tree if tough checking required, because these data are also untrustworthy. In our current implementation, we have tainted these environment variables when loading FASTV by using the external variable `environ` to locate their address.

Tainting the Propagation of Original Untrusted Data. In our approach, tainting the propagation of original untrusted data is achieved by intercepting those *string* related `glibc` functions. Once these functions are called, we check firstly whether their corresponding original string (source string) is tainted. We determine this string is tainted on the basis of whether it exists in the red-black tree (its address equals to the node's address) or it belongs to the scope of certain node (we use the node's size to determine), for example, if address p is tainted and its length is 10, then "$p + 5$" is also implicitly tainted since it lies in the scope of p.

After that, (1) if we find the source string has been tainted, then we taint the destination string. For the destination string, they may have been tainted previously, so we need to search the tree to determine whether it has been tainted. If it has not, we insert the destination starting address and size into our tree, and next the function continues. (2) If the source string is untainted, then we also need to ensure the consistency of the destination string untainted, and subsequently if it has been tainted before this time we need to delete it from or update the node's size in our red-black tree. We believe this consistency operation is reasonable since every string related operation is been guarded, and the most recent string to format functions would be the corresponding most recent modified.

3.2 Format-String Validation

The format-string validation is used to detect the format-string attacks. As described in Section 2, not every format-string argument is necessary for the validation, only those tainted data or when tough checking needed the dynamically generated string is included for. Thus, we provide a default policy and a fine-grained policy to handle the validation flexibly.

Default Policy. Our default policy is primarily to catch those user-supplied format string, which contains dangerous format specifiers. Specifically, if the tainted data contains `%n`-directive and is used as a format-string argument, then the format-string attack is detected.

The other remained two kinds of format string are not addressed in our default policy, and we regard them as trustworthy (this is reasonable as we have stated in Section 2). However, it might lead to false negatives if attackers gain control over the dynamically generated string. So in order to complement the default policy, we provide the other policy, fine-grained policy, to handle the dynamically generated format-string argument (no need to static string since it is secure).

Fine-grained Policy. The fine-grained policy aims to determine wether the `%n`-directive corresponding argument is secure when the format string dynamically generated. If this argument points to a program return address, frame pointer, entry of GOT or DTORS, then it is insecure and the attack is detected. Otherwise, the function continues.

We construct a reference table, which contains the base address and whole size information of program GOT and DTORS, when loading FASTV. We achieve ELF binary analysis to find out these security critical address as well as their associated size to create this table. As for the return address and frame pointer, these data are dynamically changed and we cannot determine them by ELF binary analysis. Fortunately, Tsai et al. have proposed a solution to address this problem in Libsafe [14], and we adapt their approach here to locate the return address and frame pointer.

3.3 Reaction to Format-String Attacks

Many approaches when detected format-string attacks usually kill the running process. However, when attacks occur repeatedly, which is a common scenario with automated attacks, these protection mechanisms would lead to repeated restarts of the victim application and render its service unavailable. Thus, unlike their approaches we present a flexible mechanism for the preventions.

In our scheme, (1) if the format-string attacks are detected during the validation, then we report the format-string attack detected to `syslogd`, and set the `glibc` global variable `errno=EIO` to indicate this is an input/ourput error and let the program itself handle this problem. We believe for many server applications, they will take appropriate steps to deal with this input/output error. (2) If the output error is fatal (e.g., an intermediate output for after calculation) and the program itself ignores that, then here we also abort the running process in such a way as other approaches currently. We are planning to look for some alternative approaches (e.g., attack repair technique [20]) to remedy this (our aim is not to abort the running process).

4 Evaluation

We conducted a serials of experiments to evaluate the effectiveness and performance of our approach. In order to compare our results with others, we chose Libsafe in that we both adapt library interposition, and White-listing, which is the most recent work, to do the evaluation. All the experiments were carried out on two 2.4G Pentium processors with 1G RAM running Linux kernel 2.6.3. The tested programs were compiled by `gcc` 3.2.3 with default option, and linked with `glibc` 2.3.2.

4.1 Vulnerability Prevention

In our evaluation, we focused on *real-world* format-string attacks and selected six of such programs. The vulnerability of these programs and our security test against them are described below.

- `wu-ftpd`. The `wu-ftpd` 2.6.0 and earlier is vulnerable to a serious remote attack in the "SITE EXEC" implementation, in which user-supplied data can be used as a format-string argument [1]. For the security test of this program, we exploited the return address as the %`n` target.
- `rpc.statd`. The `rpc.statd` program (for the version of `nfs-utils` before 0.1.9.1), passes user-supplied data to the `syslog` function as a format string. Thus, attackers can inject malicious code to be executed with the privileges of the `rpc.statd` process [7]. In our test, we tried to overwrite GOT entries as the %`n` corresponding pointer.

– `splitvt`. The `splitvt` before 1.6.5 program exists a vulnerability in the command line with `-rcfile`, which is not properly handled as a format-string argument [8]. Our attack test was targeted return address.
– `rwhoisd`. The `rwhoisd` 1.5 server contains a remotely exploitable format-string vulnerability, which allows attackers to execute arbitrary code on affected hosts by supplying malicious format specifiers as the argument to the `-soa` directive [9]. Again, our attack test was to patch the return address.
– `pfinger`. A format-string vulnerability exists in `pfinger` 0.7.5 through 0.7.7, which allows remote attackers to execute arbitrary command via format-string specifiers in a .plan file [10]. For this program test, we also tried to overwrite the return address.
– `tcpflow`. The `tcpflow` 0.2.0 program contains an exploitable format-string vulnerability during the opening of a device with the command-line argument. Thus, local users can gain an unauthorized root shell by this vulnerability [11]. We made our security attack target on GOT entries to test this program.

The results of our effectiveness evaluation are presented in Table 1. As shown in this table, our approach, not only the default policy (DP) but also the fine-grained policy (FP), successfully prevented all the format-string attacks listed above. This is expected because all the security test was used the user-supplied data which comes from local or network to launch the format-string attack.

For the White-listing approach, though it also reliably fixed all the vulnerabilities, it may lead to denial of service attack for some cases (though in our protections `pfinger` and `tcp-flow` aborted the running process, we should note this is the behaivor of program itself). The Libsafe approach, also as expected, missed the attack which does not target return address or frame pointer, and only caught 4 out of the 6 attacks.

Table 1. Results of effectiveness evaluation

CVE#	Program	Libsafe	White-listing	FASTV(DP)	FASTV(FP)
CVE-2000-0573	`wu-ftpd`	D & A	D & A	P & C	P & C
CVE-2000-0666	`rpc.statd`	M	D & A	P & C	P & C
CVE-2001-0111	`splitvt`	D & A	D & A	P & C	P & C
CVE-2001-0913	`rwhoisd`	D & A	D & A	P & C	P & C
CVE-2001-1215	`pfinger`	D & A	D & A	P & A	P & A
CAN-2003-0671	`tcp-flow`	M	D & A	P & A	P & A

D: Detected, A: Aborted, P: Prevented, C: Continued, M: Missed

4.2 Performance Overhead

In order to test the performance overhead of our approach, we first did the micro-benchmark test to measure the overhead at the function call level, and then measured the overall performance at the application level by running a typical `printf`-intensive application `man2html` (to test the overhead of *print* and *string* related functions), and a network program `tcpdump` (to test the overhead of *input* related functions, such as `read`, `recv`). All the tested programs were run multiple times with the highest priority in single-user-mode except for `tcpdump` which run in the network-mode due to its network requirement.

Micro Benchmarks. To determine the overhead of per `printf`-style function, we ran a serial of simple programs consisting of a single loop containing one single `sprintf` call, with a varied number of format string length. We choose `sprintf` in that we can use this function to test the performance of both format-string validation (parse and check the format string) and related string taint (taint the relevant destination string if the corresponding printed string is tainted), and its performance is greater than that of other `printf`-family functions. In addition, we choose `strcpy` to test the micro-benchmark of string related functions since in our approach we also wrapper them.

With static format string which contains no format specifiers, our approach as well as Libsafe added almost no performance overhead (the performance added rate is zero). As for White-listing, it had a different performance added rate, which is greater than ours and Libsafe's. To be more specific, when the format-string length is not too long in our test, e.g., less than 100, White-listing only incurred little performance overhead; and when format-string length is added, e.g., to 1k, it added the performance overhead of 3μs (about 75%).

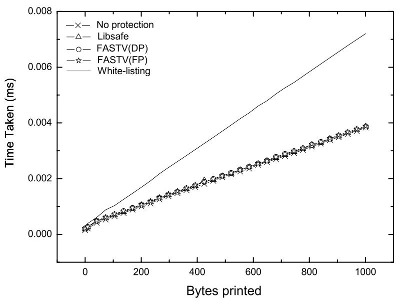
For dynamically generated string which contains two %`n` directives, our approach with DP did not add any overhead, which is similar to Libsafe. This is expected, because our default policy does not check these program dynamically generated string. As for FP of our approach, which will check all the dynamically generated string, the performance added rate was less than White-listing's; and in the worst case of our test, it added 2.4μs (about 60%), and White-listing added 3.5μs (about 90%).

With user-supplied different length format string (it is just a performance test here, though it appears insecure) which contains no format specifier, our overhead for both DP and FP was similar to the result of dynamically generated string with FP: the performance added rate was less than White-listing's.

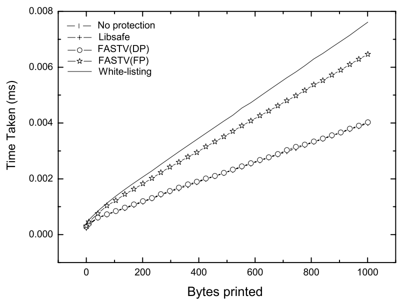
We also tested `vsprintf` by replacing the printing loop with `vsprintf`, and correspondingly modifying its relevant argument, to test the performance of per `vprintf`-style function call. We observed a similar performance overhead to `sprintf` function for the above three test cases, respectively.

For the micro-benchmark of `strcpy` function, in our test, except for Libsafe which improves the performance by replacing `strcpy` with `memcpy` [14] (this is based on the fact that copying with `memcpy` is 6 to 8 times faster than that of `strcpy` for large buffers [21]), our approach as well as White-listing almost did not add any performance overhead.

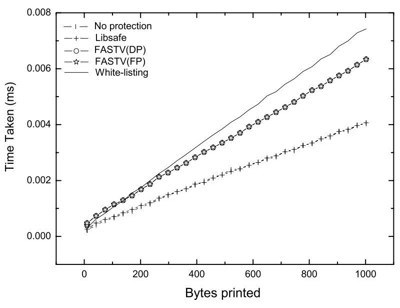
All the results for our micro benchmarks are depicted in Figures from 2(a) to 2(d). Some of these overheads may seem relatively high, but we stress that these are micro benchmarks and not realistic programs. And as we show below for real-world applications, our approach only incurs a little performance overhead.



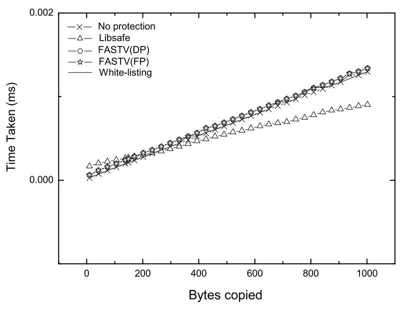
(a) Sprintf micro-benchmark with static format string containing no specifiers



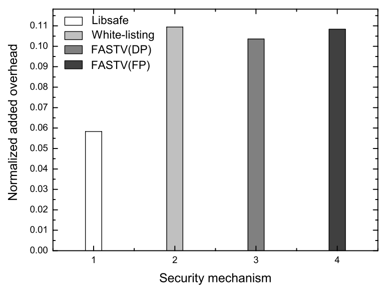
(b) Sprintf micro-benchmark with dynamically generated format string containing 2 %n directives



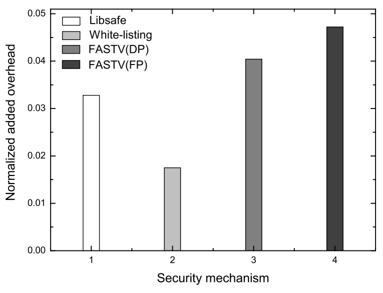
(c) Sprintf micro-benchmark with user-supplied format string



(d) Strcpy micro-benchmark with user-supplied string



(e) Man2html macro-benchmark



(f) Tcpdump macro-benchmark

Fig. 2. Results of benchmark test

Macro Benchmarks. We used `man2html` program to evaluate the macro-benchmark. Our test was to batch translate 1k man pages which is 129.6k bytes each, via `man2html-1.6`. The result for the macro benchmark of this program is presented in

Figure 2(e). As shown in the Figure, our approach incurred the performance overheadof 10.36% with DP and 10.84% with FP, which is a little less than that of White-listing with the overhead of 10.94%. For the Libsafe, the performance overhead was 5.83%.

We also tested our approach against a network program `tcpdump-3.9.4`. We ran this program by capturing 1k network packets in a high-speed transferring LAN. Our approach added the overhead of 4.04% with DP and 4.72% with FP. Libsafe added 3.28%, and White-listing added 1.75%. Figure 2(f) summarizes the result of this program.

5 Discussion

In this section, we discuss the false positive and false negative of our approach with different security policy, and its limitations when applied to the software protections.

False Positive and False Negative. As stated, our default policy guards all the dangerous tainted data partly or completely acting as the format string, there would be few false negatives (false negatives may exist in the un-caught string propagation if the application does not use the string related `glibc` functions) in this policy when handling user-supplied data. From theoretic analysis, we could find our default policy may have false positives, but from practice we could say almost *no* false positive. We have examined a large number of real-world applications and found none of them needs user-supplied %`n`-directive except for attacks. We believe this case, i.e., requiring users to input the %`n`-directive, does not exist in normal released software. So we believe it is not a common case and does not deserve our attention.

While our default policy is adequate for the format-string attack prevention in most circumstances, those untainted dynamically generated string might cause false negatives since we ignore the validation of these untainted data. As a result, we provide the fine-grained policy to complement our default policy. In our fine-grained policy, there are very few false positives since we used the mis-use detection approach (each %`n` corresponding argument is validated whether or not pointing to our guarded area). But there are false negatives in this policy, which is caused by the limitations described below.

Limitations. One of the limitations of our approach is in the fine-grained policy, we only guard the most common attacked areas: return address, frame pointer, GOT and DTORS. If an attack makes use of a program legal function-pointer for the %`n` corresponding written address, we would not be able to detect it. Because there is no useful information (e.g., type to tell us this is a function pointer) in the normal binary file to let us identify program legal function-pointers from other variables. As for the GOT and DTORS, due to a different memory region they reside in, we can find out their address via ELF binary analysis.

There is another limitation. Our approach requires the program being dynamically linked (this is because library interposition technique only intercepts the function references to dynamic library). However, the statically linked applications are not too much used if we consider Xiao's study that 99.78% applications in Unix platform are dynamically linked [12]. In addition, if the program invokes its own functions instead of `glibc` for *I/O* and *string* manipulation, our approach also would not work.

6 Related Work

A considerable amount of approaches have been developed for the defense against format-string attacks. Those related to ours could be divided into three categories: compile-time analysis, run-time techniques, and combined compile-time/run-time techniques.

Compile-Time Analysis. This technique typically analyzes and/or instruments program source code to detect possible format-string attacks. PScan [22] is such a kind of simple and notable tool for discovering format-string vulnerabilities. It works by looking for the common case of the `printf`-style function in which the last parameter is the format string and none-static. Similar to the functionality of PScan, `gcc` itself also provides a compiler flag, "-Wformat=2", to cause `gcc` to complain about the non-static format string [23]. Both PScan and the "-Wformat" enhancement operate on the lexical level, and they offer the advantage of fixing bugs before software release. However, these two approaches are less complete, and usually subject to both missing format-string vulnerabilities and issuing warnings about safe code.

Compile-time analysis with taint technique is useful to find out bugs or identify potentially sensitive data. Perl's taint mode [24] is the first proposed solution showing this idea, which taints all the input to applications, and enforces a runtime checking to see whether the untrusted data is used in a security-sensitive way. Inspired by Perl's taint analysis, many approaches then have been proposed (e.g., [25,26,27,28]). One of them for particularly detecting format-string attacks is TypeQualifier presented by Shankar et al. [17]. In their approach, if the potentially tainted data is used as the format string, then an error is issued. From this point, it seems very similar to ours. However, these two approaches are based on different mechanism to implement (static analysis for this technique, run-time tracking in ours). Besides, this technique requires programmers' efforts to specify which object is tainted, and consequently presents an additional burden on developers. In contrast, our approach appears almost transparent. In addition, this approach is more conservative than necessary because static analysis is inherently limited and much supposedly tainted data is actually perfectly safe, whereas our approach is not so conservative since we only prevent those tainted data which contains dangerous format specifier and is used as a format string.

Run-Time Techniques. In contrast to compile-time analysis, run-time techniques present a low burden on developers and uniformly improve the security assurance of applications. Libformat [13] (a preloaded shared library) is a case of such examples, which aborts any processes if they call `printf`-family functions with a format string that is writable and contains `%n`-directive. This technique is effective in defending against real format-string attacks, but in most cases both writable format strings and `%n`-directives associated destination are legal, and consequently it generates many false alarms. Despite the similarity of both our approach and Libformat guard the writable and `%n`-directive contained in format-string argument, ours is not so much conservative as this technique, and we provide more flexibility.

Another notable run-time approach is referred as Libsafe [14], which implements a safe subset of format functions that will abort the running process if the inspected corresponding pointer of `%n` directive points to a return address or frame pointer. This approach also looks very close to ours. However, we should note the major difference is Libsafe provides every format string (despite static or not) checking on limited scope (i.e., return address and frame pointer), whereas we only check apparently-tainted areas to identify the root-cause (untrusted data) of format string attacks. Thus, as shown in our experiment, Libsafe would sometimes lead to false negatives, while our approach could catch almost all of them.

The idea of using dynamic taint analysis for detecting security attacks was attracted a lot of attention. Suh et al. [29] proposed a *dynamic information flow tracking* scheme to protect programs against malicious attacks. Chen et al. [30] developed a *pointer taintedness* detection architecture to defeat the most memory corruption attacks. These two approaches were demonstrated useful and efficient, but both of them require processor modifications to support taint-tracking. Unlike these two hardware solutions, Newsome et al. proposed a software approach, TaintCheck [31], to monitor and control the program execution at a fine-grained level. While this approach is very promising and can defend against a large number of security attacks with fewer false positives, the main drawback is that it incurs significant performance overhead by a factor of 10 or more because of its emulator-based implementation. Our approach follows their way in dynamic taint but differs in the granularity and interception.

Combined Compile-Time/Run-Time Approaches. FormatGuard [16] is an extension to `glibc` that provides argument number checking for `printf`-like functions with the support of GNU C Preprocessor. Programs need to be recompiled but without any modifications for its protection. Although FormatGuard can protect the `printf`-like functions efficiently, it cannot protect the format functions which use `vararg` such as `vprintf` (in this case it is not possible to count the actual number of parameters at compile time). Besides, FormatGuard may result in false negatives when another format specifier is replaced with `%n`-directive.

White-listing [15] is another approach which tries to achieve the benefits of both static and run-time techniques. By cleverly using a source code transformation, this approach automatically inserts the code that maintains and checks against the white-listing containing safe %n-writable address ranges via the knowledge gleaned from static analysis. White-listing gains high precision with very few false negatives and few false positives, and imposes little performance overhead. However, one limitation of this approach is that applications which are only re-compiled using White-listing can benefit from its protection.

7 Conclusion and Future Work

In this paper, we have proposed a practical and transparent approach to the detection and prevention of format-string attacks. We exploit the dynamic taint analysis and library interpositions technique, which allow us to protect the software without any recompilation, to achieve our goals. Through the thorough analysis and empirical evaluation, we show that our approach has very few false negatives and false positives, and just imposes a little performance overhead on the protected software.

Due to the similarity to format-string attacks, SQL injection is another dangerous attack caused by unvalidated input. We feel our approach is also applicable for the prevention of this attack, for instance, we can taint the input data and check against SQL syntax to see if these data represent an invalid user input. One of our future work will apply our approach to deal with this attack. Other future work includes to port our approach to other platforms (e.g., Windows), to investigate attack repair approaches and so on.

Acknowledgements

We thank Michael F. Ringenburg, Erwang Guo, Yi Wang, Yi Ge and anonymous reviewers for their support during this work. This research was supported in part by Chinese National Science Foundation under grant 60373064 and Chinese National 863 High-Tech Program under grant 2003AA144010.

References

1. "tf8". Wu-Ftpd Remote Format String Stack Overwrite Vulnerability. At *http://www.securityfocus.com/bid/1387*. (2000)
2. NIST National Vunerability Database. At *http://nvd.nist.gov*. (2006)
3. Scut, team teso:Exploiting Format String Vulnerabilities. At *http://www.team-teso.net/releases/formatstring-1.2.tar.gz*. (2001)
4. Riq and Gera. Advances in format string exploitation. *Phrack Magazine*, 59(7). At *http://www.phrack.org/phrack/59/p59-0x07*. (2002)
5. Lhee, K. and Chapin, S.:Buffer overflow and format string overflow vulnerabilities. *Software-Practice & Experience.* Vol 33(5). Pages: 423-460. (2003)
6. Anley, C.:Advanced SQL Injection In SQL Server Applications. Technical Report, NGSSoftware Insight Security Research. (2002)
7. Jacobowitz, D.:Multiple Linux Vendor rpc.statd Remote Format String Vulnerability. At *http://www.securityfocus.com/bid/1480*. (2000)
8. Kaempf, M.:Splitvt Format String Vulnerability. At *http://www.securityfocus.com/bid/2210/*. (2001)
9. NSI Rwhoisd Remote Format String Vulnerability. At *http://www.securityfocus.com/bid/3474*. (2001)
10. Pelat, G.:PFinger Format String Vulnerability. At *http://www.securityfocus.com/bid/3725*. (2001)
11. Goldsmith, D.:TCPflow Format String Vulnerability. At *ttp://www.securityfocus.com/bid/8366* . (2003)
12. Xiao, Z.:An Automated Approach to Software Reliability and Security. Invited Talk, Department of Computer Science, University of California at Berkeley. (2003)
13. Robbins, T.:Libformat. At *http://www.wiretapped.net/~fyre/software/libformat.html*. (2001)
14. Tsai, T. and Singh, N.:Libsafe 2.0: Detection of Format String Vulnerability Exploits. At *http://www.research.avayalabs.com/project/libsafe/doc/ whitepaper-20.pdf*. (2001)
15. Ringenburg, M. and Grossman, D.:Preventing Format-String Attacks via Automatic and Efficient Dynamic Checking. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*, Alexandria, Virginia. (2005)

16. Cowan, C., Barringer, M., Beattie, S. and Kroah-Hartman, G.:FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium (Security'01)*, Washington DC. (2001)
17. Shankar, U., Talwar, K., Foster, J. S., and Wagner, D.:Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium (Security'01)*, Washington DC. (2001)
18. TIS. Executable and Linkable Format Version 1.1. At *ftp://download.intel.com/perftool/tis/elf11g.zip*
19. Cormen, T., Stein, C., Rivest, R. and Leiserson, C.:*Introduction to Algorithms*. MIT Press, second edition. (2002)
20. Smirnov, A. and Chiueh, T.:DIRA: Automatic Detection, Identification and Repair of Control-Hijacking Attacks. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*, San Jose, CA. (2005)
21. Avijit, K., Gupta, P., and Gupta, D.:TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection. In *Proceedings of the 13th USENIX Security Symposium (Security'04).* (2004)
22. DeKok, A.:PScan: A limited problem scanner for C source files. At *http://www.striker.ottawa.on.ca/∼aland/pscan/*. (2000)
23. The GNU Compiler Collection. Free Software Foundation. At *http://gnu.gcc.org/*
24. Perl security manual page. At *http://www.perldoc.com*.
25. Zhang, X., Edwards, A. and Jaeger, T.:Using CQual for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium (Security'02).* (2002)
26. Foster, J., Fahndrich, M. and Aiken, A.:A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99).* (1999)
27. Evans, D. and Larochelle, D.:Improving Security Using Extensible Lightweight Static Analysis. In *IEEE Software*, January/February. (2002)
28. Tuong, A.N., Guarnieri, S., Greene, D., Shirley, J. and Evans, D.:Automatically hardening web applications using precise tainting. In *Proceedings of the 20th IFIP International Information Security Conference (SEC'05).* (2005)
29. Suh, G., Lee, J., Zhang, D. and Devadas, S.:Secure program execution via dynamic information flow tracking. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, Boston, MA. (2004)
30. Chen, S., Xu, J., Nakka, N., Kalbarczyk, Z. and Iyer, R. K.:Defeating memory corruption attacks via pointer taintedness detection. In *Proceedings of IEEE International Conference on Dependable Systems and Networks (DSN'05).* (2005)
31. Newsome, J. and Song, D. :Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*, San Jose, CA. (2005)