# FormatShield: A Binary Rewriting Defense against Format String Attacks

Pankaj Kohli and Bezawada Bruhadeshwar

Centre for Security, Theory and Algorithmic Research (C-STAR)
International Institute of Information Technology
Hyderabad - 500032, India
pankaj_kohli@research.iiit.ac.in, bezawada@iiit.ac.in

**Abstract.** Format string attacks allow an attacker to read or write anywhere in the memory of a process. Previous solutions designed to detect format string attacks either require source code and recompilation of the program, or aim to defend only against write attempts to security critical control information. They do not protect against arbitrary memory read attempts and non-control data attacks. This paper presents FormatShield, a comprehensive defense against format string attacks. FormatShield identifies potentially vulnerable call sites in a running process and dumps the corresponding context information in the program binary. Attacks are detected when malicious input is found at vulnerable call sites with an exploitable context. It does not require source code or recompilation of the program and can defend against arbitrary memory read and write attempts, including non-control data attacks. Also, our experiments show that FormatShield incurs minimal performance overheads and is better than existing solutions.

**Keywords:** Format String Attacks, Binary Rewriting, Intrusion Detection, System Security.

## 1 Introduction

Format string vulnerabilities are a result of the flexible features in the C programming language in the representation of data and the use of pointers. These features have made C the language of choice for system programming. Unfortunately, this flexibility comes at a cost of lack of type safety and function argument checking. The format string vulnerability applies to all format string functions in the C library, and exists in several popular software and server programs [4, 6, 9, 14, 16, 28]. Attackers have exploited format string vulnerabilities on a large scale [12, 36], gaining root access on vulnerable systems. As of January 2008, Mitre's CVE project [3] lists more than 400 entries containing the term "format string".

Format string vulnerabilities occur when programmers pass user supplied input to a format function, such as `printf`, as the format string argument i.e., using code constructs such as `printf(str)` instead of `printf("%s", str)`. This

Y. Mu, W. Susilo, and J. Seberry (Eds.): ACISP 2008, LNCS 5107, pp. 376–390, 2008.

```
int main(int argc, char **argv) {
    char string[8] = "DATA";
    if (argc > 1)
        printf(argv[1]);
    return 0;
}
```

**Fig. 1.** A program vulnerable to format string attack

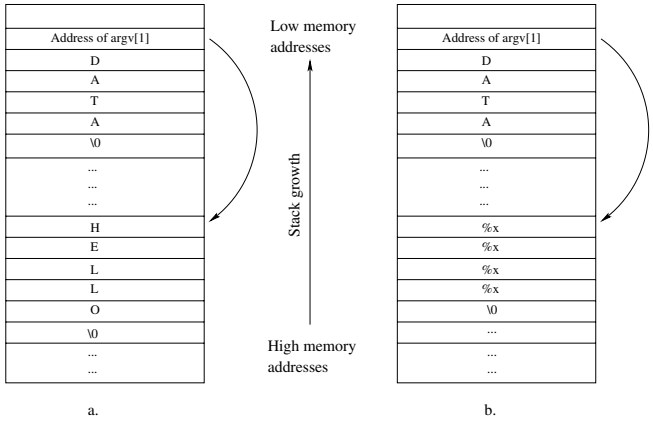


**Fig. 2.** Stack layout for the program given in Figure 1 when `printf` is called. **a.** On giving a legitimate input, the program prints HELLO. **b.** On giving a malicious input ("`%x%x%x%x`"), the program prints 44415441 (hex equivalent of "DATA").

input is interpreted by the format function as a format string, and is scanned for the presence of format specifiers such as `%x`, `%s`, `%n` etc. For each format specifier, corresponding value or address is picked from the stack and is read or written, depending on the format specifier. For example, the format specifier `%d` specifies an integer value to be read from the stack, while the format specifier `%n` specifies a value to be written to the address picked from the stack. An attacker can use this to perform reads or writes to arbitrary memory locations. Vulnerable functions in `libc` include `printf` family, `warn` and `err` family, `syslog`, `vsyslog` and several others.

In Figure 1, we show an example of a program vulnerable to a format string attack that passes the user input to `printf`. Here a malicious user could insert format specifiers in the input to cause the program to misbehave. Figure 2 shows the stack for this program when a legitimate and a malicious input in given. The malicious user gives as input `%x%x%x%x`, which causes `printf` to pick and display the next few bytes from the stack (44415441 - hex representation of "DATA" in this case), allowing him to read the stack. Similarly, use of `%s` format specifier makes the format function interpret the four bytes on the stack as the address of the string to be displayed. Using direct parameter access, i.e. `%N$d`,

allows an attacker to access the $N^{th}$ argument on the stack without accessing the previous $N-1$ arguments. The value of $N$ is so specified by the attacker such that the corresponding address is picked from the format string itself and hence can be controlled by the attacker. This allows an attacker to read any memory location within the address space of the process. The common form of the attack uses `%n` format specifier, which takes an address to an integer as an argument, and writes the number of bytes printed so far to the location specified by the address. The number of bytes printed so far can easily be controlled by printing an integer with large amount of padding such as `%656d`. Using `%n` format specifier, an attacker can overwrite the stored return address on the stack with the address of his own code, taking control of the program when the function returns. Other targets include address of destructor functions in `DTORS` table, address of library functions in Global Offset Table (`GOT`), function pointers and other security critical data. Also, an attacker can crash the process using several `%s` format specifiers, when an illegal address is picked from the stack, the program terminates with a segmentation fault.

Many techniques have been devised to defend against format string attacks [18, 15, 13, 10, 8, 30, 25, 19, 7]. All these approaches are valuable and defend against arbitrary code execution attacks. However, each of them suffers from at least one of the two drawbacks: either they require source code and recompilation of the program to be protected, or they aim to defend only against write attempts to security critical control information, such as return address, `GOT` and `DTORS` entries, etc. They do not guard against arbitrary memory read attempts, which can lead to critical information disclosure leading to further attacks. Also, none of the previously proposed solutions protects against non-control data attacks, in which an attacker targets a program specific security critical data, such as a variable storing user privileges, instead of control information. Such attacks have been studied in the past [20].

In this paper, we present FormatShield, a comprehensive solution to format string attacks. FormatShield does not require source code and recompilation, and can be used to protect legacy or proprietary programs for which source code may not be available. It does not take into consideration the presence of any specific format specifier such as `%n` in the format string, and thus, it can defend against both types of format string attacks, i.e. arbitrary memory read attempts and arbitrary memory write attempts. Also, FormatShield is capable of defending against non-control data attacks. It does not rely on the target of the format specifiers, and thus can protect against both, attacks that target control information such as return address on the stack, and those which target program specific security sensitive non-control information.

**Organization of the Paper.** The rest of the paper is organized as follows. Section 2 presents the technical description of the approach used by FormatShield. Section 3 describes the design and implementation of FormatShield. Section 4 presents experimental results on the effectiveness and performance evaluation. Limitations are discussed in section 5, followed by related work in section 6. Finally, section 7 concludes the paper.

## 2 Overview of Our Approach

FormatShield works by identifying call sites in a running process that are potentially vulnerable to format string attacks. A potentially vulnerable call site is identified when a format function is called with a probable legitimate user input as a format string argument. Further, a probable legitimate user input can be identified by checking whether the format string is writable and without any format specifiers. The format string argument of a non-vulnerable call site, such as in `printf("%s", argv[1])`, would lie in a non-writable memory segment, while that of a vulnerable call site, such as in `printf(argv[1])`, would lie in a writable memory segment. The key idea here is to augment the program binary with the program context information at the vulnerable call sites. A program context represents an execution path within a program and is specified by the set of return addresses on the stack. Since all execution paths to the vulnerable call site may not lead to an attack, FormatShield considers only those with an exploitable program context. For e.g., Figure 3 shows a vulnerable code fragment. Here the vulnerability lies in the function `output()`, which passes its argument as the format string to `printf`. Although `output()` has been called from three different call sites in `main()`, note that only one of these three, i.e. `output(argv[1])`, is exploitable. Here, the contexts, i.e. the set of return addresses on the stack, corresponding to all the three calls will be different, and thus the context corresponding to `output(argv[1])` can easily be differentiated from those of other two calls to `output()`. As the process executes, the exploitable program contexts are identified. The next time, if the vulnerable call site is called with an exploitable program context with format specifiers in the format string, a violation is raised. When the process exits, the entire list of exploitable program contexts is dumped into the binary as a new loadable read-only section, which is made available at runtime for subsequent runs of the program. If the section already exists, the list of exploitable program contexts is updated in the section. The program binary is updated with context information over use and becomes immune to format string attacks.

```
void output(char *str) {
    printf(str);
}

int main(int argc, char **argv) {
    output("alpha");
    .....
    output("beta");
    .....
    output(argv[1]);
    .....
}
```

**Fig. 3.** Only the third call to `output()` is exploitable

## 3 Implementation

FormatShield is implemented as a shared library that intercepts calls to the vulnerable functions in `libc`, preloaded using `LD_PRELOAD` environment variable. This section explains the design and implementation of FormatShield. First we explain how FormatShield identifies vulnerable call sites in a running process. Then we describe the binary rewriting approach used to augment the binary with the context information.

### 3.1 Identifying Vulnerable Call Sites

During process startup, FormatShield checks if the new section (named `fsprotect`) is present in the binary of the process. This is done by resolving the symbol `fsprotect`. If present, the list of exploitable program contexts is loaded. During process execution, whenever the control is transferred to a vulnerable function intercepted by FormatShield, it checks if the format string is writable. This is done by looking at the process memory map in `/proc/pid/maps`. If the format string is non-writable, corresponding equivalent function (such as `vprintf` for `printf`) in `libc` is called since a non-writable format string cannot be user input and hence cannot lead to an attack. However, if the format string is writable, FormatShield identifies the current context of the program, and checks if this context is in the list of exploitable program contexts. The current context of the program, i.e. the set of return addresses on the stack, is retrieved by following the chain of stored frame pointers on the stack. Instead of storing the entire set of return addresses on the stack, FormatShield computes a lightweight hash of the return addresses. If the current context is not present in the list of exploitable contexts, FormatShield checks if the format string is without any format specifiers. If the format string does not contain any format specifiers, it is identified as a legitimate user input, and the current context is added to the list of exploitable contexts. Any future occurrences of format specifiers in the format string of such a call with an exploitable context is flagged as an attack. Otherwise, if the format string contains format specifiers, it is not added to the list of exploitable contexts. In either case, FormatShield calls the equivalent function in `libc`. However, if the current context is already in the list of exploitable contexts, FormatShield checks if there are any format specifiers in the format string. If the format string does not contains any format specifiers, FormatShield calls the equivalent function in `libc`. Otherwise, if the format string contains format specifiers, FormatShield raises a violation. On detecting an attack, the victim process is killed, and a log is written to `syslog`.

Note that, if the format string is writable and contains format specifiers, it could be a case when an exploitable context is not yet identified by FormatShield and is being exploited by an attacker. However, FormatShield takes a safer step of not identifying it as an attack, since dynamically created format strings with format specifiers are commonly encountered and identifying such cases as attacks would terminate an innocent process which is not under attack. Also, the default

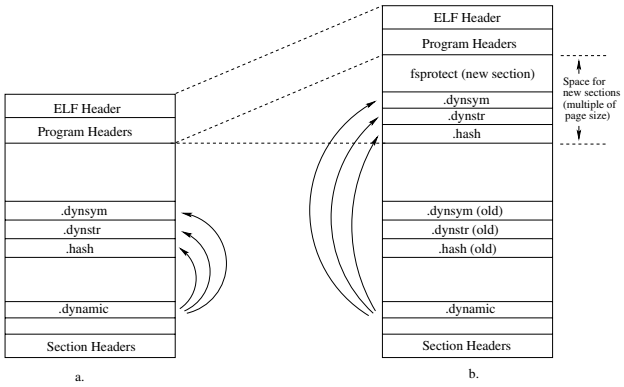


**Fig. 4.** ELF binary **a.** before rewriting **b.** after rewriting

```
....
[003] 0x08048148 a------ .hash       sz:00000040 link:04
[004] 0x08048170 a------ .dynsym     sz:00000080 link:05
[005] 0x080481C0 a------ .dynstr     sz:00000076 link:00
....
```

**Fig. 5.** Sections before rewriting the binary

```
....
[003] 0x080480E8 a------ .hash       sz:00000044 link:04
[004] 0x08048030 a------ .dynsym     sz:00000096 link:05
[005] 0x08048090 a------ .dynstr     sz:00000088 link:00
....
[026] 0x08047114 a------ fsprotect   sz:00003868 link:00
[027] 0x08048170 a------             sz:00000080 link:00
[028] 0x080481C0 a------             sz:00000076 link:00
[029] 0x08048148 a------             sz:00000040 link:00
```

**Fig. 6.** Sections after rewriting the binary. A new loadable read-only section named `fsprotect` is added which holds the context information. The `.dynsym`, `.dynstr` and `.hash` sections shown are extended copies of the original ones. The original `.dynsym`, `.dynstr` and `.hash` are still loaded at their original load addresses.

action to terminate the process can be used as a basis to launch denial of service (DoS) attack against the victim process by an attacker. However, silently returning from the vulnerable function without terminating the process may lead to an application specific error. For e.g., if the vulnerability lies in a call to `vfprintf`, skipping the call may lead to no output being printed to terminal if the string is being printed to `stdout`, which may not be fatal. However, if the string is being printed to a file, skipping the `vfprintf` call may lead to a corrupted file. Terminating the victim process would create "noise" that a conventional host-based intrusion detection system can detect the intrusion attempt.

```
DYNAMIC SYMBOL TABLE:
00000000      DF    *UND*      000000e7    __libc_start_main
00000000      DF    *UND*      00000039    printf
080484a4  g   DO    .rodata    00000004    _IO_stdin_used
00000000  w   D     *UND*      00000000    __gmon_start__
```

**Fig. 7.** Dynamic symbol table before rewriting the binary

```
DYNAMIC SYMBOL TABLE:
00000000      DF    *UND*      000000e7    __libc_start_main
00000000      DF    *UND*      00000039    printf
080484a4  g   DO    .rodata    00000004    _IO_stdin_used
00000000  w   D     *UND*      00000000    __gmon_start__
08047114  g   DO    fsprotect  0000000a    fsprotect
```

**Fig. 8.** Dynamic symbol table after rewriting the binary. A new dynamic symbol named `fsprotect` is added while rewriting the binary which points to the new section at address `0x08047114`.

### 3.2 Binary Rewriting

FormatShield uses an approach (Figure 4) similar to that used by TIED [22] to insert context information in the program binary. FormatShield currently supports only ELF [11] executables. In the ELF binary format, `.dynsym` section of binary contains symbols needed for dynamic linking, `.dynstr` section contains the corresponding symbol names, and `.hash` section holds a hash look up table to quickly resolve symbols. `.dynamic` section holds the addresses of these three sections. The information to be inserted is a list of hashes of stored return addresses corresponding to exploitable contexts at different vulnerable call sites in the program. During process exit, the entire list is dumped into the executable as a new read-only loadable section. If the section is already present, the context information in the section is updated. A typical ELF binary loads at virtual address `0x08048000`. To add a new section (Figures 5,6), FormatShield extends the binary towards lower addresses, i.e. lower than address `0x08048000`. This is done to ensure that the addresses of existing code and data do not change. To make the context information available at run time, a new dynamic symbol (Figures 7,8) is added to the `.dynsym` section and the corresponding address is set to that of the new section. Since this requires extending `.dynsym`, `.dynstr` and `.hash` sections which cannot be done without changing the addresses of other sections, FormatShield creates an extended copy of these sections, i.e. `.dynsym`, `.dynstr` and `.hash`, and changes their addresses in `.dynamic` section. The address of the new section is so chosen such that the sum of the sizes of the four new sections is a multiple of page size[1]. The space overhead is of the order of few kilobytes (less than 10 KB for most binaries).

[1] As per ELF specification [11], loadable process segments must have congruent values of the load address, modulo the page size.

### 3.3 Implementation Issues

One of the issues with FormatShield is when the program uses some kind of Address Space Randomization (ASR) [2, 21, 23]. ASR involves randomizing base addresses of various segments so as to make it difficult for an attacker to guess an address. Since the base addresses of various code segments are randomized, the absolute memory locations associated with the set of return addresses will change from one execution of the program to the next. To compensate for this, we decompose each return address into a pair {`name, offset`}, where `name` identifies the executable or the shared library, and `offset` identifies the relative distance from the base of the executable or shared library.

## 4 Evaluation

We conducted a series of experiments to evaluate the effectiveness and performance of FormatShield. All tests were run in single user mode on a Pentium-4 3.2 GHz machine with 512 MB RAM running Linux kernel 2.6.18. All programs were compiled with `gcc` 4.1.2 with default options and linked with `glibc` 2.3.6.

### 4.1 Effectiveness

We tested FormatShield on five programs with known format string vulnerabilities:

- `wuftpd` version 2.6.0 and earlier suffer from a format string vulnerability [6] in the "`SITE EXEC`" implementation. A remote user can gain a root shell by exploiting this vulnerability.
- `tcpflow` 0.2.0 suffers from a format string vulnerability [28], that can be exploited by injecting format specifiers in command line arguments. A local user can gain a root shell by exploiting this vulnerability.
- `xlock` 4.16 suffers from a format string vulnerability [29] when using command line argument `-d`, that can be used by a local user to gain root privileges.
- `rpc.statd` (`nfs-utils` versions 0.1.9.1 and earlier) suffers from a format string vulnerability [9], which allows a remote user to execute arbitrary code as root.
- `splitvt` version 1.6.5 and earlier suffer from a format string vulnerability when handling the command line argument `-rcfile`. A local user can gain a root shell[2] by exploiting this vulnerability.

The above programs were trained "synthetically" with legitimate inputs before launching the attacks so as to identify the vulnerable call sites and the corresponding exploitable contexts. FormatShield successfully detected all the above attacks, and terminated the programs to prevent execution of malicious code. The results are presented in Table 1.

[2] The attack gives a root shell if the program is installed suid root, otherwise it gives a user shell.

**Table 1.** Results of effectiveness evaluation

| Vulnerable program | CVE # | Results without FormatShield | Results with FormatShield |
|---|---|---|---|
| `wuftpd` | CVE-2000-0573 | Root Shell acquired | Process Killed |
| `tcpflow` | CAN-2003-0671 | Root Shell acquired | Process Killed |
| `xlock` | CVE-2000-0763 | Root Shell acquired | Process Killed |
| `rpc.statd` | CVE-2000-0666 | Root Shell acquired | Process Killed |
| `splitvt` | CAN-2001-0112 | Root Shell acquired [2] | Process Killed |

**Table 2.** Comparison with previous approaches

| Feature | LibFormat | FormatGuard | Libsafe | WhiteListing | FormatShield |
|---|---|---|---|---|---|
| Works without source code | ✓ | × | ✓ | × | ✓ |
| Supports vprintf like functions | ✓ | × | ✓ | ✓ | ✓ |
| Supports wrapper functions | × | × | ✓ | ✓ | ✓ |
| Prevents read attacks | × | ✓[3] | × | × | ✓ |
| Prevents write attacks | ✓ | ✓ | ✓[4] | ✓ | ✓ |
| Prevents non control data attacks | × | ✓[3] | × | × | ✓ |
| Not format string specific | ✓ | ✓ | ✓ | × | ✓ |

To check the effectiveness of FormatShield on non-control data attacks, we modified the publicly available exploit for the wuftpd 2.6.0 format string vulnerability [6] to overwrite the cached copy of user ID `pw->pw_uid` with 0 so as to to disable the server's ability to drop privileges. FormatShield successfully detected the write attempt to the user ID field and terminated the child process. Table 2 shows a detailed comparison of FormatShield and the previous approaches to detection of format string attacks.

### 4.2 Performance Testing

To test the performance overhead of FormatShield, we performed micro benchmarks to measure the overhead at function call level, and then macro benchmarks to measure the overhead at application level.

**Micro benchmarks.** To measure the overhead per function call, we ran a set of simple benchmarks consisting of a single loop containing a single `sprintf`

[3] FormatGuard protects by counting arguments and number of format specifiers, and thus can protect against arbitrary memory reads and non-control data attacks. However, it does not work for format functions called from within a wrapper function and those with variable argument lists such as `vprintf`.

[4] Libsafe defends against writes to stored return address and frame pointer, but does not protect against writes to `GOT` and `DTORS` entries.

**Table 3.** Micro benchmarks

| Benchmark | FormatGuard | White-Listing | FormatShield |
|---|---|---|---|
| `sprintf`, no format specifiers | 7.5% | 10.2% | 12.2% |
| `sprintf`, 2 %d format specifiers | 20.9% | 28.6% | 4.6% |
| `sprintf`, 2 %n format specifiers | 38.1% | 60.0% | 3.3% |
| `vsprintf`, no format specifiers | No protection | 26.4% | 15.5% |
| `vsprintf`, 2 %d format specifiers | No protection | 39.8% | 1.9% |
| `vsprintf`, 2 %n format specifiers | No protection | 74.7% | 3.4% |

call. A six character writable string was used as the format string. With no format specifiers, FormatShield added an overhead of 12.2%. With two `%d` format specifiers, overhead was found to be 4.6%, while with two `%n` format specifiers the overhead was 3.3%. We also tested `vsprintf` using the same loop. The overheads were found to be 15.5%, 1.9% and 3.4% for no format specifiers, two `%d` format specifiers, and two `%n` format specifiers respectively. The overheads were found to be much less than those with the previous approaches. Table 3 compares micro benchmarks of FormatShield with those of FormatGuard and White-Listing.

**Macro benchmarks.** To test the overhead at the application level, we used `man2html` since it uses `printf` extensively to write HTML-formatted man pages to standard output. The test was to translate 4.6 MB of man pages. The test was performed multiple times. It took `man2html` 0.468 seconds to convert without FormatShield, and 0.484 seconds with FormatShield. Thus, FormatShield imposed 3.42% run-time overhead.

## 5 Discussion

In this section, we discuss the false positives and false negatives of FormatShield, and its limitations when applied to the software protection.

### 5.1 False Positives and False Negatives

FormatShield can give false positives or false negatives in certain cases. It is when a format string is dynamically constructed as a result of a conditional statement and then passed to a format function. A false positive can be there if one outcome of the condition creates a format string with format specifiers and the other outcome creates one without format specifiers. Similarly, a false negative can be there when one outcome of the condition reads user input into the format string and the other outcome creates a format string with format specifiers. Also, there can be a false negative when format specifiers are present at the vulnerable call site but the corresponding context is not yet identified.

### 5.2 Limitations

FormatShield requires frame pointers to obtain the set of stored return addresses on the stack, which are available in most cases. However, it may not be able to

protect programs compiled without frame pointers, such as those compiled with `-fomit-frame-pointer` flag of `gcc`. Also, FormatShield requires that exploitable contexts of the vulnerable call sites are identified before it can detect attacks. This may require the program to be trained, either by deploying or by exercising "synthetically". Another limitation of FormatShield is that it requires programs to be dynamically linked (since library call interpositioning works only with dynamic linked programs). However, this is not a problem if we consider Xiao's study [31] according to which 99.78% applications on Unix platform are dynamically linked. Also, since FormatShield keeps updating the context information in the program binary till it becomes immune to format string attacks, it may interfere with some integrity checkers.

## 6 Related Work

Several techniques have been proposed to defend against format string attacks. These can be divided into three categories: compile-time approaches, run-time approaches, and combined compile-time and run-time approaches.

### 6.1 Compile-Time Approaches

PScan [7] works by looking for `printf`-style functions where the last parameter is a non-static format string. Similar to PScan's functionality, `gcc` itself provides flags such as "-Wformat=2" to statically check the format string and issue warnings for dangerous or suspect formats. Both PScan and `gcc` work at the lexical level. They require source code, are subject to missing format string vulnerabilities and even issue warnings about safe code. Another compile-time technique for detecting format string attacks is presented by Shankar et al [8]. In their approach, all untrusted inputs are marked as tainted, and the propagation of tainted data is tracked throughout the program operation. Any data derived from tainted data is itself marked as tainted. If at some point in the program, the tainted data is used as a format string, an error is raised. This approach does not work for already compiled code. Moreover, it requires programmers' efforts to specify which objects are tainted.

### 6.2 Run-Time Approaches

LibFormat [10] works by intercepting calls to `printf` family of functions, and aborts any process if the format string is writable and contains `%n` format specifier. This technique is quite effective in defending against real format string attacks, but in most cases writable format strings containing `%n` format specifier are legal, and consequently it generates many false alarms. Libsafe [13] implements a safe subset of format functions that will abort the running process if the address corresponding to a `%n` format specifier points to a return address or a frame pointer. However, it still allows writes to `GOT` and `DTORS` entries, and therefore is subject to missing many attack attempts. Lin et al [15] use dynamic taint

and validation to detect format string attacks. In their approach, if the format string is non-static and contains `%n` format specifier, and if the corresponding address points to the return address, frame pointer, or `GOT` or `DTORS` entries, an attack is detected and the process is aborted. The approach is effective in preventing arbitrary memory write attempts to control sensitive addresses, but does not defends against arbitrary memory read attempts and non-control data attacks. Kimchi [25] is another binary rewriting defense technique, that inserts code in the binary which prevents a format string to access memory beyond the stack frame of its parent function. However, it is subject to missing many attack attempts when the format string itself is declared in the parent function, and therefore lies in the parent function's stack frame. Lisbon [30] identifies the input argument list, and places a canary word immediately after the list's end. A violation is raised if the program attempts to access the canary word. This approach works for attacks that aim to probe the underlying stack using a series of `%x%x%x...` format specifiers. However, the approach will miss all the read and write attempts where the attacker uses a format specifiers with direct parameter access. For example, the input `%18$x` will read the $18^{th}$ argument without accessing the canary. All the above approaches detect write attempts using `%n` format specifiers but fail to detect arbitrary memory read attempts.

Address Space Randomization (ASR) [2, 21, 23, 34] is a generic technique to defend against any kind of memory corruption attack. The idea behind ASR is that the successful exploitation of such an attack requires an attacker to have knowledge of the addresses where the critical information is stored and/or where the attacker specified code is present. By randomizing the locations of various memory segments and other security critical structures, ASR makes it hard for an attacker to guess the correct address. For the Intel x86 architecture, PaX ASLR [1, 2] provides 16, 16 and 24 bits of randomness for the executable, mapped and stack areas respectively. However, many successful derandomization attacks against PaX have been studied in the past. Durden [32] uses a format string attack to deduce the value of `delta_mmap`. Another brute force derandomization attack has been presented by Shacham et al [17], which defeats PaX ASLR in less than 4 minutes. Instruction Set Randomization (ISR) [24, 27] is another generic defense technique that defends against code injection attacks by randomizing the underlying instruction set. Since the attacker does not know the randomizing key, his injected code will be invalid for the injected process, causing a runtime exception. However, overheads associated with ISR make it an impractical approach to defend against attacks. Also, attacks have been published [33] capable of defeating ISR in about 6 minutes. Moreover, both kinds of randomizations still allow information disclosure attacks.

### 6.3 Combined Compile-Time and Run-Time Approaches

FormatGuard [18] provides argument number checking for `printf`-like functions using GNU C compiler. Programs need to be recompiled without any modification. It provides protection against only a subset of functions and does not work for functions that expect variable argument lists such as `vprintf`. White-listing

[19] uses source code transformation to automatically insert code and maintains checks against the whitelist containing safe `%n` writable address ranges via knowledge gained from static analysis. Both FormatGuard and White-Listing require source code and recompilation of the program.

## 7 Conclusion and Future Work

Format string vulnerabilities are one of the few truly threats to software security. This paper described the design, implementation and evaluation of FormatShield, a tool that protects vulnerable programs by inserting context information in program binaries. Although the current implementation is designed to work on Linux platform, the same approach can be made to work on Win32 platform as well, using the Detours [35] framework. We have shown that FormatShield is effective in stopping format string attacks, and incurs a very nominal performance penalty of less than 4%. However, FormatShield requires the process to be trained using synthetic data or by deploying in order to identify vulnerable call sites. We believe static analysis can be used to identify such vulnerable call sites. Hence, the future work involves covering this limitation of FormatShield to make it much more effective in defending against format string attacks.

## References

[1] PaX. Published on World-Wide Web (2001), `http://pax.grsecurity.net`
[2] PaX Team. PaX address space layout randomization (ASLR), `http://pax.grsecurity.net/docs/aslr.txt`
[3] CVE - Common Vulnerabilities and Exposures, `http://www.cve.mitre.org`
[4] Kaempf, M.: Splitvt Format String Vulnerability, `http://www.securityfocus.com/bid/2210/`
[5] CWE - Vulnerability Type Distributions in CVE, `http://cve.mitre.org/docs/vuln-trends/index.html`
[6] tf8.: Wu-Ftpd Remote Format String Stack Overwrite Vulnerability, `http://www.securityfocus.com/bid/1387`
[7] De Kok, A.: PScan: A limited problem scanner for C source files, `http://www.striker.ottawa.on.ca/~aland/pscan/`
[8] Shankar, U., Talwar, K., Foster, J.S., Wagner, D.: Detecting format string vulnerabilities with type qualifiers. In: Proceedings of the 10th USENIX Security Symposium (Security 2001), Washington, DC (2001)
[9] Jacobowitz, D.: Multiple Linux Vendor rpc.statd Remote Format String Vulnerability, `http://www.securityfocus.com/bid/1480`
[10] Robbins, T.: Libformat, `http://www.wiretapped.net/~fyre/software/libformat.html`
[11] Tool Interface Standard (TIS) Committee: Executable and linking format (ELF) specification, version 1.2 (1995)
[12] CERT Incident Note IN-2000-10, Widespread Exploitation of rpc.statd and wu-ftpd Vulnerabilities (September 15, 2000)
[13] Tsai, T., Singh, N.: Libsafe 2.0: Detection of Format String Vulnerability Exploits, `http://www.research.avayalabs.com/project/libsafe/doc/whitepaper-20.pdf`

[14] Pelat, G.: PFinger Format String Vulnerability, http://www.securityfocus.com/bid/3725
[15] Lin, Z., Xia, N., Li, G., Mao, B., Xie, L.: Transparent Run-Time Prevention of Format-String Attacks Via Dynamic Taint and Flexible Validation. In: De Meuter, W. (ed.) ISC 2006. LNCS, vol. 4406, Springer, Heidelberg (2007)
[16] NSI Rwhoisd Remote Format String Vulnerability, http://www.securityfocus.com/bid/3474
[17] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: Proceedings of the 11th ACM conference on Computer and communications security, Washington DC, USA, October 25-29 (2004)
[18] Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G.: FormatGuard: Automatic protection from printf format string vulnerabilities. In: Proceedings of the 10th USENIX Security Symposium (Security 2001), Washington, DC (2001)
[19] Ringenburg, M., 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 2005), Alexandria, Virginia (2005)
[20] Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: Proceedings of the 14th conference on USENIX Security Symposium, Baltimore, MD (2005)
[21] Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In: USENIX Security Symposium, Washington, DC (August 2003)
[22] Avijit, K., Gupta, P., Gupta, D.: TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection. In: Proceedings of the 13th USENIX Security Symposium, San Diego, CA (2004)
[23] Bhatkar, S., Sekar, R., DuVarney, D.C.: Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In: Proceedings of the 14th USENIX Security Symposium, July 31-August 05, p. 17 (2005)
[24] Barrantes, E.G., Ackley, D.H., Palmer, T.S., Stefanovic, D., Zovi, D.D.: Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In: Proceedings of the 10th ACM conference on Computer and communications security, Washington D.C, USA (October 27-30, 2003)
[25] You, J.H., Seo, S.C., Kim, Y.D., Choi, J.Y., Lee, S.J., Kim, B.K.: Kimchi: A Binary Rewriting Defense Against Format String Attacks. In: WISA 2005 (2005)
[26] Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, pp. 63–78 (January 1998)
[27] Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering Code-Injection Attacks with Instruction-Set Randomization. In: Proceedings of the 10th ACM conference on Computer and Communications Security, Washington D.C, USA, October 27-30 (2003)
[28] @stake, Inc. tcpflow 0.2.0 format string vulnerability (August 2003), http://www.securityfocus.com/advisories/5686
[29] bind: xlockmore User Supplied Format String Vulnerability, http://www.securityfocus.com/bid/1585
[30] Li, W., Chiueh, T.-c.: Automated Format String Attack Prevention for Win32/X86 Binaries. In: Proceedings of 23rd Annual Computer Security Applications Conference, Florida (December 2007)

[31] Xiao, Z.: An Automated Approach to Software Reliability and Security. Invited Talk, Department of Computer Science. University of California at Berkeley (2003)
[32] Durden, T.: Bypassing PaX ASLR protection. Phrack Magazine 59(9) (June 2002), http://www.phrack.org/phrack/59/p59-0x09
[33] Sovarel, N., Evans, D., Paul, N.: Where's the FEEB? The Effectiveness of Instruction Set Randomization. In: 14th USENIX Security Symposium (August 2005)
[34] Xu, J., Kalbarczyk, Z., Iyer, R.: Transparent Runtime Randomization for Security. In: Fantechi, A. (ed.) Proc. 22nd Symp. on Reliable Distributed Systems –SRDS 2003, pp. 260–269. IEEE Computer Society, Los Alamitos (2003)
[35] Hunt, G., Brubacher, D.: Detours: Binary interception of Win32 functions. In: Proceedings of the 3rd USENIX Windows NT Symposium, Seattle, WA, pp. 135–143 (1999)
[36] Lemos, R.: Internet worm squirms into Linux servers. Special to CNET News.com (January 17, 2001), http://news.cnet.com/news/0-1003-200-4508359.html