

Libsafe 2.0: Detection of Format String Vulnerability Exploits

Timothy Tsai and Navjot Singh
Avaya Labs, Avaya Inc.
600 Mountain Ave
Murray Hill, NJ 07974 USA
{ttsai,singh}@avaya.com

February 6, 2001

Abstract

This white paper describes a significant new feature of libsafe version 2.0: the ability to detect and handle format string vulnerability exploits. Such exploits have recently garnered attention in security advisories, discussion lists, web sites devoted to security, and even conventional media such as television and newspapers. Examples of vulnerable software include `wu-ftpd` (a common FTP daemon) and `bind` (A DNS [Domain Name System] server). This paper describes the vulnerability and the technique libsafe uses to detect and handle exploits.

NOTE: This paper only describes one particular feature of libsafe version 2.0: the ability to detect and handle format string vulnerability exploits. Other features include support for code compiled without frame pointer instructions, extra debugging facilities, and bug fixes. See [1] for details of the original version of libsafe.

1 Introduction

Buffer overflow exploits constitute perhaps the most common form of computer security attack [4, 5, 6]. Such exploits take advantage of programming errors to overflow buffers, thus writing unintended data to the part of memory that immediately follows the targeted buffers. If the targeted buffer exists on the process stack, then the exploit often attempts to overwrite a return address on the stack, which often results in obtaining root access to that machine. The original version of libsafe, version 1.3 [1], presented a significant advance in the detection and handling of buffer overflow attacks by offering a solution that detects a large number of exploits with low overhead and tremendous ease of use¹.

Recently, another widespread vulnerability has received a great deal of attention: the format string vulnerability[2, 7]. The latest version of libsafe, version 2.0, implements a solution for detecting and handling the most dangerous format string vulnerability exploits, while preserving the low overhead and ease of use of the original libsafe.

The most common source of this vulnerability is the ubiquitous `printf()` function. Consider the following vulnerable piece of code:

```
printf("%x %x %x %x\n");
```

The above code will usually compile with no warnings², even though it obviously lacks the required number of arguments. If this code is executed, it will print out four hexadecimal numbers, corresponding to the values on the stack where it expects the missing arguments to be present. This allows an attacker to examine the contents of the stack.

The following code illustrates an even more insidious form of the format string vulnerability:

¹Libsafe requires no specific security expertise and can be installed in under one minute!

²For `gcc`, warnings are produced with the `-Wall` option, but not with the default warning level.

```
printf("%.*d%n\n", (int) start_attack_code, 0, return_addr_ptr);
```

The above example takes advantage of a relatively seldom used `printf()` specifier: `%n`. This specifier calculates the current number of characters produced by the `printf()` function and writes this number to the memory location indicated by the corresponding pointer in the argument list. In our example, the pointer is `return_addr_ptr`. The astute observer may realize at this point that a malicious attacker can potentially overwrite any memory location, including locations containing return addresses. Furthermore, the above form of the `printf()` statement controls the exact number that is written to the memory location. Our example writes the value `start_attack_code` to the location `return_addr_ptr`. Assuming that `start_attack_code` is the starting address for some attack code, the next return from that exploited function will cause the attack code to be executed. Often, this attack code causes a shell to be started, and if the process under attack is privileged (as is the case with many daemon processes), then an attacker can obtain a root shell.

Fortunately, it takes a bit more ingenuity to actually take advantage of this vulnerability. Usually, vulnerable code occurs in a form similar to the following:

```
if (illegal_command(command)) {  
    sprintf(error_msg, "Illegal command: %s", command);  
    ...  
    syslog(LOG_WARNING, error_msg);  
    return;  
}
```

In this example, `command` is a character buffer that contains a command from the user. If the command is illegal, then the `sprintf()` statement forms an error message that is passed to `syslog()`. Under normal circumstances, `syslog()` will simply append `error_msg` to the appropriate log file. However, if `command` contains `printf()` specifiers, such as those in the first two code examples, then bad things can happen.

Such code vulnerabilities exist in real life, and the corresponding exploits also exist. In fact, existence of these and similar vulnerabilities and the relative ease of obtaining exploits has largely led to the prevalence of so-called “script kiddies,” or attackers who systematically attack remote machines using downloaded scripts in the hopes of finding a machine that is vulnerable. Such attackers often possess only a rudimentary knowledge of networks and systems. However, they often find great success due to the surprisingly large number of Internet-connected machines that execute vulnerable software. Part of the problem is the complexity of system maintenance. Making sure that one’s machine has the latest version of every software package is not simple, especially since system maintenance is often a secondary responsibility. Also, some vulnerabilities are still mostly unknown, and software updates to fix the problem may not yet be available.

This is where `libsafes` version 2.0 is valuable. `Libsafes` version 2.0 will foil all format string vulnerability exploits that attempt to overwrite return addresses on the stack. If such an attack is attempted, `libsafes` will log a warning and terminate the targeted process. As with version 1.3, installation is extremely easy and requires no knowledge of the system, applications, exploits, or even `libsafes` itself. Also, because `libsafes` incurs relatively little overhead, it can be used to protect all processes on a machine, thereby potentially detecting instances of vulnerabilities that may yet be unknown.

2 Implementation

The implementation of format string vulnerability detection in `libsafes` version 2.0 borrows heavily from the basic detection mechanism in version 1.3. There are three main steps in the detection mechanism:

Interception: `Libsafes` executes its own version of selected vulnerable functions.

Safety check: `Libsafes` determines if the function can be safely executed.

Violation handling: If the function cannot be safely executed, `libsafes` executes warning and termination actions.

2.1 Interception

The basic idea behind libsafe is the interception of vulnerable functions by safer alternatives that first check to make sure that the functions can be safely executed based on their arguments. If the check passes, libsafe either calls the original function or executes code that is functionally equivalent. Otherwise, warnings are posted and the process is terminated.

Libsafe is able to intercept functions (i.e., substitute its alternatives in place of the original functions) because it is implemented as a shared library that is loaded into memory before the standard library (i.e., `/lib/libc.so`). For Linux systems, the run-time loader, `ld.so`, is responsible for loading the various program code and libraries into memory. For programs that require the standard library, `ld.so` loads this library into memory and links all references to library functions in the program code to the library functions. If libsafe is activated, `ld.so` loads the libsafe library into memory before the standard library. Because the libsafe alternative functions have the same names as the original standard library functions, `ld.so` uses the libsafe functions in place of the standard library functions.

Most of the libsafe functions perform a safety check and then call the original function or a safer alternative (e.g., `snprintf()` is called in place of `sprintf()`). However, two functions are treated differently: `_I0_vfprintf()` and `_I0_vfscanf()`³. For `_I0_vfprintf()` and `_I0_vfscanf()`, the original source code from `libc-2.1.3-91` is incorporated directly into libsafe. Libsafe needs the original source code because the safety checks for these two functions require knowledge of local variables.

2.2 Safety check

The safety checks for each function are highly specific to each function. For `_I0_vfprintf()`, libsafe performs two checks:

Return address and frame pointer check:

For each `%n` specifier, libsafe checks the associated pointer argument. Each such pointer argument is passed to `_libsafe_raVariableP(void *addr)`, where `addr` is the pointer argument. `_libsafe_raVariableP(void *addr)` returns 1 only if it determines that `addr` points to a return address or a frame pointer on the stack. Otherwise, it returns 0, which means that `addr` points to an address that is either not on the stack or which is on the stack, but which is not a return address or a frame pointer. If `_libsafe_raVariableP()` returns 1, then libsafe has found a violation.

Frame span check:

The argument list for any function should always be contained within a single stack frame. Thus, attacks that attempt to probe the stack using statements such as `printf("%x %x ...")` might require arguments that extend beyond the current stack frame. The `_libsafe_span_stack_frames(void *start_addr, void *end_addr)` function returns 1 only if `start_addr` and `end_addr` are located in two different stack frames. If `_libsafe_span_stack_frames()` returns 1, then libsafe has found a violation.

To perform these two checks, libsafe determines the locations and sizes of the frames on the stack. Figure 1 illustrates the organization of a process stack. The beginning of each stack frame is indicated by the presence of a frame pointer that points back to the previous stack frame. Libsafe finds each stack frame by starting at the top-most frame and traversing the frame pointers until it finds the stack frame for `main()`. The top-most frame corresponds to a libsafe function. Within this libsafe function, the frame pointer is found by using the `gcc` function `__builtin_frame_pointer(0)`. The return address back into the calling function is located immediately before each frame pointer. This technique works for most processes, with a few exceptions. Certain compilers may not produce code that places frame pointers on the stack (e.g., `gcc -fomit-frame-pointer`), and some customized compilers may not locate return addresses immediately next to the frame pointer (e.g., the StackGuard compiler [3]).

³ `_I0_vfprintf()` and `_I0_vfscanf()` are the core functions that all other `*printf()` and `*scanf()` functions eventually call. Thus, intercepting these two core functions effectively intercepts the entire family of `*printf()` and `*scanf()` functions. Note: `syslog()` also eventually calls `_I0_vfprintf()`

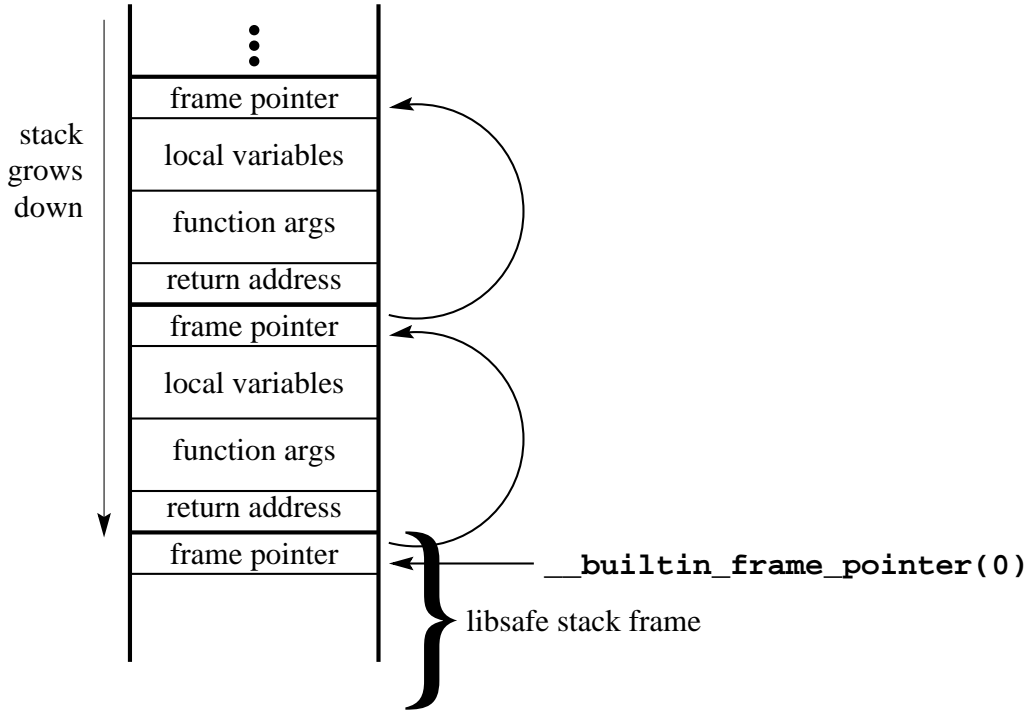


Figure 1: Stack Frames

2.3 Violation handling

If libsafe finds a violation during a safety check, then it performs the actions in Table 1.

Table 1: Libsafe Actions After Finding a Violation

Action	Default	Optional?
Terminate process	Off/On	Not optional
Add a entry to <code>/var/log/secure</code> using <code>syslog()</code>	On	Optional
Print a warning to <code>stderr</code>	On	Not optional
Dump a hexadecimal version of the stack contents to a file	Off	Optional
Send email to a list of recipients	Off	Optional
Produce a core dump by calling <code>abort()</code>	Off	Optional

The main libsafe action after detecting a violation is to terminate the process. Data integrity after a violation cannot be assured, and therefore, the safest course of action is to terminate the entire process. However, for violations of the return address and frame pointer check, libsafe can optionally allow the process to continue execution. This exception is based on the assumption that programmers will almost never (or at least should never) produce code that attempts to use the `%n` specifier to overwrite a return address or frame pointer. In practice, most occurrences of such attacks result from processing user input that unexpectedly contains the `%n` specifier. In such instances, since the input is garbage, libsafe can usually allow the process to continue to process the input as long as the `%n` specifier is not permitted to write to memory.

2.4 Notes

1. Libsafe relies on the location of frame pointers on the stack to determine the location of stack frames and return addresses. Some programs have been compiled without code to embed frame pointers on the stack (e.g., by using `gcc -fomit-frame-pointer`). For such code, libsafe will automatically detect

the absence of frame pointers on the stack and allow the program to execute normally. However, it will not be able to detect any exploits for such programs.

2. Libsafe is linked with glibc and is incompatible with libc5. If you have a program that is linked with libc5, you will need to either obtain an updated version linked with glibc or recompile the source code yourself with glibc.

3 Software Availability

Libsafe version 2.0 has not yet been released to the general public. However, it is our intention to release the software under the Lesser GNU Public License sometime in the near future. Please contact Timothy Tsai (ttsai@avaya.com) if you have any questions or are interested in evaluating the software.

References

- [1] Arash Baratloo, Timothy Tsai, and Navjot Singh. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.
- [2] Bind holes mean big trouble. <http://securityfocus.com/frames/?content=/templates/article.html?id=144%>.
- [3] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, 1998.
- [4] Mark W. Eichen and Jon A. Rochlis. With microscope and tweezers: An analysis of the internet virus of november 1988. In *Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy (SSP '89)*, 1989.
- [5] Jon A. Rochlis and Mark W. Eichen. With microscope and tweezers: The worm from MIT's perspective. *Communications of the ACM*, June 1989.
- [6] Donn Seeley. A tour of the worm. In *Proceedings 1989 Winter USENIX Technical Conference*, January 30 - February 3 1989.
- [7] Wu-ftpd remote format string stack overwrite vulnerability. <http://www.securityfocus.com/frames/?content=/vdb/bottom.html%3Fvid%3D1%387>.