# Program transformation to add-on protection against buffer overflow

Munawar Hafiz, Ralph E. Johnson
University of Illinois at Urbana-Champaign
(mhafiz, rjohnson)@cs.uiuc.edu

## Abstract

Security requirements change. Many legacy systems fail to cope with the changing requirements because it is infeasible to redesign these systems. This paper is an example that protection can be added on to an existing system by program transformation. We successfully transformed the source code of three open source C programs to introduce protection against buffer overflow attacks. By describing security solutions as program transformations, it is possible to retrofit security to an existing system.

***Categories and Subject Descriptors*** D.1.2 [*Automatic Programming*]: Program transformation; D.2.9 [*Software Management*]: Software maintenance; D.2.11 [*Software Architectures*]: Patterns.

***General Terms*** Security.

***Keywords*** Program Transformation, Buffer Overflow, Security.

## 1. Introduction

Security is architectural; it is a property of the entire system, not one part of it. Security experts generally say that "security cannot be added on, it must be designed from the beginning" (Anderson 1972). Security solutions cannot be added to a system by adding a module, but it can be added in other ways. In particular, it is possible to improve the security of a system by "adding on" a program transformation.

This paper provides an example that security can be added on to an existing system. We introduce a program transformation that changes vulnerable C source code and makes it safe from buffer overflow attacks. The purpose of this paper is to show that program transformations are useful for making software secure, not to provide new ways of analyzing programs or representing programs. We implemented our program transformation using crude techniques, and there are certainly better ways to implement it. However, even this simple example shows that program transformations can help make software more secure.

Removing the buffer overflow vulnerability is one step; it might require multiple steps to make a program more secure. If each of these steps is described as a program transformation, a number of such transformations can be applied to a program to introduce various types of protection.

Our paper starts with a comparison of the common practices for preventing buffer overflow attacks. Then we describe our program transformation methodology and the results of applying this on three open source C software. Finally, we argue how other security solutions can be described as program transformations.



## 2. Buffer Overflow Attacks

Buffer overflow is the oldest and the most common security vulnerability (Qualys 2007). To find out the relevance of this vulnerability on today's software, we surveyed the top ten most active projects in sourceforge.net (SourceForge.net 2007) in September 2007. Six projects use C or C++ as one of the languages, three use PHP and one uses Java. Among the six C/C++ projects, four have at least one buffer overflow vulnerability reported. These projects are *pidgin*, *gimp-print*, *licq* and *python*. *Ghostscript*, written in C, contributes to a buffer overflow vulnerability in *gv*, but it does not have a direct vulnerability. The only C++ based project not to have a buffer overflow vulnerability is *Crystal Space 3D SDK*.

Research efforts to mitigate buffer overflow attacks primarily focus on the detection of buffer overflow in source code. The programmer, after generating a list of potential buffer overflows using one of these detection tools, is in charge of making the changes in the source code. This approach has two problems. First, the coverage of the generated list is limited to the accuracy of the detection algorithm. There may be a lot of false positives or false negatives. Second, the programmers do not have a guideline how to change the source code without any side effects so that the buffer overflow is removed but no new vulnerability is introduced.

As a result, despite the effort of the software companies, commercial software are still not free from this vulnerability.

Buffer overflow attacks can be launched on the stack or the heap. The simplest attack is stack smashing (Aleph One 1996), where the attacker overwrites the return address on the stack and redirects the program counter to point to his malicious script. This is done by passing a long input buffer to a buffer handling routine. In C, the string routines do not check the buffer size while copying a source buffer to a destination buffer; this creates an opportunity to launch a buffer overflow attack. A very famous exploit of this vulnerability is the Morris worm, which used a vulnerability in the *fingerd* daemon (Eichin and Rochlis 1989).

Heap based buffer overflow attacks are more sophisticated (Conover 1999). This attack is launched on the dynamically allocated memory in the heap. For example, an attacker can overflow a dynamically allocated temporary file name pointer to make it point to a separate string, e.g. the command line argument $argv[1]$. The attacker can then supply important configuration files like $/root/.rhosts$ as the command line argument and corrupt the file, eventually gaining access to the system.

A language with garbage collection and bounds checking like Java and Smalltalk, does not have this vulnerability.

## 3. Buffer Overflow Detection and Prevention

The buffer overflow detection algorithms rely on static or dynamic analysis of programs. Static analysis tools range from very simple lexical analyzer (Viega et al. 2000) to tools performing integer analysis to approximate the pointer arithmetic in the C source code (Wagner et al. 2000; Dor et al. 2001). These algorithms often generate too many false positives. For example, one lexical analyzer based buffer overflow detection scheme checks each method and rates it as vulnerable or not vulnerable based on whether they use vulnerable buffer manipulation routines (e.g. $strcpy$ and $strcat$). A program is not vulnerable to buffer overflow attacks only because it is using an unsafe string routine; hence this approach returns many false positives. On the other hand, the integer analysis routines often analyze the program flow imprecisely because they make impractical assumptions (e.g. assuming C strings are only manipulated by $strcpy$ and $strcat$ functions) to make their algorithm efficient. In some cases, the programmer has to annotate the program (Dor et al. 2001), which is difficult.

Dynamic checking algorithms monitor memory access at runtime by keeping state information for each memory byte (Haugh and Bishop 2003; Hastings and Joyce 1992). These algorithms have a data storage overhead. Sometimes, the detection algorithm does not keep the state of the entire memory; it only keeps the critical information related to a function call in a private data area and checks this area when the function returns (Libenzi 2004). However, the private data area has to be implemented. Another technique that requires extra implementation uses markers to denote data endpoint in stack and heap (Cowan et al. 1998, 2003). The program dynamically examines the markers for corruption. A corrupt marker indicates an overflow in a buffer resident before the marker in memory.

The best way to prevent buffer overflow it to use a programming language that does not have the vulnerability; but this is often not the most feasible option. Another option is to rewrite the string I/O library to include support for array bounds checking. For example, string copy and concatenation functions $strcpy$ and $strcat$ are vulnerable to buffer overflow attacks. Safe alternatives of these functions are available as a library; e.g. $strncpy$ and $strncat$, or $strlcpy$ and $strlcat$ (Miller and de Raadt 1999) functions do not allow overwriting beyond the buffer bounds. Another option is to allow overwriting beyond the buffer bounds by dynamically resizing the destination string to hold the longer source string, e.g. the SafeStr (Messier and Viega) library. Also, C++ std::string (std::string) library provides safe I/O by dynamically resizing strings.

Special implementation of stacks can also prevent buffer overflow. Making the stack non-executable prevents buffer overflow (OpenWall Project), but some applications might need the execution feature of the stack. Another option is to split the control and data stack (Xu et al. 2002), but it is hard to calculate the function return address in this case.

## 4. Program Transformation to Remove Buffer Overflow

The buffer overflow attack prevention techniques are applicable to a software system in the design phase. It is not easy to apply these prevention techniques to transform a legacy program written in C to a program that is not vulnerable. We are describing a program transformation approach to remove buffer overflow vulnerability in existing C programs. Our program transformation follows the 'search and replace' approach; it searches for all the unsafe functions that may cause buffer overflow and replaces them with a safer alternative. We have written a Perl script to search and replace $strcat$ and $strcpy$ functions in C programs. We applied this simple transformation in three C programs. The transformed programs are not vulnerable to known buffer overflow attacks.

It is important to understand the significance of our contribution. Our program transformation script only replaces $strcat$ and $strcpy$ functions; other replacements are required to completely remove buffer overflow vulnerability. Developing a commercial quality tool is not our goal; however, developing such a tool would not be very difficult. We want to show that a security solution to remove buffer overflow vulnerabilities can be added on to an existing system by a program transformation.

Many of the security solutions are described as security patterns. Security patterns document best practices and

proven solutions for security problems. The program transformation to remove buffer overflow vulnerability follows the Safe Data Buffer security pattern. This section starts with a description of the security pattern. Then we describe the desirable properties of the replacement functions. This is followed by the details of the program transformation script. Finally, we describe the effect of the program transformation.

### 4.1 Safe Data Buffer Pattern

The Safe Data Buffer (Hafiz 2005) pattern provides a guideline to implement a safe data structure to prevent buffer overflow vulnerability in languages like C. C strings are NUL-terminated. A string function like $strcpy$ blindly copies all characters from the source string into the destination until it finds a NUL. This is vulnerable to buffer overflow attacks like 'smashing the stack' or 'overrun screw' (Aleph One 1996). The safe data buffer pattern suggests that every data buffer should have length information associated with it and the library should use this length information when it manipulates the buffer. Table 1 summarises the pattern.

**Table 1.** Safe Data Buffer Pattern

**Context**
You are designing a system in a programming language that does not have built-in array bounds checking.

**Problem**
Buffer overflow occurs when a process attempts to store data beyond the boundary of a fixed-length buffer. The problem is caused by bad programming practice. If every buffer handling routine checked allocated memory and operated within that memory bounds, buffer overflow would not occur. In practice, the buffer handling routines do not handle these tasks. An attacker exploiting a buffer overflow can execute arbitrary code and take complete control of the operating system. How can you design buffers that do not have this vulnerability?

**Forces**

- Programming languages that do not check for buffer bounds are used in practice.
- Programmers can prevent buffer overflow by checking buffer bounds every time they are using a buffer handling routine; instead they call the library functions without checking any preconditions.
- Buffer handling library functions do not check for buffer bounds.

**Solution**
Keep length information and allocated memory information with the buffer. In all buffer handling functions, check for length and available memory before updating the data buffer.

Length and memory information can be kept by creating a separate data structure. *qmail*, written in C, follows this approach (Hafiz 2005). *qmail* uses its own string library. Strings are not NUL-terminated. Instead, they are encapsulated in a structure named $stralloc$. This data structure keeps the length information of the string buffer. The $len$ field keeps the length of the buffer in bytes and the $a$ filed keeps the count of allocated bytes in the string.

```
typedef struct stralloc {
     char *s;
     unsigned int len;
     unsigned int a;
     }
```

All string-manipulation functions check if the input data fits the buffer before attempting to update the buffer. Another mail transfer agent, *sendmail X*, has a data structure named $sm\_str\_S$, that keeps length information (Aßmann 2004).

Creating a new data structure is not an essential part of the solution, changing the buffer handling functions is. The changed functions should check for length and available memory before updating a data buffer. For example, *qmail* author wrote the data buffer handling functions in order to incorporate the $stralloc$ data structure. The popular C function $strcpy$ has the following signature.

```
char* strcpy (char *destination,
                     const char *source);
```

In *qmail*, this function is replaced by the $stralloc\_copy$ function that has the following signature.

```
int stralloc_copy(stralloc *destination,
                       stralloc *source);
```

String functions can be rewritten without introducing a new data structure. For example, the $strcpy$ function is replaced by the $strlcpy$ function (Miller and de Raadt 1999).

```
size_t strlcpy ( char *destination,
         const char *source, size_t size);
```

The new function has one more parameter. The $size$ parameter takes the size of the destination and does not allow writing beyond the specified length. It is passively introducing length information; it does not require a new data structure.

In fact, some alternative safe libraries do not require an explicit length argument. The *libmib* library defines an alternative for the $strcpy$ function. The $astrcpy$ function does not require the length argument (Forrest J. Cavalier III).

```
char* astrcpy (char **destination_address,
                      chonst char *source);
```

This function takes the address of the destination and calculates the length information.

The $strcpy$ and $strcat$ functions are not the only string handling routines that are vulnerable. The string input functions like $gets$ and $sscanf$, and the formatted string printing functions like $sprintf$ and $vsprintf$ are also vulnerable. There are safe replacements for these functions. For example, the $sprintf$ function has the following signature.

```
int sprintf (char *str,
                  const char *format, ... );
```

This function might take additional arguments based on the $format$ argument. A safe replacement is the $asprintf$ function defined in the *libmib* library.

```
int asprintf (char **str_address,
              const char *format, ... );
```

The vulnerability introduced by the $strcpy$ and $strcat$ functions has been known for a long time. Our survey of the top ten most active projects shows that at least three of the six C/C++ projects (*Crystal Space 3D SDK*, *gimp-print*, and *licq*) used $strcpy$ and $strcat$ functions initially, but changed to safer C/C++ string libraries when buffer overflow attacks were reported. This replacement was done by manual code inspection. Our program transformation automatically replaces the string functions.

### 4.2 Choice of the Library

The library that will be used as the substitutes should have functions with clear semantics. The replacement functions should have similar signatures or at least they should be as close as possible. The new library should be easy to adopt and it should add minimal computational overhead. The library should be introduced with minimal program transformation. Most importantly, the library should be safe, i.e. it should not introduce any new vulnerabilities.

The safe library options have their own trade offs. For example, the $strncpy$ and $strncat$ functions have complicated semantics, making them difficult to use. The $strcpy$ function NUL-terminates the string. It is a common misconception that its safe alternative $strncpy$ does the same. The $strncpy$ function NUL-terminates only if the length of the source string is less than the size parameter. The safest way to use $strncpy$ is to NUL-terminate by explicit assignment after every function call. But programmers often rely on the functions' capability. In large projects, it might happen that some strings are NUL-terminated after the use of $strncpy$ function, while others are not. This causes a maintenance nightmare.

The $strncat$ function has another problem. A common mistake that programmers make while using the $strncat$ functions is that they use an incorrect size parameter. $strncat$ guarantees the NUL-termination, but NUL byte should not be counted as part of the size parameter. Again, the size parameter reflects the amount of space available, instead of the size of the destination string. The programmers have to calculate this value before using the function, which is error-prone. It recreates the original problem that $strcpy$ and $strcat$ had. If used correctly, $strcpy$ and $strcat$ do not have buffer overflow vulnerability. But programmers make mistaken assumptions about the size of the source and the destination buffer. The safe library option should not rely on the programmers' calculation of string size; it should not overflow a buffer when the programmers make mistakes.

*qmail* provides a safe string copy and concatenation function. These functions take the same number of parameters as $strcpy$ and $strcat$, but these parameters have a different type, the $stralloc$ data structure of *qmail*. $stralloc\_copy$ and $stralloc\_cat$ are easy to use but the programmer has to declare and allocate the buffer data structure before using them. All the instances where the buffer is used would now have to use the $stralloc$ data structure. Additionally, the library functions to initialize the buffer, e.g. $malloc$, should be replaced with the $stralloc\_ready$ function. This means that the library is not program transformation friendly.

A library does not have to ensure that the related program transformation is easy. However, if there are several library options, the one that involves an easier program transformation should be chosen for efficiency. The $strlcpy$ and $strlcat$ functions retain the original parameters and add one length parameter. Both $strlcpy$ and $strlcat$ take the full size of the destination string as the length parameter, which is easy to calculate. Both the functions have clear semantics and guarantee that the destination string will be NUL-terminated. More importantly, operating systems have libraries that implement these functions. For example, the *glib 2.0* library implements these functions as $g\_strlcat$ and $g\_strlcpy$. These functions have the same semantics as $strlcpy$ and $strlcat$. The *glib* library also provides safe implementation of the string print functions, e.g. $g\_printf$, $g\_fprintf$, $g\_sprintf$, $g\_vprintf$, $g\_vfprintf$, $g\_vsprintf$, $g\_vasprintf$, etc.

The *libmib* library is another suitable option. It includes the $astrcpy$ and $astrcat$ functions. The library also provides safe alternatives for string print functions, $avsprintf$, $asprintf$, $afgets$, etc.

### 4.3 Details of the Program Transformation Script

Our proof-of-concept Perl implementation reads the source code, searches for patterns to identify the $strcpy$ and $strcat$ functions, and replaces them with the functions provided in the *glib 2.0* library. The patterns are not exhaustive. There are other patterns that we did not implement in our Perl script. Furthermore, a commercial quality tool should replace other unsafe functions with safe alternatives.

Our tool reads the source code line by line. When it finds the text $strcat$ or $strcpy$, it identifies the source and destination parameters by pattern matching and then replaces them with the $g\_strlcat$ or $g\_strlcpy$ library. The patterns are described in table 2.

The replacement patterns in table 2 use both the $sizeof$ and the $malloc\_usable\_size$ function. The $sizeof$ function is only good for determining the size of a buffer that is statically allocated. For the dynamically allocated variables, the $sizeof$ function returns the size of the pointer variable pointing to the first memory address. For these variables, the $malloc\_usable\_size$ function returns the size. The $malloc\_usable\_size$ function, on the other hand cannot be used with statically allocated buffers; it produces a segmentation fault when done so. Hence, the Perl script has to identify which variables are statically allocated and which are dynamically allocated.

We have a dirty fix in our script. The script identifies all the dynamically allocated variables within each functions

| Pattern Name | Parameters | Search Pattern | Replacement Pattern |
|---|---|---|---|
| 1. *Function with two variables* | Two parameters each of which are variables | `strcpy (var1, var2);` | `g_strlcpy (var1, var2, sizeof(var1));` |
| 2. *Function with string* | The source is a string literal, the destination is a variable | `strcpy (var1, "C Str");` | `g_strlcpy (var1, "C str", sizeof(var1));` |
| 3. *Function with pointer* | Any of the source or destination parameter is a pointer | `strcpy (*var1, "C Str");` | `g_strlcpy (*var1, "C str", malloc_usable_size(*var1));` |
| 4. *Function with array* | Any of the source or destination parameter is an array | `strcpy (var1[index], var2);` | `g_strlcpy (var1[index], var2, sizeof(var1));` |
| 5. *Function with structure* | Any of the source or destination parameter is a member variable in a structure | `strcpy (struct1.var1, var2);` | `g_strlcpy (struct1.var1, var2, sizeof(struct1.var1));` |
| 6. *Function with type cast* | Any of the parameters have a type cast | `strcpy ((char *) var1, var2);` | `g_strlcpy ((char *) var1, var2, malloc_usable_size((char *) var1));` |
| 7. *Function with pointer arithmetic* | Any of the parameters have pointer arithmetic with integer | `strcpy (*var1 + intvalue, var2);` | `g_strlcpy (*var1 + intvalue, var2, malloc_usable_size(*var1) - intvalue);` |
| 8. *Function with memory address* | Any of the parameters contain a memory address | `strcpy (&var1, var2);` | `g_strlcpy (&var1, var2, malloc_usable_size(&var1));` |

**Table 2.** Patterns for $strcat$ and $strcpy$ functions

and keeps them in an array. In all the projects where we have applied the program transformation, the curly braces that start and end a function are in a separate line with no indentation. We have looked for this pattern. Whenever we encounter an end curly brace ('}') as a first character in a line, we clear the variable list (i.e. the array) because we have reached the end marker of the function. Then we start to add newly encountered variables in the empty list, doing so until we reach another curly brace marking a function endpoint. This will not work in a project that uses no indentation and all the curly braces, including the ones that denote the start and end of a loop, are the first character in a line. Such an indentation scheme is very unlikely; but even in that case, a sophisticated tool can be developed to identify the function boundary.

To identify the dynamically allocated variables, we search for the 'assignment statement followed by an alloc keyword' pattern. These variables are stored in the list. During the search and replace task, the first variable in the $strcpy$ or $strcat$ function is identified and matched with the entries in the list. If it is a match, the size of the variable is calculated with the $malloc\_usable\_size$ function; otherwise the $sizeof$ function is used.

When the new functions are included, the corresponding header files have to be loaded at compile time and the library has to be loaded at link time. In our case, we did not include the $glib/gstrfuncs.h$ file in all the C source files. The quickest fix is to include the library during link time. Our script modifies the $Makefile$s and the $config.status$ file of the project. In each of these files, we look for the pattern '`gcc`' as a single word and replace it with '`gcc` `` `pkg-config --libs glib-2.0` ``'. This ensures that the library is loaded during link time. Instead of updating all the make files, it is possible to parse the make hierarchy and intelligently update. However, our solution is the simplest and the most perfect for the scenario.

String search and replace provides a quick solution, but a commercial quality tool can be developed for this program transformation by utilizing the parse trees and abstract syntax trees of the C program, or by parsing the make files and configuration files. Our goal is to show that security solutions can be added as program transformations, and our crude transformation is sufficient to make the point.

### 4.4 Effect of Program Transformation

The destination buffer is overflown when it is smaller than the source buffer. The functions that we used as replacement prohibits this by truncating the source buffer when it is larger than the destination buffer. This makes the program behave in a different manner because the data is corrupted. But this is not a problem, because the buffer overflow attacks are only exploited by malicious users. When the attackers produce the corrupt truncated data, they will get a different behavior from the program. Many buffer overflow exploits ask a program to open a malformed file. In this case, the program will get a truncated version of the file name or some file parameter and will fail to recognize it, thereby stating that it is unable to open the file. Again, consider a mail transfer agent program receiving a mail with a long header or body that corrupts the buffer. The transformed program will truncate the data and produce garbage mail that can later be removed by a user's spam filter.

Buffer overflow attacks are attempted by people with malicious intent. The solution is to give them malformed data response and continue to run the service. The resources employed by the attacker is wasted, and he is also not getting the intended service. The good users are not punished with bad data because they do not misbehave. This punishment orientation is how human society retains order, and the same lesson can be adopted here to introduce protection against buffer overflow.

## 5. Case Study

We have applied the program transformation in three open source C programs: a pdf/ps file viewer (*gv*), a zip library (*zziplib*) and the implementation of the kerberos protocol

(*kerberos*). These programs are selected because buffer overflow exploits have been reported in the programs recently, and the exploit codes are available.

The program transformation impacts three types of files. First, the C files are searched for instances of $strcpy$ and $strcat$, and are replaced with $g\_strlcat$ and $g\_strlcpy$. Second, the Makefiles are searched for the word 'gcc' and are replaced with 'gcc `pkg-config --libs glib-2.0`'. Finally, the $config.status$ file is searched for the word 'gcc', and replaced with the same pattern.

### 5.1 Gnu Viewer

*gv* is used to view and navigate through postscript and pdf documents on an X display. *gv* provides a graphical user interface for the ghostscript interpreter. It is available for Unix and Linux operating systems. We examined version 3.6.2 of *gv*. It contains 46 C files with 27000 lines of C code.

Renaud Lifchitz reported a stack buffer overflow vulnerability on November 2006 (Bugtraq ID 20978). *gv* versions 3.5.8, 3.6, 3.6.1 and 3.6.2 are vulnerable. An attacker convinces a person to use *gv* to view a malformed ps file; the result is a segmentation fault. The error is in the $ps\_gettext$ function defined in the $ps.c$ file. Line 1382 of the file contains a $strcpy$ call which causes the buffer overflow.

Our Perl script replaces the unsafe $strcpy$ and $strcat$ functions. There are 37 instances of $strcpy$ and 51 instances of $strcat$ in the source code. The Perl script replaces 86 of these 88 instances. Our tool is not sophisticated enough to identify the parameter variables in the two remaining cases. The first case is easy to fix,

```
strcpy(tmp,&(label[progress]));
```

Our Perl script misses it because the second variable is enclosed by a parentheses, which we have not specified in our rule. For the second case, the second variable is an expression which is complex to describe with regular expression.

```
strcat(msg, count == 0 ?
           ", expecting `" : " or `");
```

In both these cases, we made the changes by hand and compiled the resulting program.

4 configuration file changes are made. For example line 553 of the $config.status$ file,

```
s,@ac_ct_CC@,|#_!!_#|gcc,g
```

is replaced with

```
s,@ac_ct_CC@,|#_!!_#|gcc
       `pkg-config --libs glib-2.0`,g
```

The resultant program compiles without any errors, and shows the same behavior. When *gv* tries to open the malformed postscript file, there is no segmentation fault.

The source code has other transformation opportunities. We did not change these because the buffer overflow protection only requires the change on the $strcat$ and $strcpy$ files. There are 7 instances of $strncpy$, 8 instances of $strncat$, 47 instances of different flavors of $scanf$, and 211 instances of different types of $printf$ commands that should be replaced, to make the program more robust.

### 5.2 *zziplib*

*zziplib* is a lightweight zip library to extract data from files archived in a single zip file. The current version of *zziplib* library is version 0.13.49, which was produced in response to a buffer overflow vulnerability reported in version 0.13.48 in March 2007 (Bugtraq ID 23013). We have transformed version 0.13.47 of the *zziplib* program. This version has 7346 lines of C code in 33 C files.

The stack based buffer overflow vulnerability occurs because *zziplib* fails to properly bounds-check user-supplied input before copying it to an insufficiently sized buffer. The $zzip/file.c$ file defines the $zzip\_open\_shared\_io$ function that takes a file name parameter named $filename$ and copies it to a buffer $basename$.

```
char basename[PATH_MAX];
char* p;
strcpy (basename, filename);
```

[Lines 726-728 from $zzip/file.c$ file]

When an attacker makes the user open a zip file with long filename, the program crashes. Exploiting the issue may allow attackers to execute arbitrary machine code in the context of the application using the library. Even a failed exploit attempt would cause a denial of service scenario, because the program crashes.

The Perl script changes all the 5 instances of $strcpy$ and $strcat$ in the source code. Also, the $Makefile$ is changed in 15 places to include the library information. The transformed program successfully compiles and displays the same behavior. It does not crash when a long file name is presented. Rather the file name is truncated when the buffer is copied. The program fails to open the file but it does not crash.

*zziplib* is chosen for two reasons. First, it is small; hence all the changes can be manually inspected for correctness. Second, the subsequent version of *zziplib* is available that removes the buffer overflow vulnerability. Version 0.13.49 replaces the previous code segment with,

```
char basename[PATH_MAX];
char* p;
int filename_len = strlen (filename);
if (filename_len >= PATH_MAX)
    { errno = ENAMETOOLONG; return 0; }
memcpy (basename, filename, filename_len+1);
```

[Lines 741-745 from $zzip/file.c$ file]

Our version just replaces line 728 from previous version with the $g\_strlcpy$ alternative and achieves the same effect.

Other unsafe functions are not replaced. There are 121 instances of various flavors of $printf$ functions and 1 instance

| Pattern Description | Search Pattern | Replacement Pattern |
|---|---|---|
| 1. *Function with expression* - Evaluates an expression to find the parameter. | `strcat (var1, (bool_expression? var2_a : var2_b));` | `g_strlcat (var1, (bool_expression? var2_a : var2_b), sizeof (var1));` |
| 2. *Function with pointer arithmetic* - Pointer arithmetic with variables | `strcat (var1 + varinc, var2);` | `g_strlcat (var1 + varinc, var2, sizeof(var1) - varinc);` |
| 3. *Function with complex array indexes* - Array indexes are complex | `strcat (var1, var2[index + 2]);` | `g_strlcat (var1, var2[index + 2], sizeof(var1));` |
| 4. *Function with non-standard typecast* - Parameters with non-standard type cast | `strcat (var1, (varcast *) var2);` | `g_strlcat (var1, (varcast*) var2, sizeof(var1));` |
| 5. *Function with a function parameter* - Parameter is the return value of a function | `strcat (var1, fn_returning_var2(...));` | `g_strlcat (var1, fn_returning_var2(), sizeof (var1));` |
| 6. *Function with special operators* - Special operators | `strcat (var1, ++var2);` | `g_strlcat (var1, ++var2, sizeof(var1));` |
| 7. *Functions used in assignment* - The return value is assigned to a variable | The $strcat$ function returns the destination buffer while $g\_strlcat$ returns the size of the destination buffer after concatenation. They cannot be substituted in an assignment statement. | |
| 8. *Function not ending in one line* - Functions written in multiple lines | The $strcat$ functions may be written in multiple lines. Since our script only reads one line every time, this cannot be detected. | |

**Table 3.** Patterns that were not covered for $strcat$ and $strcpy$ functions

of $strncpy$ function. Replacing them would make the program more robust.

### 5.3 *Kerberos*

*Kerberos* is an implementation of the network authentication protocol named Kerberos (Neuman and Ts'o 1994). The protocol is designed to provide strong authentication for client/server applications by using secret key cryptography. A number of recent buffer overflow vulnerabilities have been reported for different versions of Kerberos (Bugtraq ID 23285; Bugtraq ID 24653). The $kadmind$ server stack buffer overflow vulnerability was reported in April 2007 (Bugtraq ID 23285). An attacker can exploit this issue to execute arbitrary code with administrative privileges. A successful attack can result in the complete compromise of the application. Failed attempts results in denial-of-service conditions. Besides, all $kadmind$ servers run on the master Kerberos server. Since the master server holds the KDC principal and policy database, an attack may not only compromise the affected computer, but could also compromise multiple hosts that use the server for authentication. *Kerberos 5 kadmind 1.6* and prior versions are vulnerable.

We have transformed *kerberos 5 kadmind 1.5.1*. This contains 1045 C programs with 369,442 lines of code, 129 Makefiles containing 102,018 lines of code, and 11 $config.status$ files containing 9718 lines of code. Our Perl script converts 470 $strcat$ and $strcpy$ instances. It also replaces 110 instances of the word 'gcc' in the Makefiles and 5 instances of the same word in the $config.status$ files. 35 instances of $strcpy$ and 3 instances of $strcat$ are not modified. These instances have complex patterns for the parameters. Table 3 lists these patterns. It is not difficult to develop a tool that also finds these patterns, but this is beyond our scope.

The compiled version shows the same functionality as the original version. However, we could not test whether the new version does not have the buffer overflow vulnerability. The buffer overflow scenario could not be simulated because of the lack of information about the exploit code. In fact, the buffer overflow in this case is not because of $strcpy$, but because of the $vsprintf$ function. The buffer overflow error reported is in the $klog\_vsyslog$ function in the $lib/kadm5/logger.c$ file [Some irrelevant code portion has been removed].

```
static int
klog_vsyslog(int priority, const
             char *format, va_list arglist)
{
  char outbuf[KRB5_KLOG_MAX_ERRMSG_SIZE];
  char *syslogp;
  strncpy(outbuf, ctime(&now) + 4, 15);
  cp += 15;
  syslogp = &outbuf[strlen(outbuf)];
  vsprintf(syslogp, format, arglist);
```

The buffer overflow occurs because the $arglist$ parameter is not checked before it is used in $vsprintf$. Nevertheless, we have chosen kerberos to apply our transformation to a larger program and show that the program behavior does not change.

There are 4487 instances of different type of $printf$ functions, 313 instances of $strncpy$ functions and 155 instances of $strncat$ function instances in the source code that should be removed to make the program safe from buffer overflow vulnerabilities.

## 6. Improved Tool for Program Transformation

We have shown a proof-of-concept implementation for introducing buffer overflow protection in C programs. A commercial quality tool for the program transformation can be developed using the following guidelines.

Our implementation only replaces the $strcat$ and $strcpy$ functions. It should be extended to replace the unsafe string input and print functions. We have already discussed the problems associated with the $strncat$ and $strncpy$ functions. These should also be replaced. In this case, the size parameter has to be calculated. The simplest way to do it is to ignore the size parameter in the $strncpy$ signature, and

calculate the size in the $g\_strlcpy$ function by applying the $malloc\_usable\_size$ or the $sizeof$ function over the destination string parameter.

Our implementation does not match all the parameter patterns for $strcat$ and $strcpy$ functions. Our lexical analysis based approach should be replaced with an improved approach that uses the parse tree or the abstract syntax tree. An approach in this direction is the Gemini tool (Dahn and Mancoridis 2003). It uses TXL to transform all the stack allocated buffers in a C program to heap allocated buffers, because heap allocated buffers are less vulnerable against buffer overflow attacks. TXL has a grammar for C and it can be used for more efficient source code transformation (Cordy 2006) than our approach.

Our program transformation is only dependent on replacing the library. Bad function calls are not the only candidates causing buffer overflow. In many cases pointers are directly manipulated in C programs, and bad pointer arithmetic can lead to a buffer overflow vulnerability. Again, Gemini (Dahn and Mancoridis 2003) tries to solve it by allocating buffers on the heap. However, transforming the stack based buffers to heap based buffers does not entirely remove the buffer overflow vulnerability. A combination of the two approaches, replacing the functions and relocating the buffers, would be more effective.

Our understanding is that programmers are more careful when they are manipulating buffers directly. Bad assumptions about the library functions are the primary cause of buffer overflow vulnerability. Nevertheless, a more detailed program transformation technique should analyze the pointers and add necessary constraints at the program points where bad pointer arithmetic can happen.

## 7. Other Program Transformations for Security

Eliminating buffer flow vulnerability is not the only security solution that can be automated by program transformations. In fact, most security solutions seem like they could be automated. For example, the security solution described in the single access point pattern suggests that the access points in a system should be minimized so that the system can not be attacked from multiple entry points (Yoder and Barcalow 1997). This can be done by making a Façade (Gamma et al. 1995) in the object-oriented world, or by introducing a gatekeeper component that also has the same API as the original program in C. The gatekeeper should run with minimum privileges and delegate the incoming requests only. For example, the *TIS firewall toolkit* had an SMTP service that acted as the wrapper of a *sendmail* program. The toolkit intercepts requests from the outside world and then forwards it to the *sendmail* program. The SMTP service runs as an unprivileged user; hence outside attackers cannot compromise the whole system by compromising the gatekeeper process.

Different security solutions are applicable at different levels of a system. For example, one way to secure a monolithic process is to partition it into multiple processes. Compartmentalization (Viega and McGraw 2002) limits the exploits of an attacker to a single compromised process. A monolithic process is compartmentalized by describing the partitions and slicing the program based on these descriptions. On the other hand, the buffer overflow problem is solved by removing the vulnerable function calls that create a buffer overflow. This is done by searching for the vulnerable function calls in a program and replacing them with a secure alternative. These two approaches work at different granularities of the system; partitioning works on the whole program and needs a description of how the system is to be partitioned, while eliminating buffer overflow operates on each part of the program individually.

There are some approaches to partition a system for privilege separation (Kilpatrick 2003; Brumley and Song 2004), but these approaches only split the program into a privileged and unprivleged partition instead of partitioning into multiple processes based on the task distribution. Compartmentalization has to consider how the tasks are distributed between the partitioned processes (Veryard and Ward 2001), and apply the least privilege principle (Saltzer and Schroeder 1975). The program transformation has to take these issues into account, making it a hard problem.

Sometimes a program transformation is not enough; changes have to be made in a program's runtime environment. Suppose, a software engineer wants to run an existing program inside a chroot jail (Friedl 2002) to limit the exploits of an attacker. This is done by setting the jail with a *chroot()* (Linux Man Pages) call passing a pathname as parameter. After the call, the pathname becomes the root directory ('/') for the process. Thus, files outside the specified directory structure are considered 'safe' from the jailed process. A number of things have to be considered prior to the setup of a chroot jail. First, a replica of the file system is created inside the chroot directory. This is typically done by creating the $/etc$, $/usr$, $/dev$ and the $/tmp$ directory. Then, all the required libraries and resources of the jailed process have to be copied inside the jail directory with exact file system hierarchy. Finally, the process inside the jail is run as a user with limited privileges because root privilege enables an attacker to break the jail (Friedl 2002). It is possible to automate these steps. The program transformation introduces the *chroot()* call only; but the task of setting up the necessary environment should also be automated.

Even if the security solutions can be described as program transformations, applying one solution does not guarantee security. A system should have defense in depth (Viega and McGraw 2002), i.e. it should have multiple layers of security tactics instead of a single security strategy. An example is the architecture of *qmail*, a secure MTA. At the process level, *qmail* is compartmentalized into modules which limits

the exploits of an attacker after a security break-in. Compartmentalization also makes each module simpler; a person can inspect the modules for correctness. Second, *qmail* communicates with the outside world with a few chosen processes that run as a user with limited privileges. This follows the single access point pattern. Third, the low-level coding patterns in *qmail* eliminate important classes of errors such as buffer overflows. Applying one security solution is not enough, multiple solutions have to be applied in an order.

The order in which the program transformations are applied is probably important. Many sequences might not make sense at all. Research on security patterns might be useful in identifying an appropriate order. Security patterns are organized as a pattern language (Hafiz et al. 2007). The pattern language provides a guideline about the order in which the patterns, i.e. the security solutions, should be applied. This guideline can be used to determine the sequence of program transformations.

## 8. Conclusion

This paper shows that a particular kind of security flaw, buffer overflow, can be eliminated by program transformation. There are probably many more security flaws that can be eliminated in this way. This is important, because new security threats are continuously arising, and old software needs to be modified to deal with these threats. As new security threats emerge, new program transformations should be made available to make software fit for the changed reality.

## Acknowledgments

The authors would like to thank Paul Adamczyk, Jeffrey Overbey, Maurice Rabb, Nicholas Chen and Farhana Hafiz for providing valuable feedback. Also, the authors would like to thank the people from the mailing lists of all the open source projects listed here for providing information about the projects.

## References

Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49):File 14, 1996.

J. P. Anderson. Computer security technology planning study. Technical report, ESD-TR-73-51, October 1972.

Claus Aßmann. Sendmail X: Requirements, architecture, functional specification, implementation, and performance. `http://www.sendmail.org/ ca/email/sm-X/design -2004-09-29/main/main.html`, 2004.

David Brumley and Dawn Xiaodong Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, pages 57–72. USENIX, 2004.

Bugtraq ID 20978. GNU gv stack buffer overflow vulnerability. `http://www.securityfocus.com/bid/20978`.

Bugtraq ID 23013. zzipLib zzip_open_shared_io stack buffer overflow vulnerability. `http://www.securityfocus.com/bid/23013`.

Bugtraq ID 23285. MIT kerberos 5 kadmind server stack buffer overflow vulnerability. `http://www.securityfocus.com/bid/23285/`.

Bugtraq ID 24653. MIT kerberos 5 kadmind server rename_principal_2_svc function stack buffer overflow vulnerability. `http://www.securityfocus.com/bid/24653`.

Matt Conover. w00w00 on heap overflows. `http://www.w00w00.org/files/articles/heaptut.txt`, January 1999.

James R. Cordy. Source transformation, analysis and generation in TXL. In John Hatcliff and Frank Tip, editors, *PEPM*, pages 1–11. ACM, 2006. ISBN 1-59593-196-1.

Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In USENIX, editor, *Seventh USENIX Security Symposium proceedings: conference proceedings: San Antonio, Texas, January 26–29, 1998*, pages ??–??, pub-USENIX:adr, 1998. USENIX. ISBN 1-880446-92-8.

Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard™: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104. USENIX, August 2003.

Christopher Dahn and Spiros Mancoridis. Using program transformation to secure c programs against buffer overflows. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 323, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2027-8.

Nurit Dor, Michael Rodeh, and Shmuel Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In Patrick Cousot, editor, *SAS*, volume 2126 of *Lecture Notes in Computer Science*, pages 194–212. Springer, 2001. ISBN 3-540-42314-1.

Mark W. Eichin and Jon A. Rochlis. With microscope and tweezers: An analysis of the Internet virus of November 1988. In *IEEE Security and Privacy*, pages 326–343, 1989.

Forrest J. Cavalier III. Libmib allocated string functions. `http://www.mibsoftware.com/libmib/astring/`.

Stephen Friedl. Go directly to jail. *Linux Magazine*, December 2002.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

Munawar Hafiz. Security architecture of Mail Transfer Agents. Master's thesis, University of Illinois at Urbana-Champaign, 2005.

Munawar Hafiz, Paul Adamczyk, and Ralph E. Johnson. Organizing security patterns. *IEEE Software*, 24(4):52–60, July/August 2007.

R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 USENIX*

*Conference*, pages 125–136, 1992.

Eric Haugh and Matt Bishop. Testing C programs for buffer overflow vulnerabilities. In *NDSS*. The Internet Society, 2003. ISBN 1-891562-16-9; 1-891562-15-0.

Douglas Kilpatrick. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 273–284. USENIX, 2003. ISBN 1-931971-11-0.

Davide Libenzi. Guarded memory move (GMM), February 10 2004. URL `http://citeseer.ist.psu.edu/637889.html`; `http://www.xmailserver.org/gmm.pdf`.

Linux Man Pages. chroot. `http://linux.die.net/man/2/chroot`.

Matt Messier and John Viega. Safe c string library v1.0.3. `http://www.zork.org/safestr/safestr.html`.

Todd C. Miller and Theo de Raadt. `strlcpy` and `strlcat` — consistent, safe, string copy and concatenation. In USENIX, editor, *Usenix Annual Technical Conference. June 6–11, 1999. Monterey, California, USA*, pages ??–??, pub-USENIX:adr, 1999. USENIX. ISBN 1-880446-33-2. URL `http://www.openbsd.org/papers/strlcpy-paper.ps`.

B. Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, September 1994.

OpenWall Project. Linux kernel patch from the openwall project. `http://www.openwall.com/linux/`.

Qualys. Top ten vulnerabilities. `http://www.qualys.com/research/rnd/top10`, September 2007.

Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *procieee*, 63(19):1278–1308, September 1975.

SourceForge.net. Most active projects - all time. `http://sourceforge.net/top/mostactive.php`, September 2007.

std::string. C++ std::string. `https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/coding/295.html`.

Richard Veryard and Adrian Ward. Trusting components and services. `http://cbdiforum.com/report_summary.php3?topic_id=23&report=411&order=author&start_rec=15`, 2001.

John Viega and Gary McGraw. *Building secure software: How to avoid security problems the right way*. Addison-Wesley Publishing Co., Indianoplis, IN, 2002.

John Viega, J. T. Bloch, Y. Kohno, and Gary McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *ACSAC*, page 257. IEEE Computer Society, 2000. ISBN 0-7695-0859-6.

David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS '00)*, pages 3–17, San Diego, CA, February 2000. Internet Society.

J. Xu, Z. Kalbarczyk, S. Patel, and R. Iyer. Architecture support for defending against buffer overflow attacks, 2002. URL `citeseer.ist.psu.edu/xu02architecture.html`.

J. Yoder and J. Barcalow. Architectural patterns for enabling application security. In Proceedings of the 4th Conference on Patterns Language of Programming (PLoP'97)., 1997.