Using Program Transformation to Secure C Programs Against Buffer Overflows

Christopher Dahn, Spiros Mancoridis
Department of Computer Science
College of Engineering
Drexel University, Philadelphia, PA, USA
{Christopher.Stephen.Dahn, Spiros.Mancoridis}@drexel.edu

Abstract

Buffer overflows are the most common source of security vulnerabilities in C programs. This class of vulnerability, which is found in both legacy and modern software, costs the software industry hundreds of millions of dollars per year.

The most common type of buffer overflow is the run-time stack overflow. It is common because programmers often use stack allocated arrays. This enables the attacker to change a program's control flow by writing beyond the boundary of an array onto a return address on the run-time stack. If the arrays are repositioned to the heap at compile time, none of these attacks succeed. Furthermore, repositioning buffers to the heap should perturb the heap memory enough to prevent many heap overflows as well.

We have created a tool called Gemini that repositions stack allocated arrays at compile time using TXL. The transformation preserves the semantics of the program with a small performance penalty. This paper discusses the semantics-preserving transformation of stack allocated arrays to heap allocated "pointers to arrays". A program that is amenable to a buffer overflow attack and several Linux programs are used as examples to demonstrate the effectiveness and overhead of our technique.

1. Introduction

C is a widely used programming language for critical software (e.g., operating systems and system software). Most of the software that is bundled with Linux and Sun Solaris are written in C. Furthermore, the most popular servers on the Internet for e-mail, the World Wide Web, and the Domain Name System are implemented in C [3, 18, 22].

C programmers often use arrays to store data gathered from external input. Stack allocated arrays are automatic variables, hence they are allocated and de-allocated during run-time without programmer intervention. This is convenient since the input is often used immediately [8]. Despite their convenience, stack allocated arrays are vulnerable to buffer overflow attacks. Fortunately, allocating all arrays to the heap can mitigate such attacks.

Stack buffer overflows are the most common form of security vulnerability found in C programs [19]. This vulnerability alone costs industry hundreds of millions of dollars per year [1]. For example, *bind*, the software responsible for 95% of the Domain Name System, was discovered to contain a buffer overflow as recently as November, 2002 [3, 2]. After discovery of a vulnerability in infrastructure-critical software, many man hours of software analysis, reinstallation, and testing are required to fix it.

Moving stack allocated arrays to the heap accomplishes two things. First, it disrupts the attack vectors of known stack buffer overflow exploits and all future stack buffer overflow exploits. Second, it can disturb the heap memory enough to eliminate known heap buffer overflow attack vectors also. Moving a stack allocated array to the heap does not fix the bug that causes the buffer overflow, it only prevents the overflow from providing the attacker with elevated privileges, such as a command shell. This leads to fewer vulnerabilities in the long run since it is very difficult, and in many cases impossible, for an attacker to leverage a heap buffer overflow [19].

In C, a heap allocated buffer is actually a pointer to contiguous memory. Pointers are not automatic variables, hence they require explicit memory management by the programmer. The added complication of explicit memory management often leads to bugs such as uninitialized pointers and memory leaks.

Memory management can be automated by a program that transforms arrays into "pointers to arrays". Such a program should preserve the semantics of the original program so that the transformation is transparent. In C, this is a problem since arrays and pointers are not equivalent types.

Preserving the semantics of the program after the transformation allows code to be developed using conventional programming practices (i.e., allocating certain buffers on the stack). Furthermore, maintenance and debugging need not be hampered by the prolific use of pointers. Rather, the code is automatically transformed to use heap allocated “pointers to arrays” immediately prior to compilation.

We have created a tool called Gemini that uses TXL rules to transform stack allocated arrays into heap allocated “pointers to arrays” automatically [10]. This transformation preserves the semantics of the original program, allowing it to be inserted into the end of the development process transparently and with a small amount of run-time overhead.

The remainder of this paper is structured as follows: Section 2 outlines related research, Section 3 discusses the differences between arrays and pointers in C, Section 4 outlines the transformation process, Section 5 presents an example that demonstrates the effectiveness and overhead of our technique, Section 6 describes the limitations of this work and our future plans to overcome them, and Section 7 outlines the conclusions of this work.

2. Related Research

This work is related to two major areas of research. The first is software security, specifically as it applies to buffer overflow vulnerabilities in code. The second is the use of source code transformation for code re-engineering.

2.1 Buffer Overflows

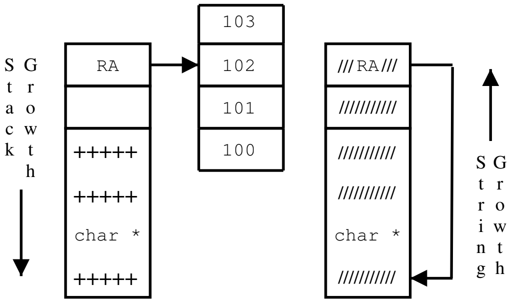


Figure 1. A stack buffer overflow

Buffer overflows may occur when a fixed size memory allocation is used to store a variable-size data entry. There are conflicts when the variable-size data entry overruns the bounds of the fixed-size memory. In Figure 1, ‘+++++’ represents valid data in the buffer and ‘/////’ represents the attacker’s data. These overflows are typically exploited by entering a string that is larger than the buffer assigned to hold it. If the return address (RA) is part of the overwritten run-time stack, an attacker may execute arbitrary code, such as spawning a remote terminal session [15, 19]. This is discussed in more detail in Section 5.1.

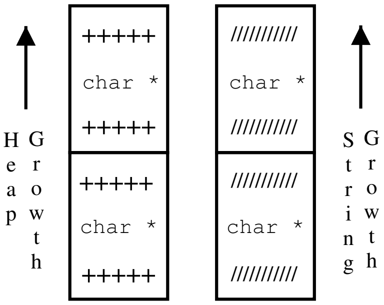


Figure 2. A heap buffer overflow

In Figure 2, the same buffer overflow is executed against a heap allocated buffer. Unlike the stack, the heap does not contain return addresses, making it harder to change the program’s control flow [19]. This particular overflow will overwrite the data in the second buffer, which may lead to erroneous behavior during program execution. However, such behavior is unlikely to be a security vulnerability [19].

2.1.1 Technologies to Detect & Prevent Buffer Overflows

Some security tools, such as Splint [13], perform static analysis to find code that is likely to be vulnerable. Unlike our technique, however, they require programmers to annotate their source code with constraints. Not all of the existing source code analysis tools require code annotations, however. Flawfinder [23], RATS [16], and ITS4 [4] are all tools that examine source code and report possible weaknesses.

An overview of these tools, along with a comparison of their capabilities, can be found in a Linux Journal article [14]. In general, these tools direct the attention of code auditors to C/C++ functions that are known to be associated with security problems, such as buffer overflows, and produce a list of vulnerable code statements.

StackGuard [7] has been reasonably successful at reporting buffer overflows immediately after they happen at runtime. Specifically, StackGuard inserts code into the application at compile time and a ‘canary’ value just before the return addresses on the run-time stack. When the function returns, the added code checks if this canary value is still in place. If the canary value is no longer present, a buffer overflow must have occurred. When this happens, the application terminates with a notification.

A similar solution to StackGuard is StackGhost [9]. StackGhost is a modification to the OpenBSD 2.8 kernel on the Sun Sparc architecture. This modification assigns the return address of all processes in a way that exposes modified return addresses. One way this is accomplished is by encrypting the return address.

Richard Jones and Paul Kelly patched the GNU C Compiler (GCC) to perform bounds checking at run-time [11]. This approach protects static, stack and heap allocated buffers. Every buffer has extra information stored in a table, such as its bounds. This table is then referenced to determine if buffer accesses are legal. However, the overhead generated by these references can increase the run-time of the application by several orders of magnitude [11].

A way to avoid the side effects of an exploited vulnerability is to disallow the execution of the run-time stack. This prevents executable code, such as shell instructions that may have been placed on the stack during a buffer overflow, from being executed.

One way to get around a non-executable run-time stack is to perform a heap overflow, followed by a stack overflow. The heap overflow is used to insert the binary instructions for a command shell into the program's executable memory space. A stack overflow is then used to modify the return address of the current stack frame to point to the executable shell instructions in the heap.

Some work has been done to define a framework for discovering and patching vulnerabilities at run-time in response to a network worm [17]. A worm is a program that propagates itself by discovering hosts on a network that are vulnerable to an attack, such as a buffer overflow, and then leveraging that attack to take over the computer. Once the new computer is compromised, the worm uses it to continue searching for another vulnerable host. The work describes the components necessary to detect an attack, and then find and correct the vulnerable source code automatically. The framework defines a program to discover when a service offered on a host is vulnerable to a network worm. The vulnerable service is placed in a sand-boxed environment to discover the specific attack the worm is using. The framework then uses a program to re-engineer the source code of the service so that it is no longer vulnerable to the worm. The patched source code is automatically tested for proper inoculation to the worm, recompiled and reinstalled. Our work can be used by the framework to re-engineer software so that it is no longer vulnerable to buffer overflow attacks.

Dynamic binary translation is a technique that modifies the binary code of a program at run-time. It can be used to patch software without recompilation. These techniques are platform dependent and often require special run-time libraries in order to function correctly. One example of such a system is Dyninst [**?**], a C++ class library for performing run-time code patching.

2.2 Source Code Transformation

Several languages have been created to perform source code transformation. Two such languages are TXL [5, 6] and Stratego [21, 20].

TXL uses a grammar for the input text to be transformed and a set of rules for performing the transformations. TXL can be thought of as a mixture of a functional programming language and the UNIX tools `lex` and `yacc`.

The TXL grammar files are specified in extended Backus-Nauer form. First, TXL uses the specified grammar files to produce a scanner and parser for that grammar. Second, it generates a parse tree from the input using the scanner and parser. Finally, it applies the transformation rules to the tree. The scope of any rule is a subtree of the tree, as described below. An example of a parse tree is provided in Figure 4. Elliptical nodes represent non-terminal symbols and rectangular nodes represent terminal symbols in the grammar.

```
rule remExactMatch Def [externaldefinition]
    replace [repeat externaldefinition]
        First [externaldefinition]
        Rest [repeat externaldefinition]
    where
        First [= Def]
    by
        Rest
end rule
```

Figure 3. A sample TXL rule that removes all duplicate external definitions from a C program.

Every TXL transformation rule must have a **replace** directive and a **by** directive. The **replace** directive specifies the type of tree the rule will replace, and the **by** directive specifies what the rule will replace a matching tree with.

A rule searches its scope and matches each subtree that is of the type specified by the **replace** directive. For example, the rule in Figure 3 only matches trees of type [**repeat** externaldefinition]. In the TXL grammar for C, this specifies any sequence of zero or more external definitions, such as global declarations and function definitions. The remainder of the **replace** directive specifies what the tree must be composed of to match successfully. In this example, the [**repeat** externaldefinition] must begin with an external definition and be followed by a sequence of zero or more external definitions.

If a matching tree is found, `First` is bound to the first element of type [externaldefinition] and `Rest` is bound to the remaining elements that follow First. This rule also re-

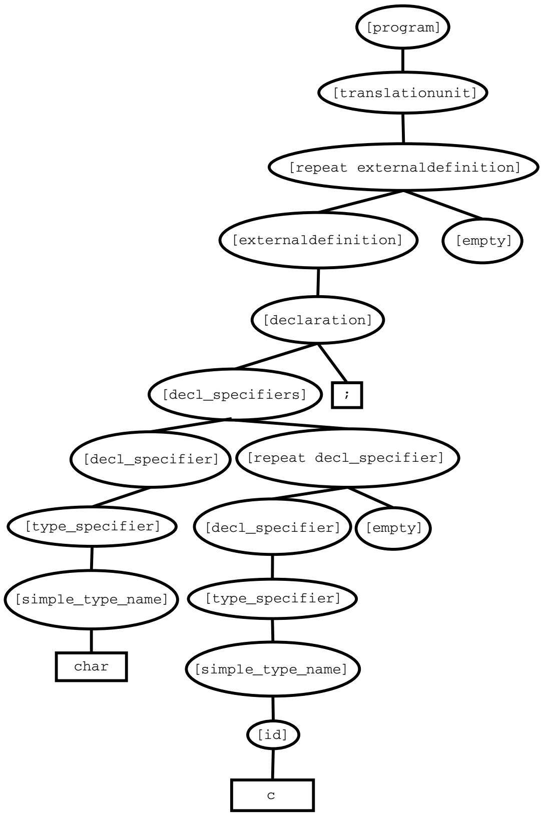


Figure 4. The TXL parse tree for the file containing the C declaration '`char c;`'

quires an argument whose type is specified by the variable `Def`. `Def` is bound to the tree that is supplied as the argument.

The **where** directive specifies that the tree bound to `First` must be equal to the tree bound to `Def`. Equality is determined by the types of the trees and the values of the trees' terminal symbols. If the **where** directive succeeds, the tree that matched the rule is replaced by the tree in the **by** directive. This tree must be of the same type specified by the **replace** directive.

The transformation always preserves syntax since the replacement tree is of the same type as the matching tree.

Stratego, which is another text transformation language, transforms text given a set of *signatures*, *rules* and *strategies* for applying those rules. The signature is similar to a grammar in that it defines how to construct the first-order terms that Stratego operates on. The rules define how signatures are transformed by combining term re-writing strategies.

3. C Arrays and Buffers

There is a common misconception that C arrays and pointers are equivalent. This is not true for two reasons. First, arrays are not variables, and second, the memory layout for an array is different than that of a "pointer to array" [12].

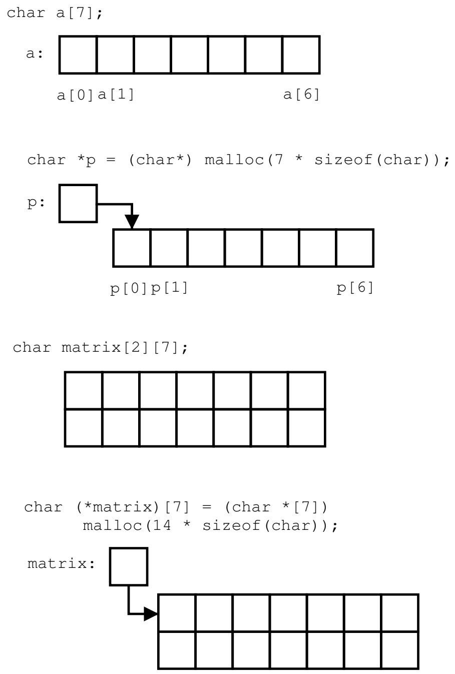

```
char a[7];
```
```
char *p = (char*) malloc(7 * sizeof(char));
```
```
char matrix[2][7];
```
```
char (*matrix)[7] = (char *[7])
      malloc(14 * sizeof(char));
```

Figure 5. Array allocation versus pointer to array allocation

Figure 5 shows the memory allocation for an array `a` and a "pointer to array" `p`. In the first case, the address of `a` is equivalent to the address of `a[0]`. The type of `a` is "array of 7 `char`", and it is allocated 7 bytes. Thus, `sizeof(a)` is 7. Furthermore, `a` is not a variable, and hence, not a valid *lvalue*. For example, '`a++;`' is not a valid C statement.

The address of `p` is not equivalent to the address of `p[0]`. Rather, the memory at address `p` contains the address of `p[0]`. Thus, `sizeof(p)` is 4, not 7, as is the case with array `a`. The total amount of memory associated with `p` is 11 bytes (i.e., 7 bytes for the array and 4 bytes for the pointer). The compiler recognizes that this is a pointer to an array, and de-references `p` automatically when the programmer references `p[0]`. The C statement '`p++;`' is legal since `p` is a variable, and hence, a valid *lvalue*.

When an array is a parameter to a function, the compiler automatically performs a transformation similar to our technique. Specifically, the compiler transforms the array declaration into a "pointer to array" declaration. Arrays can be degenerated into “pointers to arrays” without loss of data, since it can automatically adjust how the memory is accessed.

When an n-dimensional array is implicitly transformed by the compiler, the first dimension is lost. Instead, the compiler creates a pointer to an array, where the dimensions of the new array are dimensions 2 through n.

For example, suppose a function declares a formal argument, ‘`char matrix[2][7]`’. The programmer believes this specifies an array with 2 rows and 7 columns. However, the declaration is implicitly transformed to ‘`char (*matrix)[7]`’. This declares a pointer to an array of 7 `char`s. `matrix` is now a pointer to a row of the original array. If the programmer references `matrix[1][5]`, the compiler will dereference `matrix`, add 12 (i.e., $(1 \times 7) + 5$) to the address stored there, and load the value at that location.

4. TXL Transformation Process

In order to perform the transformation, one simplifying assumption is made. The C source code being transformed must be compilable to an executable binary.

Recall that we are attempting to prevent stack allocated buffer overflows. This is accomplished by transforming all stack allocated arrays into heap allocated “pointers to arrays”. This transformation preserves the semantics of array access and function argument declarations.

Array declarations within functions are the only arrays that must be transformed. Arrays that are declared outside the scope of a function are allocated in the block storage segment and data segment of the executable, and hence, they are not vulnerable to stack overflows. Similarly, pointers to arrays and pointers to pointers, allocated with `malloc`, are on the heap and therefore not vulnerable to stack-based buffer overflow attacks. The following steps outline our transformation process.

Step 1: Declaration Expansion. This step expands declarations that contain a list of declarators. An example of this transformation is shown in Figure 6. Expanding declarations in this fashion simplifies the rest of the transformations.

Step 2: `typedef` **Flattening.** A `typedef` can either alias a type or an array of types. Figure 7 shows the state of the source code after flattening an array alias. Without doing the flattening, there could be many nested `typedef` aliases, making it difficult to determine the correct “pointer to array” declaration.

Step 3: Declaration Transformation. This step transforms all local array declarations to “pointer to array” declarations, the results of which are shown in Figure 8. An initialization function, shown in Figure 11, is created to perform all memory allocation and initialization for the pointer to array.

Before:

```
char fun(char *data)
{
  typedef char my_char[5];
  my_char buf1[10], buf2 = ”test”;
  char *i = buf1;
  printf(
     "I received the string:  %s\n",
     data);
  strcpy(i, data);
  return buf2[sizeof(buf2)-2];
}
```

After:

```
char fun(char *data)
{
  typedef char my_char[5];
  my_char buf1[10];
  my_char buf2 = ”test”;
  char *i = buf1;
  printf(
     "I received the string:  %s\n",
     data);
  strcpy(i, data);
  return buf2[sizeof(buf2)-2];
}
```

Figure 6. Step 1: Declaration expansion

The memory allocation and initialization cannot be performed within the body of the function due to an ambiguity in the C grammar concerning declarations and statements. For example, parsing the declaration ‘`my_char (a);`’ and the statement ‘`printf (a);`’ can result in the same parse tree. Therefore, if the `printf` statement occurs immediately after the declaration of `a`, the C grammar is such that the `printf` statement is the last declaration in the forward declarations block. This ambiguity would be resolved during the semantic analysis phase of compilation, however our transformation is limited to syntax.

Due to this ambiguity, it is impossible to guarantee that the allocation and initialization of each transformed array will occur before any statements reference the resulting pointer. To solve this problem, we perform all of the work in a separate function. This function returns a pointer to the prepared memory.

The ISO C99 specification allows the dimensions of locally defined arrays to be variable sized. After the transformation, the dimensions of the resulting pointer will be

```
char fun(char *data)
{
  typedef char my_char[5];
  char buf1[10][5];
  char buf2[5] = "test";
  char *i = buf1;
  printf(
    "I received the string:  %s\n",
    data);
  strcpy(i, data);
  return buf2[sizeof(buf2)-2];
}
```

Figure 7. Step 2: `typedef` **flattening**

referenced several times during initialization. Simply copying the expression to several areas of the source code would result in a failure to preserve the semantics of the program. For example, if the dimension of `buf1` was a call to the `random` function, then a verbatim copy of the dimension would have unpredictable results on the run-time behavior of the application.

To solve this problem, the dimensions of the original array are extracted and stored in a local integer variable. This placeholder is substituted for the original expression during allocation and initialization of the memory.

Step 4: `sizeof` **Alias Declarations.** This step inserts a new, unique array declaration for each array declaration that was transformed. This new declaration is the same as the original array declaration, except that it is not initialized with data. The results are shown in Figure 9.

The purpose of the unique declaration is to preserve the semantics of `sizeof`. To be proper, the `sizeof` constant should only be passed a type. However, many programmers will pass it a variable or an expression. If a `sizeof` constant references the transformed array, it will no longer evaluate to the same value as the original program since the type has changed from array to pointer. In order to solve this special case, we search through the scope of each transformed array declaration and replace every reference to the original array, within a `sizeof`, with the name of the new unique declaration, as shown in Figure 10.

Step 5: Add `free` **and Transform** `return` **and** `sizeof`**.** As shown in Figure 10, this step adds the appropriate calls to `free`, and transforms the `return` and `sizeof` statements. The calls to `free` are necessary to preserve the behavior of the original arrays, which are automatic variables. The transformation will insert the `free` calls at the end of every block where the original array would have run out of scope.

```
/* See Figure 11 for definition of */
/* buf1_init1 and buf2_init1 */
char fun(char *data)
{
  typedef char my_char[5];
  int size_var1 = 10;
  int size_var2 = 5;
  char (* buf1) [size_var2] =
     buf1_init1 (size_var1, size_var2);
  int size_var3 = 5;
  char (* buf2) =
     buf2_init1 (size_var3);
  char *i = buf1;
  printf(
    "I received the string:  %s\n",
    data);
  strcpy(i, data);
  return buf2[sizeof(buf2)-2];
}
```

Figure 8. Step 3: Declaration transformations

If the `return` statement references one of the buffers, a segmentation fault may occur since the `return` statement will attempt to dereference an invalid pointer. Hence, the expression that would have been returned is stored in a local variable, and the contents of the variable are returned instead.

Step 6: Initialization Functions. This step adds the initialization functions to the end of the source file to ensure that any necessary header files are included above them, such as `stdlib.h`. Figure 11 shows these functions. Prototypes for the new functions are inserted at the top of the source file so that the compiler can resolve the symbol names of the functions.

5. Effectiveness and Efficiency of the Transformation

To demonstrate the effectiveness of our transformation, we show how the transformation of source code that is amenable to a buffer overflow prevents the exploit from occurring. Several transformed Linux programs have been tested to demonstrate the expected efficiency of the transformed code, as shown in Tables 1 and 2.

```
char fun(char *data)
{
  typedef char my_char[5];
  int size_var1 = 10;
  int size_var2 = 5;
  char buf1_sizeof_alias1 [size_var1]
     [size_var2];
  char (* buf1) [size_var2] =
     buf1_init1 (size_var1, size_var2);
  int size_var3 = 5;
  char buf2_sizeof_alias1 [size_var3];
  char (* buf2) =
     buf2_init1 (size_var3);
  char *i = buf1;
  printf(
     "I received the string:  %s\n",
     data);
  strcpy(i, data);
  return buf2[sizeof(buf2)-2];
}
```

Figure 9. Step 4: `sizeof` **alias declarations**

5.1 Example of a Buffer Overflow

The code for both the vulnerable program and the exploit program were adapted from source code originally written by John Viega and Gary McGraw [19]. Both programs are available from the project web site [10].

Figure 6 shows part of the source code of `breakme.c`, a C program that contains a buffer overflow vulnerability. The vulnerability lies in the call to the `strcpy` function on line 9 without verifying that array `buf1` is large enough to hold the contents of `data`. The `strcpy` function is the cause of many buffer overflows.

The attacker is usually interested in exploiting a program that is running as the `root` user. The string that overflows the buffer generally contains binary instructions to execute a shell. If the vulnerable program is running as `root`, the resulting shell will too.

The source code indicates that `buf1` is allocated 50 bytes. Assuming `buf1` is the first variable allocated on the stack (i.e., located highest in memory), the return address must be located at least 50 bytes above the start of `buf1`.

The precise location of the return address can be found easily via stack inspection at runtime [19], or in a debugger. Once the exact offset is found for the return address, the attacker can write a program that performs the buffer overflow.

```
char fun(char *data)
{
  typedef char my_char[5];
  int size_var1 = 10;
  int size_var2 = 5;
  char buf1_sizeof_alias1 [size_var1]
     [size_var2];
  char (* buf1) [size_var2] =
     buf1_init1 (size_var1, size_var2);
  int size_var3 = 5;
  char buf2_sizeof_alias1 [size_var3];
  char (* buf2) =
     buf2_init1 (size_var3);
  char *i = buf1;
  printf(
     "I received the string:  %s\n",
     data);
  strcpy(i, data);
  {
    int txl_return_temp1;
    txl_return_temp1 =
       buf2[sizeof(buf2_sizeof_alias1)-2];
    free (buf1);
    free (buf2);
    return txl_return_temp1;
  }
}
```

Figure 10. Step 5: Adding `free`**, and transforming** `return` **and** `sizeof`

Figure 12 shows the C program that exploits the buffer overflow in `breakme.c`. To exploit the buffer overflow, an attacker must send 81 bytes to `breakme`. The first 53 of the bytes is the executable binary code to spawn a `root` shell, the next 23 bytes are non-null filler characters to pad the array until the return address can be overwritten at position 77, the next 4 bytes are the new return address, and the final byte is a terminating `null`.

This will result in the function jumping down to the attacker's code on the stack instead of back to the calling function on return. If everything is laid out in memory correctly, the attacker will receive a `root` owned command prompt on the screen.

Figure 13 shows the vulnerable program, `breakme`, being executed normally by the attacker, and execution of the `exploit` program resulting in a `root` owned shell. Figure 14 shows the `exploit` program failing to gain a `root` shell on the transformed version of `breakme`. Instead, the buffer overflow results in a segmentation fault and subse-

```
char * buf2_init1 (int size_var3)
{
 char (* buf2);
 char buf2_alias1 [5] = "test";
 buf2 = (char (*))
   calloc (size_var3,
   sizeof (char));
 memcpy (buf2, buf2_alias1,
   sizeof(buf2_alias1));
  return buf2;
}

char * buf1_init1 (int size_var1,
   int size_var2)
{
 char (* buf1) [size_var2];
 buf1 = (char (*) [size_var2])
   calloc (size_var1,
   sizeof(char) * size_var2);
 return buf1;
}
```

Figure 11. Step 6: Memory allocation and initialization functions

quent termination of `breakme`. The transformed version of `breakme.c` is shown in Figures 10 and 11.

5.2 Efficiency of Transformed Linux Programs

To show the amount of overhead that can be expected from using heap allocated buffers in place of stack allocated arrays, we transformed several Linux programs, each with varying degrees of size and complexity, as shown in Table 1. If a program came with a regression test suite, these tests were performed on both the original code and the transformed code, the results of which are shown in Table 2.

Table 1 shows the number of arrays that were transformed into “pointers to arrays” and the average run-time increase of the resulting binary. The binaries were compiled without optimizations in each case. The time increase was calculated in one of two ways. If the program did not include a suite of regression tests, it was executed fifty times with standard options. The result of this was compared to the same tests being executed on the non-transformed binary. If the program did include a suite of regression tests, the time increase was calculated by taking the difference in fifty runs of the test suite of the transformed and non-transformed binaries.

```
int main(int argc, char** argv)
{
 char *shellcode =
   "\x31\xdb\x89\xd8\xb0\x17\xcd\x80"
   "\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
   "\x88\x46\x07\x89\x46\x0c\xb0\x0b"
   "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
   "\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
   "\x80\xe8\xdc\xff\xff\xff/bin/sh";
 char *overflow = (char *)
   calloc(81, sizeof(char));
 char **args = (char **) calloc(3,
   sizeof (char*));
 int i, explen=53, offset=76;
 strncpy(overflow, shellcode,
   explen);
 for(i=explen; i<offset; i++)
   overflow[i] = 42;
 overflow[offset] = 0x20;
 overflow[offset+1] = 0xf5;
 overflow[offset+2] = 0xff;
 overflow[offset+3] = 0xbf;
 overflow[offset+4] = 0x0;
 args[0] = "./breakme";
 args[1] = overflow;
 args[2] = 0x0;
 execv("./breakme", args);
}
```

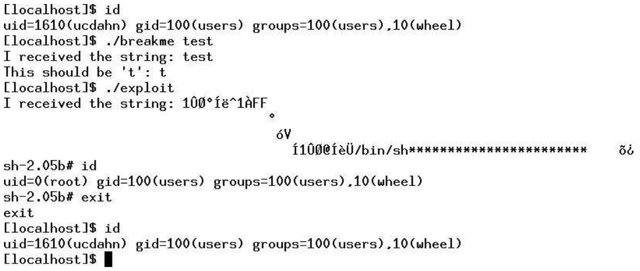
Figure 12. `exploit.c`**. This is the code to exploit the buffer overflow in** `breakme.c`

Table 2 shows the results of the regression tests. In every case, every test passed in both the transformed and non-transformed binaries.

5.3 Transformation Environment

The TXL program must operate on pre-processed source code. Until preprocessing, it is impossible to know what portions of the source will be compiled. Furthermore, `typedef` flattening may require information from files that are included by the C file.

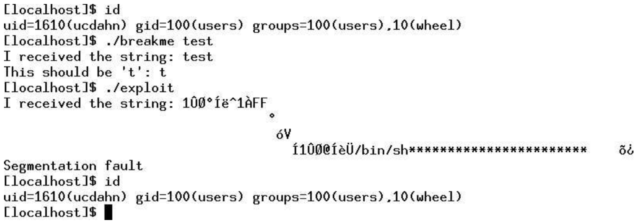
We used GCC for the purposes of the experiments. GCC has its own extensions to C that it adds to the preprocessor output. Many of these additions are optimization hints for the compiler, such as specifying the size of types. In order to handle these extensions, we have extended the C grammar



```
[localhost]$ id
uid=1610(ucdahn) gid=100(users) groups=100(users),10(wheel)
[localhost]$ ./breakme test
I received the string: test
This should be 't': t
[localhost]$ ./exploit
I received the string: 1Û0°Íë^1ÀFF
                                   °

                                    óV
                                      Í1Û0@ÍèÜ/bin/sh************************    õ¿
sh-2.05b# id
uid=0(root) gid=100(users) groups=100(users),10(wheel)
sh-2.05b# exit
exit
[localhost]$ id
uid=1610(ucdahn) gid=100(users) groups=100(users),10(wheel)
[localhost]$ █
```

Figure 13. The buffer overflow being exploited



```
[localhost]$ id
uid=1610(ucdahn) gid=100(users) groups=100(users),10(wheel)
[localhost]$ ./breakme test
I received the string: test
This should be 't': t
[localhost]$ ./exploit
I received the string: 1Û0°Íë^1ÀFF
                                   °

                                    óV
                                      Í1Û0@ÍèÜ/bin/sh************************    õ¿
Segmentation fault
[localhost]$ id
uid=1610(ucdahn) gid=100(users) groups=100(users),10(wheel)
[localhost]$ █
```

Figure 14. The buffer overflow is prevented using the transformed version of `breakme`

included with TXL.

The following steps constitute the pipeline for transforming C code. This pipeline can be inserted directly into the build process of most open source software.

Step 1: Configure and build the program. This ensures that all necessary build files are created and that the program holds the simplifying assumption specified in Section 4. From this step we obtain the names of the files that need to be transformed.

Step 2: Automatically modify the `Makefile`. Use a `sed` script to automatically change the `Makefile` so that it produces pre-processed C code instead of object files and binaries. Finally, update the modification time of each C file using the `touch` program. This ensures that `make` will attempt to generate object files and binaries in the next step.

Step 3: Generate pre-processed C. Execute `make` again for each file that was produced during the initial execution of `make`. This time the pre-processed C code will be produced for each file. The debugging directives found in the pre-processor output from GCC are removed since the TXL grammar cannot parse them.

Step 4: Transform the program. Backup the original C source files and transform each pre-processed C file, overwriting the original C source file with the transformed output. After all of the files have been transformed, execute `make` again to create the transformed program, then restore the original C source files.

Program	Arrays Fixed	Time Increase	Regression Tests
apache	30	-3%	No
bind	486	0%	Yes
bison	45	-2%	Yes
find	6	0%	Yes
flex	30	0%	No
openssh	221	2%	Yes
which	1	1%	No
whois	6	1%	No

Table 1. Efficiency results of transformed Linux programs

Program	Number of Tests	Old Code Passed	New Code Passed
bind	146	146	146
bison	56	56	56
find	5	5	5
openssh	21	21	21

Table 2. Results of regression tests on transformed code

6. Limitations & Future Work

Currently, `static`, global and `extern` buffers, and `structs` are not transformed. With the exception of structs, all of these buffers are either allocated on the heap, or within the block storage segment or data segment of the executable. Since such buffers cannot result in stack allocated buffer overflows, we do not need to transform them.

There are two types of `struct` declaration that must be transformed. The first is arrays of `structs` and the second is declarations involving `structs` which contain arrays. A lookup table of each `struct` type must be generated to determine which contain arrays. For each matching declaration, the same algorithm that is outlined in Section 4 should be followed.

7. Conclusions

The number of vulnerabilities found in software continues to rise each year. There have been many proposed solutions to this problem, many of which work very well, but often without guarantees.

Our solution guarantees that current and future stack buffer overflow attack vectors will fail when used against a transformed program, since the heap does not contain return addresses. Our solution does not fix the bug that causes a buffer overflow, but it does mitigate the risk of such a bug by preventing the attacker from inserting executable instructions, such as shell instructions, and overwriting the return address to jump to those instructions.

Furthermore, our technique preserves the semantics of the program. This allows Gemini to be inserted into the regular development process effortlessly. The performance associated with using heap memory instead of stack memory will increase with the amount of use the stack allocated buffers receive, and by the number of times the functions containing the arrays are called.

A fortunate side-effect of our technique is that by inserting more buffers onto the heap, the heap memory becomes perturbed. This perturbation lends itself to thwarting current and future heap buffer overflow attack vectors.

The tool which applies our technique, as well as source code examples from this paper are available from our website [10].

Acknowledgments

Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-01-2-0534. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

We would like to thank Dr. Vassilis Prevelakis and Dan DaCosta, whose intuition and understanding of GCC, C, and software security were invaluable to this work. We would also like to thank Mike Andrews and Bill Mongan who helped brainstorm names for our tool.

References

[1] Computer Security Institute, Computer Crime and Security Survey. 2002. http://www.gocsi.com/press/20020407.html.

[2] Buffer overflow in DNS, http://icat.nist.gov/icat.cfm?cvename=CAN-2002-1219.

[3] Percentage of bind installations for DNS, http://www.isi.edu/ bmanning/in-addr-versions.html.

[4] Cigital. ITS4 http://www.cigital.com/its4/.

[5] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider. Software Engineering by Source Transformation - Experience With TXL. In *Proceedings from SCAM'01 - IEEE 1st International Workshop on Source Code Analysis and Manipulation*, pages 168–178, November 2001.

[6] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider. Source Transformation in Software Engineering Using the TXL Transformation System. *Journal of Information and Software Technology*, 44(13):827–837, October 2002.

[7] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Automatic Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[8] D. DaCosta, C. Dahn, S. Mancoridis, and V. Prevelakis. Characterizing the 'Security Vulnerability Likelihood' of Software Functions. In *Proceedings from the IEEE International Conference on Software Maintenance*, September 2003.

[9] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[10] Gemini, http://serg.cs.drexel.edu/gemini/.

[11] R. W. M. Jones and P. H. J. Kelly. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[12] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., Upper Saddle River, New Jersey, 1988.

[13] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilites. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[14] J. Nazario. Source code scanners for better code. *Linux Journal*, January 2002. http://www.linuxjournal.com/article.php?sid=5673.

[15] A. One. Smashing The Stack For Fun and Profit. *Phrack Magazine*, 7(49), November 1996.

[16] SecureSoftware. RATS http://www.securesoft.com/rats.php.

[17] S. Sidiroglou and A. D. Keromytis. A Network Worm Vaccine Architecture. In *IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, June 2003.

[18] Percentage of SMTP mail transfer daemons, http://www.bbv.com/SMTP/SMTP-2001-Stats.jpg.

[19] J. Viega and G. McGraw. *Building Secure Software*. Addison Wesley, 2002.

[20] E. Visser. A Survey of Rewriting Strategies in Program Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 57, 2001.

[21] E. Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. *Lecture Notes in Computer Science*, 2051, May 2001.

[22] Percentage of web server installations, http://news.netcraft.com/archives/web_server_survey.html.

[23] D. Wheeler. Flawfinder http://www.dwheeleer.com/flawfinder/.