# Exploring Security Vulnerabilities by Exploiting Buffer Overflow Using the MIPS ISA

**Andrew T. Phillips and Jack S.E. Tan**
**Computer Science Department**
**University of Wisconsin-Eau Claire**
**Eau Claire, WI 54702**
**{phillipa, tanjs}@uwec.edu**

## Abstract

By exploiting a well known security vulnerability in many C library implementations, it is possible for an unprivileged user to gain unrestricted system privileges. With an understanding of how the process execution stack is allocated and managed during process execution, a user can override the return address of a C library routine and thereby resume execution at a different address where a set of malicious functions can be invoked [1]. This is known as the buffer overflow exploit. With buffer overflow as the underlying theme, an example will be described using C and the MIPS assembly language that simultaneously exposes students to issues in computer security, operating systems concepts such as memory management and function invocation/return, and the MIPS instruction set architecture.

## Categories and Subject Descriptors

K.3 [**Computers & Education**]: Computer & Information Science Education – *Computer Science Education*.

## General Terms

Security

## Keywords

Security, Pedagogy, Buffer Overflow

## 1 Introduction

Many implementations of C library routines, such as strcat, strcpy, and sprintf, do not have any automatic bounds checking mechanisms, so that writing past the end of an array is not just a common programming error, but it can also lead to classic computer security exploits involving buffer overflow. For example, as a result of non-existent bounds checking, characters can be written past the end of a fixed length buffer and into the next immediate location in the program's execution stack. If that location, or any other nearby location, holds the return address of the calling routine, the return address can be overwritten with a different address that can then be used to launch code with malicious intent ([2], [3], [4]).



*SIGCSE 2003,* February 19-23, R. Nevada, USA.


Understanding stack pointers and return addresses in an instruction set architecture such as MIPS is usually accomplished by an example that requires simple manipulation of the stack pointer during function invocation and return. By extending the scope and complexity of teaching about function calls and returns to include the discussion and potential exploitation of buffer overflow, several additional fundamental concepts can easily be infused and illustrated. Some of these are:

- basic compiler code generation techniques
- dynamic memory allocation techniques & activation records
- security issues in program language design and implementation
- memory management
- the von Neumann model of computation.

In this paper, we describe a simple programming example that incorporates and illustrates the effects of buffer overflow for the purpose of teaching students about low-level implementations of language constructs, flow of control, and the management of process execution and its stack. The example is a simple implementation of a routine function call that overflows the buffer during invocation. The return address is overwritten with an address that can subsequently invoke code to perform an unintended operation on a data set, thereby mimicking the result of malicious code. Errors involving segmentation faults or core dumps will be precluded with careful coding of the overwritten return address. This is intended to produce results from the unexpected diversion with error-free yet baffling results and no compile errors whatsoever.

The level of simplicity in the requirements for the implementation will be described in detail in the next section. Subsequent sections will describe the actual program example and its MIPS translation.

## 2 Process Execution Stack

To understand buffer overflow and its associated computer security exploit, the concept of how a process is organized and managed in memory is essential. A typical memory organization for a process includes the text, data, and stack segments, and Figure 1 illustrates this organization for the widely used MIPS instruction set [5].

Of primary importance here is that the process stack is dynamic in nature, and that it grows from higher memory addresses toward the lower memory address region occupied first by the static data segment, and then by the text (code) segment.

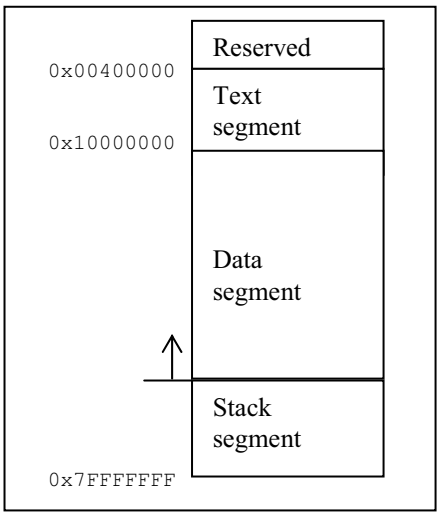


**Figure 1: MIPS Memory Layout**

When a function is invoked, several actions are undertaken to preserve the state of the calling function and to provide additional memory to the called function. A stack frame, or activation record, is pushed onto the stack, and information pertaining to the called function is contained in this frame, e.g., certain function arguments, the return address if another function call will be made, a copy of the global pointer, and any local variables statically allocated in the called function. Figure 2 and Figure 3 show an example of this general process, where `$sp` represents the MIPS stack pointer register, `$ra` represents the MIPS return address register, and `$gp` represents the MIPS global pointer register. Note that this is not a comprehensive representation of an actual execution stack as other types of information may also pushed onto the stack, and of course an optimizing compiler may elect not to allocate space on the stack for all local variables, since it may set aside registers for some of them. To provide a simple explanation, only the bare minimum of information is shown.

```
void bar(int y) {
   int bar_q; // will be allocated on the stack
   bar_q = y;
}

void foo(int x) {
   int foo_p; // will be allocated on the stack
   foo_p = x + 2;
   bar(foo_p);
   /* return address points to here */
}

void main() {
   foo(8);
   /* return address points to here */
}
```

**Figure 2: main() calls foo() which calls bar()**

Notice from the stack frames in Figure 3 that when `main()` calls `foo()`, the local variable `foo_p` is pushed onto the stack after the return address and the global pointer corresponding to `main()`. The return address register `$ra` is pushed onto the stack in foo's stack frame in order to preserve the return address from `foo()` back to `main()` once `bar()` is called. Note that register `$ra` in the MIPS instruction set architecture always holds the return address corresponding to the most recent function call only. Also note that a good compiler would optimize this code by storing the local variables `foo_p` and `bar_q` in registers instead of on the stack. This will be reflected in subsequent examples, but at this point we just wish to note that local variables, when allocated on the stack, are pushed onto the stack after the return address.

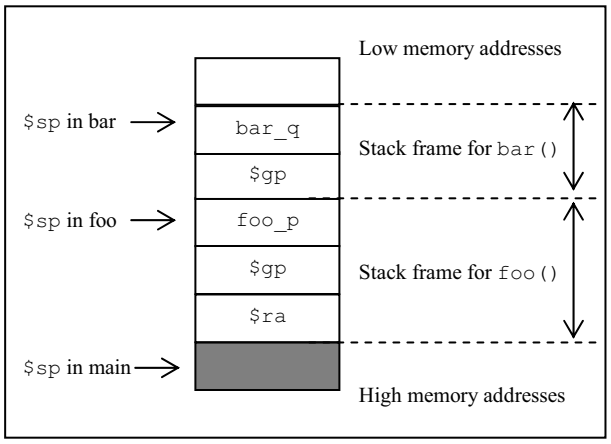


**Figure 3: Process stack associated with calling sequence**

What happens if `foo()` defines the local variable `foo_p` to be a fixed length character array instead of a single integer? Then space for the entire character array must be allocated on the stack, with the beginning of the array (at index 0) located at the lower end of the memory address space and growing toward the higher end. This innocent stack based memory allocation scheme potentially spells disaster if the main program calls and passes `foo()` a character array that is longer than the space set aside for `foo_p`.

Since neither C nor MIPS provide bounds checking for arrays, the characters in the longer array can be written past the boundary of the local buffer and thereby overflow into the region of the stack that contains the return address and global pointer saved at the invocation of the function. If the return address is overwritten with a value that represents some other legitimate address, then the code pointed to by that address will be run upon the function return.

For example, if the function `foo()` were to be defined as in Figure 4, the result would certainly be a buffer overflow since the C library function `strcpy()` does not perform bounds checking but rather simply copies characters until it encounters a null value. This means that the contents of the longer character array will be written past the end of the smaller character array, which is stored on the stack, and ultimately overwrite the return address on the stack (see Figure 5). Of course, the return address is placed on the stack in the first place since `foo()` calls `strcpy()` and therefore needs to save the return address back into `main()`.

```
void foo(char *buf) {
   char local_buf[10];
   strcpy(local_buf,buf);
}

void main() {
   char *large_buf = "a……a"; // lots of a's
   foo(large_buf);
   /* location of return from foo() */
}
```

**Figure 4: Buffer overflow example code**

Typically, a segmentation violation or core dump is a result of such innocent ignorance because the value overwritten in place of `$ra` on the stack does not represent a valid address, or even if it

does, it is an address outside of the user's allowable process address space.

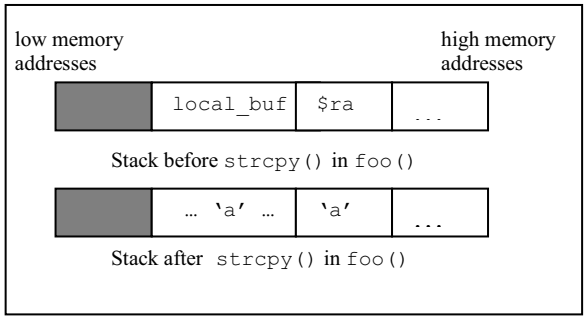


**Figure 5: Overflowing `foo()`'s local buffer**

In summary, the following key points are sufficient to cause a buffer overflow:

1. An array, named `buf` in our example, is passed as an argument to the function `foo()`.
2. A fixed size array, named `local_buf` in our example, is allocated on the stack as a local variable to `foo()`.
3. The length of the actual argument array (`buf`) happens to be longer than the size allocated to the local one (`local_buf`), a copy is made from `buf` to `local_buf` in `foo()`, but the bounds are not checked in the process.
4. The function `foo()` makes at least one other function call, in this case to `strcpy()`, but it makes no difference what action this function performs.

## 3 32 Bit MIPS Implementation of Buffer Overflow

The example code from Figure 4 is further expanded in Figure 6 in order to allow for a careful examination of a typical C to MIPS code translation that would occur as a result of compilation using a standard optimizing compiler. The only difference in the code is that the C library function `strcpy()` is replaced with four instructions that perform the same byte-by-byte copy operation, and by a separate function call to `bar()` that does nothing except force the return address onto the stack just as the call to `strcpy()` would have done. We have done this in order to clearly show how a standard compiler translates this C code into MIPS in a way sufficient to permit the buffer overflow programming error to occur and then result in a potential computer security exploit.

```
void foo(char *buf) {
   char local_buf[10];
   int i = 0;
   while (buf[i] != 0) {
      local_buf[i] = buf[i];
      i++;
   }
   bar(); /* code for bar() not shown */
}

void main() {
   char *large_buf = "a……a"; // lots of a's
   foo(large_buf);
   /* location of return from foo() */
}
```

**Figure 6: Expanded buffer overflow example**

The top of the run-time stack in the 32 bit MIPS model is referenced by the contents of register `$sp` which always identifies the top most occupied position of the stack. In the 32 bit MIPS model, each word is 4 bytes in length, so it is most common to see the stack pointer `$sp` incremented (for pop) or decremented (for push) by multiples of 4.

Regardless of the semantic content of a function, a standard compiler will always set aside a certain amount of stack space for the stack frame of the function. For our purposes we need only note that standard compilers always do this consistently. In particular, for a function that satisfies requirement 4 above, the return address back to the calling function is always pushed onto the stack first, before any other registers, and before any local variables. In fact, relative to the top of the stack prior to the function call, `$ra` is always saved at an offset of -4 bytes, followed by `$gp` at –8 bytes, followed by any local variables allocated to the function. Figure 7 shows the situation immediately following the call to our new function `foo()`.

| | addr | Stack contents |
|---|---|---|
| new `$sp`→ | X-28 | `allocated by the compiler` |
| | X-24 | `allocated by the compiler` |
| | X-20 | `local_buf[0] … local_buf[3]` |
| | X-16 | `local_buf[4] … local_buf[7]` |
| | X-12 | `local_buf[8], local_buf[9] …` |
| | X-8 | Global pointer ($gp) from `main()` |
| | X-4 | Return address ($ra) back into `main()` |
| old `$sp`→ | X | … |

**Figure 7: MIPS stack during call to foo()**

Notice that the local buffer, `local_buf`, requires three 4 byte words since MIPS allocates stack space in multiples of a word, even though `local_buf` would in fact only need 10 bytes total. So, `local_buf` occupies the stack positions X-20 to X-11 with the remaining positions X-10 and X-9 representing allocated but unused memory. What is really important here though is that we can be assured that the local buffer is stored on the stack below (in memory address space) the global pointer and return address. Also notice that the local variable "`i`" is not allocated any space on the stack since a good compiler, as we have previously stated, would optimize its use and place it in a register instead.

Figure 8 shows the 32 bit MIPS translation of the code shown above. The code for function `bar()` is omitted since the only important issue here is the memory access pattern of `local_buf`, which starts at stack index X-20 and moves up by one on each iteration of the loop. Since the bounds of the copy process are not checked, then if the size of `buf` exceeds the size of `local_buf` by more than 2 bytes (in this example), the saved global pointer (`$gp`) and then possibly the saved return address (`$ra`) will be overwritten on the stack by the buffer overflow flaw. This could be fatal once `foo()` attempts to execute the return to `main()`.

In fact, if the length of `buf` exceeds `local_buf` by at least 10 bytes, then `$ra` is certain to be overwritten by the contents of `buf` itself. This is how the buffer overflow flaw becomes a security exploit.

## 4 Exploiting the buffer overflow flaw

So how do you use the buffer overflow in an exploit? The idea is simple and makes for a nice (if even slightly devious) example of the von Neumann model of computation at work. Simply put, to exploit the buffer overflow flaw, the main program (in our example) should:

1. Pass a buffer of length at least 21 bytes to the function `foo()`.
2. Ensure that none of first 20 bytes in the buffer have the value 0 (which would prematurely terminate the copy), but that some byte past index 19 does have value 0.
3. Ensure that the bytes in the buffer represent the MIPS code that you wish to execute.
4. Ensure that the value of the word starting at byte index 16 of the buffer (since this is where `$ra` would be on the stack) represents an address that points back into the buffer at the location where the exploit code begins.

Items 1 and 2 ensure that we will indeed overwrite the return address on the stack, while item 4 ensures that the return address is overwritten with another address to the exploit code, which is stored neatly in the buffer itself.

```
foo:
   subu  $sp,$sp,28
   sw    $ra,24($sp)
   sw    $gp,20($sp)
   lbu   $v0,0($a0)
   beq   $v0,$zero,.L4
   addi  $a1,$zero,0
   addu  $a2,$sp,8      # address of local_buf[0]
.L5
   lbu   $v0,0($a0)
   addu  $a0, $a0, 1
   addu  $v1,$a2,$a1
   sb    $v0,0($v1)     # write onto the stack
   lbu   $v0,0($a0)
   bne   $v0,$zero,.L5
   addu  $a1,$a1,1
.L4:
   jal   bar
   lw    $ra,24($sp)
   lw    $gp,20($sp)
   jr    $ra
   addu  $sp, $sp, 28
.end    foo

.LC0:                   # lots of a's
   .byte 0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61
   .byte 0x61,0x61,0x61,0x61,0x61,0x61,0x61,0x61
   .byte 0x61,0x61,0x61,0x61,0x00

main:
   subu  $sp,$sp,16
   sw    $ra,12($sp)
   sw    $gp,8($sp)
   la    $a0,.LC0
   jal   foo
…
.end    main
```

**Figure 8: 32 bit MIPS translation using "gcc –S –O2"**

But how can you know what address to place into the buffer `large_buf` starting at index 16, given that the stack is allocated in memory by the operating system when the program is loaded? The answer is that most programs have a fixed starting address (relative to the user process's address space) for the stack prior to program invocation. If we know that not much information has been pushed onto the stack prior to our call to `foo()`, then we can effectively guess the address we need. This is precisely how malicious hackers make use of the buffer overflow exploit. They guess, and if they guess well or are persistent enough at guessing, they eventually get it right. In our example, the return address is saved on the stack only 24 bytes below the top of the stack at program invocation, and this is fairly typical. That means that the "new" exploit code starting address should point somewhere between 24 and 40 bytes below the original top of stack address.

In fact, if the local buffer is moderately large, say 256 bytes or more, then the guessing is made even easier. In this case, the buffer passed to `foo()` could contain the exploit code preceded by a long sequence of `NOP` instructions and followed by a repeating block of identical words representing the exploit code starting address guess. The idea here is that as long as you guess an address that points back into the buffer anywhere in the `NOP` region (along a word boundary), then the resulting code will not core dump, and the exploit will work. Figure 9 shows a very simple (and non-malicious) example of this idea. In this example, we are using the MIPS instruction "`add $t1, $t1, $zero`" in place of `NOP` so that none of the bytes in the machine translation (which is \x01204820) are 0, and yet the effect is still a `NOP`. Our exploit code is simply to subtract 1 from the address representing `buf` ("`addi $a0, $a0, -1`" which translates to \x2084FFFF). While this is just a single line exploit, obviously much more complicated code could be executed instead. And of course, we have neglected to deal with the termination of our exploit code. Following the exploit code, we have indicated a sequence of three repeated address "guesses" marked in our example by \x12345678. These addresses are fictional of course, and all that is required for the exploit to work is that one of these words overwrite the old `$ra` and that it's value point into the `nop` region.

```
void foo(char *buf) {
   char local_buf[256];
   strcpy(local_buf,buf);
}

void main() {
   char *large_buf =   "\x01\x20\x48\x20"
                       "\x01\x20\x48\x20"
                       …
                       "\x20\x84\xFF\xFF"
                       "\x12\x34\x56\x78"
                       "\x12\x34\x56\x78"
                       "\x12\x34\x56\x78"
                       "\x00";

   foo(large_buf);
   /* location of return from foo() */
}
```

**Figure 9: Buffer overflow with code exploit**

Of course, malicious code that is launched via this stack based buffer exploit also can easily cause interrupts that relinquish privileged, or root, control to unauthorized users. But operating system security violations are not the only consequence of the buffer overflow exploit. Security can be compromised in network protocols, databases, and internet web servers as well, with one of the most infamous examples of this being the breach of the NCSA Web server in 1994.

Of course our main point here is this: with an understanding of how the process execution stack is allocated and managed during

process execution, we can simultaneously expose students to issues in computer security, operating systems concepts, and the MIPS instruction set architecture. And to do so, we can use the buffer overflow exploit as a unifying theme without "crossing the line" by actually producing any malicious code.

## 5 Summary

The classic buffer overflow computer security exploit need not be "hidden" from computer science students. In fact, discussing how buffer overflow is exploited is an excellent means for integrating topics in compiler code generation, memory management, function calls/returns, and the basics of instruction set architectures. We also hope that by being forthright about the potential for malicious code exploits of buffer overflow, computer science students will become more careful and defensive in their program design in the future.

## References


[1] Aleph One, *Smashing the Stack for Fun & Profit*, http://www.phrack.com/show.php?p=49&a=14

[2] CERT® Vulnerability Note CU#259787, http://www.kb.cert.org/vuls/id/259787.

[3] CERT® Advisory CA-2002-26 Buffer Overflow in CDE ToolTalk, http://www.cert.org/advisories/CA-2002-26.html.

[4] CERT® Advisory CA-2002-19 Buffer Overflows in Multiple DNS Resolver http://www.cert.org/ advisories/CA-2002-19.html

[5] Patterson, D., and Hennessey, J., *Computer Organization and Design: A Hardware/Software Interface,* Appendix A, Morgan Kaufmann (2001).

[6] Sweetman, D., *See MIPS Run,* Morgan Kaufmann, San Francisco, CA, (1999).