Global Information Assurance Certification Paper

GIAC Advanced Incident Handling and Hacker Exploits
Practical Assignment – Option 2

Solaris Loadable Kernel Modules and Their Use in Rootkits

William S. Davis

April 4th, 2001

Table of Contents:

1.0 Exploit Details:

Name: Solaris Integrated Trojan Facility (sitf0.2)
Variants: linspy, heroin, itf, Knark, adore (all are for Linux)
Operating System: Solaris 7 & 8
Protocols/Services: Loadable Kernel Modules (LKMs)
Description:
The Solaris Integrated Trojan Facility enables an attacker to hide files, processes and installed kernel modules, while allowing the attacker to redirect program execution calls and grant root access to the system. The software uses loadable kernel modules, code that runs within the operating system kernel and not at the user application level.

2.0 Introduction:

The Solaris Integrated Trojan Facility (SITF) is a kernel-level rootkit. A rootkit is generally a series of steps or procedures that an attacker, once they have gained root access to a host or server, will use to hide their continued access and illicit activity. To do this, the SITF installs loadable kernel modules (LKMs) that perform these procedures by modify the functioning of the operating system itself.

In the past, a rootkit typically contained a collection of trojaned user programs that allowed them to alter the output for their own purposes. For example, a trojaned Unix "ps" program would be used to hide processes run by the attacker, or a trojaned Unix "ls" program that would not list files the attacker wanted to remain unseen by anyone else. As a defense against rootkits, system administrators began to use integrity checking, cryptographic hashes or a program like Tripwire, to ensure that critical programs were not altered.

What makes a kernel-level rootkit a particularly insidious exploit is that it is the operating system kernel, rather than user programs, which is altered. This means that integrity checking may fail to detect any modification of a system, since the user programs have not been replaced with trojaned versions, and the operating system itself may give false information to the integrity-checking program to begin with. For this reason, the LKMs provide a significant, inherent vulnerability within the Solaris operating system.

2.1 Operating Systems and the Solaris Kernel

Before giving a description of LKMs and how they can be exploited, the following is a very brief introduction to the Unix operating system, its kernel and the Solaris boot process. It is important to understand these basic concepts to fully evaluate the risk of kernel-level rootkits. References for much more thorough discussions of the Solaris kernel are listed at the end of this paper.

The operating system, simply put, is a collection of system programs, which allow users to run other application programs. By abstracting the machine hardware into a "virtual" machine, the operating system provides a consistent environment for the software that runs on the machine and gives the user a "look and feel" to the computer system. (1)

The "kernel" is the core of the operating system whose primary functions are to manage the hardware by allocating its resources among the programs running on it, and to supply a set of system services for those programs to use. (2)

Operating systems are generally classified as having either a microkernel or monolithic design. A microkernel design has separate processes (modules) that run in a privileged mode, but communicate with each other by passing messages. The "microkernel" itself is little more than a message hub, while the modules provide the functionality. The goal of this design is to keep the microkernel as small as possible. On the other hand, the monolithic design is one large process, which may be subdivided into modules internally, but when run, is a single large binary image. Its modules do not pass messages, but communicate directly by calling functions in other modules. (3)

An advantage to a microkernel design is a potential for more efficient use of memory, as modules are loaded into memory only as they are called upon, and unneeded modules are never loaded. The LKMs mechanism provides this dynamic capability to the operating system kernel by loading or unloading modules in response to system calls, or the kernel's resource requirements. Furthermore, modules can be developed, tested and modified, without having to add the code to the "kernel", recompile the kernel and reboot the system.

An advantage of a monolithic kernel design is that it provides a wholly contained binary that cannot be altered without recompilation and rebooting. The security implication of this is obvious. An operating system that cannot be altered while running has a lower degree of vulnerability than one that can be modified while the system is running. However, this does not mean a monolithic kernel has no vulnerabilities.

Solaris is a Unix operating system of a microkernel design. It is not possible to create a monolithic Solaris kernel (4). The Unix operating systems Linux and BSD are originally of a monolithic kernel design, but have added the ability to dynamically load or unload modules. Although this is somewhat of a hybrid of the two kernel designs, this functionality can be ignored, and a fully monolithic kernel produced. For this reason, Solaris is more vulnerable to a kernel-rootkit exploit, but Linux and FreeBSD are also susceptible to the same kind of exploit.

2.2 Booting the Kernel

Understanding the bootstrapping and initialization of the Solaris operating system can be very helpful towards auditing and defending Solaris against kernel rootkit exploits. The following draws heavily from "Solaris Internals" by Jim Mauro and Richard McDougall, whose book is highly recommended.

Booting the Solaris operating system from a local disk can be divided into six steps.

Step 1: The boot command - loading the bootblock

The first step in the boot process is to read and load the bootblock into memory. This process uses the system's firmware in PROM, known as Open-Boot PROM (OBP) in Solaris, to load the bootblock located at physical sectors 1-15 of the boot disk, provide NVRAM for setting system parameters, build the hardware device tree, and provide bootstrap support for manual or automatic booting of the system.

Step 2: The bootblock program – loading ufsboot

The second step is for the bootblock to locate and load the secondary boot program, ufsboot (for a local disk boot) or inetboot (for a network boot). The path and name of the secondary boot program is hardcoded into the bootblock program as /platform/<arch>/ufsboot, where <arch> is the hardware architecture type and can be determined by the "uname –m" command. The bootblock program cannot be larger than 7680 bytes (15 * 512 bytes), so it contains just enough code to read a Unix file system (UFS) directory, locate a file and load it into memory. Once ufsboot is loaded, the bootblock passes control to ufsboot.

Step 3: The ufsboot program – loading the core kernel and linker

The ufsboot program locates and loads the core kernel binary at /platform/<arch>/kernel/unix and the kernel linker program at /kernel/misc/krtld. The core kernel binary, unix, is the platform dependent component of the core kernel and is an executable and linking format (ELF) binary image file. The ufsboot program can parse the ELF headers, and based on that information loads the required krtld program and passes control to krtld.

Step 4: The krtld program – loading required kernel modules

The krtld program examines the ELF header information of the unix program and determines the dependencies the program has on other binary images. For the unix program, this includes /kernel/genunix, the platform and hardware independent binaries of the core kernel, /platform/<arch>/kernel/misc/platmod, the platform specific binaries of the core kernel, and /platform/<arch>/kernel/cpu/$CPU, the processor specific binaries of the core kernel.

As krtld encounters these dependencies, it searches for these specified modules. A key variable determines the path for which krtld will search for these modules. This variable is set in the OBP firmware or can be manually entered on the boot program's command line (boot –a). Late in the boot process, this path can be set within the /etc/system file. This is an important point from a security aspect as will be seen in section 8.2 below.

After the core kernel binaries (unix, krtld, genunix, platmod, and $CPU) have been loaded, krtld passes control to unix.

Step 5: Initializing the kernel

At this point, the Solaris kernel is running and is using virtual memory address space, but some further initialization is required before the first real user application is started. The kernel initializes some processor registers, and makes calls to mlsetup(), main() and startup(). These functions create the initial processes, map and initialize hardware devices and initialize memory. When the above initializations have completed, the operating system banner is displayed.

After some additional platform checking, the /etc/system kernel configuration file is accessed to create a linked list of system parameter data structures in kernel memory. The /etc/system file contains commands used to customize the operating environment of the kernel and are useful in controlling some aspects of LKMs, notably what modules cannot or must be loaded, and what the module search path should be.

LKMs have actually been loading at various times prior to this during the boot process. During startup(), the modules swap, specfs, procsfs and tod were loaded. Other times that loading occurs is during kernel subsystem or platform specific module initializations. As intended by the microkernel design, these modules are loaded as they are called, or dependencies are determined. However, once /etc/system has been accessed, LKMs can be force loaded into the kernel by commands within that file.

Note that at this juncture, the preliminary memory initialization determines how much physical memory is available after the core kernel modules have been loaded. This value can be seen in the boot logging information as "mem" and "avail mem."

Step 6: The init process – the first user

The kernel function newproc() is called from main() to create the init process that is the first real user process. The kernel allocates user address space to init rather than kernel address space, so that init does not use or execute within the kernel's memory address space. Init is the last process created by the kernel to get the system running. Init is the ancestor of all subsequent unix processes and the direct parent of login shells.

The remaining bootup processes are completed by init, take place within user memory address space and are determined by entries in the file /etc/inittab. These entries define the system's default state and controls the execution of scripts in the /etc/rc*.d directories. These scripts are run to bring the system to a know status, specifying which services are to be started. Init checks the integrity of the root and usr file systems first, mounts local disks, performs file system cleanup, starts system and network services, mounts remote disks, and finally, enables logins by starting getty.

3.0 Description of Protocols/Services:

As mentioned in the introduction, LKMs are binary object files that are code modules that can be loaded or unloaded from the running Solaris kernel based on code dependencies and resource requirements. LKMs are defined in /usr/include/sys/modctl.h and are one of seven types; device drivers, system calls, file systems, misc (miscellaneous), streams modules, scheduling classes and exec file type.

Pragmatic (pseudonym), who has written in-depth articles about LKMs, loosely compared them to “old DOS TSR programs, they were our gate to staying resident in memory and catching every interrupt we wanted.”

3.1 Loading and Linking LKMs

Each of the LKMs types has their own specific installation steps, but the steps are similar in nature. The module is loaded into memory and kernel address space is mapped to the modules’ text and data segments.

The kernel function modload() starts this process, and is initiated by calls within the running kernel, or by the user program modload(1). The kernel maintains a linked list of structures for all the modules loaded in the kernel. These structure are defined by modctl and module in /usr/include/sys/modctl.h and /usr/include/sys/kobj.h. Some important structure elements that will come into play are the module name, mod_modname, the module id, mod_id, and additional module information in mod_modinfo and mod_linkage.

When modload() is called, it will initially search the linked list of module structures to see if the desired module’s structure has already been created. If it does not exist, a new structure is created and added to the linked list. It is interesting to note that even if a module is unloaded, its module structure remains in the linked list, and an element in the structure, mod_loaded, is cleared. Thus, all of the modules loaded while the system has been running can be determined from this linked list.

If the module does need to be loaded, the krtld module is called to create address space segments and bindings, and load the binary object into memory, and sets the mod_loaded element in the module’s modctl structure. Finally, it executes the module’s _init() routine to complete the task of initializing the module for use within the kernel.

3.2 Kernel Symbols and Module Information

Since modules can be loaded and unloaded as needed, the kernel’s table of module symbols must remain dynamic. A pseudodevice, /dev/ksyms, contains the currently loaded module symbols and is maintained by the device driver /usr/kernel/drv/ksyms. It is important to understand that this list of module symbols is just a list of names of variables and functions contained in the modules and their associated virtual addresses. You can actually view this table using the command *nm –x /dev/ksyms.* I have found it useful to modify the output using the awk command, so that the address is printed first, rather than the symbol id. The advantage is that you can sort the list by virtual address. The command is as follows: *nm –s /dev/ksyms | awk '{print $2, $1, $3, $4, $5, $6}' | sort*.

The modinfo(1M) command is another useful tool for listing what modules are currently loaded. The output from this command lists the module's id, the virtual address at which it was loaded (in hex), size of the module (in hex bytes), some module-specific data (info), a revision number, and the module's name. The id numbers will not necessarily be contiguous. As a module is unloaded, its id may be released for use by another module, so that at any given time, gaps in the sequence of module ids will be present. Solaris 7 typically has around 90 modules listed, while Solaris 8 has about 110 (5).

3.3 Module Coding Requirements

As stated above, a module must have an init() routine for the proper completion of loading and initialization. Required within the init() function must be a call to modinstall function, specific to the module type, which declares and initializes the associated mod_linkage structure and a generic modlinkage for the generic module abstraction.

In addition, a module must have _fini() and _info() functions. The _fini() function prepares a module for unloading, and the _info() function which provides information about a module while it is loaded.

The coding of LKMs is beyond the scope of this paper, but there are several sources listed in the references section that are helpful. The manual pages are worth looking at (_info(9E), mod_install(9F)), but an excellent introduction to coding Solaris LKMs is presented in the paper by plasmoid (pseudonym) entitled "Solaris Loadable Kernel Modules."

When these modules are compiled and linked, it is necessary to include the –D_KERNEL switch when compiling, and the –r flag when linking. Furthermore, since the kernel does not contain many standard C functions, it may be necessary to extract them from the /lib/libc.a library using the *ar –x* command, and then linking them in manually. The process is seen below:

```
ar –x /lib/libc.a c_function.o
gcc –D_KERNEL –DSVR4 –DSOL2 –o2 module_name.c
ld –o module_name –r module_name.o c_function.o
```

The binary image file must now be placed in a directory within the kernel module search path before it can be loaded into the kernel.

4.0 Description of Variants:

This exploit has been "in the wild" for some time, though not specifically for Solaris. SunOS 4.x did have a loadable module interface, and an attack to snoop tty used LKMs called *tap* (6).

There were earlier discussions about utilizing LKMs, but the first major article was published in Prack 50 Article 5, "Abuse of the Linux Kernel for Fun and Profit" (April 9, 1997.) It was written by halflife (pseudonym) and discussed TTY hijacking using LKMs in a Linux kernel. This module was called *linspy*.

Another extensive paper written by pragmatic entitled "(nearly) Complete Linux Loadable Kernel Modules" was released in March of 1999, which went into extensive detail on writing LKMs for Linux, discussed ways in which the kernel could be subverted, and gave numerous code examples from many sources, including most of the "classic" code on which others have based their versions of this exploit. Among the many examples are the modules *heroin*, one of the first examples of an LKM used to hide files and processes, and *itf*, the Integrated Trojan Facility, which was based on *heroin* and in pragmatic's words, "has everything you need to backdoor a system in a very effective way." *Itf*, was published in Prack 52, Article 18, "Weakening the Linux Kernel" (January 26, 1998) and was written by plaguez (pseudonym). Another popular Linux module is *Knark*, which was written by Creed and released around November of 1999. It was based on *itf*. Also TESO has released a Linux module named *adore* that is similar to *itf*.

Pragmatic also released a paper entitled "Attacking FreeBSD with Kernel Modules" in June of 1999, which covered the same kinds of methods from the point of view of the BSD kernel.

In December of 1999, plasmoid released an article entitled "Solaris Loadable Kernel Modules" which discussed similar techniques from the point of view of Solaris. The code examples used in his paper were taken from the Solaris Integrated Trojan Facility (SITF), a small collection of coded modules that illustrate the basic exploit techniques. The module *sitf0.2* incorporates these techniques into one loadable module, providing a general kernel rootkit. *Sitf0.2* is also based on the *itf* module for Linux.

The basic set of "features" for these modules are module hiding, file and directory hiding, process hiding, execution redirection, grant root access to a uid, and promiscuous flag hiding.

The differences between the modules have to do with the specifics of the operating system and the methods approach, rather than the concepts. Although it is a non-trivial task, these modules can be ported to various Unix operating systems that support LKMs, but attention must be paid to the details of the structures and system calls. A difference in methods is seen by *Knark's* use of a signal 31 to hide a process, while SITF uses a remote switch to allow the attacker to hide or unhide processes based on a key embedded in their name. As with any programming, there are many solutions for a problem, so there may be a variety of modules providing a number of features, but the basic concepts of exploiting LKMs remains the same, and provides a very fertile ground for future development.

5.0 How the Exploit Works:

It should be noted right at the beginning that the user must have root access to use this kind of exploit. As mentioned above, the purpose of a rootkit is to cover the activity of an attacker once they have gained root access, and ensure that they can maintain root access.

A kernel rootkit installs LKMs that modify or replace the actions and output of other existing LKMs that are a normal part of the operating system. These modules are able to operate at a privileged level within the kernel, and can operate within the kernel memory space, and to some degree, interface with the user memory space. The LKMs can hide their presence in the running kernel, redirect kernel system calls, hide files and directories, and redirect calls of user executable binaries.

The sitf0.2 module, within the SITF, specifically takes advantage of Solaris kernel modules and several deficiencies in some of the Solaris module code. The sitf0.2 module is declared as a miscellaneous operations type (misc) module which is defined by the mod_miscops structure in /usr/include/sys/modctl.h. Once it has been loaded into the system, it is capable of the following features detailed below.

5.1 Stealth Modules

As mentioned above, the module name is stored in the module's linkage structure. Normally, the module's name is a character string and is usually a short descriptive phrase about the module's functionality. For example, the kb module's name is "stream module for keyboard" and the *modinfo* command would show an entry for kb such as:

```
 41  f5a95bf0   3b19  8  1  kb (stream module for keyboard)
```

However, if the module's name is null (""), no information about the module is printed by the modinfo command, even though the module is loaded, has an assigned id, and is fully operational. Plasmoid admits in his paper on Solaris LKMs "even if this protection leaving the module's name blank is weak, it will fit your needs, if the system administrator is not a real system programmer."

The reason it is considered a weak technique is that when a module is loaded, its symbols are mapped and listed in the kernel symbols table, /dev/ksyms. Plasmoid, in his discussion of this fact, indicated a more complete method for hiding the module would be to patch the Solaris module that lists and manages all kernel symbols, and suggested he would explain the technique in a second version of his article. As yet, I have been unable to find any reference that he has ever released this second version. If the related symbols were excluded from the list in the /dev/ksyms, it would be much more difficult to detect a hidden module and might require "real system programmer" skills.

Another technique mentioned by pragmatic was to avoid exporting any symbols used in the LKMs, defining a symbol table within the module itself, and thus avoiding any exposure within the kernel symbol table. However, this was specific for Linux, and I have not seen this technique used in a Solaris module, but something similar may be possible.

5.2 Redirection of System Calls

By intercepting and redirecting system calls within the kernel, it is possible to change the way the operating system reacts to various calls or commands. System calls are the basic kernel functions that are used to perform most operations on a system. They are callable interfaces available to user programs so that the user program can request the kernel to perform specific actions on their behalf. For example, the open64() system call opens a file in a filesystem and the read() system call extracts data from an opened file. A list of system calls is available in the file /usr/include/sys/syscall.h.

System calls are referenced through a kernel table named sysent. Sysent contains structures for each system call available and is indexed by a system call number, specified in the /etc/name_to_sysnum file. Many of the system calls are implemented as LKMs and are stored in /kernel/sys and /usr/kernel/sys directories.

Redirection of system calls requires three things. There must be a replacement function, in jargon, a faked syscall, the sysent table must be modified to point to the faked syscalls structure. Finally, the LKM stores the original pointer of the syscalls so that it maintains full functionality.

An important aid to faking a system call, is the */usr/bin/truss* command. Truss will output a trace of system calls that are made for a command. The command */usr/bin/truss touch test_file* will show all the system calls that are made while executing the command to create the file test_file. It includes such system calls as execve(), open(), stat(), fstat(), mmap(), close(), time(), stat64(), creat64(), utime() and _exit().

By determining what system calls a particular command of interest has, will determine what system calls might be affected by redirection.

5.3 File and Directory Hiding

There are actually two aspects to hiding files and directories. Not only are files and directories hidden from being listed, but the user is also prevented from even opening the file or changing the current directory to a hidden one.

Listing files and directories uses the getdents64() system call (syscall) from such commands as *ls* or *du*. (This can be seen by using the *truss* program mentioned above) If a faked syscall routine is created to simply not list certain files, then the output will never contain entries for those files. To avoid creating some lengthy list of files or directories to hide, the technique used by SITF is to include a “magic” string within the file or directory name that is specified by the attacker within the LKM. The default value in the sitf2.0 module is “blah” and any name containing that string is not listed in the output. Using the methods described in 5.2 above, the attacker crafts a faked getdents64() routine, such as faked_getdents64(), stores the original pointer of getdents64(), and loads the new pointer to faked_getdents64() into the sysent table. When a call is made by a user program to getdents64(), the faked_getdents64() routine handles the request, using the actual getdents64() routine to retrieve the information, deleting any entries in the list that contain the magic string.

A similar technique is used to prevent users from opening or entering a hidden file or directory. Faked open64() and chdir() routines intercept the user request. If the request is for a file that contains the magic string, the faked routine returns the error message: “No such file or directory.” See diagram in section 6.0.

5.4 Process Hiding

Every process has an associated prot_t structure which is defined in /usr/include/sys/proc.h. The process structure provides the basis for creating and managing processes in the Solaris operating system. Within the prot_t structure is the structure name *user*, which is defined in /usr/include/sys/user.h. One of the members of the *user* structure is the member u_psargs, which contains the name of the binary image file and its arguments.

Solaris creates special files based on the entries in prot_t and places them in the /proc directory. This is actually a pseudo file system that exports the kernel’s process model and abstractions by providing a file-like interface to the user so that they can retrieve information about the processes and have the capability to control processes and debug system problems.

Since the name of the executable can be determined for every process, and that this information is retrieved through a filesystem type of interface, then the faked syscall for getdents64(), mentioned above in 5.3, can be slightly modified to include the code to search for the process name, and omit from any listing a process name which contains the magic string. Thus, neither the use of the *ps* command or a directory listing of /proc would indicate the presence of the hidden process.

5.5 Remote Switch

As mentioned in section 4.0, SITF makes use of a “remote switch” to toggle whether or not files, directories and processes containing the magic string will be hidden or not. This provides the attacker with a means of debugging the installed rootkit, or working with other files that have been loaded onto the compromised system.

A faked syscall is again utilized to intercept a request that contains a special string, this time referred to as a security “key.” If the key string is present, then the security bit is toggled to either turn on hiding or turn off hiding of names that contain the magic string.

SITF implements this through the *touch* command and its use of the syscall creat64(). A faked version of creat64() checks for the security key string in the request to create a new file, and if the key string is present, it toggles the security switch.

5.6 Program Redirection

Redirecting the execution of an intended user program to another alternate program is not a new concept and there are numerous viruses and Trojan programs that exist to do this. Usually, these programs can be detected and eradicated with antiviral or integrity checking software.

In this technique, a faked execve() syscall is used to check the name of the requested program to execute, and replaces the name with an alternate and then lets the original execve() function execute.

The implication of this, is that an alternate program could be placed anywhere on the system and hidden. When a call is made to a specific program, like passwd for example, than an alternate program is run that would most likely perform additional functions to passwd's more traditional ones, like collecting passwords in a hidden file.

In SITF, only one program is redirected, with the original, and alternate program being specified within the source code of the LKM.

5.7 Root Access

This is actually a very simple technique than can give a user full root access. A faked setuid() syscall merely checks to see if a specific uid is being requested, and if so, makes syscalls passing id 0 to seteuid(), setgid(), setegid() and finally the original setuid() granting superuser rights to that uid.

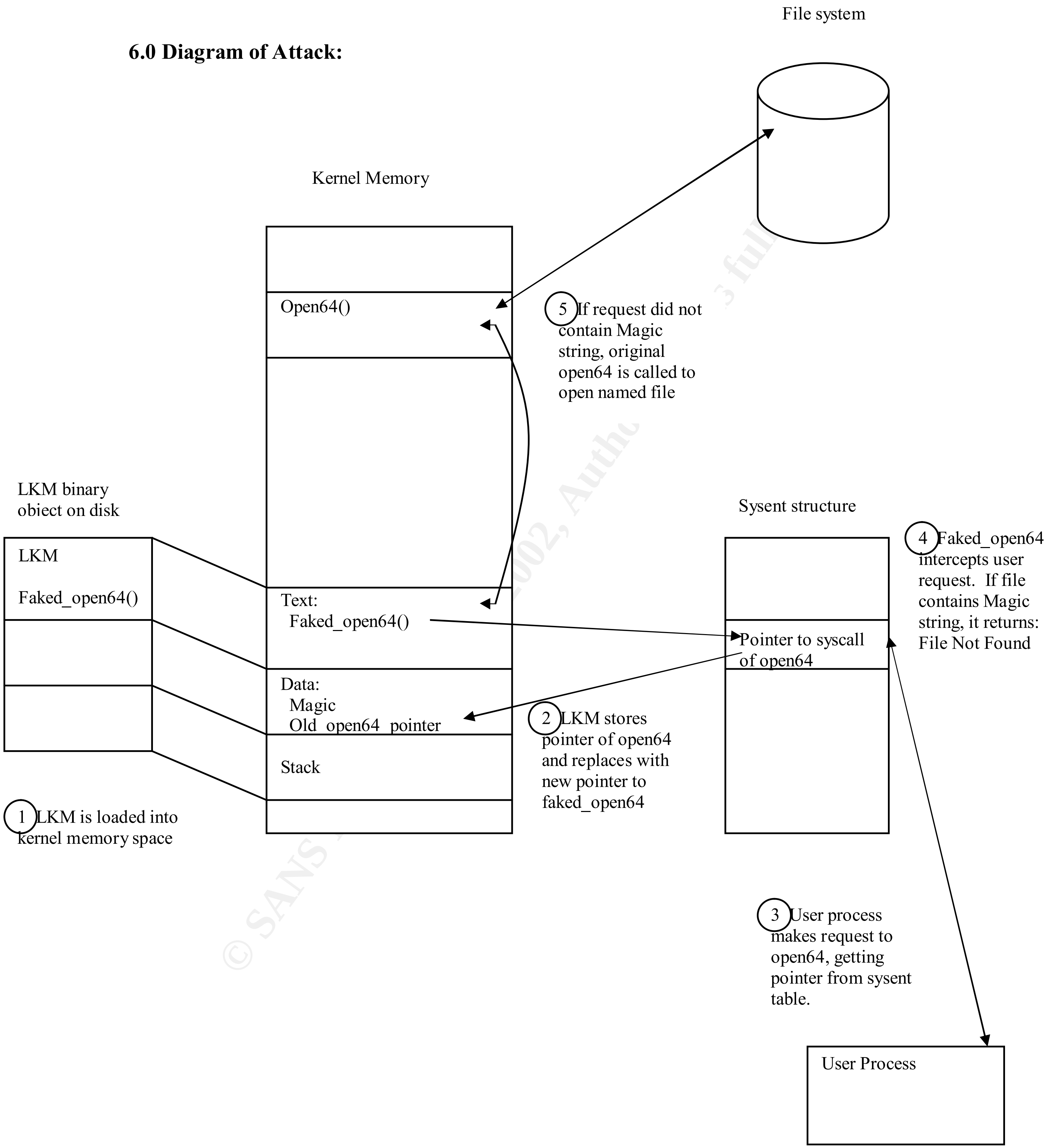
What is especially disturbing about this is that the faked function is only 13 lines of code, and if just this function was included in a LKM, it could be a very effective backdoor with a very small signature.

5.8 Promiscuous Flag Hiding

This feature follows the same scheme as those above; fake the ioctl() syscall, modify the output from the original ioctl() based on the status of the of the interface and return the results to the user.

In sitf0.2, this is only done once, so that if a subsequent command by a user to actually place the interface into promiscuous mode is given, the user would not detect that it had previously been hidden.

6.0 Diagram of Attack:



7.0 Signature of the Attack:

What makes this exploit difficult to detect is that you cannot always trust the output that the kernel is providing you with. Since the very heart of the operating system is being compromised, files may be hidden from any integrity checks. It may be that all the original system files have not been modified at all, it is just that their requests are being intercepted and their output altered without any indication this is occurring.

However, there appears to be several pieces of information that you could use as an audit trail, in case you suspect your system has been compromised. Additionally, I have found that comparisons of the output from modinfo, crossed referenced with the kernel symbol table, may be of use in trying to detect the presence of an unauthorized LKM.

7.1 Modinfo and the Kernel Symbol Table /dev/ksyms

After learning how a module could keep its information from being displayed by the command *modinfo*, I examined the output from the command *nm –x /dev/ksyms* and was able to locate module names based on the load address in virtual memory. I decided to try to correlate the output from the two commands to see if it was possible to identify all of the kernel symbols with a corresponding module id in the modinfo output. I wrote a simple perl program based on the following algorithm:

- Load all known modules from modinfo output into a hash table indexed by load address
- Sort the output of the command *nm –x /dev/ksyms* by load address
- Loop through until the symbols of the core binary modules from bootup have been passed. These are the unix, krtld, genunix, platmod and $CPU discussed in section 2.0
- Search for a function symbol (FUNC) of name _init, or a transition from an object symbol (OBJT) to a FUNC symbol
- Using the load address of the current FUNC symbol, see if an element exists in the modinfo hash
- If the element does not exist, flag the FUNC symbol as suspect
- If the element exists, list the FUNC symbol as a known LKM
- Using the size of the module from the modinfo hash, loop through the address range bounded by the module load address and that address plus the size of the module
- Start searching for new function symbols

For the most part, this algorithm worked, but did give a few false positives. By eliminating most of the symbols in the kernel symbol table, it was possible to manually compare the symbols for the flagged functions and discern whether they were symbols associate with modinfo-listed modules.

Below is an example of the signature I was able to detect when I loaded the sitf0.2 module on a Solaris 7, 32-bit, sparc architecture. Section 7.2 lists the output of the *modinfo* command, sorted by load address in virtual memory. Section 7.3 lists the search results from the perl program mentioned above. The initial four functions are false positives, but the last entry is the sitf0.2 LKM. By looking at the sorted kernel symbol table output, and examining the symbols around the address indicated by the unknown _init() function, it can be quickly seen that this is not a normal module. The extracted output is listed in section 7.4. Note the references to newioctl, newcreat64, newchdir, newopen64, newgetdents64, newexecve, and newsetuid. Definitely symbols to be concerned about.

A couple of notes about this output are in order. This was produced right after the system had been rebooted. Other times that I repeated this process, the unknown module would appear at addresses mixed in between existing modules. This method also worked on Solaris 7 64-bit as well as Solaris 8, though with more false positives.

As long as the kernel symbol table contains the module symbols and has not been subverted by patching the kernel symbol table device driver, then the above concept may be useful in locating unauthorized modules. However, additional techniques should be attempted. These are discussed in section 7.5 below.

7.2 Sorted Modinfo Output with sit0.2 Module Installed:

```
Load Addr    ID Load Addr  Size   Module Name
---------------------------------------------
f59e1000 --   5 f59e1000   4577     specfs
f59e5994 --  78 f59e5994   1c19     tlimod
f59e7378 --  80 f59e7378    2d8        ipc
f59e7670 --   7 f59e7670   2ddc         TS
f59ea45c --   8 f59ea45c    4f0   TS_DPTBL
f59ea94c --   9 f59ea94c  27c28        ufs
f5a12574 --  10 f5a12574   ec4c     rpcmod
f5a211c0 --  11 f5a211c0  28f84         ip
f5a4bfb8 --  12 f5a4bfb8    ce3    rootnex
f5a4cc9c --  13 f5a4cc9c    1ec    options
f5a4ce88 --  14 f5a4ce88    76c        dma
f5a4d5f4 --  15 f5a4d5f4    cb7       sbus
f5a4e2ac --  16 f5a4e2ac   1ae7      iommu
f5a4fd94 --  17 f5a4fd94   1648        sad
f5a513e8 --  18 f5a513e8    61f     pseudo
f5a51a0c --  19 f5a51a0c  103bc         sd
f5a61dc8 --  20 f5a61dc8   7136       scsi
f5a68f18 --  21 f5a68f18   d6f5        esp
f5a78378 --  28 f5a78378  12926     procfs
f5a89cac --  35 f5a89cac   45d0        udp
f5a8d27c --  77 f5a8d27c   92a3     rpcsec
f5a93ef0 --  87 f5a93ef0   163b       ptem
f5a952dc --  71 f5a952dc    7f6      kstat
f5a95e44 --  32 f5a95e44    616      clone
f5a9afd4 --  34 f5a9afd4   11a1        md5
f5a9d178 --  86 f5a9d178    e53        pts
f5a9dd04 --  64 f5a9dd04    4c5   intpexec
f5a9e094 --  90 f5a9e094    5dd      ledma
f5a9e94c --  26 f5a9e94c   15c3       dada
f5a9ff60 --  30 f5a9ff60   d008     sockfs
f5aac528 --  33 f5aac528  17b50        tcp
f5abf500 --  38 f5abf500   45b7      timod
f5ac3ab8 --  85 f5ac3ab8    f0f        ptm
f5ac4da4 --  40 f5ac4da4   868f         zs
f5acd434 --  41 f5acd434    58b       obio
f5ad1290 --  81 f5ad1290    29b     connld
f5ad13cc --  82 f5ad13cc    105         IA
```

```
f5ad1444 --  43 f5ad1444   1800         ms
f5ad2c44 --  44 f5ad2c44    a1c     consms
f5ad3660 --  45 f5ad3660   3d42         kb
f5ad73a4 --  46 f5ad73a4    b55    conskbd
f5ad7efc --  47 f5ad7efc   1955         wc
f5ad9854 --  48 f5ad9854    d64      iwscn
f5ada5b8 --  49 f5ada5b8   234f    elfexec
f5adc908 --  50 f5adc908   103d         mm
f5add948 --  51 f5add948   328c     fifofs
f5ae0cf0 --  52 f5ae0cf0   5926     ldterm
f5ae6618 --  53 f5ae6618   2381   ttcompat
f5ae899c --  54 f5ae899c   14d0       ptsl
f5ae9e6c --  55 f5ae9e6c   2053        ptc
f5aebec0 --  84 f5aebec0   1670        hwc
f5aed6f8 --  88 f5aed6f8    259   redirmod
f5aed848 --  61 f5aed848   4683         tl
f5af1ecc --  62 f5af1ecc   160a     sysmsg
f5af34d8 --  63 f5af34d8    6d8         cn
f5af4078 --  65 f5af4078    2fc       pipe
f5af51c4 --  68 f5af51c4    d70       fdfs
f5af71f4 --  67 f5af71f4   730e    ufs_log
f5afdc88 --  69 f5afdc88   3e12     doorfs
f5b0153c --  70 f5b0153c   1488     namefs
f5b026d4 --  72 f5b026d4   d8a2      tmpfs
f5b0ff78 --  73 f5b0ff78    9db        log
f5b10954 --  74 f5b10954    8c3         sy
f5b11218 --  75 f5b11218   4f90        vol
f5b161a8 --  76 f5b161a8  262f4        nfs
f5b3b138 --  79 f5b3b138   2290     semsys
f5b3d1a8 --  83 f5b3d1a8   2ea6         pm
f5b3fbc8 --  42 f5b3fbc8   3c34      cgsix
f5b43230 --  89 f5b43230   5f0e         le
f5b49140 --  37 f5b49140   51a7        arp
f5b4e2e8 --  59 f5b4e2e8   1988        rts
f5b4fc70 --  36 f5b4fc70   3b58       icmp
f5b537c8 --  91 f5b537c8    858      ksyms
```

7.3 Output from Perl Search Program

```
              Modinfo entry                    Kernel Symbol Table entry
Status      ID Loadaddr   Size     Mod Name      Load addr     size      Type   symbol name

UNKNOWN --                                   -- |0xf007e700|0x00000b20|FUNC |_kobj_boot
UNKNOWN --                                   -- |0xf008dda8|0x000001a0|FUNC |true_add
UNKNOWN --                                   -- |0xf012f8f0|0x00000020|FUNC |tsu_module_identify
UNKNOWN --                                   -- |0xf027c694|0x00000010|FUNC |.mul
FOUND   --   5 f59e1000   4577     specfs -- |0xf59e1000|0x00000270|FUNC |specvp
FOUND   --  78 f59e5994   1c19     tlimod -- |0xf59e5994|0x000019b4|FUNC |_init
FOUND   --  80 f59e7378    2d8        ipc -- |0xf59e7378|0x00000234|FUNC |_init
FOUND   --   7 f59e7670   2ddc         TS -- |0xf59e7670|0x000026c4|FUNC |_init
FOUND   --   8 f59ea45c    4f0   TS_DPTBL -- |0xf59ea45c|0x00000050|FUNC |_init
FOUND   --   9 f59ea94c  27c28        ufs -- |0xf59ea94c|0x000001dc|FUNC |alloc
FOUND   --  10 f5a12574   ec4c     rpcmod -- |0xf5a12574|0x0000c9f4|FUNC |_init
FOUND   --  11 f5a211c0  28f84         ip -- |0xf5a211c0|0x00020178|FUNC |_init
FOUND   --  12 f5a4bfb8    ce3    rootnex -- |0xf5a4bfb8|0x00000a1c|FUNC |_init
FOUND   --  13 f5a4cc9c    1ec    options -- |0xf5a4cc9c|0x00000118|FUNC |_init
FOUND   --  14 f5a4ce88    76c        dma -- |0xf5a4ce88|0x000004fc|FUNC |_init
FOUND   --  15 f5a4d5f4    cb7       sbus -- |0xf5a4d5f4|0x00000990|FUNC |_init
FOUND   --  16 f5a4e2ac   1ae7      iommu -- |0xf5a4e2ac|0x00001764|FUNC |_init
FOUND   --  17 f5a4fd94   1648        sad -- |0xf5a4fd94|0x000011b0|FUNC |_init
FOUND   --  18 f5a513e8    61f     pseudo -- |0xf5a513e8|0x000003b8|FUNC |_init
FOUND   --  19 f5a51a0c  103bc         sd -- |0xf5a51a0c|0x0000da28|FUNC |_init
FOUND   --  20 f5a61dc8   7136       scsi -- |0xf5a61dc8|0x00000010|FUNC |scsi_ifgetcap
FOUND   --  21 f5a68f18   d6f5        esp -- |0xf5a68f18|0x0000babc|FUNC |_init
PROCFS  --  28 f5a78378  12926     procfs -- |0xf5a783e8|0x000000b4|FUNC |ctlsize
FOUND   --  35 f5a89cac   45d0        udp -- |0xf5a89cac|0x000035c4|FUNC |_init
FOUND   --  77 f5a8d27c   92a3     rpcsec -- |0xf5a8d27c|0x000068d0|FUNC |_init
FOUND   --  87 f5a93ef0   163b       ptem -- |0xf5a93ef0|0x000013c0|FUNC |_init
FOUND   --  71 f5a952dc    7f6      kstat -- |0xf5a952dc|0x0000062c|FUNC |_init
FOUND   --  32 f5a95e44    616      clone -- |0xf5a95e44|0x000003e0|FUNC |_init
```

```
FOUND   --  34 f5a9afd4   11a1        md5 -- |0xf5a9afd4|0x00000fa4|FUNC |_init
FOUND   --  86 f5a9d178    e53        pts -- |0xf5a9d178|0x00000b80|FUNC |_init
FOUND   --  64 f5a9dd04    4c5   intpexec -- |0xf5a9dd04|0x00000380|FUNC |_init
FOUND   --  90 f5a9e094    5dd      ledma -- |0xf5a9e094|0x00000364|FUNC |_init
FOUND   --  26 f5a9e94c   15c3       dada -- |0xf5a9e94c|0x00000074|FUNC |dcd_initialize_hba_interface
FOUND   --  30 f5a9ff60   d008     sockfs -- |0xf5a9ff60|0x000000f0|FUNC |sogetvp
FOUND   --  33 f5aac528  17b50        tcp -- |0xf5aac528|0x00012a64|FUNC |_init
FOUND   --  38 f5abf500   45b7      timod -- |0xf5abf500|0x00003674|FUNC |_init
FOUND   --  85 f5ac3ab8    f0f        ptm -- |0xf5ac3ab8|0x00000bec|FUNC |_init
FOUND   --  40 f5ac4da4   868f         zs -- |0xf5ac4da4|0x0000002c|FUNC |zsa_null
FOUND   --  41 f5acd434    58b       obio -- |0xf5acd434|0x0000038c|FUNC |_init
FOUND   --  81 f5ad1290    29b     connld -- |0xf5ad1290|0x00000138|FUNC |_init
FOUND   --  82 f5ad13cc    105         IA -- |0xf5ad13cc|0x0000004c|FUNC |_init
FOUND   --  43 f5ad1444   1800         ms -- |0xf5ad1444|0x000014dc|FUNC |_init
FOUND   --  44 f5ad2c44    a1c     consms -- |0xf5ad2c44|0x000006a4|FUNC |_init
FOUND   --  45 f5ad3660   3d42         kb -- |0xf5ad3660|0x000028c8|FUNC |_init
FOUND   --  46 f5ad73a4    b55    conskbd -- |0xf5ad73a4|0x000007b4|FUNC |_init
FOUND   --  47 f5ad7efc   1955         wc -- |0xf5ad7efc|0x00000d38|FUNC |_init
FOUND   --  48 f5ad9854    d64      iwscn -- |0xf5ad9854|0x00000ab4|FUNC |_init
FOUND   --  49 f5ada5b8   234f    elfexec -- |0xf5ada5b8|0x000009d8|FUNC |elfexec
FOUND   --  50 f5adc908   103d         mm -- |0xf5adc908|0x000000e8|FUNC |mm_attach
FOUND   --  51 f5add948   328c     fifofs -- |0xf5add948|0x00002db0|FUNC |_init
FOUND   --  52 f5ae0cf0   5926     ldterm -- |0xf5ae0cf0|0x00004d40|FUNC |_init
FOUND   --  53 f5ae6618   2381   ttcompat -- |0xf5ae6618|0x00002120|FUNC |_init
FOUND   --  54 f5ae899c   14d0       ptsl -- |0xf5ae899c|0x00001124|FUNC |_init
FOUND   --  55 f5ae9e6c   2053        ptc -- |0xf5ae9e6c|0x00001cf4|FUNC |_init
FOUND   --  84 f5aebec0   1670        hwc -- |0xf5aebec0|0x0000156c|FUNC |_init
FOUND   --  88 f5aed6f8    259   redirmod -- |0xf5aed6f8|0x000000f4|FUNC |_init
FOUND   --  61 f5aed848   4683         tl -- |0xf5aed848|0x00004174|FUNC |_init
FOUND   --  62 f5af1ecc   160a     sysmsg -- |0xf5af1ecc|0x00000de4|FUNC |_init
FOUND   --  63 f5af34d8    6d8         cn -- |0xf5af34d8|0x000004ac|FUNC |_init
FOUND   --  65 f5af4078    2fc       pipe -- |0xf5af4078|0x000001d8|FUNC |_init
FOUND   --  68 f5af51c4    d70       fdfs -- |0xf5af51c4|0x00000a6c|FUNC |_init
FOUND   --  67 f5af71f4   730e    ufs_log -- |0xf5af71f4|0x0000002c|FUNC |lufs_sv_constructor
FOUND   --  69 f5afdc88   3e12     doorfs -- |0xf5afdc88|0x00000008|FUNC |door_open
FOUND   --  70 f5b0153c   1488     namefs -- |0xf5b0153c|0x00000024|FUNC |nameinsert
FOUND   --  72 f5b026d4   d8a2      tmpfs -- |0xf5b026d4|0x00000040|FUNC |tmpfs_hash_init
FOUND   --  73 f5b0ff78    9db        log -- |0xf5b0ff78|0x00000040|FUNC |log_info
FOUND   --  74 f5b10954    8c3         sy -- |0xf5b10954|0x000006a8|FUNC |_init
FOUND   --  75 f5b11218   4f90        vol -- |0xf5b11218|0x00003c38|FUNC |_init
FOUND   --  76 f5b161a8  262f4        nfs -- |0xf5b161a8|0x00000138|FUNC |nfs_validate_caches
FOUND   --  79 f5b3b138   2290     semsys -- |0xf5b3b138|0x00001fd0|FUNC |_init
FOUND   --  83 f5b3d1a8   2ea6         pm -- |0xf5b3d1a8|0x00002788|FUNC |_init
FOUND   --  42 f5b3fbc8   3c34      cgsix -- |0xf5b3fbc8|0x00003570|FUNC |_init
FOUND   --  89 f5b43230   5f0e         le -- |0xf5b43230|0x000050a4|FUNC |_init
FOUND   --  37 f5b49140   51a7        arp -- |0xf5b49140|0x00003e04|FUNC |_init
FOUND   --  59 f5b4e2e8   1988        rts -- |0xf5b4e2e8|0x000011a0|FUNC |_init
FOUND   --  36 f5b4fc70   3b58       icmp -- |0xf5b4fc70|0x00002a7c|FUNC |_init
FOUND   --  91 f5b537c8    858      ksyms -- |0xf5b537c8|0x000000d4|FUNC |ksyms_mapin
UNKNOWN --                               -- |0xf5b54354|0x00000b5c|FUNC |_init
```

7.4 Kernel Symbol Table Entries for sitf0.2 Module

Data Segment

Load addr	Size	Type	Symbol name
0xf5af6eb0	0x00000005	OBJT	magic
0xf5af6eb8	0x00000006	OBJT	key
0xf5af6ec0	0x00000009	OBJT	oldcmd
0xf5af6ed0	0x0000001d	OBJT	newcmd
0xf5af6ef0	0x00000004	OBJT	security
0xf5af6ef4	0x00000004	OBJT	promisc
0xf5af6ef8	0x00000008	OBJT	modlmisc
0xf5af6f00	0x00000014	OBJT	modlinkage

Text Segment

Load addr	Size	Type	Symbol name
0xf5b53db8	0x00000000	NOTY	gcc2_compiled.
0xf5b53db8	0x0000006c	FUNC	check_process
0xf5b53e24	0x0000003c	FUNC	check_for_process
0xf5b53e60	0x00000054	FUNC	sitf_isdigit

```
|0xf5b53eb4|0x0000007c|FUNC |sitf_atoi
|0xf5b53f30|0x000000c4|FUNC |newioctl
|0xf5b53ff4|0x00000084|FUNC |newcreat64
|0xf5b54078|0x00000074|FUNC |newchdir
|0xf5b540ec|0x00000080|FUNC |newopen64
|0xf5b5416c|0x0000010c|FUNC |newgetdents64
|0xf5b54278|0x0000008c|FUNC |newexecve
|0xf5b54304|0x00000050|FUNC |newsetuid
|0xf5b54354|0x00000b5c|FUNC |_init
|0xf5b54434|0x0000001c|FUNC |_info
|0xf5b54450|0x00000b5c|FUNC |_fini
|0xf5b544dc|0x000001f4|FUNC |_memmove
|0xf5b544dc|0x000001f4|FUNC |memmove
|0xf5b54518|0x00000000|NOTY |s1algn
|0xf5b54538|0x00000000|NOTY |s2algn
|0xf5b54554|0x00000000|NOTY |aldst
|0xf5b54558|0x00000000|NOTY |ald
|0xf5b54568|0x00000000|NOTY |w3cp
|0xf5b545c0|0x00000000|NOTY |w1cp
|0xf5b5460c|0x00000000|NOTY |w2cp
|0xf5b54660|0x00000000|NOTY |w4cp
|0xf5b54684|0x00000000|NOTY |dbytecp
|0xf5b546a8|0x00000000|NOTY |ovbc
|0xf5b546d0|0x000001b0|FUNC |_memcpy
|0xf5b546d0|0x000001b0|FUNC |memcpy
|0xf5b54880|0x00000094|FUNC |strstr
```

7.5 Additional Auditing

The dynamic nature of LKMs, and the number of LKMs that may be loaded make it difficult to get a nice clean audit trail. It is worth looking at the output from modinfo, and getting some idea of what may be considered normal for a system. Keep in mind that hidden modules will not show, and often, the activities of the unauthorized LKMs will be using the standard modules as well.

In section 2.2, step 5, I mentioned that as the kernel is initializing, it displays the total physical memory and the total available memory after the core kernel was loaded. On the test system I was using, the values were:
 unix: mem = 49152K (0x3000000)
 unix: avail mem = 44257280
The installed core kernel image was 5932K. This value is worth noting, as this size should not usually change for a stable hardware configuration.

Another interesting audit that could be performed, but would take some system programming, is to walk through the linked list of module structures (see section 3.1) after the system had been running normally for some period of time. This could produce a list of all the modules commonly accessed by the running system. It would also be worthwhile to check this linked list occasionally for any modules with names set to null or other strange names.

Auditing system calls might detect unusual calls to suspicious system functions, such as newcreat64(). Performance might be a big issue if all processes had all system calls logged. Again, I have not encountered a tool to do this and would require some system level programming. Also, it would not be too difficult to create your own LKM that intercepts system calls specifically to create modules, so that each time a module is loaded, it could be logged using the cmn_err() syscall. An auditing technique similar to auditing system calls is to monitor and log execve() calls and trigger actions as a result of irregular activity. Finally, an initial audit of the system call table *sysent* itself after boot up would provide the basis for monitoring changes to the table.

There is a method available to watch, in real time, the automatic loading and unloading of kernel modules by setting the variable *moddebug* in the kernel using the *adb* command according as follows:

```
# adb –kw /dev/ksyms /dev/mem
physmem 1661
moddebug /W 0x80000000
moddebug:  0x0 = 0x80000000
```

While running this on my test system, the output showed the following when I loaded the sitf0.2 module:

```
unix:  load '/home/gcih/slkm-1.0/sitf0.2' id 92 loaded @ 0xf5b53db4/0xf5af6eac size 2985/108
unix:  installing sitf0.2, module id 92.
```

As seen in sections 7.2 – 7.4 above, the module was not listed by modinfo. It does correlate with the above load addresses. The first load address indicates the text segment of the module while the second segment indicates the data segment of the module. This is a good indication that a useful monitoring tool or script could be developed.

8.0 How to Protect Against it

I have found no specific tools or articles that focus on hardening the kernel to protect against unauthorized module loading, but this topic appears to be gaining attention. The following suggestions come from a variety of sources listed in the references section. Some of them come from those who are "exposing" this exploit to the Internet community and after detailing what mischief can be done, offer a few suggestions on possible techniques to protect the kernel. Others come from reliable sources within the security community. Most are in the realm of possibilities, or proof of concept stage at this point, rather than actual procedures available for download.

8.1 Creating a Monolithic Kernel is Not an Option

The most common suggestion I encountered was to disable the LKMs capability. This is not possible for Solaris (4), and even for a system like Linux, this seems unfeasible, since more modules are created as LKMs to keep the core kernel to a size that would fit on a floppy disk.

8.2 The Kernel Search Path

When a call is made to load a LKM, the kernel searches for it based on a search path variable as mentioned in section 2.2, step 4. By narrowly defining and protecting this path, it may be possible to limit where the LKMs binary images may be loaded from.

Initially, the path is retrieved from PROM. The OBP program has two security modes, one which prevents EEPROM changes and hardware command execution while at the OPB level, and a full security mode that adds the additional requirement that the system will not boot without the correct OBP password. This can be set from the OBP prompt using the command:
Ok *setenv security-mode level* where level is either *command* or *full*
This can also be done from a root shell using the command:
eeprom security-mode=level where level is either *command* or *full*

At a later stage, the path variable is read in from the kernel configuration file */etc/system*. Obviously if the attacker already has root privilege, this file could be modified, so its integrity would be critical.

8.3 LKMs Loading from Readonly Media

If the search path can be secured, then limiting LKMs loading from readonly media could secure these modules. Running a system from a CD has been suggested as a general defense against rootkits, and is used in the incident handler's jumpkit to maintain known good system command.

8.4 Disabling Specific LKMs Loading

Another aspect of the */etc/system* file is the ability to not only forceload an LKM, but to exclude an LKM from being loaded. A list of modules to exclude is created from all of the exclude statements in this kernel configuration file. This might be useful in disabling certain capabilities of a system, but is probably of limited use, based on the types of exploit features mentioned in section 5.0.

Unfortunately, the default is to include a LKM as loadable. What would be useful is to specify which modules could be loaded, and once loaded, which module could not be unloaded. This generally defeats the purpose of LKMs, and is a backwards way of creating a monolithic kernel, but it could be a way of securing critical modules during the initial booting of a system. Pragmatic give some example code for the scheme in his Linux paper.

8.5 Encryption and Authentication

A fellow system administrator put the need for encryption and authentication succulently. "What is needed it a tool that verifies the kernel and LKMs signatures (md5 hashes) before loading into memory, and that can verify these signatures on the fly. It would provide a means of determining if the system was executing truly known code, without having to reboot the system to get it back to a known good state." (7)

Again, pragmatic gives some example code as a starting point for authenticating module loading in his Linux paper and some thoughts on using md5 hashes in his FreeBSD paper. I have not seen anything for Solaris as yet.

8.6 Kernel Hypervisors to Secure Applications

An interesting paper I came across entitled "Using Kernel Hypervisors to Secure Applications" by Mitchem, Lu and O'Brien written in December of 1997, proposed the concept of using LKMs to provide security wrappers for user application. In essence, this is a tcp_wrappers idea, implemented at the kernel level. Another avenue of research might be whether this concept could be extended to wrapping other LKMs. They might intercept system calls to other modules, verify the integrity of the module, do any additional fine grained security controls or authentication, and logging. Their URL is: www.securecomputing.com/khyper.

8.7 Runtime Kernel Patching

What makes defense against this kind of exploit extremely difficult is that, first of all, the attacker has root level access, and secondly, is working with kernel processes which have privileged access to all the kernel objects. What could be worse?

An paper released in November of 1998 by Silvio Cesare entitled "Runtime Kernel Kmem Patching" described the technique of modifying a running Linux kernel using direct access to kernel memory. Even a monolithic kernel would be vulnerable to such techniques.

8.8 Final Comments

Kernel rootkits are an extremely difficult and insidious exploit to detect and defend against. Although it requires a higher skill level, it is not that difficult, and others will develop the nice kinds of interfaces that will broaden the base of potential attackers. For these reasons, plus the role that Solaris servers play in the corporate world, Solaris kernel rootkits are going to be a severe problem if counter measures are not taken. Research and development along the lines discussed above could provide some additional lines of defense against kernel exploits. As one individual at a website that "exposes" vulnerabilities stated, "Security is an illusion. It's really just called 'risk management.'" (8)

9.0 Sited References

(1) Rusling, David A. "The Linux Kernel." 1999. http://www.linuxHQ.com/guides/TLK/tlk.html (April 4, 2001)

(2) Mauro, Jim, Richard McDougall. *Solaris Internals.* Palo Alto, CA: Sun Microsystems Press, 2001.

(3) Maxwell, Scott. *Linux Core Kernel Commentary.* Scottsdale, AZ: Coriolis Open Press, 1999.

(4) Mauro, Jim. Sun Microsystems. Personal Correspondence. March 27, 2001. solaris-internals-feedback@devnull.eng.sun.com.

(5) Boran, Sean. "Weekly Solaris Security Digest 2001/01/22 to 2001/01/28." January 29, 2001. http://securityportal.com/topnews/weekly/solaris20010129.html (April 4, 2001)

(6) Dittrich, Dave. "Root Kits and hiding files/directories/processes after a break-in." March 7, 2001. http://staff.washington.edu/dittrich/misc/faqs/rootkits.faq (April 4, 2001)

(7) Plotner, Steffen. Yankee Environment Systems. Turners Falls, MA. Personal Correspondence. April 3, 2001.

(8) Hoglund, Greg. "A Moment of Clarity." Unspecified 2001. http://www.rootkit.com/ (February 23, 2001)

9.1 Additional References:

Cesare, Silvio. "Runtime Kernel KMEM Patching." November, 1998. URL: http://www.big.net.au/~silvio/runtime-kernel-kmem-patching.txt (April 4, 2001)

Clemens, Jonathan. "Knark: Linux Kernel Subversion." Unspecified 2000. URL: http://www.sans.org/newlook/resources/IDFAQ/knark.htm (April 4, 2001)

Mauro, Jim. "The dynamic Solaris kernel" February, 2000. URL: http://www.unixinsider.com/swol-02-2000/swol-02-insidesolaris.html (March 5, 2001)

Mauro, Jim. "The kernel directory" April, 2000. URL: http://www.unixinsider.com/swol-04-2000/swol-04-insidesolaris.html (March 5, 2001)

Mitchem, Terrence, Raymond Lu, and Richard O'Brien. "Using Kernel Hypervisors to Secure Applications." December, 1997. URL: http://www.securecomputing.com/khyper/acsac97.pdf (April 4, 2001)

Plaguez (pseud.). "Weakening the Linux Kernel." Phrack. No. 52. January 26, 1998. URL: http://packetstorm.securify.com/mag/phrack/phrack52/P52-18 (April 4, 2001)

Plasmoid (pseud.). "Solaris Loadable Kernel Modules." Unspecified 1999. http://packetstorm.securify.com/groups/thc/slkm-1.0.html (April 4, 2001)

Pragmatic (pseud.). "(nearly) Complete Linux Loadable Kernel Modules", March, 1999. URL: http:// packetstorm.securify.com/docs/hack/LKM_HACKING.html (April 4, 2001)

Pragmatic (pseud.). “Attacking FreeBSD with Kernel Modules.” June, 1999. URL: http://packetstorm.securify.com/groups/thc/bsdkern.html (April 4, 2001)

9.2 Useful Man Pages:

adb(1)
dump(1)
_info(9E)
ksyms(7D)
modinfo(1)
mod_install(9F)
modldrv(9S)
nm(1)
savecore(1M)
system(4)

9.3 Location of Exploit Source Code:

Plasmoid (pseud.) *slkm-1.0.tar.gz* December 20, 1999. URL: http://packetstorm.securify.com/groups/thc/slkm-1.0.tar.gz (April 4, 2001)